Optimised Predecessor Data Structures for Internal Memory*

Naila Rahman[1], Richard Cole[2], Rajeev Raman[3]

[1] Department of Computer Science, King's College London, naila@dcs.kcl.ac.uk.
[2] Computer Science Department, Courant Institute, New York University, cole@cs.nyu.edu.
[3] Department of Maths and Computer Science, University of Leicester, r.raman@mcs.le.ac.uk.

Abstract. We demonstrate the importance of reducing misses in the *translation-lookaside buffer (TLB)* for obtaining good performance on modern computer architectures. We focus on data structures for the dynamic predecessor problem: to maintain a set S of keys from a totally ordered universe under insertions, deletions and predecessor queries. We give two general techniques for simultaneously reducing cache and TLB misses: simulating 3-level hierarchical memory algorithms and cache-oblivious algorithms. We give preliminary 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.

1 Introduction

Most algorithms are analysed on the random-access machine (RAM) model of computation, using some variety of the unit-cost criterion. This postulates that accessing a location in memory costs the same as a built-in arithmetic operation, such as adding two word-sized operands. Over the last 20 years or so CPU clock rates have grown explosively, but the speeds of main memory have not increased anywhere near as rapidly. Nowadays accessing a main memory location may be over a hundred times slower than performing an operation on data held in the CPU's registers. In addition, accessing (off-chip) main memory dissipates much more energy than accessing on-chip locations [7], which is especially undesirable for portable and mobile devices where battery life is an important consideration.

In order to reduce main memory accesses, modern computers have multiple levels of *cache* between CPU and memory. A cache is a fast associative memory, often located on the same chip as the CPU, which holds the values of some main memory locations. If the CPU requests the contents of a main memory location, and the value 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

* Research supported in part by EPSRC grant GR/L92150 (Rahman, Raman), NSF grant CCR-98-00085 (Cole) and UISTRF project 2001.04/IT (Raman).

the main memory (a cache *miss*). A cache hit has small penalty (1-3 cycles is typical) but a cache miss is very expensive. To amortise the cost of a main memory access, an entire *block* of consecutive main memory locations which contains the location accessed is brought into cache on a miss. Thus, a program that exhibits good *locality*, i.e. one that accesses memory locations close to recently accessed locations, will incur fewer cache misses and will consequently run faster and consume less power. Much recent work has focussed on analysing cache misses in algorithms [11, 15, 17, 16, 18, 13, 6]. Asymptotically, one can minimise cache misses by emulating optimal 2-level external-memory algorithms [18], for which there is a large literature [21].

There is another source of main memory accesses which is frequently overlooked, namely misses in the *translation-lookaside buffer (TLB)*. The TLB is used to support *virtual memory* in multi-processing operating systems. Virtual memory means that the memory addresses accessed by a process refer to its own unique logical address space, and it is considered "essential to current computer systems" [8]. To implement virtual memory, most operating systems partition main memory and the logical address space of each process into contiguous fixed-size *pages*, and store (recently-accessed) logical pages of each active process in main memory at a time [1]. This means that every time a process accesses a memory location, the reference to the corresponding logical page must be translated to a physical page reference. Unfortunately, the necessary information is held in the *page table*, a data structure in main memory! To avoid looking up the page table on each memory access, translations of some recently-accessed logical pages are held in the TLB, which is a fast associative memory. If a memory access results in a TLB hit, there is no delay, but a TLB miss can be much more expensive than a cache miss.

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 make many TLB misses. Cache and TLB sizes are usually small compared to the size of main memory. Typically, cache sizes vary from 256KB to 8MB, and TLBs hold between 64 and 128 entries. Hence *simultaneous* locality at the cache and page level is needed even for internal-memory computation.

In [17] a model incorporating both cache and TLB was introduced, and it was shown that radix sort algorithms that optimised both cache and TLB misses were significantly faster than algorithms that optimised cache misses alone. Indeed, the latter made many more TLB misses than cache misses. In [17] it was also shown that by emulating optimal 2-level external-memory algorithms one ensures that the numbers of TLB and cache misses are of the same order of magnitude, and the number cache misses is optimal. This minimises the *sum* of cache and TLB misses (asymptotically). However, this is not enough. In [17] it is argued that due to the different characteristics of a cache miss and a TLB miss—in particular, a TLB miss can be more expensive than a cache miss—these should be counted separately. Even if cache and TLB misses cost about the same, opti-

[1] In case a logical page accessed is not present in main memory at all, it is brought in from disk. The time for this is not counted in the CPU times reported.

mising cache and TLB misses individually will usually reduce memory accesses by nearly a half. For example, sorting n keys requires $\Theta((n/B)\log_?(n))$ cache misses, where B is the cache block size, and $\Theta((n/P)\log_?(n))$ TLB misses, where P is the page size. Ignoring the bases of the logarithms for the moment, the fact that $P \gg B$ (our machine's values of $P = 2048$ and $B = 16$ are representative) means that the latter should be much smaller than the former. We argue that the same holds for searching, the problem that this paper focusses on.

In this paper we suggest two techniques for simultaneously minimising cache and TLB misses. The first is to simulate algorithms for a 3-level hierarchical memory model (3-HMM). The second is the use of *cache-oblivious* algorithms [6]. Cache-oblivious algorithms achieve optimal performance in a two-level memory but without knowledge of parameters such as the cache block size and the number of cache blocks. As a consequence, they minimise transfers across each level of a multi-level hierarchical memory.

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 (DS) makes an optimal $O(\log_B n)$ cache misses and executes $O(\log n)$ instructions[2] for all operations, where $|S| = n$ [9, 5]. However, a B-tree also makes $\Omega(\log_B n)$ TLB misses, which is not optimal. In this paper we consider the practical performance of three implementations.

Firstly, we consider the B*-tree [5], 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 subtrees 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 DS which uses the standard idea for 3-HMM. At the top level the DS 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 DS makes $O(\log_B n)$ cache misses and $O(\log_P n)$ TLB misses for all operations (the update cost is amortised).

Finally, we consider a cache-oblivious tree. The tree of Bender et al. [4] makes the optimal $O(\log_B n)$ cache misses and $O(\log_P n)$ TLB misses. However it appears to be too complex to get good practical performance. Our implementation is based on a simpler cache-oblivious DS due to Bender et al [3]. This data stuc-

[2] Logs are to base 2 unless explicitly noted otherwise.

ture makes $O(\log_B n + \log\log n)$ cache misses and $O(\log_P n + \log\log n)$ TLB misses.

Our preliminary experimental figures suggest that the relative performance is generally as follows:

$$\text{B}^* < \text{Cache-oblivious} < \text{3-HMM} < \text{Paged B}^*.$$

It is noteworthy that a non-trivial DS such as the cache-oblivious tree can outperform a simple, proven *and* cache-optimised DS such as the B^* tree.

2 Models Used

The basic model we use was introduced in [17] and augmented earlier cache models [13, 11] with a TLB. The model views main memory as being partitioned into equal-sized and aligned *blocks* of B memory words each. The cache consists of S *sets* each consisting of $a \geq 1$ *lines*. a is called the *associativity* of the cache, and typical values are $a = 1, 2$ or 4. Each line can store one memory block, and the i-th memory block can be stored in any of the lines in set (i mod S). We let $C = aS$. 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 mod 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. 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 on a LRU basis. Our results on searching do not depend strongly on the associativity or the cache/TLB replacement policy.

We now give some assumptions about this model, which are justified in [17]:
(A1) TLB and cache misses happen independently, i.e., a memory access may result in a cache miss, a TLB miss, neither, or both;
(A2) B, C, P and T are all powers of two, and
(A3) i. $B \ll P$; ii. $T \ll C$; iii. $BC \leq PT$; iv. $T \leq P$, $B \leq C$.

The performance measures are the number of instructions, the number of TLB misses and the number of cache misses, all counted separately.

Our experiments are performed on a Sun UltraSparc-II, which has a word size of 4 bytes and a block size of 64 bytes, giving $B = 16$. Its L2 (main) cache, which has an associativity $a = 1$ (i.e. it is direct-mapped), holds $C = 8192$ blocks and its TLB holds $T = 64$ entries [20]. Finally, the page size is 8192 bytes, giving $P = 2048$ and also (coincidentally) giving $BC = PT$.

3 General Techniques

An Emulation Theorem. Let $\mathbf{C}(a, B, C, P, T)$ be the model described in Section 2, 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. 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 an 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 1. *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 $i = 1, \ldots, C/4$. The last two locations of $Main$ are buffers that are used for copying data to avoid conflict misses [18]. For convenience, we assume that the arrays Mem and $Tert$ are aligned on cache block and page boundaries respectively.

We now assume an 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 [19], 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 [18]. 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.This proves the theorem. □

Cache-oblivious data structures. As noted in [6, 14] 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).

4 Data Structures

We now describe the three DSs in more detail, starting with the cache oblivious search tree, then the paged B*-tree and then the 3-HMM search tree. Below, the term 'I/O' refers to both TLB and cache misses.

Cache-oblivious search tree. Prokop [14] 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 L, searches take $O(\log_L n)$ I/Os. If h is the height of T, Prokop divides T at $\lfloor h/2 \rfloor$, which separates T into the top subtree T_0 of height $\lfloor h/2 \rfloor$ and $k \leq 2^{\lfloor h/2 \rfloor}$ subtrees $T_1, \ldots, T_k$ 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 1. This DS is static and requires $O(n)$ time and $O(n/L)$ I/Os to construct. Bender et al. [3] note that Prokop's tree can be simply and efficiently dynamised by using *exponential* search trees [2]. Our algorithm is closely related to that of Bender et al [3], 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 $\Theta(\sqrt{n})$. [3] After this, we recurse on each set. In contrast to [3], 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 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 2). As shown in [3], the number of I/Os during searches, excluding searching in the external leaves, is at most $O(\log_L n + \log\log n)$. Searching in the external leaves requires $O((\log n)/L)$ I/Os, which is negligible. As shown in [3], the DS (excluding the external leaves) can

[3] 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.

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 DS is rebuilt when n squares.

Implementation details. We have implemented two versions of our cache oblivious DS. 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 DS 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 DS 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 again 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 a external leaf with k keys is stored in a block of memory sufficient for $k + O(1)$ keys. Increasing this to $O(k)$ would improve insertion times, at the cost of increased memory. A final note is that the emulation of Section 3 is not required in this case to achieve optimal cache and TLB performance.

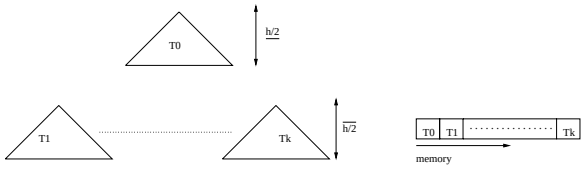

Fig. 1. The recursive memory layout of a complete binary search tree according to Prokop's scheme

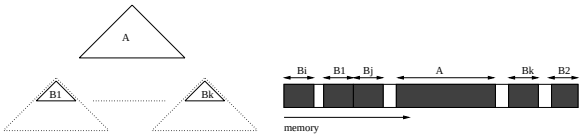

Fig. 2. Memory layout of the cache oblivious search tree with an external root node A of degree $\sqrt{n}$. The subtrees of size $\Theta(\sqrt{n})$ rooted at A are shown in outline, and their roots are $B_1, \ldots, B_k$. The memory locations used by the subtrees under $B_1, \ldots, B_k$ are not shown.

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 [5]. By packing more keys in a node, B*-trees are shallower than B-trees. 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, TLB misses dominate.

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 oustide the page are recursively organised in a similar manner (see Fig 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 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 [1, 22], but such trees seem to have poorer constant factors than ordinary B-trees.

Optimal 3-HMM Trees. We now describe the implementation of a DS that is optimised for the 3-HMM. 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

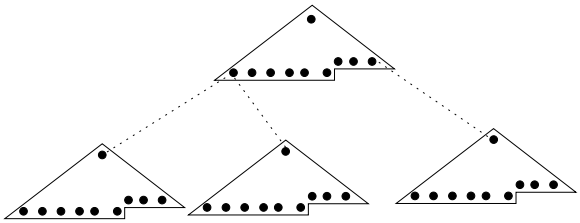

Fig. 3. A paged B-tree. The nodes in the top part of the tree are in a memory page. The top parts of sub-trees that remain outside the page with the root are recursively placed on a memory page.

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. At the moment we have not fully investigated all possible approaches to this problem.

5 Experimental Results

We have implemented B*-trees, paged B*-trees, optimal 3-HMM trees and our cache oblivious search trees with and without internal pointers. Our DSs were coded in C++ and all code was compiled using `gcc 2.8.1` with optimisation level 6. Our experiments were performed on a Sun UltraSparc-II with 2×300 Mhz processors and 1GB main memory. This machine has 64 TLBs, 8KB pages, a 16KB L1 cache, and a 512KB L2 cache; both caches are direct-mapped. The L1 cache miss penalty is about 3 cycles on this machine, and the L2 cache and TLB miss penalties are both about 30 cycles.

In each experiment, a DS 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-HMM 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 DS 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. These are preliminary results and at each data point we report the average search time per query for 10 experiments.

Each DS 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"). Figure 4 shows results for the random queries and Figure 5 shows results for the repeated random queries.

In Fig 4 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 43% and almost 70% more time than on paged B*-trees with a branching factor of 7. Optimal 3-HMM trees perform quite well, being at-most 14% slower than paged B*-trees. Of the DSs suited for both the cache and TLB, the cache oblivious search trees are the slowest. On the other hand, the 3-HMM trees are carefully tuned to our machine, whereas the cache-oblivious trees have no machine-specific parameters.

As can be seen from Figure 5, the better-performing DSs 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 DS 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, one of the authors reported a time of 9.01 μs/search for $n = 2^{22}$ using LEDA 3.7 Red-Black trees [12, 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.

6 Conclusions and future work

Our preliminary experimental results show that optimising for three-level memories gives large performance gains in internal memory computations as well. In particular we have shown that cache-oblivious data structures may have signifi-

Time per-search (μs) on UltraSparc-II 2 × 300 Mhz							
n	B* (BF=7)	B* (BF=8)	Paged B* (BF=7)	Paged B* (BF=8)	Optimal 3-HMM	Cache Obl. (int. ptrs)	Cache Obl. (no int. ptrs)
2^{18} abs	2.384	2.216	1.541	1.572	1.655	1.875	2.109
2^{18} rel	**1.547**	**1.438**	**1.000**	**1.020**	**1.074**	**1.217**	**1.369**
2^{20} abs	3.329	3.054	1.979	2.398	2.210	2.563	2.770
2^{20} rel	**1.682**	**1.543**	**1.000**	**1.212**	**1.117**	**1.295**	**1.400**
2^{22} abs	4.113	3.797	2.487	2.913	2.843	3.299	3.369
2^{22} rel	**1.654**	**1.527**	**1.000**	**1.171**	**1.143**	**1.326**	**1.355**
2^{23} abs	4.529	4.156	2.691	3.127	3.054	3.635	3.658
2^{23} rel	**1.683**	**1.544**	**1.000**	**1.162**	**1.135**	**1.351**	**1.359**

Fig. 4. Query times for $2n$ independent random queries on n keys in DS.

Computation time per-search (μs) on UltraSparc-II 2 × 300 Mhz machine							
n	B* (BF=7)	B* (BF=8)	Paged B* (BF=7)	Paged B* (BF=8)	Optimal 3-HMM	Cache Obl. (int. ptrs)	Cache Obl. (no int. ptrs)
2^{18} abs	0.793	0.751	0.736	0.744	0.904	0.885	1.262
2^{18} rel	**1.077**	**1.020**	**1.000**	**1.011**	**1.228**	**1.202**	**1.715**
2^{20} abs	0.891	0.868	0.867	0.862	1.024	0.941	1.315
2^{20} rel	**1.034**	**1.007**	**1.006**	**1.000**	**1.188**	**1.092**	**1.526**
2^{22} abs	0.976	0.955	0.962	0.947	1.129	1.170	1.399
2^{22} rel	**1.031**	**1.008**	**1.016**	**1.000**	**1.192**	**1.235**	**1.477**
2^{23} abs	0.982	0.979	0.975	0.966	1.151	1.244	1.401
2^{23} rel	**1.017**	**1.013**	**1.009**	**1.000**	**1.192**	**1.288**	**1.450**

Fig. 5. Query times for $n/512$ independent random queries repeated 1024 times each, on n keys in DS.

cant practical importance. Although the trends in our preliminary experiments are clear, these need to be rigourously established with a larger suite of experiments, including cache and TLB simulations. We would also like to test performance of these structures on secondary memory.

References

1. L. Arge, J. S. Vitter. Optimal Dynamic Interval Management in External Memory (extended abstract). *FOCS 1996*, pp. 560-569.
2. A. Andersson. Faster Deterministic Sorting and Searching in Linear Space. In *Proc. 37th IEEE FOCS*, pp. 135–141, 1996.
3. Bender, M., Cole, R. and Raman. R. Exponential trees for cache-oblivious algorithms. In preparation, 2001.
4. Bender, M., Demaine, E. and Farach-Colton, M. Cache-oblivous B-trees. In *Proc. 41st IEEE FOCS*, pp. 399–409, 2000.
5. Comer, D. The Ubiquitous B-Tree. *ACM Comput. Surv.* **11** (1979), p.121.
6. Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. Cache-oblivious algorithms. In *Proc. 40th IEEE FOCS*, pp. 285–298, 1999.
7. Furber, S. B. *Arm System-On-Chip Architecture* Addison-Wesley Professional, 2nd ed., 2000.
8. Hennessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach* (Second ed.). Morgan Kaufmann, 1996.
9. D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching, 3rd ed.* Addison-Wesley, 1997.
10. Ladner, R. E., Fix, J. D., and LaMarca, A. Cache performance analysis of traversals and random accesses. In *Proc. 10th ACM-SIAM SODA* (1999), pp. 613–622.
11. LaMarca, A. and Ladner, R. E. The influence of caches on the performance of sorting. *J. Algorithms* **31**, 66–104, 1999.
12. Korda, M. and Raman, R. An experimental evaluation of hybrid data structures for searching. In *Proc. 3rd WAE*, LNCS 1668, pp. 213–227, 1999.
13. Mehlhorn, K. and Sanders, P. Accessing multiple sequences through set-associative cache, 2000. Prel. vers. *Proc. 26th ICALP*, LNCS 1555, 1999.
14. H. Prokop. Cache-oblivious algorithms. MS Thesis, MIT, 1999.
15. Rahman, N. and Raman, R. Analysing cache effects in distribution sorting. *ACM J. Exper. Algorithmics*, WAE '99 special issue, to appear. Prel. vers. in *Proc. 3rd WAE*, LNCS 1668, pp. 184–198, 1999.
16. Rahman, N. and Raman, R. Analysing the cache behaviour of non-uniform distribution sorting algorithms. In *Proc. 8th ESA*, LNCS 1879, pp. 380–391, 2000.
17. Rahman, N. and Raman, R. Adapting radix sort to the memory hierarchy. TR 00-02, King's College London, 2000, `http://www.dcs.kcl.ac.uk/technical-reports/2000.html` Prel. vers. in *Proc. ALENEX 2000*.
18. Sen, S. and Chatterjee, S. Towards a theory of cache-efficient algorithms (extended abstract). In *Proc. 11th ACM-SIAM SODA* (2000), pp. 829–838.
19. Sleator, D. D. and Tarjan, R. E. Amortized efficiency of list update and paging rules. *Communications of the ACM* **28**, 202–208, 1995.
20. Sun Microsystem. UltraSPARC User's Manual. Sun Microsystems, 1997.
21. Vitter, J. S. External memory algorithms and data structures: Dealing with MASSIVE data. To appear in *ACM Computing Surveys*, 2000.
22. D. E. Willard. Reduced memory space for multi-dimensional search trees. In *Proc. STACS '85*, pages 363–374, 1985.