Memory Reference Locality in Binary Search Trees

Master's Thesis

Kenneth Oksanen

Teknillinen korkeakoulu	Helsinki University of Technology
Tietotekniikan osasto	Faculty of Information Technology
Tietojenksittelytekniikan laitos	Department of Computer Science

Otaniemi 1995

HELSINKI UNIVERSITY OF TECHNOLOGY		ABSTRACT OF THE MASTER'S THESIS
Author:	Kenneth Oksanen	
Name of the thesis:	Memory Reference Locality in Binary Search Trees	
Date:	May 30, 1995	**Number of pages:** 52
Department:	Faculty of Information Technology	**Professorship:** Tik-76
Supervisor:	Professor Eljas Soisalon-Soininen	
Instructor:	Lecturer Lauri Malmi	

Balanced binary search trees are widely used main memory index structures. They provide for logarithmic cost for searching, insertion, deletion, and efficient ordered scanning of keys.

Long term trends in computer technology have emphasized the effect of memory reference locality on algorithm performance. For example, the search performance of large structurally equivalent binary trees can double if nodes are located optimally in memory relative to each other. Unfortunately the traditional Random Access Memory (RAM) model cannot distinguish algorithms with good memory reference locality from algorithms with poor memory reference locality. We therefore define a new memory model, the Modular Block Cache (MBC) memory model, and present some mathematical tools to be used with it.

We analyze the behavior of searches in b-way trees in the MBC model. The analysis is used to compare T-trees, B-trees, various memory layouts of binary trees, and the effect of various placements of keys and data.

In this thesis, we will also present two new methods to improve the memory reference locality of binary search trees. Periodic relocation rebuilds the tree so that the nodes are located optimally in memory. AB-trees are based on representing the internal structure of B-tree nodes with AVL-trees instead of an arrays. We will discuss the advantages and drawbacks of these methods and discuss their feasibility in various circumstances.

Keywords: Analysis of algorithms, program behavior, model of computation, memory model, memory hierarchy, cache memories, translation lookaside buffer (TLB), spatial locality, binary search trees, AVL-trees, B-trees.

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Kenneth Oksanen		
Työn nimi:	Muistiviittauksien paikallisuus binäärisissä hakupuissa		
Päivämäärä:	30. toukokuuta 1995	**Sivuja:**	52
Osasto:	Tietotekniikan osasto	**Professuuri:**	Tik-76
Työn valvoja:	Professori Eljas Soisalon-Soininen		
Työn ohjaaja:	Lehtori Lauri Malmi		

Tasapainoitetut binääripuut ovat yleisesti käytettyjä keskusmuistitietorakenteita, jotka tarjoavat logaritmisen haku-, lisäys- ja poistoajan sekä mahdollisuuden selata avaimia tehokkaasti niiden suuruusjärjestyksessä.

Tietokoneteknologian pitkäaikaiset trendit ovat painottaneet muistiviittauksien paikallisuuden merkitystä ohjelmien ajoaikaan. Esimerkiksi puun rakennetta muuttamatta on mahdollista kaksinkertaistaa hakunopeus suurissa binääripuissa, jos puun solmut sijoitetaan muistissa toisiinsa nähden parhaimmalla mahdollisella tavalla. Valitettavasti perinteinen RAM-muistimalli ei pysty ottamaan huomioon algoritmien tekemien muistiviittauksien paikallisuutta. Siksi tässä työssä esitetään uusi MBC-muistimalli ja sen soveltamiseen tarvittavia matemaattisia työkaluja.

Muistiviittauksien paikallisuutta b-haarautuvissa puissa analysoidaan matemaattisesti MBC-muistimallissa. Analyysiä sovelletaan eri puutyyppien sekä avaimien ja datan talletusvaihtoehtojen vertailuun.

Tässä työssä esitetään myös kaksi uutta menetelmää, joilla binääripuiden muistiviittauksien paikallisuutta voidaan parantaa. Ensimmäinen menetelmä rakentaa ajoittain koko puun uudelleen niin, että puun solmut sijoitetaan muistissa toisiinsa nähden parhaimmalla mahdollisella tavalla. Jälkimmäinen menetelmä, AB-puu, perustuu siihen, että B-puun solmun sisäisenä tietorakenteena käytetään taulukon sijasta AVL-puuta. Työssä esitetään myös vertailuja näiden kahden ratkaisun soveltuvuudesta eri tilanteisiin.

Avainsanat: Algoritmien analyysi, ohjelmien käyttäytyminen, laskennan malli, muistimalli, muistihierarkia, kätkömuisti, TLB, paikallisuus, binäärinen hakupuu, AVL-puu, B-puu.

Acknowledgements

Swallow all your learning in the morning, but digest it in company in the evenings

Philip Dormer Stanhope,
Earl of Chesterfield (1694-1773)

I wish to express my deepest gratitude to my instructor Lauri Malmi for his diligent guidance, tolerance, and thoughtful support during the deliverance of this thesis. Also my supervisor, professor Eljas Soisalon-Soininen, professors Otto Nurmi, Markku Syrjänen, Heikki Saikkonen, lecturer Olli Yliharja, and researcher Tatu Ylönen have all counseled me on numerous cumbersome issues.

Without the insistence and concern of doctor Patrik Floréen, my dear parents, and professor Eljas Soisalon-Soininen, my graduation, already long overdue, might still be on its way.

There is no doubt that this work has been strongly influenced by the innovative working environment and lengthy midnight debates by the vending machines with my friends Tatu Ylönen, Johannes Helander, Tero Kivinen, Sivalingam Kengatharan, Heikki Suonsivu, and several others, all collectively known as the Gypsy Camp of MIT^2.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> Light thinks it travels faster than anything else, but it is wrong. No matter how fast light travels, it finds darkness has got there first, and is waiting for it.
>
> *Terry Pratchett, "Reaper Man"*

A long term trend has been that processor speed has doubled every two years and memory latencies have halved every ten years. Today elaborate hierarchical memory and cache designs are used to feed the processor with the data it needs [38]. This has emphasized the effect of *locality*; memory references close to recently referenced memory locations are more efficient than memory references elsewhere. In current systems memory accesses with poor locality cost easily 100 times more than memory accesses with good locality.

The potential of long memory latencies has influenced the design of computationally intensive algorithms and the choice of compiler optimizations. Hedayat [22] describes how numerical analysts have always been very aware of the hardware they are using. Several articles have been written on how compilers should reorganize loops so that arrays are scanned in an optimal order [12], or how the runtime system should allocate dynamically used memory [20].

The Random Access Memory (RAM) model traditionally used in algorithm analysis is unable to measure the memory reference locality of algorithms, and more accurate memory models tend to be too complicated for mathematical algorithm analysis. We therefore begin our work by presenting a new memory model, the *Modular Block Cache* (MBC) memory model, as a basis for analyzing the memory reference locality of programs. The fundamental idea in combining mathematical analyzability and accuracy into the same memory model is to give a simplified abstract view of the model to the algorithm analysts, and rules which can be used mechanically to apply the analysis to real computer systems.

Balanced binary search trees, such as AVL-trees and red-black trees, are widely used main memory index structures [1, 19, 27, 33]. They provide for logarithmic cost for searching, insertion, deletion, and, contrary to hashing, they also cater for efficient ordered scanning of keys. Numerous studies have been presented on how to maintain the trees in balance, but little has been done to analyze and improve their memory reference locality. Yet experiments we have conducted with large binary search trees on over ten different computers show that an optimally located binary tree on the average doubles the search performance. Such an improvement is of practical value in applications operating intensively on large trees.

The MBC model is a general purpose memory model, but in this thesis we will concentrate on applying it to the analysis of search trees. We first present a general analysis of b-way search trees of given height and node size. Specializing the general analysis to various trees shows that the highest locality is exhibited by B-trees and binary search trees located in memory so that the top level nodes of subtrees are adjacent to each other.

Keys and data can be stored in the binary tree nodes, separate from the nodes, or so that nodes contain only the most frequently used parts of keys and data. Also this choice has a significant impact on the memory reference locality. We analyze the alternatives and present guidelines to the optimal choice.

The optimal, or nearly optimal, memory layout of binary tree nodes can be maintained either by periodical relocations of all nodes in the tree, or in conjunction with each update operation. Relocation is temptingly simple to implement, but unfortunately updates disturb the relative locations of nodes gradually degrading the locality of the tree. Experiments suggest that periodic relocation is feasible if updates are rare.

Locality can be maintained in conjunction with each update operation by combining balanced binary trees with B-trees. B-trees [9, 15] exhibit good locality in search and scan operations, but have a considerable overhead in insertions and deletions. Small AVL-trees [1] provide for efficient searches and updates, but large AVL-trees have poor locality. The best of both data structures can be combined by using AVL-trees instead of arrays inside the B-tree nodes. This data structure, the AB-tree, provides for both efficient searches and updates, but incurs a memory overhead similar to B-trees.

The rest of this thesis is organized as follows: The MBC memory model and tools to be used with it are presented in Chapter 2. The memory reference locality of various variants and organizations of search trees is analyzed in Chapter 3. Chapters 4 and 5 present two means of maintaining locality: periodic relocation and AB-trees. Chapter 6 discusses the relevance of our work and suggest numerous topics for further research. Chapter 7 summarizes our results.

Chapter 2

Modular Block Caching

> If you are out to describe the truth, leave elegance to the tailor.
>
> *Albert Einstein (1879–1955)*

An ideal memory model must be realistic, able to predict real execution times of programs with sufficient accuracy, and yet it must be simple enough to be amenable for mathematical algorithm analysis. Unfortunately these objectives often contradict, and a suitable balance has to be found. We are willing to sacrifice a few percent error in predicted execution time if it makes mathematical analysis more lenient, but a 50% error would be unacceptable for our purposes.

2.1 Previous Work in Memory Modeling

It seems unlikely there could be a memory model satisfying all purposes. Instead, numerous special purpose memory models have been developed. Hardware designers use trace-driven simulations to learn how to develop better hardware caches. Such models and simulations are very realistic, but rather difficult as a basis for mathematical algorithm analysis.[1]

[1] Agarwal et al. [2] extract statistical information of traces and develop a more tractable model, yet mainly intended for hardware designers.

Several models have been proposed to handle two-level memories and I/O-complexity [5, 17, 23, 28], whereas only few models for multilevel memory hierarchies in single-processor machines have been presented.

Aggarwal et al. [3] proposed a *Hierarchical Memory Model* (HMM) in which memory access time $f(x)$ depends on the address x of the memory unit. For ideal RAM model $f(x) = 1$. They analyzed the complexity of various algorithms including matrix multiplication, sorting, fast fourier transform, and searching in this model using cost functions $f(x) = \lceil \log x \rceil$ and $f(x) = \lceil x^{\alpha} \rceil$, where $\alpha > 0$. The same authors presented also a revised version of HMM, the *Hierarchical Memory with Block Transfer* (BT_f) [4] in which a contiguous block can be copied in unit time per word after the startup time.

The *Limiting Technology RAM model* (LT-RAM) was originally proposed for parallel computation [11] and later applied also to ordinary processing [36]. In this model, the latency of a memory access is proportional to the physical distance the signal must travel.

Neither HMM nor BT_f is derived from common hardware solutions, and neither can be considered particularly realistic. LT-RAM is abstract and can be suitable for asymptotic analysis, but it does not appear to be accurate enough for our purposes.

A major step forward towards realistic memory models was the *Memory Hierarchy model* (MH) of Alpern, Carter, Feig and Selker [7]. The MH model consists of a sequence of *memory modules.* A module at *level* u is specified by a tuple $\langle s_u, n_u, l_u \rangle$, where s_u denotes the size of *blocks* in the module, n_u denotes the number of blocks, and l_u denotes the time needed to copy a block between levels u and $u+1$. All computations take place at level 0.

The MH model is the first model in which the effect of caches and translation lookaside buffers (TLBs) on algorithm performance could be analyzed. The authors, however, consider it too hard for asymptotic analysis and simplify it to the *Uniform Memory Hierarchy model* (UMH). In the UMH model the parameters of the memory levels are connected to each other so that for all levels u the *aspect ratio* $\alpha = n_u/s_u$ and the *packing factor* $\rho = s_u/s_{u-1}$ are constant, and the transfer cost between two levels u and $u+1$ is given as function $f(u)$.

We do not necessarily regard the MH model as "too baroque for clean algorithm analysis" [7]. Instead we develop it further and propose a scheme where mathematical algorithm analysis is performed in a simple special case of MH, and the analysis can then be mechanically specialized to arbitrary cache hierarchies by applying given rules.

2.2 Abstract View of Block Caching

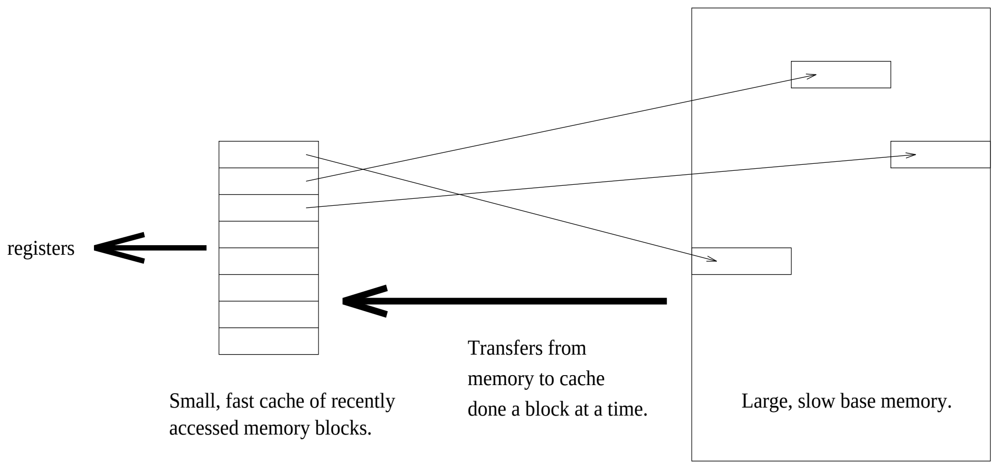

Figure 2.1: Abstract view of block caching

Figure 2.1 depicts the algorithm developer's view of block caching. It consists of exactly three levels of memory modules:

Registers are scarce, and designated to variables and temporaries[2] by the compiler. Registers cannot be accessed as elements in an array or through a pointer.

Base Memory corresponds to RAM in the traditional RAM model.

Cache is a rapidly accessible subset of the base memory.

[2]A temporary value of computation is, for example, $a + b$ in the sum $a + b + c$ if the processor cannot compute the whole sum in one instruction

When the processor issues an instruction to read data from memory to a register, it first checks to see whether the data is in the cache. If the data is in the cache, a *cache hit* has occurred. Otherwise a *cache miss* occurs, and the data is automatically fetched from the slow base memory and inserted to the target register and the cache. The *replacement policy* of the cache dictates what is discarded from the cache in order to make room for the new data.

Data is not transferred from base memory to the cache as individual bytes or words, but as larger blocks of memory. The cache contains n blocks, each containing s bytes. Each block contains the data of consecutive memory locations $ks, ks+1, \ldots, ks+(s-1)$ for some integer k.

The aim of the mathematical analysis of algorithms is to discover the average number of cache misses for the given input size, cache dimensions, and replacement policy. We denote by $C_r(s,n)$ the average number of cache misses for read instructions, or *cache read miss count* for short, and $C_w(s,n)$ respectively for writes. If we set n (or s) to zero, we have the total number of memory reads and writes the algorithm issues.

Analyses should also specify whether they assume a *warm* or a *cold cache.* A cold cache is initially empty, while a warm cache contains data similar to what is left in the cache when the algorithm is finished.

Table 2.1 gives the rest of our notation. A practical example of an analysis of a simple algorithm is given in Section 2.4.

n	number of blocks in the cache	
s	size of blocks in bytes	
$C_r(s,n)$	cache read miss count	
$C_w(s,n)$	cache write miss count	
$C_x(s,n)$	cache access miss count	$= C_r(s,n) + C_w(s,n)$
$R_r(s,n)$	cache read miss ratio	$= C_r(s,n)/C_r(s,0)$
$R_w(s,n)$	cache write miss ratio	$= C_w(s,n)/C_w(s,0)$
$R_x(s,n)$	cache access miss ratio	$= C_x(s,n)/C_x(s,0)$

Table 2.1: Notation for cache miss measures of algorithms

2.2.1 Applicability of the Block Cache Model

So far we have discussed block caching as an abstract concept, intensionally without specifying what hardware, or software, components the concepts "register", "cache" and "base memory" represent. Indeed the block cache abstraction succeeds in capturing many important aspects in numerous real systems, including processor caches, TLBs, and even virtual memory.

Current *processor caches* contain several hundred, even thousands of blocks usually called *cache lines* [38], each containing for example 16 or 32 bytes. *External caches* may contain tens of thousands of cache lines. In most current computers a processor cache hit typically costs from 10 to 30 ns, an external cache hit typically costs 50 to 200 ns, while filling a cache line from main memory typically costs 200 to 500 ns [34].

The *translation lookaside buffer* (TLB) is a cache of recently used mappings from logical addresses seen by the user program to physical addresses understood by the memory hardware [38]. Currently typical TLBs contain from tens up to a few hundred *TLB entries*, each covering the mapping of one virtual memory page, typically 4 or 8 kB. If the physical address being accessed to cannot be mapped by the TLB, a *TLB miss* has occurred and the mapping is resolved by referring to data structures of the operating system. Note that filling the TLB entry does not transfer the data of the virtual memory page but merely gives the processor the ability to access the physical memory that contains the page. For most current computers estimates for the cost of handling TLB misses vary from 20 to 400 cycles [14], and from 150 to 1000 ns [34].

Block caching can also be used to model *virtual memory*. The main memory of the system can be viewed as a cache of many thousands of virtual memory pages. If the virtual memory page is not present in the main memory it must be read from disk at a cost of perhaps tens of milliseconds.

The success of cache memories is explained by *temporal locality* [38] of memory references — most programs tend to access more likely the data they have accessed recently. The most recently accessed data will be cached, reducing the time to access it in the near future. Most programs also exhibit *spatial locality* by accessing more likely memory locations near recently accessed memory locations. It is also more efficient to read a blockful of data at once than to read each data item separately.

2.2.2 Latencies

We divide the cost of executing an algorithm into three parts. The *computational cost* l_c of the algorithm is the average time the algorithm utilizes the processor in arithmetic and logic computations. The *read latency* l_r is the average time the processor waits for data being read to a register, and the *write latency* l_w is correspondingly the average time the processor waits for data being written to the memory. The *execution time* of an algorithm is

$$l_c + C_r(s, 0)\, l_r + C_w(s, 0)\, l_w. \tag{2.1}$$

If we denote the latency of searching data from the cache with l_0 and the latency for retrieving data from the base memory with l_1, we would have $l_r = l_0 + l_1 R_r(s, n)$ in the abstract view of block caching.

Proper modeling of memory writes is more difficult since there is much more variance in hardware mechanisms implementing memory writes than in mechanisms implementing reads. If writes are infrequent, modern processors can continue with further computations in parallel with the writing, and except for TLB misses no considerable latency incurs. Fortunately reads dominate in most applications, and we can approximate $l_w = 0$ except for the TLB misses.

If, however, the application is write-intensive, the bus bandwidth will become a bottleneck. Furthermore, intensive writing increases read latencies since the bus has a higher probability of being reserved by writes when a read instruction is issued. Means for proper modeling of write-intensive programs are not yet thoroughly investigated and further research remains to be done. In the rest of this thesis analyses concentrate on read-intensive algorithms.

2.3 Practical View of Block Caching

Real systems rarely contain only one level or kind of block caching. Instead they contain numerous caches with intricate interactions, caches of "metadata", and several algorithms sharing mutual caches. We will now develop rules that can be used to specialize the abstract analysis of an algorithm to some particular hardware system. The application of these rules is mechanical, and could even be done by the compiler or the runtime, either numerically or symbolically. Possible applications include automatic choice of algorithm, automatic algorithm tuning and adaptation, running time estimation, and aiding the choice of hardware.

As in the Memory Hierarchy model [7], memory modules are specified by tuples $M_u = \langle s_u, n_u, l_u \rangle$ where l_u is the latency of moving a block between memory module u and $u+1$. We often omit the latency if it is redundant. Level 0 denotes the register, level v usually denotes the base memory.

2.3.1 Chained Caches

We assume hereafter that the block replacement policy is random, and no prefetching is employed. This is a reasonable assumption for hardware-implemented caches since sophisticated replacement policies are laborious to implement in hardware. However, the assumption is no more valid, for example, in page replacement where variants of the LRU or GCLOCK [16] are common. Block caching could still be used to analyze the performance of separate memory modules, but the rules given below may not be applicable.

Assuming random replacement is useful in the analysis, since we can employ the following properties:

Property 1. For a random replacement cache and any algorithm $C_r(s,n) \geq C_r(s,n+1)$.

Property 2. The time a data item remains in a random replacement cache $\langle s,n \rangle$ before being replaced is geometrically distributed and the expectation is proportional to n and inversely proportional to the frequency of cache misses. Particularly, the time a data item is cached is independent of how frequently the location is accessed.

We lack an external time frame so we adopt the practice of using the number of memory read instructions issued as our time. We thus write the average caching time as $n/R_r(s,n)$.

Lemma 3. Consider a chain of two random replacement caches $M_1 = \langle s, n_1 \rangle$ and $M_2 = \langle s, n_2 \rangle$, where a data item to be read is first searched in M_1 and if a miss occurs, then from M_2. For any algorithm, the system of these caches has on average at most as many misses as a cache $\langle s, n_2 \rangle$, and at least as many misses as a cache $\langle s, n_1 + n_2 \rangle$.

Proof. Consider the cases where a memory read is concealed from M_2 by a successful search in M_1. Property 2 implies that the miss ratio $R_{r,2}$

of M_2 is not affected by reads which would succeed in M_2, and that reads which would fail in M_2 affect M_2 only by decreasing $R_{r,2}$. This proves the first part of the lemma.

The second part of the lemma gives an upper bound to how much $R_{r,2}$ can decrease due to reads which would fail in M_2 but are concealed by success in M_1. We compare the performance of the system M_1, M_2 to a single random replacement cache $\langle s, n' \rangle$ which we define by requiring it to perform as well as the system M_1, M_2. We therefore have $R_r(s, n') = R_{r,2}$, and by requiring the average caching times of memory locations to be equal we have

$$\frac{n'}{R_{r,2}} = \frac{n_2}{R_{r,2}} + p\frac{n_1}{R_{r,1}},$$

where p is the probability that the memory location remains cached in M_1 when it is replaced from M_2. Since p is a probability, $0 \leq p \leq 1$, and we have

$$\frac{n'}{R_{r,2}} \leq \frac{n_2}{R_{r,2}} + \frac{n_1}{R_{r,1}}.$$

As a miss in M_2 requires a miss in M_1, we have $R_{r,1} > R_{r,2}$ and

$$\frac{n'}{R_{r,2}} \leq \frac{n_2}{R_{r,2}} + \frac{n_1}{R_{r,2}} \quad \Rightarrow \; n' \leq n_2 + n_1$$

□

In practice $n_2 \gg n_1$ as caches later in the chain are much larger than caches close to the register, $R_{r,1} \gg R_{r,2}$, and p is small. The effect of caches close to the registers on the miss ratio of caches distant from the processor is therefore negligible, and n' is close to n_2.

Adding the latency of transferring data between levels u and $u+1$ to l_{u-1} we can approximate a chain of caches $\langle s_{u-1}, n_{u-1}, l_{u-1} \rangle$, $\langle s_u, n_u, l_u \rangle$, $\langle s_{u+1}, n_{u+1} \rangle$ with the chain $\langle s_{u-1}, n_{u-1}, l_{u-1} + l_u R_r(s_u, n_u) \rangle$, $\langle s_{u+1},\ n_{u+1} \rangle$, and vice versa. Applying the approximation repetitively we have the following rule.

Rule 4. A chain of random replacement caches

$$\langle s_{u-1}, n_{u-1}, l_{u-1} \rangle, \langle s_u, n_u, l_u \rangle, \ldots, \langle s_v, n_v \rangle$$

can be reduced to the sequence of random replacement caches

$$\langle s_{u-1}, n_{u-1}, l_{u-1} + \sum_{i=u}^{v-1} l_i R_r(s_i, n_i)\rangle, \langle s_v, n_v\rangle,$$

and vice versa.

The read latency of the cache chain is thus $l_r = l_0 + \sum_{i=1}^{v-1} l_i R_r(s_i, n_i)$.

2.3.2 Shared Caches

Next, consider two processes, P1 and P2 with known cache read miss counts $C_{r,1}(s,n)$ and $C_{r,2}(s,n)$ respectively, accessing separate blocks with constant frequencies, and using a shared cache $M = \langle s, n\rangle$. Our goal is to approximate the behaviour of the two processes running concurrently using known properties of P1, P2, and M.

Denote the number of blocks of P1's data in M by i, and the number of blocks of P2's data by $n-i$. We regard the partition of M between P1's and P2's data as a birth and death Markov process where each state represents some partition $i \in \{0, \ldots, n\}$. P1's part of M increases when P1 issues a read instruction which causes a miss in M, and the data brought into M replaces data of P2. Analogously for the increase of P2's part. We therefore have the relative birth and death probabilities

$$\lambda_i = \frac{C_{r,1}(s,i)}{C_{r,1}(s,i) + C_{r,2}(s,n-i)} \, \frac{n-i}{n}$$
$$\mu_i = \frac{C_{r,2}(s,n-i)}{C_{r,1}(s,i) + C_{r,2}(s,n-i)} \, \frac{i}{n}.$$

In birth and death processes the stationary probabilities π_i and π_{i-1} of adjacent states i and $i-1$ satisfy $\pi_i = \pi_{i-1}\lambda_{i-1}/\mu_i$. A locally maximal stationary probability of state must satisfy

$$\begin{cases} \pi_i \geq \pi_{i-1} \\ \pi_i \geq \pi_{i+1} \end{cases} \Longleftrightarrow \begin{cases} \pi_i \geq \pi_i \mu_i/\lambda_{i-1} \\ \pi_i \geq \pi_i \lambda_i/\mu_{i+1} \end{cases} \Longleftrightarrow \begin{cases} \mu_{i+1} \geq \lambda_i \\ \lambda_{i-1} \geq \mu_i \end{cases}$$

Using Property 1 it is easy to see that $\lambda_i > \lambda_{i+1}$ and $\mu_i < \mu_{i+1}$ for any i. This implies there is either a unique most probable state or two adjacent most probable states. If there is an integer ν which satisfies $\lambda_\nu = \mu_\nu$ then ν satisfies also the above conditions for the most probable partition of M. If no integer ν can be found, but if $C_{r,1}(s,n)$ and $C_{r,2}(s,n)$ are defined, contiguous, and decreasing also for real n, then either $\lfloor \nu \rfloor$ or $\lceil \nu \rceil$ is the most probable state and we use ν as a close approximation of it.

Assume $i > \nu$ and we have an integer ν which satisfies $\lambda_\nu = \mu_\nu$. Because $\mu_{i+1} > \mu_i > \mu_\nu = \lambda_\nu \geq \lambda_{i-1} > \lambda_i$ we have $1 > \lambda_{i-1}/\mu_i > \lambda_i/\mu_{i+1}$. Applying the rule of adjacent stationary probabilities repetitively we have $\pi_i/\pi_\nu = \prod_{j=\nu+1}^{i} \lambda_{j-1}/\mu_j$, a product of $i-\nu$ successively decreasing factors all below 1. Hence π_i decreases rapidly as i grows. An analogous inspection holds for $i < \nu$ and we can see π_i becomes rapidly negligible as i deviates from ν. Hence the terms close to ν are dominant in the sum $\sum_{i=0}^{n}(C_{r,1}(s,i)+C_{r,2}(s,n-i))\,\pi_i$ justifying the approximation $C_{r,1}(s,\nu) + C_{r,2}(s,n-\nu) \approx \sum_{i=0}^{n}(C_{r,1}(s,i) + C_{r,2}(s,n-i))\,\pi_i = E\,[C_{r,1}(s,i) + C_{r,2}(s,n-i)] = C_r(s,n)$. We therefore have

Rule 5. Given two processes, P1 and P2 with known cache read miss counts $C_{r,1}(s,n)$ and $C_{r,2}(s,n)$ accessing separate blocks with constant frequencies, and using a shared cache $\langle s,n\rangle$, the two processes' combined number of cache misses $C_r(s,n)$ is given approximately by

$$\begin{cases} C_r(s,n) = C_{r,1}(s,\nu) + C_{r,2}(s,n-\nu) \\ (n-\nu)\,C_{r,1}(s,\nu) = \nu\,C_{r,2}(s,n-\nu) \\ 0 \leq \nu \leq n \end{cases}$$

This rule has proven useful in many cases, not least in analyzing two algorithms running concurrently, communicating at most through common registers.

2.3.3 Translation Lookaside Buffer

When a TLB miss occurs, the processor calls a dedicated routine called the *TLB miss handler.* The TLB miss handler reads some data from the data structures of the operating system, stores the data to the TLB, and returns allowing the processor to resume the execution of the program. Calling, executing, and returning from the TLB miss handler incurs a computational cost l_{TLB}, and the memory reads performed by the TLB miss handler incur a memory read latency. Modeling the TLB is based on regarding the calls to the TLB miss handler as a process that runs concurrently with the program incurring the TLB misses. We can then apply Rule 5.

There are variations in how much data the TLB miss handler reads and whether the data is stored in caches as in normal memory accesses, but variations to the formula we derive below can capture these discrepancies. We will discuss the case where the TLB miss handler performs normal memory reads to access one element in a large hash table.[3] The elements are small and fit in one cache block, but the references to the hash table have little spatial locality and it is unlikely that two frequently used elements reside in the same block.

Assume the TLB is a random replacement cache $\langle s_{TLB}, n_{TLB}\rangle$, where s_{TLB} denotes the page size and n_{TLB} the number of TLB entries. The probability of a TLB miss is given by $R_x(s_{TLB}, n_{TLB})$. Assuming the TLB miss handler uses ν cache lines, the probability of a TLB miss *and* the TLB miss handling to cause a cache miss is given by $R_x(s_{TLB}, n_{TLB})\, R_r(s_{TLB}, \nu)$, where ν satisfies $(n-\nu)\, R_x(s_{TLB}, n_{TLB})\, R_x(s_{TLB}, \nu) = \nu\, R_r(s, n-\nu)$. As TLBs are usually the smallest of caches, we can apply Rule 4 to contract the TLB and the cache chain. Furthermore including the l_0s caused by the TLB miss handler into l_{TLB} we have the average latency l_r of a memory read

$$l_r = l_0 + \sum_{i=1}^{v-1} l_i R_r(s_i, n_i - \nu_i) + R_x(s_{TLB}, n_{TLB})\, (l_{TLB} + \sum_{i=1}^{v-1} l_i R_r(s_{TLB}, \nu_i)) \tag{2.2}$$

where for all $i \in \{1, \ldots, v-1\}$ holds

$$\begin{cases} (n-\nu)\, R_x(s_{TLB}, n_{TLB})\, R_x(s_{TLB}, \nu) = \nu\, R_r(s, n-\nu) \\ 0 \le \nu_i \le n_i \end{cases}$$

Note that hardware implementors have devised means to perform the TLB search in parallel with the search in M_1, and successful searches in the TLB incur no additional cost [42].

As discussed earlier, in modeling infrequent writes it is sufficient to consider only TLB misses. The average latency l_w of memory writes is thus

$$l_w = R_x(s_{TLB}, n_{TLB})\, (l_{TLB} + \sum_{i=1}^{v-1} l_i R_r(s_{TLB}, \nu_i)) \tag{2.3}$$

[3]This technique is known as hashed page tables [24].

2.4 Example

As an instructive example consider an algorithm that reads a random element of k bytes in an array of N elements employing a warm random replacement cache of n blocks each containing s bytes. Assume also each element fits in one cache block. If the array fits entirely in the cache, no misses occur; otherwise a miss occurs with a probability equal to the ratio of the cached and uncached part of the array.

$$C_{r,array}(N,k;s,n) = \begin{cases} 0, & \text{if } Nk \le sn, \\ 1 - sn/(Nk), & \text{otherwise.} \end{cases} \tag{2.4}$$

As no writes are performed $C_w(N,k;s,n) = 0$.

The mathematical analysis of our example is now complete. Next we demonstrate how to apply the analysis to a particular computer, and compare how our analysis matches the experiments.

We specialize our analysis to a SUN SparcStation 10/402 which has a TLB, a processor cache, but no external cache. Specializing the Formulas 2.2 and 2.1, and performing some simplifications relying on the fact that in our example $C_{r,array}(s,0) = 1$ and $C_{w,array}(s,n) = 0$, we have the total execution time of the algorithm t given by the equations

$$\begin{cases} t \;=\; l_c + l_0 + l_1 C_{r,array}(s_1, n_1 - \nu_1) + \\ \qquad\quad + C_{r,array}(s_{TLB}, n_{TLB})\,(l_{TLB} + l_1 C_{r,array}(s_{TLB}, \nu_1)) \\ (n_1 - \nu_1)\, C_{r,array}(s_{TLB}, n_{TLB})\, C_{r,array}(s_{TLB}, \nu_1) = \nu_1\, C_{r,array}(s_1, n_1 - \nu_1) \\ 0 \le \nu_1 \le n_1 \end{cases}$$

In our machine the TLB has dimensions $\langle s_{TLB} = 4096, n_{TLB} = 62, l_{TLB} \approx 785 \text{ ns}\rangle$ [4], the 16 kB data cache has dimensions $\langle s_1 = 16, n_1 = 1024, l_1 \approx 275 \text{ ns}\rangle$, and we assume $l_0 \approx 20$ ns.

In order to test the above analysis we built a test program which repetitively generates a random number in range $\{0, \ldots, N-1\}$ and reads the corresponding element in the array. The elements are 4 byte words. The computational cost l_c of the test program was measured to be approximately 283 ns per iteration. Figure 2.2 compares the analytical and experimental results of various array sizes N.

[4] The machine has 64 TLB entries, but some are reserved for the operating system and to map the executable binary of the program.

Most values were given by hardware specifications, but the latencies l_0, l_1, l_{TLB}, and l_c were derived from measurements. The computational latency l_c was given by the difference between the execution times of the above algorithm and the same algorithm without the memory reference. The other three latencies were fit in the least squares sense to the measurements given in Figure 2.2. Fortunately there are eighteen data points but only three undetermined variables. The coefficient of determination is ~ 0.99919. The same values of l_0, l_1, and l_{TLB} will be used later in Section 3.2 in comparing predicted and measured performance of tree searches.

The standard RAM model predicts a horizontal line. The coefficient of determination of the RAM model is therefore zero.

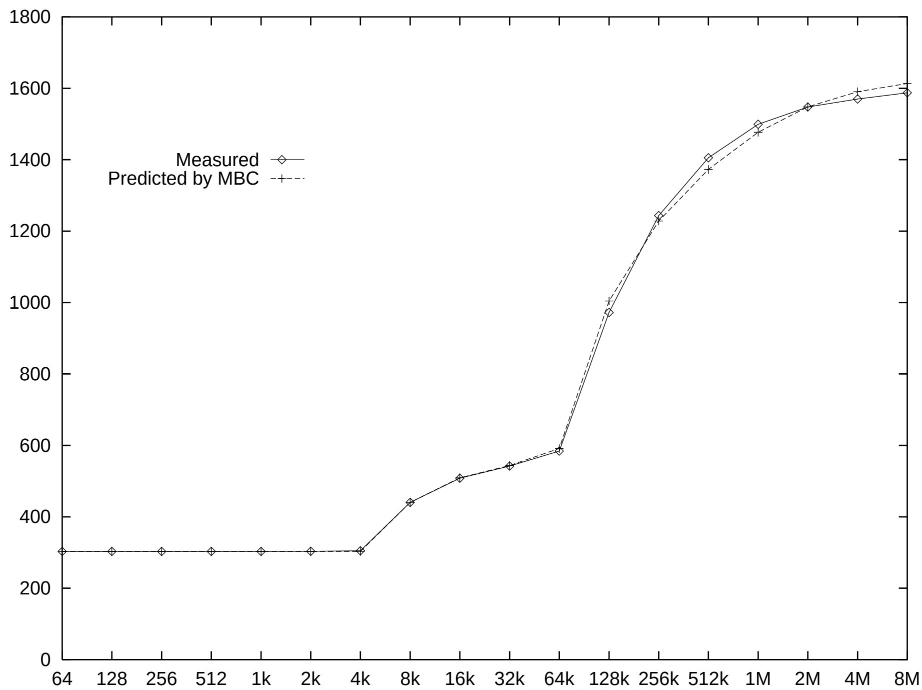

Figure 2.2: Predicted vs. measured latency of reading random elements in an array on a SUN SparcStation 10/402.

Chapter 3

Analysis of b-Way Search Trees

The purpose of computing is insight, not numbers.

R. W. Hamming, "Numerical Methods for Scientists and Engineers", 1973

3.1 Block Cache Misses

We now set our aim to mathematically analyze b-way tree searches in the Modular Block Cache model. The first goal is to find out the cache miss count $C_{r,tree}(b, z, h; s, n)$ of a b-way tree search of height h and with node size z using a warm random replacement block cache $\langle s, n \rangle$. We usually abbreviate $C_{r,tree}(b, z, h; s, n)$ to $C_{r,tree}$. We make the following assumptions:

- The variable b denotes the *branching factor* of the tree, the number of children of each non-leaf node. For binary trees b equals two.
- Each node occupies z bytes, the nodes are located randomly and as tightly as possible, but so that the contents of a node will always be cached in one block. This requires s is an integer multiple of z, and implies that once any field of the record representing a node has been accessed, accessing immediately other fields of the node will not cause a cache miss.

- The tree height is h and the tree is perfectly balanced.
- All searches reach the leaf nodes. This is true for external search trees and for failed searches in internal search trees.
- The keys stored in the tree and the keys to be searched are uniformly distributed.

Let p_i be the probability of a cache miss on the ith level of the tree, p_0 being the probability of a cache miss when referring to the the root of the tree. Because the search procedure accesses exactly once a node on each level, the average number of misses per search $C_{r,tree}$ is the sum of the probabilities of misses on each level of the tree:

$$C_{r,tree} = \sum_{i=0}^{h-1} p_i. \tag{3.1}$$

Two conditions must be satisfied for a cache miss to occur when accessing a node x. First, the cache block previously used to store x must have been replaced. Second, x must not reside on a cache block which happens to be cached due to a recent reference to a nearby node. Denote the probabilities of these two conditions by $p_{i,replaced}$ and $p_{i,random}$, respectively. We have

$$p_i = p_{i,replaced}\, p_{i,random}. \tag{3.2}$$

Because we assume random replacement policy, the probability of a particular cache block not being replaced by a cache miss is $1 - 1/n$, and after k cache misses $(1 - 1/n)^k$.

We assumed a warm cache, which corresponds to having performed infinitely many prior searches. Assume we are about to refer to node x on level i. If we referred to the node in the previous search, on average $C_{r,tree} - p_i$ cache misses have occurred since, and in general, if we previously referred to the node j searches ago, on average $jC_{r,tree} - p_i$ cache misses have occurred.

The probability of x being referred during a search is inversely proportional to the number of nodes on its level, i.e. $1/b^i$. Conversely, the probability of x not being referred to is $1 - 1/b^i$.

The probability of a node being visited after *exactly* j searches and its cache block still being unreplaced is given by

$$\left(1-\frac{1}{b^i}\right)^{j-1}\frac{1}{b^i}\left(1-\frac{1}{n}\right)^{jC_{r,tree}-p_i}.$$

As we assumed a warm cache, j may be arbitrarily large. Summing over j we have a geometric series.

$$\begin{aligned}
p_{i,replaced} &= 1-\sum_{j=1}^{\infty}\left(1-\frac{1}{b^i}\right)^{j-1}\frac{1}{b^i}\left(1-\frac{1}{n}\right)^{jC_{r,tree}-p_i} \\
&= 1-\frac{1}{b^i}\left(1-\frac{1}{n}\right)^{C_{r,tree}-p_i}\sum_{j=0}^{\infty}\left(\left(1-\frac{1}{b^i}\right)\left(1-\frac{1}{n}\right)^{C_{r,tree}}\right)^j \\
&= 1-\frac{1}{b^i}\left(1-\frac{1}{n}\right)^{C_{r,tree}-p_i}\frac{1}{1-(1-1/b^i)\,(1-1/n)^{C_{r,tree}}} \\
&= 1-\frac{(1-1/n)^{-p_i}}{1+((1-1/n)^{-C_{r,tree}}-1)\,b^i}. \qquad (3.3)
\end{aligned}$$

Consider now $p_{i,random}$ when we are about to refer to a node x. Assuming random placement in memory, the probability of some blockful of data being cached is proportional to the number of nodes s/z in that block. However, our premise is now that the block is *not* cached because of a recent reference to node x, but the blockful of data may be cached because of a recent reference to some *other* node in the block. There are $s/z-1$ other nodes in the same block, and as the probability of a cache miss of a random reference is given analogously to Equation 2.4 we have

$$p_{i,random}=\begin{cases}0, & \text{if the tree fits entirely in the cache,}\\ 1-\frac{(s/z-1)}{(s/z)}\frac{n}{m}, & \text{otherwise,}\end{cases}$$

where m is the number of blocks needed to store the tree. We will hereafter concentrate on the case where the tree does not fit in the cache. As there are $(b^h-1)/(b-1)$ nodes in the tree we have

$$p_{i,random}=1-\frac{(s-z)\,(b-1)\,n}{(b^h-1)\,z}.$$

Substituting $p_{i,replaced}$ and $p_{i,random}$ to Equation 3.2 we have

$$p_i=\left(1-\frac{(s-z)\,(b-1)\,n}{(b^h-1)\,z}\right)\left(1-\frac{(1-1/n)^{-p_i}}{1+((1-1/n)^{-C_{r,tree}}-1)\,b^i}\right). \qquad (3.4)$$

Substituting the values p_i to Equation 3.1 and performing some simplifications we obtain

$$C_{r,tree} = \left(1 - \frac{(s-z)\,(b-1)\,n}{(b^h-1)\,z}\right)\left(h - \sum_{i=0}^{h-1} \frac{(1-1/n)^{-p_i}}{1+((1-1/n)^{-C_{r,tree}}-1)\,b^i}\right). \tag{3.5}$$

Unfortunately Equation 3.5 does not appear to be analytically solvable although some simplifying approximations are possible. Because n is largish, up to several hundreds even on TLBs, $1 - 1/n$ is close to 1. The values p_i are probabilities and therefore $-1 \leq -p_i \leq 0$. We can thus approximate

$$C_{r,tree} \approx \left(1 - \frac{(s-z)\,(b-1)\,n}{(b^h-1)\,z}\right)\left(h - \sum_{i=0}^{h-1} \frac{1}{1+((1-1/n)^{-C_{r,tree}}-1)\,b^i}\right). \tag{3.6}$$

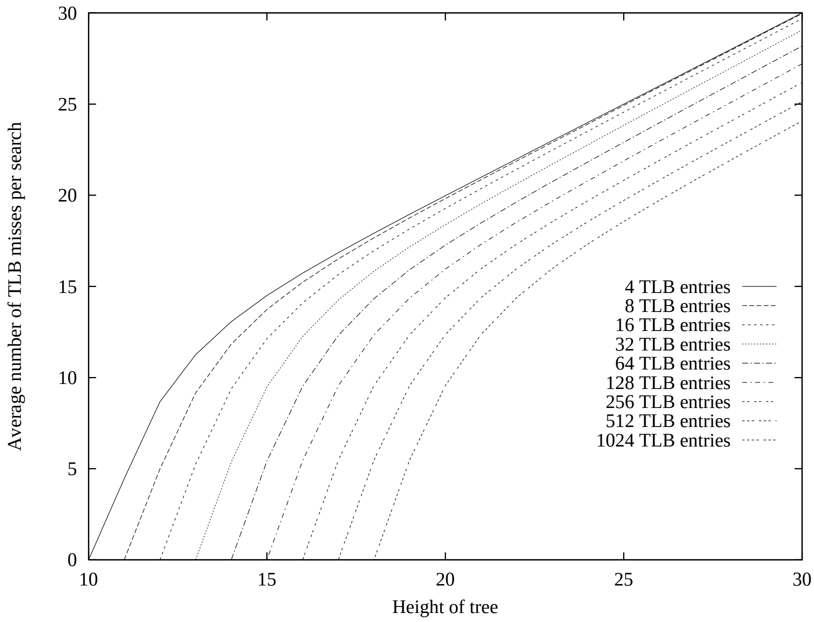

Figure 3.1: Warm random replacement cache miss count of binary tree searches with varying tree height and number of cache blocks for the case $s = 4096$ (typical for TLBs and virtual memory) and the node size $z = 16$.

We have numerically confirmed the approximation from Equation 3.5 to Equation 3.6 to incur at most a few percentage error for typical values of n. Equation 3.6 can be solved numerically to obtain approximations for $C_{r,tree}$. Figure 3.1 illustrates some solutions for binary trees.

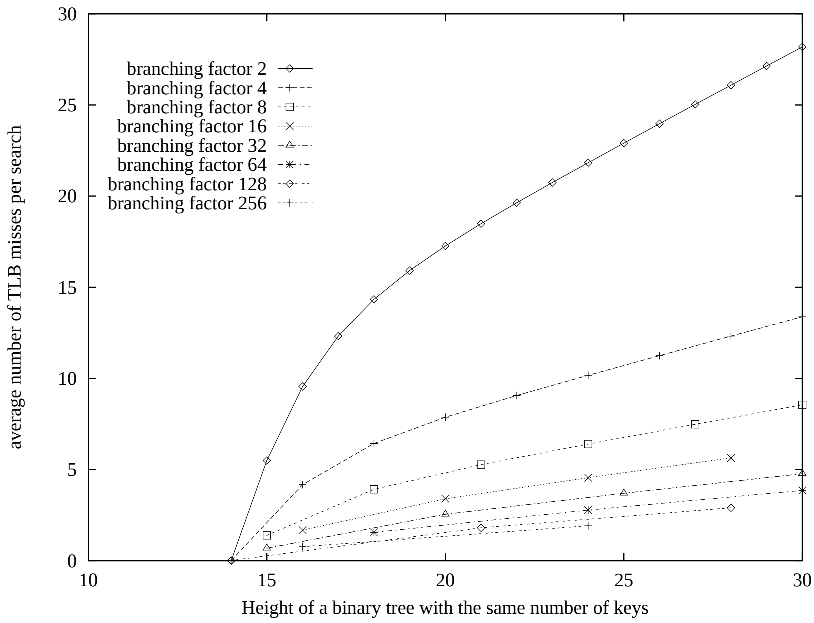

Figure 3.2: Cache miss count in a tree with branching factor b assuming $z = 16\,(b-1), s = 4096, n = 64$, and the number of keys per node is $b-1$.

As the branching factor b increases, the height h of the tree decreases, nodes contain more keys and data, and $C_{r,tree}$ decreases. Figure 3.2 illustrates some cases of b. The minimum number of cache misses is obtained when b is as large as possible and each node is matched to a block.

Our assumption of a uniform distribution of keys is the worst case in terms of the cache miss count. If the access pattern is very skewed, the conscious programmer may either use a self-adjusting tree [10, 37], or rather augment the tree with a small software managed cache [35]. The distribution of searches that miss the cache is less skewed than the distribution of all search requests.

3.2 Experimental Validation

> The fundamental principle of science, the definition almost, is this: the sole test of the validity of any idea is experiment.
>
> *Richard P. Feynman (1918–1988)*

We applied the analysis of the previous section to practice identically to what was demonstrated in Section 2.4. We also performed experiments on a binary tree which was organized in memory so that, for all integers $i \geq 0$, nodes on level $2i + 1$ were located adjacent to their parent. For caches with large blocks this has the effect of having $b = 4, z = 48$, and halving the tree height. Therefore TLB performance will improve, but as $s_1 = 16$, processor cache performance will be unaffected. Note that we use the standard binary tree search in both cases.

The computational latency was $l_c = 400$ ns $+ h_2 200$ ns, where h_2 is the height of the binary tree. The fixed 400 ns cost is caused by function call overhead and generating a random key to be searched. Comparison of predicted and measured performance is presented in Figure 3.3.

The measurements were performed 100 times. The 20 slowest executions were discarded because they were slowed down by cache pollution caused by other processes on the machine. The average of the undiscarded execution times were used for the plot. All operations were performed in main memory, and no disk I/O was involved.

Unfortunately Equation 3.6 determines $C_{r,tree}$ only for the tree sizes of $b^h - 1$ keys which makes our analysis too coarse for nice graphs. Hereafter we will employ a version of Equation 3.6 which accepts trees of any number of keys, and produces identical results with Equation 3.6 for integer h. The additional premises and treatment of that formula is rather tedious and we have had to omit it.

3.3 Optimal Layout of Binary Trees

Binary trees refer very often to the root and the top levels of the tree. Binary trees have therefore a reasonably good temporal locality. The problem,

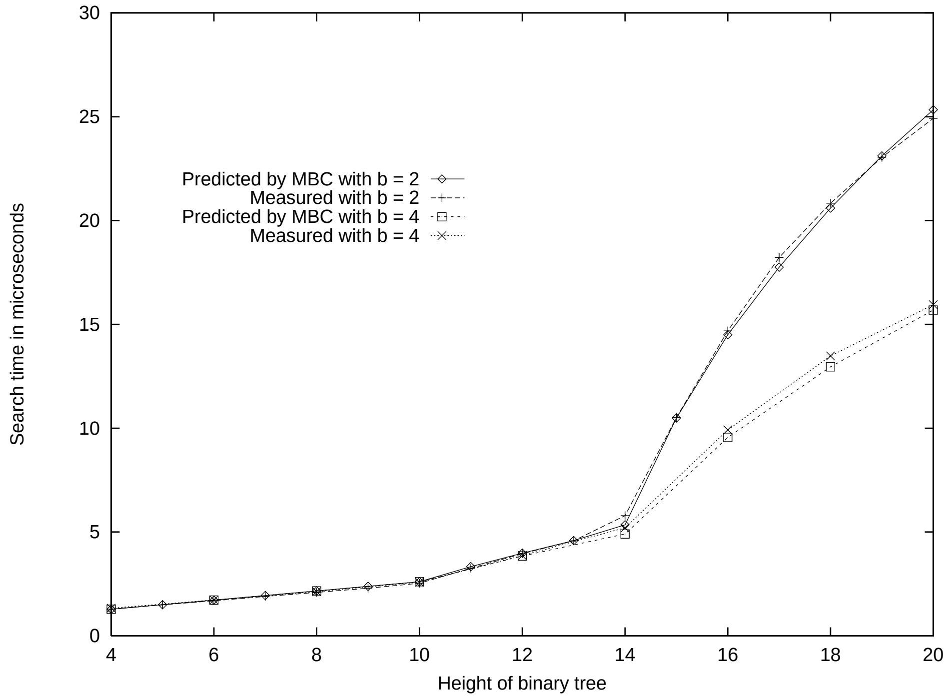

Figure 3.3: Predicted vs. measured cost of searching in binary trees on a SUN SparcStation 10/402.

however, is the lack of spatial locality. Child nodes are often far from their parents, and unless the tree is small or the child node was referred lately, descending to the child node is likely to cause a cache miss.

Cache lines of processor caches and external caches are usually not much larger than a typical binary tree node. Therefore little of the capacity of cache lines are wasted. The matter is very different, however, for disk based systems, virtual memory, and the translation lookaside buffer. Their cache blocks are much larger than typical binary tree nodes. The lack of spatial locality of binary trees has therefore lead to the development of B- and T-trees which are heavily employed in database systems [9, 15, 29].

Spatial locality can also be obtained with binary trees. As depicted in Figure 3.4, the idea is to locate the top levels of a binary subtree within a single block. Dotted lines represent the boundaries of a block.

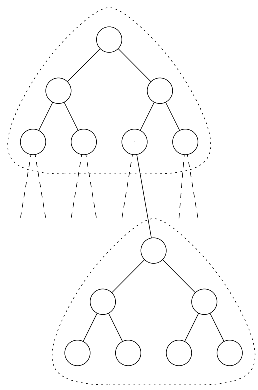

Figure 3.4: Optimal memory layout of a binary tree.

Wilson et al. call this layout of a tree the "hierarchical decomposition" of the tree [41]. Wilson's research was motivated by reducing paging in large LISP systems, but the same principles apply to reduce TLB thrashing.

We will now perform a comparison of various tree organizations using the cache miss count solved numerically from by Equation 3.6. Assume pointers require 4 bytes[1] and the key and data of a node require d bytes. Given d and block size s, the size of a binary tree node is $8 + d$ and the maximal number of nodes that a block can contain is given by $m = \lfloor s/(8+d) \rfloor$. If a binary tree has i internal nodes, it has $i + 1$ leaf nodes. Replacing the external nodes with pointers, we see that a node containing i binary tree nodes has a branching factor $i + 1$. The maximal branching factor is thus given by $b = m + 1 = \lfloor s/(8+d) \rfloor + 1$. The node size is given by $z = (8+d) \lfloor s/(8+d) \rfloor$, and the height of the tree is $h = h_2/log_2 b$, where h_2 is the height of the binary tree.

In B-trees the internal structure of a node is usually an array of pointer-key-value tuples, an additional pointer, and some control information which we assume to use 4 bytes. The size of a b-way B-tree node is thus $z = 8 + b\,(4+d)$, the maximal branching factor is $b = \lfloor (s-8)/(d+4) \rfloor$.

[1]We discuss computers with a 32-bit word size, although 64-bit machines are inevitably entering the scene. However, block and virtual memory page sizes also tend to grow respectively.

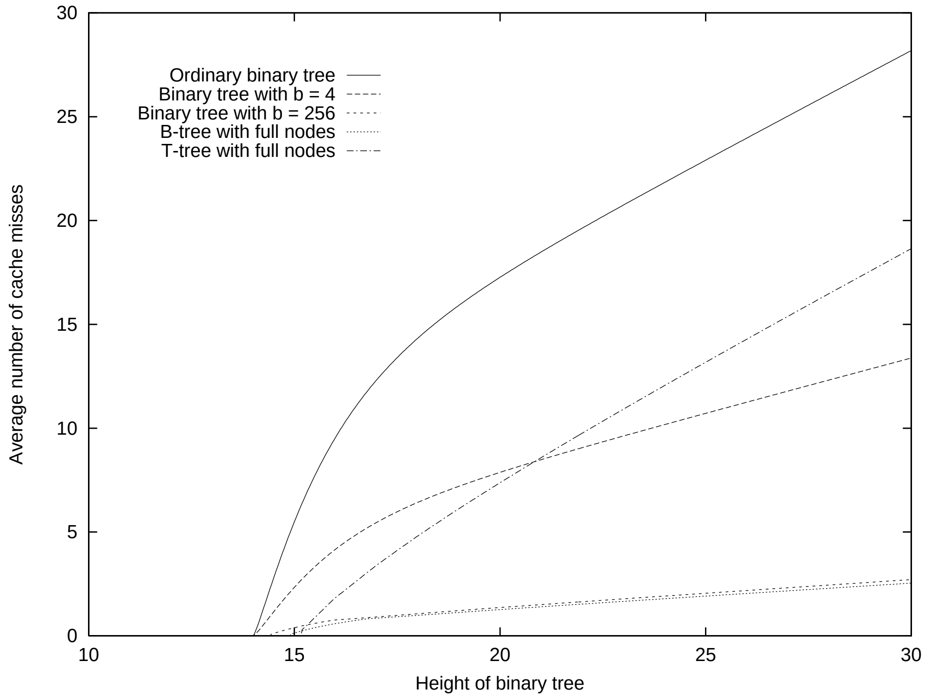

Figure 3.5: Comparison of cache miss counts for various tree types using a cache $\langle s = 4096, n = 64\rangle$, and key, data, and pointer sizes of four bytes.

T-trees [29] were designed for main memory databases. A T-tree node contains an array of key-data tuples, control information, and three pointers: a pointer to two children and the parent. T-trees have a branching factor $b = 2$, but a T-tree node can contain up to $m = \lfloor (s-16)/k \rfloor$ keys. A binary tree of height h_2 contains $2^{h_2} - 1$ keys, which fit in $\lceil (2^{h_2} - 1)/m \rceil$ T-tree nodes. The height of a T-tree with full nodes is thus given by $h = log_2(\lceil (2^{h_2} - 1)/m \rceil + 1) \approx h_2 - log_2 m$.

Figure 3.5 presents a comparison of cache miss counts of various tree types. Evidently the low branching factor of T-trees increases the cache miss count rapidly as the tree grows. B-trees provide for the lowest number of cache misses, but updating B-trees requires costly operations on long arrays. Therefore B-trees are not particularly popular as main memory search trees, but they are very popular in disk based applications where the array operations are dwarfed by even more expensive disk operations. In main memory it is often beneficial to sacrifice a small reduction in branching factor b in order to keep the cost of updates low.

As shown already in Figure 3.3, even a small increase in locality improves the performance of search. In experiments conducted with large optimally located main memory binary trees on over ten different computer systems we observed on average a doubling of search speed compared to randomly located binary trees. Scan operations have a similar speedup, while insertions and deletions have a slightly smaller slowdown because they execute more computational instructions and some memory writes, particularly if they perform balancing.

Note that because a search operation in an optimally located binary tree has a lower cache miss count than a randomly located binary tree, Rule 5 implies the other algorithms of the program get a larger portion of the cache. The performance improvement of the whole program may thus be larger than what the performance increase of the tree search portion of the program suggests.

3.4 Location of Keys and Data

So far we have discussed and experimented with small nodes. But this is not always sufficient — in many cases the key and the data are much larger than the four bytes we have assumed so far. We will first discuss means to cope with larger keys.

Should large keys be stored in the node or outside the node? We call these solutions *key inclusion* and *key exclusion* respectively. In key inclusion the size of the node z increases, which makes the maximal value of the branching factor b decrease, which in turn increases tree height h. All these cause $C_{r,tree}$ to increase. Nevertheless, as long as $b > 2$ the tree will retain some spatial locality compared to a randomly located binary tree. If the key is stored outside the node and no attention is paid to the spatial locality of keys, following the pointer from the node to the key is likely to cause a cache miss in addition to the cache misses caused by traversing in the tree. Therefore, if the sole goal is to minimize the number of cache misses, it is undoubtedly advantageous to store the key in the node. Unfortunately key exclusion is sometimes dictated by other reasons, for example the need to share the key among multiple data structures.

An interesting compromise is to store a part of the key in the node and the full key elsewhere. For example, assume the key to be searched is the string `'Helsinki'`, the search has already descended left from the node `'Hessen'` and right from the node `'Heidelberg'`, the key of the current node is `'Helvetia'`, and the part of the key cached[2] in the node is `'lve'` starting from the third letter. In this case the search procedure can determine the child to go to without accessing the full key at all. If the cached part of the key could not discern the child to descend to, the search procedure reads the full key and updates the cached part so that the next time the decision could be made by looking only at the cached part. We call this technique *key caching*. A restricted variant of this technique is sometimes called *key prefixing*.

It is quite possible that the whole key becomes checked by looking only at the cached parts. In *tries* [18, 8] this happens always.

Apparently key caching can reduce the number of times the search procedure must read the full key, and yet the cached part of the key need not be large. Moving the full key elsewhere will decrease the node size z and increase the branching factor b. We will now investigate whether this increase in b can outweigh the increase of cache misses due to the occasional need to read the full key. The investigation proceeds by regarding the traversing of the optimally located b-way tree and the reading of full keys as separate processes P1 and P2 respectively. We can then apply Rule 5.

We fix $n = 64$ and $s = 4096$. We assume the node contains a 4-byte key cache giving a 20-byte binary tree node. We then have $b = \lfloor s/20 \rfloor + 1 = 205, z = 20\,(b-1) = 4080$ [3]. The cache miss count for P1 is computed by $C_{r,tree}(b, z, h; s, \nu)$. If no effort is put on the spatial locality of the full keys, P2 has only temporal locality it inherits from the binary tree search of P1. If k denotes the key size and p denotes the probability of reading the full key, then the cache miss count for P2 is given by $p\,C_{r,tree}(2, k, h_2; s, n-\nu)$, where $h_2 = h \log_2 b \approx 7.68\,h$, the height of the corresponding binary tree.

[2]SIC! This technique is also a form of caching!

[3]16 bytes will be unused per block, which is not quite according to the requirements of the formula for $C_{r,tree}$. This, however, has a negligible effect.

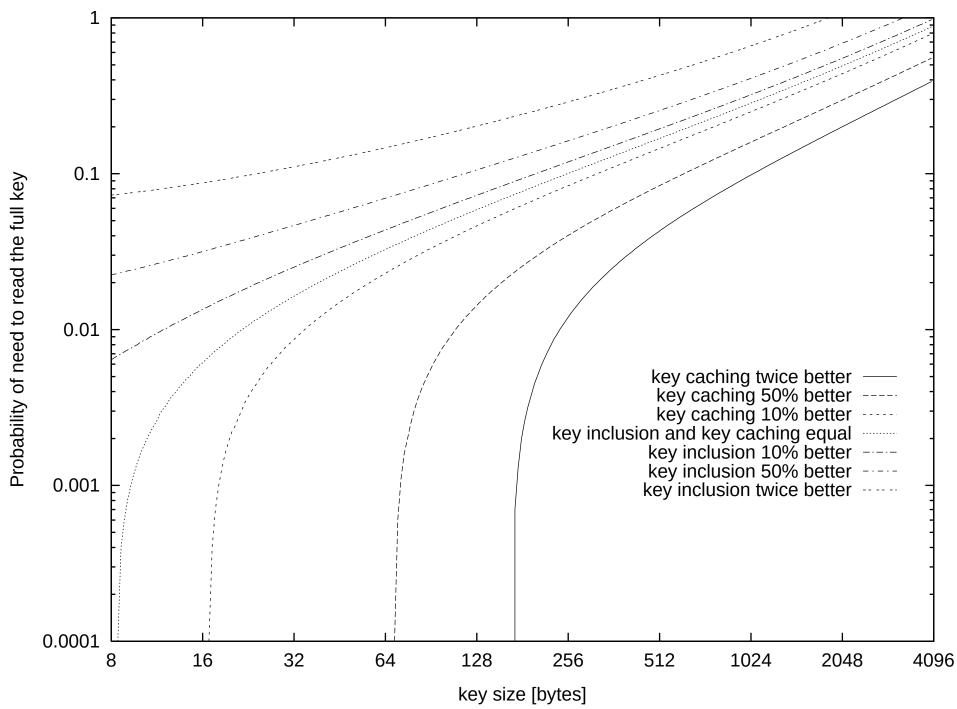

Figure 3.6: The effect of the key size k and the probability p of referring to the full key on the relative cache miss counts of key caching and key inclusion with a cache $\langle s = 4096, n = 64\rangle$ and a tree of one million keys.

Figure 3.6 compares the cache miss counts of key inclusion to key caching with various values of p. The cache miss count for key inclusion is given by $C_{r,tree}(b', z', h'; s, n)$, where $b' = \lfloor s/(12 + k)\rfloor + 1$, $z' = (12 + k)\,(b' - 1)$, and $h' = h_2/\log_2 b'$.

Changes in the size and number of cache blocks (s and n) and the height of the tree h affect the relative performance of key caching and key inclusion surprisingly little as long as the tree is much larger than the cache. The choice between key caching and key inclusion depends mostly on the technology invariant values k and p. The key size k is usually dictated by the application using the tree, while p, the probability of referring to the full key, depends also on the characteristics and distribution of the keys and how numerous aspects of key caching have been designed. Further research in key caching appears warranted.

The two last objections to key caching are the fears for computational overhead and a complex implementation. Given the general trends the computational overhead should not be feared. Actually we have witnessed key caching to be more computationally efficient for large trees and long keys ($k \approx 32$) of English-like text. There will be additional computation due to the bookkeeping performed to know how much of the beginnings of keys are necessarily equal, but the bookkeeping will pay off by speeding up key comparisons in the lower levels of the tree — the beginnings of the keys need not be compared since they are already known to be equal. This surprising result was discovered in comparing implementations of key *inclusion* with and without the bookkeeping required for key caching. The speedup was thus due to decreased computation, not decreased cache miss count.

Admittably key caching is more complex than the standard binary tree search, but it is not overly complex. Even a rather sophisticated implementation fits well within 100 lines of C.

Assuming we have the liberty of choosing where to locate data, then where should we put it? If we use key exclusion the data best be put in the same structure with its key. This will incur no additional cache misses, but it saves us a pointer in the binary tree node allowing us to increase b.

It is considerably harder to decide whether data should be stored in nodes or with full keys when key caching is employed. If key caching is very beneficial, $p \ll 1/h$ and k is largish, but the search succeeds often and the found data is accessed, then locating the data with the full key would be a mistake since referring to the data would increase p. In such cases one should seriously consider storing the data in the node, sacrificing a decrease of b in order to keep p down. If b would decrease considerably, one should use external search trees.

Chapter 4

Periodic Relocation

I remember, I remember,
The house where I was born.
The little window where the sun
Came peeping in the morn.

Thomas Hood (1799-1845)

As with maintenance of balance, also maintenance of spatial locality can be done either periodically for the whole tree or incrementally with each update operation.

4.1 Algorithm for Relocation

We give below an algorithm to rebuild a tree in hierarchical decomposition. The pseudo code uses Algol-like syntax and arrays, but augmented so that the keywords **ref** and **addr of** correspond to C prefix operators `*` and `&` respectively: The expression **ref** *ptr* evaluates to the value pointed to by *ptr* and **addr of** *item* gives the address of *item*. The syntax $ptr \rightarrow field$ is equivalent to $(\mathbf{ref}\ x).field$.

We assume the tree is perfectly balanced so that the root is on level 0 and all leaf nodes are either on level $height - 1$ or $height - 2$. If this is not the case, some global rebalancing algorithm [13, 31, 39] should be employed prior to relocation.

We also assume we have a routine ALLOCATE-NODES which allocates memory for a given number of nodes so that the allocated area does not cross a block boundary. After the relocation the old tree can be freed.

```
COUNT-NODES(tree)
if tree = NIL then
  return 0
else
  return COUNT-NODES(tree → left) + COUNT-NODES(tree → right) + 1
```

```
RELOCATE(tree, height)
if tree = NIL then
  return NIL
if 2^(height−2) ≥ max_nodes_per_page then
  n ← max_nodes_per_page
else
  n ← COUNT-NODES(tree)
mem ← ALLOCATE-NODES(n)
mem[1] ← ref tree
j ← 2
i ← 1
for i ← 1 to n do
  if mem[i].left ≠ NIL then
    if j ≤ n then
      mem[j] ← ref mem[i].left
      mem[i].left ← addr of mem[j]
      j ← j + 1
    else
      mem[i].left ← RELOCATE(mem[i].left, height − ⌊log2 i⌋ − 1)
  if mem[i].right ≠ NIL then
    if j ≤ n then
      mem[j] ← ref mem[i].right
      mem[i].right ← addr of mem[j]
      j ← j + 1
    else
      mem[i].right ← RELOCATE(mem[i].right, height − ⌊log2 i⌋ − 1)
return addr of mem[1]
```

The algorithm works similarly to Wilson's garbage collector [41]. A call to RELOCATE allocates a large contiguous piece of memory, *mem*, into which it copies nodes from the old tree by performing a breadth first traverse. The variable i is used to traverse all binary tree nodes in *mem*. The children of the node at $mem[i]$ are copied from the old tree either to a later place in *mem*, or if *mem* is full, a recursive call to RELOCATE is issued to relocate the subtree. The pointers *left* and *right* in $mem[i]$ are updated to refer to the relocated nodes. The variable j tells where in *mem* to place the next relocatable node. Note that i is always less than j.

The breadth-first traversal of function RELOCATE visits each node once and, due to the limitation on line 3 in RELOCATE, COUNT-ITEMS visits each node atmost twice. The relocation algorithm requires thus linear time.

In a practical implementation the logarithm $\lfloor \log_2 i \rfloor$ is replaced by a variable which is incremented at the end of the loop whenever the loop counter i becomes a power of two, which in turn can be tested by simple bit operations.

4.2 Degradation of Locality

Updates and particularly rotations alter the relative locations of nodes in the tree and gradually degrade the locality. The feasibility of periodic relocation depends on how often the relocation has to be performed. We performed some experiments to see how rapidly various update policies degrade the performance.

The trees used in the experiments contain $N = 2^{16} - 1 = 65535$ nodes, each node using 16 bytes, and the whole tree using one megabyte. The experiments were conducted on a 90 MHz Pentium with a 256 kB external cache and 4 kB pages. This allows us to allocate 8 full levels of nodes on each page. Unfortunately the details of the TLB structure of the Pentium have not been disclosed, but we suspect that the TLB has approximately 64 entries giving it a coverage of 256 kB. Nevertheless, this experiment shows the feasibility of periodic relocation regardless what the characteristics of the TLB are.

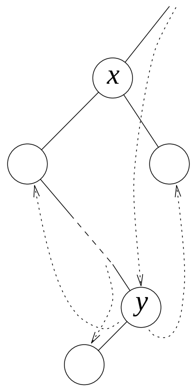

Figure 4.1: Deletion of an internal node x in a binary tree by changing pointers. Dotted lines represent new values of pointers.

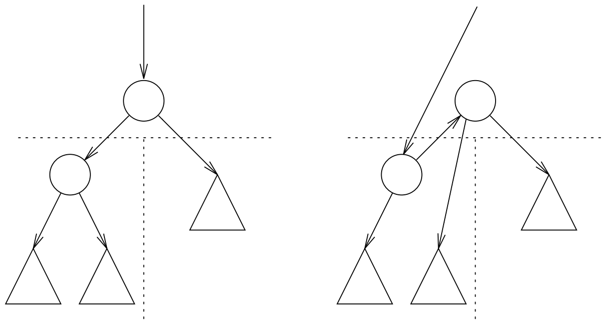

Figure 4.2: Degradation of locality in a subtree near a block boundary before and after a rotation. Dotted lines represent boundaries of blocks.

We experimented with three initial states of the tree. The first, and most efficient ($\sim 4.7\mu$s per search on the Pentium), initial tree is the perfectly balanced binary tree laid optimally out in memory so that no search path has nodes on more than two pages.

The second tree is the result of constructing the tree by its default insertion algorithm allocating the nodes from a list of successive memory locations. For ordinary binary trees the insertion was done in such an order that the resulting tree was completely balanced and a left child was allocated adjacent to its parent node. AVL-trees were constructed by adding the keys sequentially and relying on the AVL balancing algorithm.

The third and slowest tree (over 11 μs) is also perfectly balanced, but constructed from a free list shuffled in memory to random order. Its performance decreases only due to increasing imbalance. Comparing this case to the previous two cases suggests locality can have a larger effect on search performance than a slight imbalance.

There are two common ways to perform deletion in binary trees. The first one changes pointers to lift the rightmost node y of the left subtree to replace the deleted node x (see Figure 4.1), the other copies the key and data of y to x and frees y. Apparently the former method degrades locality much faster than the latter. The latter method was used also in AVL trees.

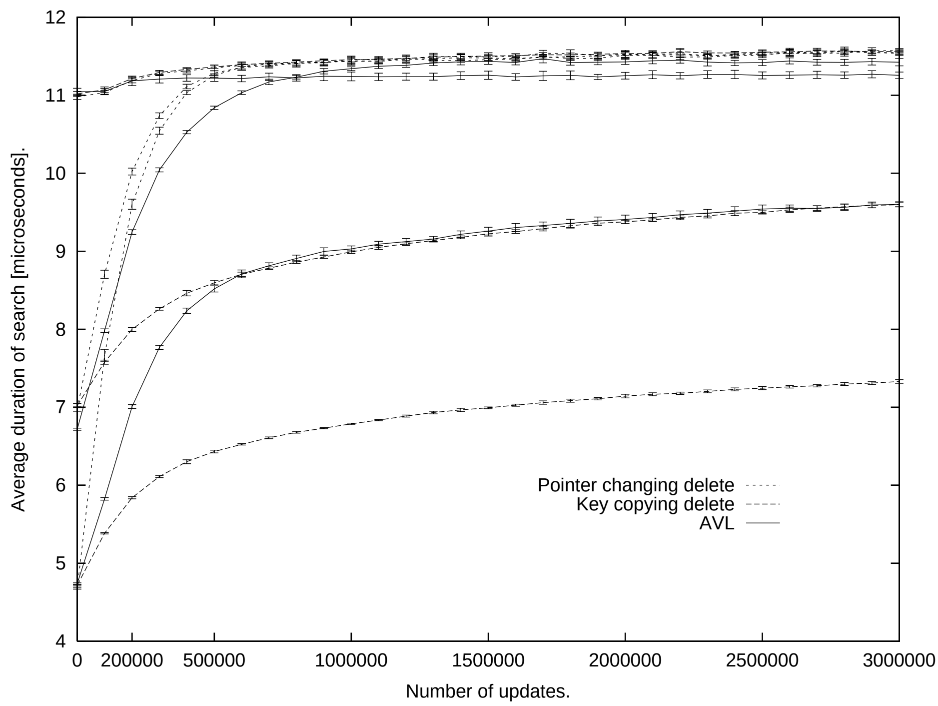

Figure 4.3: Search performance decrease due to locality degradation caused by updates. The figure presents the degradation of locality in the AVL-tree and two versions of binary tree deletion. All algorithms tested with three initial memory layouts: optimal, pre-order allocation of nodes from a linear free list, and random location of nodes. Tree size is 65535.

Test updates for ordinary binary trees were performed so that the key to be inserted or deleted was chosen randomly from the range 0 to $2N$ according to uniform distribution. As the size of the tree is N, approximately half of the searches succeeded. If the key to be inserted was already in the tree or the key to be deleted was not in the tree, nothing was done and another key was chosen instead. This key generation scheme preserves the balance rather well even if no explicit balancing was performed.

All searches were performed using exactly the same search algorithm. Search speeds were measured every 100 000th successful insertion and deletion. During the updates the tree size varied from N to $N - 10000$.

All experiments were performed 100 times. As in Section 3.2 the 20 slowest executions were discarded because they were slowed down by cache pollution caused by other processes on the machine. The undiscarded results were presumed to conform to normal distribution and the average values were used for the plots augmented with 95% confidence intervals.

AVL trees were updated differently because we wanted to stress the balancing in them. The key was chosen at random only with probability $\frac{1}{16}$, otherwise the key was the successor of the previously used key. The rotations performed in AVL trees clearly increase the degradation of locality. This is illustrated in Figure 4.2. Note that rotations rarely propagate to the top of the tree leaving it relatively untouched for rather long. Experiments we omit here for brevity suggest that the longer the successive key sequences are, or the larger the variance in tree size is, the faster the locality of the tree degrades.

Calling the routine RELOCATE periodically is a tempting approach, and if updates are relatively rare and we can tolerate the pauses caused by relocation, periodic relocation appears feasible. But, as experimental results summarized in Figure 4.3 suggest, the periodic relocation may become inferior if updates are common and particularly if balancing is required.

Chapter 5

AB-Trees

> Awaiting the sensation of a short, sharp shock,
> from a cheap and chippy chopper of a big black block.
>
> *William Schwenk Gilbert (1836-1911)*

On basis of results presented earlier in this thesis we set up a task to design and implement a binary search tree with the following properties. First, it should have good memory reference locality and be able to retain it even under heavy update load. Second, it must provide for efficient search and update operations. Third, it should maintain its balance to guard against bad sequences of update data. Fourth, it must have a reasonable memory overhead.

5.1 The Algorithm

B-trees exhibit excellent spatial locality, but because the usual internal representation of B-tree nodes is an array, insertions and deletions in the B-tree node require copying a relatively large amount of memory. This incurs both a considerable computational latency and numerous replacements in the processor and external caches.

AVL-trees, on the other extreme, provide for good insertion and deletion performance, but poor spatial locality. We pick the best of both solutions and replace the arrays of the B-tree nodes with AVL-trees. This new data structure we call the *AB-tree*. To avoid misunderstandings we call the B-tree nodes *clusters*.

The exact description of all operations on AB-trees is too voluminous to fit in this work; of numerous sources that describe the AVL-tree, only a few give an implementation of the delete operation, and we are about to describe something even more verbose. However, a C-implementation is available on request.

The search and scan algorithms for an AB-tree are identical to those of normal binary and AVL-trees. This implies that in all tree types we compare, except the B-tree, we will have the same computational cost.

The insertion to the AB-tree works by bottom-up splitting similarly to the B-tree insertion. Figure 5.1 depicts a step in splitting. Assume we are about to insert the key y to a full cluster P, and the root of the AVL-tree in P has the key x. The cluster P is first split so that the nodes referred to by xl and xr become the roots in the two new clusters. The new key y is then inserted to either one of the new clusters, and the node x is inserted to the parent cluster. If the parent cluster is also full, the splitting continues further up, possibly to the root.

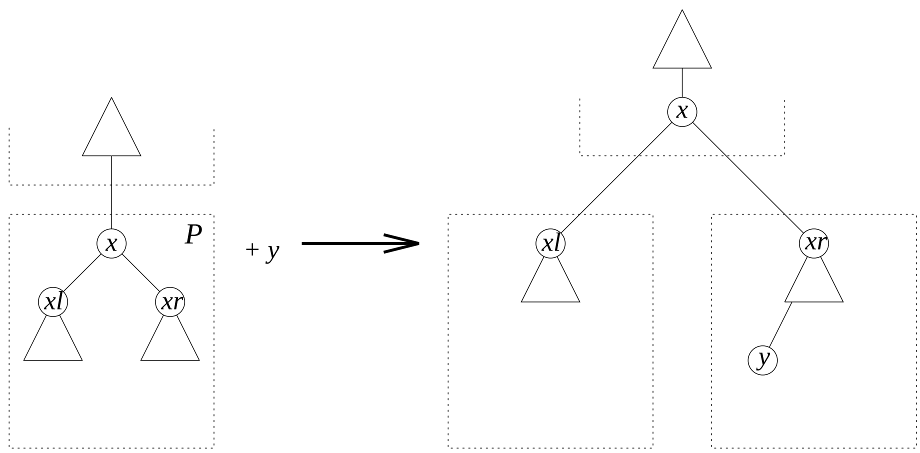

Figure 5.1: Splitting of the cluster P in inserting y. AVL-tree rotations not shown.

It would be possible to lift some other key than x to the parent cluster, but it would probably be computationally inefficient. We have therefore opted for always lifting the root node of the cluster to be split. This also implies we cannot obey the memory-conserving merge-at-half strategy. Fortunately the analysis by Johnson and Shasha [25] indicates that in practice the memory overhead is only slightly larger with the free-at-empty strategy, and the restructuring rate is much lower.

The deletion from the AB-tree is performed top-down so that whenever the deletion algorithm is about to descend to a cluster with only one node in it, the cluster is either *concatenated* [15] into its sibling cluster or the keys are *redistributed* with its sibling cluster. Figure 5.2 depicts this when the deletion algorithm is about to descend from cluster P to cluster L. If two nodes can be inserted to the sibling cluster R, then concatenation takes place by moving x and xl to R. Otherwise the subtree xrl is redistributed to cluster L. Rotations needed to maintain the AVL-trees in balance are again omitted from the figure. Descending to the right cluster is given by a mirror image of Figure 5.2.

Because descending to L first ensures L contains at least two nodes, a nonleaf cluster P can contain a single node only if it is the root cluster. If P is the root cluster, if it contains only one node, and if concatenation takes place, then R becomes the new root and the AB-tree becomes a cluster lower. Because the AB-tree gains height only by splitting the root cluster and loses height only by concatenating the root cluster into another cluster, all leaf clusters are equally far from the root cluster and the tree remains in balance.

There was no compelling reason for choosing AVL-trees instead of some other balanced tree. AVL-trees are well known balanced trees, with well known performance characteristics and good overall performance. We did consider using some tree which allows more imbalance and performs fewer rotations on average. The benefit of fewer rotations would, however, not be very large; in large trees the cost of traversing from the root to the update point is larger than the cost of rotations [26]. Larger imbalance could also risk a more uneven distribution of nodes into the new clusters when splitting.

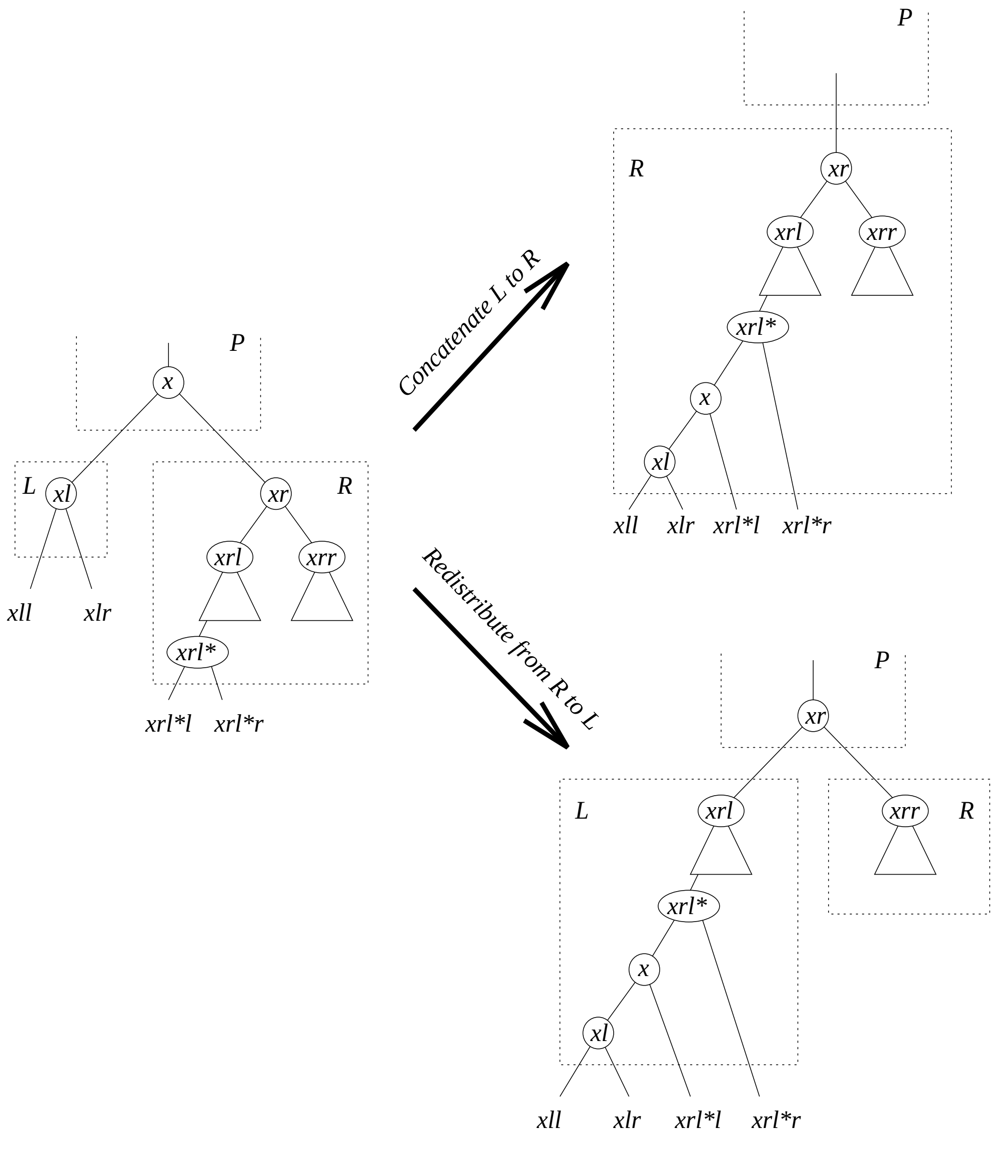

Figure 5.2: Concatenation and redistribution when about to descend to cluster L. AVL-tree rotations not shown.

5.2 Comparison

We now present some comparisons between AVL-, B-, and AB-trees on a SUN SparcStation 10/402. We will not consider T-trees, because the analysis in Section 3.3 showed them to be only a minor improvement over binary trees.

All trees contained 500 000 nodes, each node containing a four byte key and four bytes of data. The effect of tree size on the relative performances of the different tree types is small as long as the tree is much larger than the cache. The selection of keys to be inserted and deleted was done similarly to what was explained in Section 4.2 for AVL-trees.

AVL-trees were constructed either from a shuffled free list or a free list of nodes located adjacent in memory. The former case corresponds to an AVL-tree after numerous updates, the latter to a newly constructed tree.

We used 4 kB B-tree nodes and AB-tree clusters because this is also the size of virtual memory pages, s_{TLB}, on the machine and thus causes least TLB misses. The cost of B-tree insertions and deletions can be reduced by making the B-tree nodes smaller, but it will also decrease the branching factor, which in turn increases the search costs.

In B-trees the *space utilization* of nodes affects the update performance of the tree: if the B-tree nodes are full, inserting a key in the middle of the node will incur more moving of data than if the B-tree node is underutilized. Our implementations of the B- and AB-trees apply the free-at-empty principle [25]. After constructing the trees with pure inserts we obtained approximately a 2/3 space utilization. We then performed sequences of insertions and deletions which gradually decreased the space utilization down to 1/3. In between we performed benchmarks.

To avoid the effect of cache pollution and bus contention caused by other processes on the machine, we performed the AVL-tree tests 40 times and used the median of the timings. For each operation on both B- and AB-trees we performed approximately 2000 measurements with varying space utilizations. The data presented in Figures 5.3 and 5.4 was filtered with a median-of-7 filter.

Apparently B-trees are faster in search operations than AVL-trees, but probably their higher computational cost makes the B-tree slower than the AB-trees although the AB-trees have a little lower branching factor. The cost of deletions is on average 10% higher than the cost of insertions.

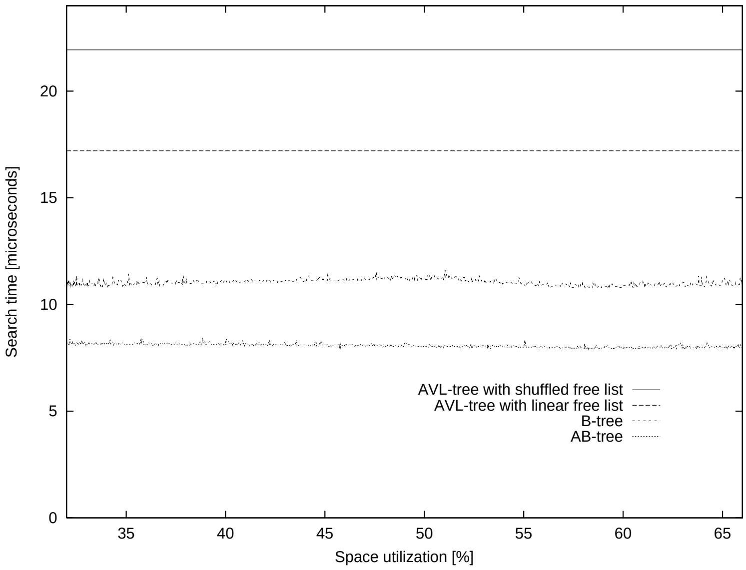

Figure 5.3: Comparison of search costs in a 500 000 node tree on a SUN SparcStation 10/402.

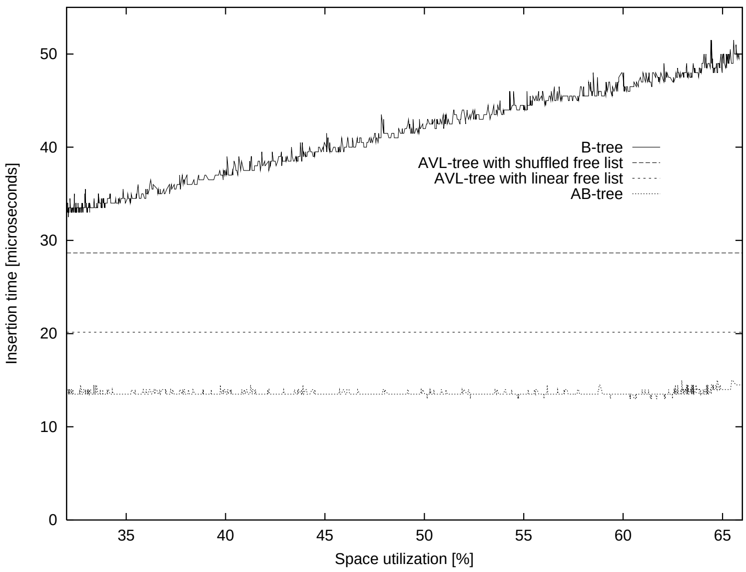

Figure 5.4: Comparison of insertion costs in a 500 000 node tree on a SUN SparcStation 10/402.

Chapter 6

Discussion

> Reason is itself a matter of faith. It is an act of faith to assert that our thoughts have any relation to reality at all.
>
> *G.K. Chesterton (1874–1936), "Orthodoxy".*

6.1 Trends in Hardware

Most of the improvements presented in Chapters 4 and 5 are based on reducing TLB thrashing. This raises the question whether TLB thrashing could be remedied by other, possibly more generic means, by improving the hardware and the operating system.

The simplest approach would be to increase all page sizes uniformly. This, however, would lead to increased overall memory consumption, decreased performance of paging, and other problems in overall system performance.

Many current processors provide for variable size TLB entries, and indeed this could be used to alleviate TLB thrashing. Unfortunately, no common operating system we know of provides means for the user to allocate very large pages. Providing variable size pages requires invasive operating system modifications that introduce considerable overhead [40].

Increasing the number of TLB entries costs expensive on-chip silicon and makes the TLBs slower. Talluri et al. [40] give means to decrease the silicon of a TLB entry by introducing only modest operating system support, but they still fail to provide satisfactory solutions for applications with large working sets.

And finally, the mapping from logical to physical addresses is the basis of so many important operating system services that it is very unlikely to be eliminated, except for some special purpose systems. We are therefore prompted to seek means to avoid TLB thrashing by restructuring the algorithms and data structures.

On the contrary, it appears that trends in parallel computation will introduce new mechanisms with similar behavior to block caching with large blocks. Kendall Square Research KSR1, for example, is a Cache-Only Memory Architecture (COMA) system [21]. KSR1 stores data in units of 16 kB *pages* and 128 byte *subpages*. The movement of data from one processor to another is done transparently a subpage at a time.

Mukherjee et al. [32] discuss three systems providing a partial shared memory abstraction on a message-passing Thinking Machines CM-5: the CHAOS irregular applications library, transparent shared memory (TSM), and extensible shared memory (XSM). All these systems move data from one processor to another in blocks of one or a few kilobytes.

6.2 Further Research Topics

The modular block cache memory model (MBC) can capture many important aspects of translation lookaside buffers as well as processor caches, external caches, disk, and probably also blockwise communication in message-passing or cache-only parallel computers. But much work remains to be done.

So far we have discussed only random replacement policies, but our recent results suggests that Rules 4 and 5 would apply in all replacement policies where successful accesses do not affect the future replacement decisions. Tools for analyzing other common replacement policies, particularly LRU, should be prepared.

We have also neglected two other techniques occasionally applied in caches: *cache hints* and *prefetching.* A cache hint might, for example, tell the cache to contain the data read by a particular instruction for an unusually long or a very short period of time. If the system or the application can anticipate the memory location to be read in the near future, it can prefetch the data so that the eventual read will have a much shorter latency. A crude approach to model cache hints would be to decrease C_r, and to model prefetching one might decrease l_u. The search for more accurate, better justified, and clearly defined means to model cache hints and prefetching is warranted.

The treatment of memory write latency in MBC is also subject to further research. This may require the incorporation of *bandwidth* to the memory model.

Analyzing algorithms in the MBC is rarely as simple as in the example given in Section 2.4. In Section 3 we analyzed searches in a b-way search tree. In Section 3.3 we applied the analysis to various binary trees, T- and B-trees, and in Section 3.4 to key inclusion and key caching. Malmi [30] has analyzed matrix transposition and matrix multiplication. These analyses suggest Rule 5 is a very useful tool. The applicability of MBC depends largely on whether other as useful tools will be discovered. Clearly more hands-on experience is needed in order to evaluate and improve the applicability of MBC in practical algorithm analysis.

Our analysis of key caching techniques showed that key caching may improve locality when the key cannot be included in the binary tree node, or if the key is large enough to significantly decrease the branching factor of the tree. In our experiments we have also observed key caching to reduce the computational cost of comparing large keys. Nevertheless, key caching does not appear to be a widely known and commonly used technique. Further research is needed to establish the benefits, drawbacks, and optimal implementation details of key caching.

We demonstrated the degradation of locality with experimental tests, but we have also made attempts to analyze degradation of locality mathematically with fringe analysis [43]. Unfortunately fringe analysis is notoriously hard and laborious, beyond the scope of this work.

It is possible to decelerate the degradation of locality by splitting the global list of free nodes into several small lists. When a node is inserted into the tree, it is allocated from the same freelist than its parent node. There appears to be an abundance of similar techniques, all of which should be investigated and evaluated.

AB-trees could be an improvement over B-trees also in database systems. Disks have developed tremendously in recent years and can now read and write several megabytes of data per second. If sufficiently many disks are attached to the computer and the transaction profile is suitable, CPU capacity becomes the performance bottleneck. If B-trees were replaced by AB-trees, insertions and deletions could be done more efficiently.

Researchers in numerical analysis often present easy to follow heuristics for choosing an appropriate algorithm for a given problem. Few such heuristics have been given to a layman wondering which search tree to use.

Chapter 7

Summary

I have crossed oceans of time to find you.

Vlad Dracula
in Bram Stoker's Dracula (1992)

We have presented a new general purpose memory model, the Modular Block Cache (MBC) memory model, for assessing the cost of memory references. It combines accuracy and analyzability by presenting a simple abstract view of the model to the algorithm analyst, and rules which can be used to apply the analysis to real computer systems.

We applied the MBC model to analyze the locality of searching in a b-way search tree. The analysis was used to compare various tree types and to suggest the optimal placing of keys and data. According to the analysis, T-trees improve memory reference locality only nominally compared to standard binary trees, but B-trees and binary trees with optimal node placement exhibit good locality. The analysis also suggests that key caching can be used to improve the memory reference locality if the keys are large or they cannot be placed in the binary tree nodes.

We have presented two methods, periodic relocation and AB-trees, to enhance memory reference locality of binary search trees. Periodic relocation is feasible when updates and the rotations needed for balancing are rare. AB-trees are based on embedding AVL-trees inside B-tree nodes. AB-trees have a memory overhead similar to B-trees, but sustain good performance of all operations.

Bibliography

[1] G.M. Adel'son-Vel'skii, Y.M. Landis, An Algorithm for the Organisation of Information. *Doklady Akademi Nauk* **146** (1962), pp. 263–266. English translation in *Soviet Math. Dokl.* **3**, pp. 1259–1263.

[2] A. Agarwal, M. Horowitz, J. Hennessy, An Analytical Cache Model. *ACM Transactions of Computer Systems*, **7**:2 (1989), pp. 185–215.

[3] A. Aggarwal, B. Alpern, A.K. Chandra, M. Snir, A Model for Hierarchical Memory. *Proceedings of 19th Annual ACM Symposium on Theory of Computing* (1987), pp. 305–314.

[4] A. Aggarwal, A.K. Chandra, M. Snir, Hierachical Memory with Block Transfer. *Proceedings of 28th Symposium on Foundations of Computer Science* (1987), pp. 204–216.

[5] A. Aggarwal, J. Vitter, IO Complexity of Sorting and Related Problems, *Communications of the ACM*, **31**:9 (1988), pp. 305-314.

[6] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley (1974).

[7] B. Alpern, L. Carter, E. Feig, T. Selker, The Uniform Memory Hierarchy Model Of Computation, *Algorithmica*, **12** (1994), pp. 72–109.

[8] J. Aoe, K. Motomoto, T. Sato, An Efficient Implementation of Trie Structures, *Software – Practice and Experience*, **22**:9 (1992), pp. 695–721.

[9] R. Bayer, E. McCreight, Organization and Maintenance of Large Ordered Indexes, *Acta Informatica* **1** (1972), pp. 173-189.

[10] J. Bell, G. Gupta, An Evaluation of Self-Adjusting Binary Search Tree Techniques, *Software – Practice and Experience*, **23**:4 (1993), pp. 369–382.

[11] G. Bilardi, F.P. Preparata, Horizons of Parallel Computations. *Proceedings of 25th Anniversary of INRIA*, Bensoussan, Verjus (Eds.), Paris, France (1992).

[12] S. Carr, K.S. McKinley, C-W. Tseng, Compiler Optimizations for Improving Data Locality, *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (1994), pp. 252-262.

[13] H. Chang, S.S. Iyengar, Efficient Algorithms to Globally Balance a Binary Search Tree. *Communications of the ACM* **27**:7 (1984), pp. 695–702.

[14] J.B. Chen, A. Borg, N.P. Jouppi, A Simulation Based Study of TLB Performance. *Proceedings of the 19th Annual International Symposium on Computer Architecture* (1992), pp. 114–123.

[15] D. Comer, The Ubiquitous B-tree. *ACM Computing Surveys* **11** (1979), pp. 121–137.

[16] W. Effelsberg, Principles of Database Buffer Management, *ACM Transactions on Database Systems*, **9**:4 (1984), pp. 560–595.

[17] R.W. Floyd, Permuting Information in Idealized Two-Level Storage, *Complexity of Computer Computations*, Plenum, New York (1972), pp. 105-109.

[18] E. Fredkin, Trie Memory, *Communications of the ACM*, **3**:9 (1960), pp. 490–500.

[19] L.J. Guibas, R. Sedgewick, A Dichromatic Framework for Balanced Trees. *Proceedings of the 19th Symposium on Foundations of Computer Science* (1983), pp. 8–21.

[20] D. Grunwald, B. Zorn, R. Henderson, Improving the Cache Locality of Memory Allocation, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation* (1993), pp. 177–186.

[21] E. Hagersten, A. Landin, S. Haridi, DMM – A Cache-Only Memory Architecture, *Computer*, **25**:9 (1992), pp. 45–54.

[22] G.A. Hedayat, Numerical Linear Algebra and Computer Architecture: An Evolving Interaction, *Technical Report UMCS-93-1-5*, Department of Computer Science, University of Manchester (1993).

[23] J.-W. Hong, H.T. Kung, I/O Complexity: The Red-Blue Pebble Game, *Proceedings of 13th Annual ACM Symposium on Theory of Computing* (1981), pp. 326–333.

[24] J. Huck, J. Hays, Architectural Support for Translation Table Management in Large Address Space Machines, *Proceedings of the 20th Annual International Symposium on Computer Architecture* (1993), pp. 39–50.

[25] T. Johnson, D. Shasha, B-Trees with Inserts and Deletes: Why Free-at-Empty is Better Than Merge-at-Half, *Journal of Computer and System Sciences*, **47**:1 (1993), pp. 45-76.

[26] P.L. Karlton, S.H. Fuller, R.E. Scroggs, E.B. Kaehler, Performance of Height-Balanced Trees, *Communications of the ACM*, **19**:1 (1976), pp. 23–28.

[27] D.E. Knuth, The Art of Computer Programming 3, Sorting and Searching, Addison-Wesley (1973).

[28] T. Lam, P. Tiwari, M. Tompa, Tradeoffs between Communication and Space, *Proceedings of 21st Annual ACM Symposium on Theory of Computing* (1989), pp. 217–226.

[29] T.J. Lehman, Design and Performance Evaluation of a Main Memory Relational Database System, *Technical Report 656*, Computer Sciences Department, University of Wisconsin-Madison (1986).

[30] L. Malmi, personal communication.

[31] A. Moitra, S.S. Iyengar, Maximally Parallel Algorithms to Balancing Binary Search Trees. *IEEE Transactions on Computers* **34**:6 (1985), pp. 563–565.

[32] S.S. Mukherjee, S.D. Sharma, M.D. Hill, J.R. Larus, J. Saltz, Efficient Support for Irregular Applications on Distributed-Memory Machines, to appear in *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1995).

[33] J. Nievergelt, E.M. Reingold, Binary Search Trees of Bounded Balance. *SIAM Journal of Computing* **2**:1 (1973), pp. 33–43.

[34] K. Oksanen, All RAM Is Accessible, But Some RAM Is More Accessible Than the Other. Unpublished manuscript, Department of Computer Science, Helsinki University of Technology (1995).

[35] W. Pugh, Slow Optimally Balanced Search Strategies vs. Cached Fast Uniformly Balanced Search Strategies, *Information Processing Letters*, **34** (1990), pp. 251-254.

[36] J.F. Sibeyn, T. Harris, Exploiting Locality in LT-RAM Computations. *Proceedings of Algorithm Theory – SWAT'94* (1994), pp. 338–349.

[37] D.D. Sleator, R.E. Tarjan, Self-adjusting Binary Search Trees. *Journal of the ACM* **32** (1985), pp. 652–686.

[38] A.J. Smith, Cache Memories. *ACM Computing Surveys*, **14**:3 (1982), pp. 473–530.

[39] Q.F. Stout, B.L. Warren, Tree Rebalancing in Optimal Time and Space. *Communications of the ACM* **29**:9 (1986), pp. 902–908.

[40] M. Talluri, M. Hill, Surpassing the TLB Performance of Superpages with Less Operating Systems Support, *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (1994), pp. 171–182.

[41] P.R. Wilson, M.S. Lam, T.G. Moher, Effective "Static-graph" Reorganization to Improve Locality in Garbage-Collected Systems, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (1991), pp. 177-191.

[42] C.E. Wu, Y. Hsu, Y-H. Liu, A Quantitative Evaluation of Cache Types for High Performance Computer Systems, *IEEE Transactions on Computers*, **42**:10 (1993), pp. 1154–1161.

[43] N. Ziviani, The Fringe Analysis on Search Trees, *Report CS-82-15*, Ph.D. Dissertation, Department of Computer Science, University of Waterloo (1982).

Index