The Cost of Address Translation

Extended Abstract

Tomasz Jurkiewicz* Kurt Mehlhorn†

Abstract

Modern computers are not random access machines (RAMs). They have a memory hierarchy, multiple cores, and virtual memory. In this paper, we address the computational cost of address translation in virtual memory. Starting point for our work is the observation that the analysis of some simple algorithms (random scan of an array, binary search, heapsort) in either the RAM model or the EM model (external memory model) does not correctly predict growth rates of actual running times. We propose the VAT model (virtual address translation) to account for the cost of address translations and analyze the algorithms mentioned above and others in the model. The predictions agree with the measurements. We also analyze the VAT-cost of cache-oblivious algorithms.

1 Introduction

The role of models of computation in algorithmics is to provide abstractions of real machines for algorithm analysis. Models should be mathematically pleasing and have predictive value. Both aspects are essential. If the analysis has no predictive value, it is merely a mathematical exercise. If a model is not clean and simple, researchers will not use it. The standard models for algorithm analysis are the RAM (random access machine) model [SS63] and the EM (external memory) model [AV88].

The RAM model is by far the most popular model. It is an abstraction of the von Neumann architecture. A computer consists of a control and processing unit and an unbounded memory. Each memory cell can hold a word, and memory access and logical and arithmetic operations on words take constant time. The word length is either an explicit parameter or assumed to be logarithmic in the size of the input. The model is very simple and has predictive value.

The external memory model was introduced because the RAM model does not account for the memory hierarchy and hence the RAM model has no predictive value for computations involving disks. Modern machines have an extensive memory hierarchy involving several levels of cache memory, main memory, and disks, see Section 2.2 for more details.

This research started with a simple experiment. We timed six simple programs for different input sizes, namely permuting the elements of an array of size n, random scan of an array of size n, n random binary searches in an array of size n, heapsort of n elements, introsort[1] of n elements, and sequential scan of an array of size n. For some of the programs, e.g., sequential scan through an array and quicksort, the measured running times agree very well with the predictions of the models. *However, the running time of random scan seems to grow as $O(n \log n)$ and the running time of the binary searches seems to grow as $O\left(n \log^2 n\right)$, a blatant violation of what the models predict.* We give the details of the experiments in Section 2.

Why do measured and predicted running times differ? Modern computers have virtual memories. Each process has its own virtual address space $\{0, 1, 2, \ldots\}$. Whenever, a process accesses memory, the virtual address has to be translated into a physical address. *The translation of virtual addresses into physical addresses incurs cost.* The translation process is usually implemented as a hardware-supported walk in a prefix tree, see Section 3 for details. The tree is stored in the memory hierarchy and hence the translation process may incur cache faults. The number of cache faults depends on the locality of memory accesses: the less local, the more cache faults.

We propose an extension of the EM model, the VAT(virtual address translation)-model, that accounts for the cost of address translation, see Section 4. We show that we may assume that the translation process makes optimal use of the cache memory by relating the cost of optimal use with the cost under the LRU strategy, see Section 4. We analyze a number of programs, including the six mentioned above, in the VAT model and obtain good agreement with the measured running times, see Section 5. We relate the cost of a cache-oblivious algorithm in the EM model to the cost in the

*Max Planck Institute for Informatics, Saarbrücken, Germany; The Saarbrücken Graduate School of Computer Science.

†Max Planck Institute for Informatics, Saarbrücken, Germany.

[1] Introsort is the version of quicksort used in modern versions of the STL. For the purpose of this paper, introsort is a synonym for quicksort.

VAT model, see Section 6. In particular, algorithms that do not need a tall-cache assumption incur no or little overhead. We close with some suggestions for further research and consequences for teaching, see Section 8.

Related Work: It is well known in the architecture and systems community that virtual memory and address translation comes at a cost. Many textbooks on computer organization, e.g. [HP07], discuss virtual memories. The papers by Drepper [Dre07, Dre08] describe computer memories, including virtual translation, in great detail. [Adv10] provides further implementation details.

The cost of address translation received little attention from the algorithms community. The survey paper by N. Rahman [Rah03] on algorithms for hardware caches and TLB summarizes the work on the subject. She discusses a number of theoretical models for memory. All models discussed in [Rah03] treat address translation atomically, i.e., the translation from virtual to physical addresses is a single operation. However, this is no longer true. In 64-bit systems the translation process is a tree walk. Our paper is the first that proposes a theoretical model for address translation and analyses algorithms in this model.

2 Some Puzzling Experiments

2.1 Seven Simple Programs We used the following seven programs in our experiments. Let A be an array of size n

- permute: for $j \in [n-1..0]$ do: $i \coloneqq \text{random}(0..j)$; $\text{swap}(A[i], A[j])$;
- random scan: $\pi \coloneqq$ random permutation; for i from 0 to $n-1$ do: $S \coloneqq S + A[\pi(i)]$;
- n binary searches for random positions in A; A is sorted for this experiment
- heapify
- heapsort
- quicksort
- sequential scan

On a RAM, the first two, the last, and heapify are linear time $O(n)$, and the others are $O(n \log n)$. Figure 1 shows the measured running times[2] for these programs divided by their RAM complexity; we refer to this quantity as *normalized operation time*. If RAM complexity is a good predictor, the normalized operation times should be approximately constant. We observe that two of the linear time programs show linear behavior, namely sequential access and heapify, that one of the $\Theta(n \log n)$ programs shows $\Theta(n \log n)$ behavior, namely quicksort, and that for the other programs (heapsort, repeated binary search, permute, random access), the actual running time grows faster than what the RAM model predicts.

How much faster and why?

Figure 1 also answers the "how much faster" part of the question. Normalized operation time seems to be a piecewise linear in the logarithm of the problem size; observe that we are using a logarithmic scale for the abscissa in this figure. For heapsort and repeated binary search, normalized operation time is almost perfectly piecewise linear, for permute and random scan, the piecewise linear has to be taken with a grain of salt.[4] The pieces correspond to the memory hierarchy. *The measurements suggest that the running times of permute and random scan grow like $\Theta(n \log n)$ and the running times of heapsort and repeated binary search grow like $\Theta\big(n \log^2 n\big)$.*

2.2 Memory Hierarchy Does Not Explain It We argue in this section that the memory hierarchy does not explain the experimental findings by determining the cost of the random scan of an array of size n in the EM model and relating it to the measured running time.

Let s_i, $i \geqslant 0$, be the size of the i-th level C_i of the memory hierarchy; $s_{-1} = 0$. We assume $C_i \subset C_{i+1}$ for all i. Let ℓ be such that $s_\ell < n \leqslant s_{\ell+1}$, i.e., the array fits into level $\ell + 1$ but does not fit into level ℓ. For $i \leqslant \ell$, a random address is in C_i but not in C_{i-1} with probability $(s_i - s_{i-1})/n$. Let c_i be the cost of accessing an address that is in C_i but not in C_{i-1}. The expected

[2] All programs were compiled by gcc in version "Debian 4.4.5-8" and run on Debian Linux in version 6.0.3 on a machine with processor Intel Xeon X5690 (3,46 GHz, 12MiB[3] Smart Cache, 6,4 GT/s QPI). The caption of Figure 2 lists further machine parameters. In each case we performed multiple repetitions and took the minimum measurement for each considered size of the input data. We chose the minimum as we are estimating the cost that must be incurred. We also experimented with average or median and the results did not change. We grew input sizes by factors of 1.4 to exclude influence of memory associativity and made sure that the largest problem size still fitted in main memory. We also performed the experiments on other machines and operating systems and obtained consistent results.

[3] KiB and MiB are modern, non ambiguous notations for 2^{10*2} and 2^{10*3} bytes, respevtively. For more details refer to `http://en.wikipedia.org/wiki/Binary_prefix`.

[4] We are still working on a satisfactory explanation for the bumpy shape of the graphs for permute and random access.

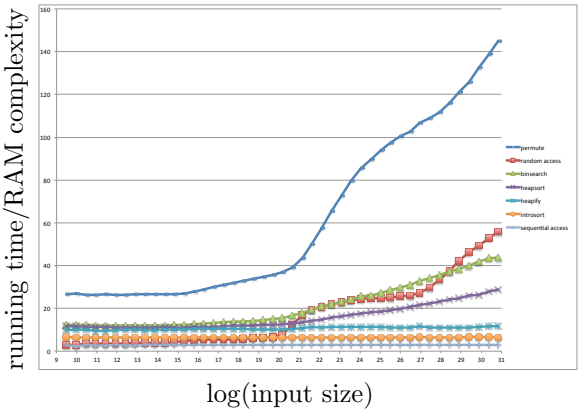

Figure 1: The abscissa shows the logarithm of the input size. The ordinate shows the measured running time divided by the RAM-complexity (normalized operation time). The normalized operation times of sequential access, quicksort, and heapify are constant, the normalized operation times of the other programs are not.

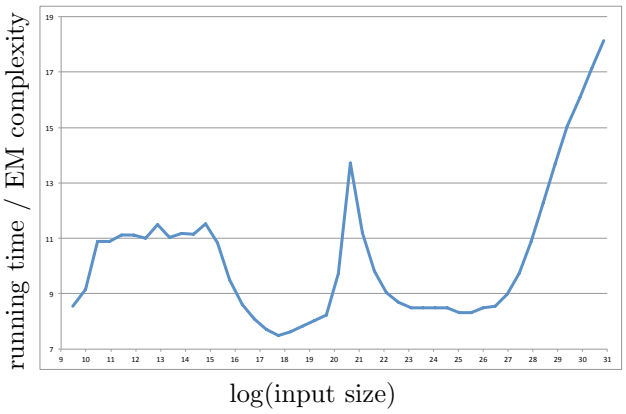

Figure 2: The running time of random scan divided by the EM-complexity. We used the following parameters for the memory hierarchy: the sizes are taken from the machine specification, and the access times were determined experimentally.

Memory Level	Size [3]	log(maximum number of elements)	Access Time in Picoseconds
L1	32kiB	12	4080
L2	256kiB	15	4575
L3	12MiB	20,58	9937
RAM			38746

total cost in the external memory model is equal to

$$T_{\mathrm{EM}}(n) := n \cdot \left(\frac{n - s_\ell}{n} c_{\ell+1} + \sum_{0 \leqslant i \leqslant \ell} \frac{s_i - s_{i-1}}{n} c_i \right) =$$
$$= n c_{\ell+1} - \sum_{0 \leqslant i \leqslant \ell} s_i (c_{i+1} - c_i).$$

This is a piecewise linear function whose slope is $c_{\ell+1}$ for $s_\ell < n \leqslant s_{\ell+1}$. The slopes are increasing, but change only when a new level of the memory hierarchy is used. Figure 2 shows the measured running time of random scan divided by EM-complexity as a function of the logarithm of the problem size. Clearly, the figure does not show the graph of a constant function.[5]

3 Virtual Memory

Virtual addressing was motivated by multi-processing. When several processes are executed concurrently on the same machine, it is convenient and more secure to give each program a linear address space indexed by the nonnegative integers. However, theses addresses are now virtual and no longer directly correspond to physical (real) addresses. Rather, it is the task of the operating system to map the virtual addresses of all processes to a single physical memory. The mapping process is hardware supported.

Memory is viewed as a collection of pages of $P = 2^p$ cells. Both virtual and real addresses consist of an *index* and an *offset*. The index selects a page and the offset selects a cell in a page. The index is broken into d segments of length $k = \log K$. For example, for processors of the x68-64 family (see http://en.wikipedia.org/wiki/X86-64) with 64 bit addresses the numbers are: $d = 4$, $k = 9$, and $p = 12$; the remaining 16 bit are used for other purposes.

Logically, the translation process is a walk in a tree with outdegree K; this tree is usually called the page table [Dre08, HP07]. The walk starts at the root; the first segment of the index determines the child of the root, the second segment of the index determines the child of the child, and so on. The leaves of the tree store indices of physical pages. The offset then determines the cell in the physical address, i.e., offsets are not translated but taken verbatim.

The page table is stored in the RAM and nodes accessed during the page table walk have to be brought to fastest memory. A small number of recent translations is stored in the translation-lookaside-buffer (TLB). The TLB is a small associative memory that contains pairs consisting of virtual and corresponding physical index. This is akin to the first level cache for data.

[5] A function of the form $(x \log(x/a))/(bx - c)$ with $a, b, c > 0$ is convex. The plot may be interpreted as the plot of a piecewise convex function.

4 The Virtual Address Translation Model (VAT model)

The full version of this section can be found in the appendix. Our model abstracts from the above. The translation is performed by a walk in a tree of outdegree K and depth d as described above. The translation process uses a translation cache TC that can store W nodes of the translation tree.[6] The TC is changed by insertions and evictions. Let a be a virtual address and let $v_d, v_{d-1}, \ldots, v_0$ be its translation path; v_d is the root, v_{d-1} is the child of the root selected by the first segment of a, and so on. Translating a requires to *access* all nodes of the translation path in order. Only nodes in the TC can be accessed. The translation ends when v_0 is accessed. The next translation starts with the next operation on the TC.

The *length of the translation* is the number of insertions performed during the translation and the *cost of the translation* is τ times the length. The length is at least the number of nodes of the translation path that are not present in the TC at the beginning of the translation.

4.1 TC Replacement Strategies Since the TC is a special case of a cache in a classic EM machine, the following classic result applies.

LEMMA 4.1. ([ST85, FLPR12]) *An optimal replacement strategy is at most by factor 2 better than LRU[7] on a cache of double size, assuming both caches start empty.*

For TC caches, it is natural to assume the initial segment property.

DEFINITION 4.1. *An **initial segment** of a rooted tree is an empty tree or a connected subgraph of the tree containing the root. TC has the **initial segment property (ISP)**, if the TC contains an initial segment of the translation tree. A TC replacement strategy has ISP, if, under this strategy, TC has ISP at all times.*

ISP is important because, as we show later, ISP can be realized at no additional cost for LRU and at little additional cost for the optimal replacement strategy. Therefore, strategies with ISP can significantly simplify proofs for upper and lower bounds. Moreover, ISP are easier to implement. Any implementation of a caching system requires some way to search the cache. This requires an indexing mechanism. RAM memory is indexed by the memory translation tree. In case of the TC itself, ISP allows to integrate the indexing structure into the cached content. One only has to store the root of the tree at a fixed position.

LEMMA 4.2. *When the LRU policy is in use, the number of TC misses in a translation is equal to the layer number of the highest missing node on the translation path.*

Proof. The content of the LRU cache is easy to describe. Concatenate all translation paths and delete all occurrences of each node except the last. The last W nodes of the resulting sequence form the TC. Observe that an occurrence of a node is only deleted if the node is part of a latter translation path. This implies that the TC contains at most two incomplete translation path, namely the least recent path that still has nodes in the TC and the current path. The former path is evicted top-down and the latter path is inserted top-down. The claim now easily follows. Let v be the highest missing node on the current translation path. If no descendant of v is contained in the TC, the claim is obvious. Otherwise, the topmost descendant present in the TC is the first node on the part of the least recent paths that is still in the TC. Thus as the current translation path is loaded into the TC, the least recent path is evicted top-down. As the consequence, the gap is never reduced.

The proof also shows that whenever LRU detaches nodes from the initial segment, the detached nodes will never be used again. This suggests a simple (implementable) way of introducing ISP to LRU. If LRU evicts a node that still has descendants in the TC, it also evicts the descendants. The descendants actually form a single path. Next, we use Lemma A.2 (see appendix) to make this algorithm lazy again. It is easy to see that the resulting algorithm is the ISLRU as defined next.

DEFINITION 4.2. ***ISLRU** (Initial Segment preserving LRU) is the replacement strategy that always evicts the lowest descendant of the least recently used node.*

PROPOSITION 4.1. *ISLRU for TCs with $W > d$ is at least as good as LRU.*

DEFINITION 4.3. ***ISMIN** (Initial Segment property preserving MIN) is the replacement strategy for TCs with ISP that always evicts the node that is not used for the longest time into the future among the nodes that are not on the current translation path and have no descendants. Nodes that will never be used again are evicted before the others in arbitrary descendant–first order.*

THEOREM 4.1. ISMIN *is an optimal replacement strategy among those with ISP.*

[6] In real machines, there is no separate translation cache. Rather, the same cache is used for data and the translation tree.

[7] LRU is a strategy that always evicts the Least Recently Used node.

Proof. Let R be any replacement strategy with ISP, and let t be the first point in time when it departs from ISMIN. We will construct R' with ISP that does not depart from ISMIN including time t and has no more TC misses than R. Let v be the node evicted by ISMIN at time t.

We first assume that R evicts v at some later time t' without accessing it in the interval $(t, t']$. Then R' simply evicts v at time t and shifts the other evictions in the interval $[t, t')$ to one later replacement. Postponing evictions to the next replacement does not cause additional insertions and does not break connectivity. It may destroy laziness by moving an eviction of a node right before its insertion. In this case R' skips both. Since no descendant of v is in the TC at time t, and v will will not be used for the longest time into the future, none of its children will be added by R before time t'; therefore the change does not break the connectivity.

We come to the case that R stores v till it is accessed for the next time, say at time t'. Let a be the node evicted by R at time t. R' evicts v instead of a and remembers a as being **special**. We guarantee that the content of the TCs in the strategies R and R' differs only by v and the current special node till time t', and is identical afterwords. To reach this goal R' replicates the behavior of R except for three situations.

1. If R evicts the parent of the special node, R' evicts the special node to preserve ISP, and and from now on remembers the parent as being special. As long as only Rule 1 is applied, the special node is an ancestor of a.

2. If R replaces some node b with the current special node, R' skips the replacement and from now on remembers b as the special node. Since a will be accessed before v, Rule 2 is guaranteed to be applied and hence R' is guaranteed to save at least one replacement.

3. At time t', R' replaces the special node with v, performing one extra replacement.

We have shown how to turn an arbitrary replacement strategy with ISP into ISMIN without efficiency loss. This proves the optimality of ISMIN.

We can now state an ISP-aware extension of Lemma 4.1.

THEOREM 4.2.

$$\mathrm{MIN}(W) \leqslant \mathrm{ISMIN}(W) \leqslant \mathrm{ISLRU}(W) \leqslant \\ \leqslant \mathrm{LRU}(W) \leqslant 2\mathrm{MIN}(W/2),$$

where MIN *is an optimal replacement strategy and* $A(s)$ *denotes a number of insertions performed by replacement strategy* A *to an initially empty TC of size* $s > d$ *for an arbitrary, but fixed sequence of translations.*

Theorem 4.2 implies $\mathrm{LRU}(W) \leqslant 2\mathrm{ISLRU}(W/2)$ and $\mathrm{ISMIN}(W) \leqslant 2\mathrm{MIN}(W/2)$. These inequalities can be sharpened considerably.

THEOREM 4.3. $\mathrm{LRU}(W + d) \leqslant \mathrm{ISLRU}(W)$ *and* $\mathrm{ISMIN}(W + d) \leqslant \mathrm{MIN}(W)$.

5 Analysis of Algorithms

In this section, we analyze the translation cost of some algorithms as a function of the problem size n and memory requirement m. For all the algorithms analyzed, $m = \Theta(n)$. We assume:

1. $\tau d \leqslant P$; the cost of moving a single translation path to the TC is no more than the size of a page, i.e., if at least one instruction is performed for each cell in a page, the cost of translating the index of the page can be amortized.

2. $K \geqslant 2$, i.e., the fanout of the translation tree is at least two.

3. $m/P \leqslant K^d \leqslant 2m/P$, i.e., the translation tree suffices to translate all addresses but is not much larger. As a consequence $\log(m/P) \leqslant d \log K = dk \leqslant 1 + \log(m/P)$ and hence $\log_K(m/P) \leqslant d \leqslant 1/k(1 + \log(m/P))$.

4. $d \leqslant W$, i.e., the translation cache can hold at least one translation path.

Sequential Access: We scan an array of size n, i.e., we need to translate addresses $b, b+1, \ldots, b+n-1$ in this order, where b is the base address of the array. The translation path stays constant for P consecutive accesses and hence at most $2n/P$ indices must be translated for a total cost of at most $\tau d(2 + n/P)$. By assumption (1) this is at most $\tau d(n/P + 2) \leqslant n + 2P$.

The analysis can be sharpened significantly. We keep the current translation path in cache and hence the first translation incurs at most d faults. The translation path changes after every P-th access and hence changes at most a total of $\lceil n/P \rceil$ times. Of course, whenever the path changes, the last node changes. The next to last node changes after every K-th access and hence changes at most $\lceil n/(PK) \rceil$ times. In total, we incur

$$d + \sum_{0 \leqslant i \leqslant d} \left\lceil \frac{n}{PK^i} \right\rceil < 2d + \frac{K}{K-1} \frac{n}{P}$$

TC faults. The cost is therefore bounded by $2P + 2n/d$, which is asymptotically smaller than RAM complexity.

Random Access: In the worst case, no node of any translation path is in cache. Thus the total translation cost is bounded by τdn. This is at most $\frac{\tau}{k}n(1+\log(n/P))$.

We will next argue a lower bound. We may assume that the TC satisfies the initial segment property. The translation path ends in a random leaf of the translation tree. For every leaf some initial segment of the path ending in this leaf is cached. Let u be an uncached node of the translation tree of minimal depth and let v be a cached node of maximal depth. If the depth of v is larger by two or more than the depth of u, then it is better to cache u instead of v (because more leaves use u instead of v). Thus up to one the same number of nodes is cached on every translation path and hence the expected length of the path cached is at most $\log_K W$ and hence the expected number of faults during a translation is $d-\log_K W$. The total expected cost is therefore at least $\tau n(d-\log_K W) \geqslant \tau n \log_K n/(PW) = \frac{\tau}{k}n\log(n/(PW))$, which is asymptotically larger than RAM complexity.

LEMMA 5.1. *The translation cost of a random scan of an array of size n is at least $\frac{\tau}{k}n\log(n/(PW))$ and at most $\frac{\tau}{k}n(1+\log(n/P))$.*

Binary Search: We do n binary searches in an array of length n. Each search searches for a random element of the array. For simplicity, we assume that n is a power of two minus one. Binary search in an array is equivalent to search in a balanced tree where the root is stored in location $n/2$, the children of the root are stored in locations $n/4$ and $3n/4$, and so on. We cache the translation paths of the top ℓ layers of the search tree and the translation path of the current node of the search. The top ℓ layers contain $2^{\ell+1}-1$ vertices and hence we need to store at most $d2^{\ell+1}$ nodes[8] of the translation tree. This is feasible if $d2^{\ell+1} \leqslant W$. For the sequel, let $\ell = \log(W/2d)$.

Any of the remaining $\log n - \ell$ steps of the binary search cause at most d cache faults. Therefore the total cost per search is bounded by

$$\tau d(\log n - \ell) \leqslant \frac{\tau}{k}(1+\log(n/P))(\log n - \ell) = \\ = \frac{\tau}{k}\log\frac{2n}{P}\log\frac{2nd}{W}.$$

This analysis may seem coarse. After all once the search leaves the top ℓ layers of the search tree, addresses of subsequent nodes differ only by $n/2^\ell$, $n/2^{\ell+1}$, ..., 1. However, we will next argue that the bound above is essentially sharp for our caching strategy. Recall that if two virtual addresses differ by D, their translation path differ in the last $\lceil \log_K(D/P)\rceil$ nodes. Thus the scheme above incurs at least

$$\sum_{\ell\leqslant i\leqslant \log n-p}\left\lceil\frac{1}{k}\log\frac{n}{2^iP}\right\rceil \geqslant \sum_{0\leqslant j\leqslant \log n-\ell-p}\frac{1}{k}\log 2^j \geqslant \\ \geqslant \frac{1}{2k}(\log n-\ell-p)^2 = \frac{1}{2k}\left(\log\frac{2nd}{PW}\right)^2.$$

TC faults. We next show that it essentially holds true for any caching strategy.

By Theorem 4.2, we may assume that ISLRU is used as the cache replacement strategy, i.e., TC contains top nodes on recent translation paths. Let $\ell = \lceil\log(2W)\rceil$. There are $2^\ell \geqslant 2W$ vertices of depth ℓ in a binary search tree. Their addresses differ be least $n/2^\ell$ and hence for any two such addresses their translation paths differ in at least the last $z = \lceil\log_K(n/(2^\ell P)\rceil$ nodes. Call a node at depth ℓ expensive if none of the last z nodes of its translation path are contained in the TC and non-expensive otherwise. There can be at most W inexpensive vertices and hence with probality at least $1/2$ a random binary search goes through an expensive node, call it v, at depth ℓ. Since ISLRU is the cache replacement strategy, the last z nodes of the translation path are missing for all descendants of v. Thus, by the argument in the preceding paragraph, the expected number of cache misses per search is at least

$$\frac{1}{2}\sum_{\ell\leqslant i\leqslant \log n-p}\left\lceil\frac{1}{k}\log\frac{n}{2^iP}\right\rceil \geqslant \sum_{0\leqslant j\leqslant \log n-\ell-p}\frac{1}{2k}\log 2^j \geqslant \\ \geqslant \frac{1}{4k}(\log n-\ell-p)^2 = \frac{1}{4k}\left(\log\frac{n}{4PW}\right)^2.$$

LEMMA 5.2. *The translation cost of n random binary searches in an array of size n is at most $\frac{\tau}{2k}n\left(\log\frac{2nd}{PW}\right)^2$ and at least $\frac{\tau}{4k}n\left(\log\frac{n}{4PW}\right)^2$.*

We know from cache-oblivious algorithms that the van-Emde Boas layout of a search tree improves locality. We will show in Section 6 that this improves the translation cost.

Heapify and Heapsort: We prove a bound on the translation cost of heapify. The following proposition generalizes the analysis of sequential scan.

DEFINITION 5.1. ***Extremal translation paths*** *of n consecutive addresses are the paths to the first and the last address in the range.* ***Non-extremal nodes*** *are the nodes on translation paths to addresses in the range that are not on the extremal paths.*

PROPOSITION 5.1. *A sequence of memory accesses that gains access to each page in a range, causes at least one*

[8] We use vertex for the nodes of the search tree and node for the nodes of the translation tree.

TC miss for each non-extremal node of the range. If the sequence of pages in the range n is accessed in the decreasing order this bound is matched by storing the extremal paths and dedicating $\log_K(n/P)$ cells in the TC for the required translations.

PROPOSITION 5.2. *Let n, ℓ and x be nonnegative integers. Number of non-extremal nodes in the union of the translation paths of any x out of n consecutive addresses is at most*

$$x\ell + \frac{2n}{PK^\ell}.$$

Moreover, there is a set of $x = \lceil n/(PK^\ell) \rceil$ addresses such that the union of the paths has size at least $x(\ell + 1) + d - \ell$.

Proof. The union of the translation paths to all n addresses contains at most n/P non-extremal nodes on the leaf level (= level 0) of the translation tree. On level i, $i \geqslant 0$, from the bottom, it contains at most $n/(PK^i)$ non-extremal nodes.

We overestimate the size of the union of x translation paths by counting one node each on levels 0 to $\ell - 1$ for every translation path and all non-extremal nodes contained in all the n translation paths on the levels above. Thus the size of the union is bounded by

$$x\ell + \sum_{\ell \leqslant i \leqslant d} n/(PK^i) < x\ell + \frac{K}{K-1}\frac{n}{PK^\ell} \leqslant x\ell + \frac{2n}{PK^\ell}.$$

A node on level ℓ lies on the translation path of $K^\ell P$ consecutive addresses. Consider addresses $z + iPK^\ell$ for $i = 0, 1, \ldots, \lceil n/PK^\ell \rceil - 1$, where z is the smallest in our set of n addresses. The translation paths to these addresses are disjoint from level ℓ down to level zero and use at least one node on levels $\ell + 1$ to d. Thus the size of the union is at least $x(\ell + 1) + d - \ell$.

An array $A[1..n]$ storing elements from an ordered set is heap-ordered if $A[i] \leqslant A[2i]$ and $A[i] \leqslant A[2i+1]$ for all i with $1 \leqslant i \leqslant \lfloor n/2 \rfloor$. An array can be turned into a heap by calling operation *sift*(i) for $i = \lfloor n/2 \rfloor$ down to 1. *sift*(i) repeatedly interchanges $z = A[i]$ with the smaller of its two children until the heap property is restored. We use the following translation replacement strategy. Let $z = \min(\log n, \lfloor (W - 2d - 1)/\lfloor \log_K(n/P) \rfloor \rfloor - 1)$. We store the extremal translation paths ($2d - 1$ nodes), non-extremal parts of the translation paths for z addresses $a_0, \ldots, a_{z-1}$ and one additional translation path a_∞ ($\lfloor \log_K(n/P) \rfloor$ nodes for each). The additional translation path is only needed when $z \neq \log n$. During the siftdown of $A[i]$, a_0 is equal to the address of $A[i]$, a_1 is the address of one of the children of i (the one to which $A[i]$ is moved, if it is moved), a_2 is the address of one of the grandchildren of i (the one to which $A[i]$ is moved, if is moved two levels down), and so on. The additional translation path a_∞ is used for all addresses that are more than z levels below the level containing i.

Let us upper bound the number of TC misses. Preparing the extremal paths causes up to $2d+1$ misses. Next, consider the translation cost for a_i, $0 \leqslant i \leqslant z - 1$. a_i assumes $n/2^i$ distinct values. Assuming that siblings in the heap always lie in the same page[9], the index (= the part of the address that is being translated) of each a_i is decreasing over time and hence proposition 5.1 bounds the number of TC misses to the number of the non-extremal nodes in the range. We use Proposition 5.2 to count them. For $i \in \{0, \ldots, p\}$ we use the Proposition with $x = n$ and $\ell = 0$ and obtain a bound of

$$\frac{2n}{P} = O\Big(\frac{n}{P}\Big)$$

TC misses. For i with $p + (\ell - 1)k < i \leqslant p + \ell k$, where $\ell \geqslant 1$ and $i \leqslant z - 1$, we use the Proposition with $x = n/2^i$ and obtain a bound of at most

$$\frac{n}{2^i} \cdot \ell + \frac{2n}{PK^\ell} = O\bigg(\frac{n}{2^i} \cdot \ell + \frac{2n}{2^i}\bigg) = \\ = O\Big(\frac{n}{2^i}(\ell + 2)\Big) = O\bigg(n\frac{i}{2^i}\bigg)$$

TC misses. There are $n/2^z$ siftdowns starting in layers z and above, they use a_∞. For each such siftdown, we need to translate at most $\log n$ addresses and each translation causes less than d misses. The total is less than $n(\log n)d/2^a$. Summation yields

$$2d + 1 + (p+1)O\Big(\frac{n}{P}\Big) + \sum_{p < i \leqslant z-1} O\bigg(n\frac{i}{2^i}\bigg) + \frac{nd\log n}{2^z} = \\ = O\bigg(d + \frac{np}{P} + \frac{nd\log n}{2^z}\bigg).$$

For any realistic values of the parameters, the third term is insignificant, hence, the cost is $O\big(\tau(d + \frac{np}{P})\big)$. We next prove the corresponding lower bound under the additional assumption that $W < \frac{1}{2}n/P$. At least one address must be completely translated, hence, cost of $\Omega(\tau d)$. The addresses in $a_0 \ldots a_{p-1}$ assume at least one address per page in subarray $[n/2..n]$, as a_i can never jump by more than 2^{i+1}. First the addresses are swept by a_0, then by a_1 and so on, and no other accesses to the subarray occur in the meantime. Hence, if LRU strategy is in use, and $W < \frac{1}{2}n/P$, there are at least $pn/(2P)$ TC misses to the lowest level of the translation tree. This gives the $\Omega\big(\frac{np}{P}\big)$ part of the misses lower bound. Hence, the total cost is $\Omega\big(\tau(d + \frac{np}{P})\big)$.

[9] This assumption can be easily lifted by allowing an additional constant in running time or in TC size.

6 Cache-Oblivious Algorithms

Algorithms for the EM model are allowed to use the parameters of the memory hierarchy in the program code. For any two adjacent levels of the hierarchy, there are two parameters. The size M of the faster memory and the size B of the blocks in which data is transfered between the faster and the slower memory. Cache-oblivious algorithm are formulated without reference to these parameters, i.e., they are formulated as RAM-algorithms. Only the analysis makes use of the parameters. A transfer of a block of memory is called an IO-operation. For a cache-oblivious algorithm let $C(M, B, n)$ be the number of IO-operations on an input of size when M is the size of the faster memory (also called cache memory) and B is the block size. Of course, $B \leqslant M$.

For several fundamental algorithmic problems, e.g., sorting, FFT, matrix multiply, and searching, there are cache-oblivious algorithm that match the performance of the best EM-algorithms for the problem [FLPR12]. These algorithms are designed such that they show good locality of reference at all scales and therefore one may hope that they also show good behavior in the VAT model. Some of these algorithms require the tall-cache assumption $M \geqslant B^2$.

THEOREM 6.1. *Consider a cache-oblivious algorithm with IO-complexity $C(M, B, n)$, where M is size of the cache, B is size of a block, and n is the input size. Let $a := \lfloor W/d \rfloor$ and let $P = 2^p$ be the size of a page. Then the number of TC faults is at most*

$$\sum_{i=0}^{d} C(aK^iP, K^iP, n).$$

Proof. We divide the translation cache into d parts of size a and reserve one part for each level of the translation tree.

Consider any level i, where the leaves of the translation tree are on level 0. Each node on level i stands for K^iP addresses and we can store a nodes. Thus the number of faults on level i in the translation process is the same as the number of faults of the algorithm on blocks of size K^iP and a memory of a blocks (i.e., size ak^iP). Therefore, the number of TC faults is at most

$$\sum_{i=0}^{d} C(aK^iP, K^iP, n).$$

Theorem 6.1 allows us to rederive some of the results in Section 5. For example, linear scan of an array of length n has IO-complexity at most $2 + \lfloor n/B \rfloor$. Thus the number of TC faults is at most

$$\sum_{i=0}^{d} \Big(2 + \frac{n}{K^iP}\Big) < 2d + \frac{K}{K-1}\frac{n}{P}.$$

It also allows us to derive new results. Quicksort has IO-complexity $O((n/B)\log(n/B))$, and hence the number of TC faults is at most

$$\sum_{i=0}^{d} O\Big(\frac{n}{K^iP}\log\frac{n}{K^iP}\Big) = O\Big(\frac{n}{P}\log\frac{n}{P}\Big).$$

Binary search in van Emde Boas layout has IO-complexity $\log_B n$, and hence the number of TC faults is at most

$$\sum_{i=0}^{d} \frac{\log n}{\log(K^iP)} = \sum_{i=0}^{d} \frac{\log n}{p+ik} \leqslant \frac{\log n}{p} + \log n \int_0^d \frac{1}{p+kx}dx$$
$$= \frac{\log n}{p} + \frac{\log n}{k}\ln\frac{p+dk}{p} \leqslant \frac{\log n}{p} + \frac{\log n}{k}\ln\frac{k+\log n}{p}$$

Matrix multiply with recursive layout of matrices has IO-complexity $n^3/(M^{1/2}B)$, and hence the number of TC faults is at most

$$\sum_{i=0}^{d} \frac{n^3}{(aK^iP)^{1/2}K^iP} < \frac{K^{3/2}}{K^{3/2}-1}\frac{n^3}{a^{1/2}P^{3/2}}.$$

7 Commentary

We received a number of comments from the program committee; we address them in this section.

7.1 The model does not cover everything Current computers are highly sophisticated machines with many features. Each single feature requires a lot of attention to be modeled properly. We concentrated on the feature that leads to the greatest analysis discrepancies for the sequential algorithms. The model in the current form applies to various architectures (even though it was developed in context of the x64 machines), too precise modeling would remove this advantage. Moreover, the model was designed as an independent extension to the RAM model. This way it can be coupled with other (for instance parallel) models as well, with little or no modification.

7.2 How does it relate to EM? In the VAT model we ignore the EM cache misses. However, since every translation is followed by a memory access, one can see the RAM memory just as one additional level of the translation tree. Therefore, in fact VAT implicitly covers the EM cache misses up to the branching factor K

7.3 The model is too complicated While we received comments that the model is too simple, we also received ones saying that the model is too complicated. This impression is probably due to the fact that some of our proofs are somewhat technical. Some arguments simplify if asymptotic notation is used earlier, or if the VAT cost is obviously upper bounded by the RAM cost (for sequential access patterns to the memory). However, as this is the first work on the subject, we find it appropriate to be more detailed than absolutely necessary. With time, more and more simplifications will appear. In particular, there is evidence that for many algorithms the exact value of K does not matter and hence $K = 2$ may be used.

7.4 The translation tree is shallow It is true that height of the translation tree on today's machines is bounded by 4, and so the translation cost is bounded. However, even though our experiments use only 3 levels, the slowdown appears to be at least as significant as one caused by a factor of $\log n$ in operational complexity. Therefore, decreasing VAT complexity has a high practical significance. Please note that while 64 bit addresses are sufficient to address any memory that can be constructed according to known physics, there are other practical reasons to consider longer addresses. Therefore, current bound for the height of the translation tree is not absolute.

8 Conclusions

We introduced the VAT model and analyzed some fundamental algorithms in this model. We showed that the predictions made by the model agree well with measured running times. Our work is just the beginning. There are many open problems, for example: Which translation cost is incurred by cache-oblivious algorithms that require a tall cache assumption? Virtual machines incur the translation cost twice. What is the effect of this? What is the optimal VAT-cost of sorting?

We believe that every data structure and algorithms course must also discuss algorithm engineering issues. One such issue is that the RAM model ignores essential aspects of modern hardware. The EM model and the VAT model capture additional aspects.

References

[Adv10] Advanced Micro Devices. AMD64 architecture programmer's manual volume 2: System programming, 2010.

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[Dre07] Ulrich Drepper. What every programmer should know about memory. 2007. `http://lwn.net/Articles/250967/`.

[Dre08] Ulrich Drepper. The cost of virtualization. *ACM Queue*, 6(1):28–35, 2008.

[FLPR12] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, pages 4:1 – 4:22, 2012. a preliminary version appeared in FOCS 1999.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Diego, 2007.

[Mic07] Pierre Michaud. (yet another) proof of optimality for min replacement. `http://www.irisa.fr/caps/people/michaud/yap.pdf`, October 30 2007.

[Rah03] Naila Rahman. Algorithms for hardware caches and TLB. In Ulrich Meyer, Peter Sanders, and Jop Sibeyn, editors, *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 171–192. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-36574-5_8.

[SS63] J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10(2):217–255, 1963.

[ST85] D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM (CACM)*, 28(2):202–208, 1985.

Appendix

A The VAT model

VAT machines are RAM machines that use virtual addresses. Virtual addresses were motivated by multiprocessing. If several programs are executed concurrently on the same machine, it is convenient and more secure to give each program a linear address space indexed by the nonnegative integers. However, now the addresses are virtual. They do no longer correspond directly to addresses in the physical memory. Rather, the virtual memories of all running programs must be simulated with one physical memory.

We concentrate on the virtual memory of a single program. Both real (physical) and virtual addresses are strings in $\{0, K-1\}^d \{0, \ldots, P-1\}$. The $\{0, K-1\}^d$ part of the address is called index, and its length d is an execution parameter fixed a priori the execution. It is assumed that $d = \lceil \log_K(\text{last used address}/P) \rceil$. The $\{0, \ldots, P-1\}$ part of the address is called page offset and P is the page size. The translation process is a tree walk. We have a K-nary tree T of height d. The nodes of the tree are pairs (ℓ, i) with $\ell \geqslant 0$ and $i \geqslant 0$. We refer to ℓ as the layer of the node and to i as the number of the node. The leaves of the tree are on layer zero and a node (ℓ, i) on layer $\ell \geqslant 1$ has K children on layer $\ell - 1$, namely the nodes $(\ell - 1, Ki + a)$, for

$a = 0 \dots K-1$. In particular, node $(d, 0)$, the root, has children $(d-1, 0), \dots, (d-1, K-1)$. The leaves of the tree store page numbers of the main memory of a RAM machine. In order to translate virtual address $x_{d-1} \dots x_0 y$, we start in the root of T, and then follow the path described by $x_{d-1} \dots x_0$. We refer to this path as the *translation path* for the address. The path ends in the leaf $(0, \sum_{0 \leqslant i \leqslant d-1} x_i K^i)$. Let z be the page index stored in this leaf. Then $zP + y$ is the memory cell denoted by the virtual address. Observe, that y is part of the real address.

The translation process uses a translation cache TC that can store W nodes of the translation tree.[10] The TC is changed by insertions and evictions. Let a be a virtual address and let $v_d, v_{d-1}, \dots, v_0$ be its translation path. Translating a requires to *access* all nodes of the translation path in order. Only nodes in the TC can be accessed. The translation of a ends when v_0 is accessed. The next translation starts with the next operation on the TC.

The *length of the translation* is the number of insertions performed during the translation and the *cost of the translation* is τ times the length. The length is at least the number of nodes of the translation path that are not present in the TC at the beginning of the translation.

A.1 TC Replacement Strategies Since the TC is a special case of a cache in a classic EM machine, the following classic result applies.

LEMMA A.1. ([ST85, FLPR12]) *An optimal replacement strategy is at most by factor 2 better than LRU*[11] *on a cache of double size, assuming both caches start empty.*

This result is useful for upper bounds and lower bounds. LRU is easy to implement. In upper bound arguments, we may use any replacement strategy and then appeal to the Lemma. In lower bound arguments, we may assume the use of LRU. For TC caches, it is natural to assume the initial segment property.

DEFINITION A.1. *An **initial segment** of a rooted tree is an empty tree or a connected subgraph of the tree containing the root. TC has the **initial segment property (ISP)**, if the TC contains an initial segment of the translation tree. A TC replacement strategy has ISP, if, under this strategy, TC has ISP at all times.*

[10] In real machines, there is no separate translation cache. Rather, the same cache is used for data and the translation tree.

[11] LRU is a strategy that always evicts the Least Recently Used node.

PROPOSITION A.1. *Strategies with ISP exist only for TCs with $W > d$.*

ISP is important because strategies with ISP are easier to implement. Any implementation of a caching system requires some way to search the cache. This requires an indexing mechanism. RAM memory is indexed by the memory translation tree. In case of the TC itself, ISP allows to integrate the indexing structure into the cached content. One only has to store the root of the tree at a fixed position. We will show that ISP can be realized at no additional cost for LRU and at little additional cost for the optimal replacement strategy.

A.2 Eager Strategies and the Initial Segment Property Before we prove an ISP analogue of Lemma A.1, we need to better understand the behavior of replacement strategies with ISP. For classic caches premature evictions and insertions do not improve efficiency. We will show that the same holds true for TCs with ISP. This will be useful as we will use early evictions and insertions in some of our arguments.

DEFINITION A.2. *A replacement strategy is **lazy** if it performs an insertion of a missing node only if the node is accessed right after, and performs an eviction only before an insertion for which there would be no free cell otherwise. In the other case the strategy is **eager**. If not stated otherwise, we assume that a strategy being discussed is lazy.*

Eager strategies can perform replacements before they are needed, and can even insert nodes that are not needed at all. Also, they can insert and re-evict, or evict and re-insert nodes during a single translation. We eliminate this behavior *translation by translation* as follows. Consider a fixed translation and define the sets of **effective evictions and insertions** as follows.

$$EE = \{evict(a) : \text{there are more } evict(a) \text{ than } insert(a) \text{ in the translation.}\}$$

$$EI = \{insert(a) : \text{there are more } insert(a) \text{ than } evict(a) \text{ in the translation.}\}$$

Please note that in this case "there are more" means "there is *one* more" as there cannot be two $evict(a)$ without an $insert(a)$ between them, or two $insert(a)$ without $evict(a)$.

PROPOSITION A.2. *The effective evictions and insertions modify the content of the TC in the same way as the original evictions and insertions.*

PROPOSITION A.3. *During a single translation while a strategy with ISP is in use:*

1. *No node from the current translation path is effectively evicted, and all the nodes missing from the current translation path are effectively inserted.*
2. *If a node is effectively inserted, no ancestor or descendant of it is effectively deleted. Subject to obeying the size restriction of the TC, we may therefore reorder effective insertions and effective deletions with respect to each other (but not changing the order of the insertions and not changing the order of the evictions).*

LEMMA A.2. *Any eager replacement strategy with ISP can be transformed into a lazy replacement strategy with ISP with no efficiency loss.*

Proof. We modify the original evict/insert/access sequence *translation by translation*. Consider the current translation and let EI and EE be the set of effective insertions and evictions. We insert the missing nodes from the current translation path exactly at the moment they are needed. Whenever, this implies an insertion into a full cache, we perform one of the lowest effective evictions, where lowest means that no children of the node are in the TC. There must be such an effective eviction as otherwise also the original sequence would overuse the cache. When all nodes of the current translation path are accessed, we schedule all remaining effective evictions and insertions at the beginning of the next translation; first the evictions in descendant-first order and then the insertions in ancestor-first order. The modified sequence is operationally equivalent to the original one, performs no more insertions, and does not exceed cache size. Moreover, the current translation is now lazy.

A.3 ISLRU, or LRU with the Initial Segment Property Even without ISP, LRU has the property below.

PROPOSITION A.4. *When the LRU policy is in use, number of the TC misses in a translation is equal to the layer number of the highest missing node on the translation path.*

Proof. The content of the LRU cache is easy to describe. Concatenate all translation paths and delete all occurrences of each node except the last. The last W nodes of the resulting sequence form the TC. Observe that an occurrence of a node is only deleted if the node is part of a latter translation path. This implies that the TC contains at most two incomplete translation path, namely the least recent path that still has nodes in the TC and the current path. The former path is evicted top-down and the latter path is inserted top-down. The claim now easily follows. Let v be the highest missing node on the current translation path. If no descendant of v is contained in the TC, the claim is obvious. Otherwise, the topmost descendant present in the TC is the first node on the part of the least recent paths that is still in the TC. Thus as the current translation path is loaded into the TC, the least recent path is evicted top-down. As the consequence, the gap is never reduced.

The proof above also shows that whenever LRU detaches nodes from the initial segment, the detached nodes will never be used again. This suggests a simple (implementable) way of introducing ISP to LRU. If LRU evicts a node that still has descendants in the TC, it also evicts the descendants. The descendants actually form a single path. Next, we use Lemma A.2 to make this algorithm lazy again. It is easy to see that the resulting algorithm is the ISLRU as defined next.

DEFINITION A.3. ***ISLRU*** *(Initial Segment preserving LRU) is the replacement strategy that always evicts the lowest descendant of the least recently used node.*

Due to the construction and Lemma A.2 we have the following.

PROPOSITION A.5. *ISLRU for TCs with $W > d$ is at least as good as LRU.*

REMARK A.1. *In fact the proposition holds also for $W \leqslant d$, even though ISLRU no longer has ISP in this case.*

A.4 ISMIN: The Optimal Strategy with the Initial Segment Property

DEFINITION A.4. ***ISMIN*** *(Initial Segment property preserving MIN) is the replacement strategy for TCs with ISP that always evicts the node that is not used for the longest time into the future among the nodes that are not on the current translation path and have no descendants. Nodes that will never be used again are evicted before the others in arbitrary descendant–first order.*

THEOREM A.1. ISMIN *is an optimal replacement strategy among those with ISP.*

Proof. Let R be any replacement strategy with ISP, and let t be the first point in time when it departs from ISMIN. We will construct R' with ISP that does not depart from ISMIN including time t and has no more TC misses than R. Let v be the node evicted by ISMIN at time t.

We first assume that R evicts v at some later time t' without accessing it in the interval $(t, t']$. Then R' simply evicts v at time t and shifts the other evictions in the interval $[t, t')$ to one later replacement. Postponing evictions to the next replacement does not cause additional insertions and does not break connectivity. It may destroy laziness by moving an eviction of a node right before its insertion. In this case R' skips both. Since no descendant of v is in the TC at time t, and v will will not be used for the longest time into the future, none of its children will be added by R before time t'; therefore the change does not break the connectivity.

We come to the case that R stores v till it is accessed for the next time, say at time t'. Let a be the node evicted by R at time t. R' evicts v instead of a and remembers a as being **special**. We guarantee that the content of the TCs in the strategies R and R' differs only by v and the current special node till time t', and is identical afterwords. To reach this goal R' replicates the behavior of R except for three situations.

1. If R evicts the parent of the special node, R' evicts the special node to preserve ISP, and and from now on remembers the parent as being special. As long as only Rule 1 is applied, the special node is an ancestor of a.

2. If R replaces some node b with the current special node, R' skips the replacement and from now on remembers b as the special node. Since a will be accessed before v, Rule 2 is guaranteed to be applied and hence R' is guaranteed to save at least one replacement.

3. At time t', R' replaces the special node with v, performing one extra replacement.

We have shown how to turn an arbitrary replacement strategy with ISP into ISMIN without efficiency loss. This proves the optimality of ISMIN.

We can now state an ISP-aware extension of Lemma A.1.

THEOREM A.2.

$$\mathrm{MIN}(W) \leqslant \mathrm{ISMIN}(W) \leqslant \mathrm{ISLRU}(W) \leqslant \\ \leqslant \mathrm{LRU}(W) \leqslant 2\mathrm{MIN}(W/2),$$

where MIN *is an optimal replacement strategy and $A(s)$ denotes a number of insertions performed by replacement strategy A to an initially empty TC of size $s > d$ for an arbitrary, but fixed sequence of translations.*

Proof. MIN is an optimal replacement strategy, so it is better than ISMIN. ISMIN is an optimal replacement strategy among those with ISP, so it is better than ISLRU. ISLRU is better than LRU by Proposition A.5. $\mathrm{LRU}(W) < 2\mathrm{MIN}(W/2)$ holds by Lemma A.1.

A.5 Improved Relationships

Theorem A.2 implies $\mathrm{LRU}(W) \leqslant 2\mathrm{ISLRU}(W/2)$ and $\mathrm{ISMIN}(W) \leqslant 2\mathrm{MIN}(W/2)$. In this section, we sharpen both inequalities.

LEMMA A.3. $\mathrm{LRU}(W + d) \leqslant \mathrm{ISLRU}(W)$.

Proof. d nodes are sufficient for LRU to store one extra path, hence, from the construction, LRU on a larger cache always stores a superset of nodes stored by ISLRU. Therefore, it causes no more TC misses as it is lazy.

THEOREM A.3. $\mathrm{ISMIN}(W + d) \leqslant \mathrm{MIN}(W)$.

In order to reach our goal, we will prove the following lemmas by modifying an optimal replacement strategy into intermediate strategies with no additional replacements.

LEMMA A.4. *There is an eager replacement strategy on TC of size $W + 1$ that except for a single special cell has ISP, and causes no more TC misses than optimal replacement strategy on TC of size W with no restrictions.*

LEMMA A.5. *There is a replacement strategy with ISP on TC of size $W + d$ that causes no more TC misses than a general optimal replacement strategy on TC of size W.*

Since ISMIN is an optimal strategy with ISP, Theorem A.3 follows from Lemma A.5.

In the remainder of this section some lemmas and theorems require the assumption $W > d$ and some do not. However, even for the latter theorems, we sometimes only give the proof for the case $W > d$.

A.6 Belady's MIN Algorithm

Recall that Belady's algorithm MIN, called also the clairvoyant algorithm is an optimal replacement policy. The algorithm always replaces the node that will not be accessed for the longest time into the future. An elegant optimality proof for this approach is provided in [Mic07]. MIN does not differentiate between nodes that will not be used again. Therefore, without loss of generality let us from now on consider descendant–first version of MIN. For any point in time, let us call all the nodes that are to be still accessed in the current translation **the required nodes**. The required nodes are exactly the nodes that are on the current translation path, and are descendants of the last accessed node (or the whole path if the translation is only about to begin).

LEMMA A.6. *1. Let w be in the TC. As long as w has a descendant v in the TC that is not a required node,* MIN *will not evict w.*

2. If $W > d$, MIN *never evicts the root.*

3. If $W > d$, MIN *never evicts a required node.*

Proof. Ad. 1. If v will be accessed ever again, then w will be used earlier (in the same translation), and so MIN evicts v before w. If v will never be accessed again, then MIN evicts it before w because it is the descendants–first version. Ad. 2. Either TC stores whole current translation path, and no eviction occurs; or there is a cell in the TC that contains a node off the current translation path, hence, the root is not evicted as it has a non required descendant in the TC. Ad. 3. Either TC stores whole current translation path, or there is a cell c in the TC with content that will not be used before any required node. Hence, no required node is the node that will not be needed for the longest time into the future.

COROLLARY A.1. *If $W > d$,* MIN *inserts root into the TC as a first thing during the first translation, and never evicts it.*

LEMMA A.7. *If $W > d$,* MIN *evicts only (non-required) nodes with no stored descendants or the node that was just used.*

Proof. If MIN evicts a node on the current translation path it cannot be descendant of the just translated node (lemma A.6, claim 3), it also cannot be ancestor of the just translated node (lemma A.6, claim 1). Hence, only the just translated node is admissible. If the algorithm evicts a node off the current translation path it must have no descendants (lemma A.6, claim 1).

LEMMA A.8. *If* MIN *has evicted the node that was just accessed, it will continue to do so also for all the following evictions in the current translation. We will refer to this as* ***round robin*** *approach.*

Proof. If MIN have evicted a node w that was just accessed, it means that all the other nodes stored in the TC will be reused before the evicted node. Moreover, all subsequent nodes traversed after w in the current translation will be reused even later than w if at all. In case of $W > d$ the claim holds by lemma A.7.

COROLLARY A.2. *During a single translation* MIN *proceeds in the following way:*

1. *It starts with the* ***regular phase*** *when it inserts missing nodes of a connected path from the root up to some node w, as long as it can evict nodes that will not be reused before just used ones.*
2. *It switches to the* ***round robin phase*** *for the remaining part of the path.*

It is easy to see that for $W > d$, in the path that was traversed in the round robin fashion, informally speaking, all gaps move up by one. For each gap between stored nodes, the very TC cell that was used to store the node above the gap now stores the last node of the gap. Storage of other nodes does not change. This way the number of nodes from this path stored in the TC does not change either. However, it reduces numbers of stored nodes on side paths attached to the path.

A.7 Proof of Lemma A.4 We introduce a replacement strategy RRMIN[12]. We add a special cell rr to the TC, and we refer to the remaining W cells as **regular TC**. We will show that the cell rr allows us with no additional TC misses, to preserve ISP in the regular TC. We start with an empty TC, and we run MIN on a separate TC of size W on a side and observe its decisions.

We keep track of a partial bijection[13] φ_t on nodes of the translation tree. We put one timestamp t on every TC access, and in the regular phase of MIN one more between each two accesses. We position evictions and insertion between the timestamps, at most one of each between two consecutive accesses. At time t, φ_t maps every node stored by MIN in its TC to a node stored by RRMIN in its regular TC. Function φ_t always maps nodes to (not necessarily proper) ancestors in the memory translation tree. We denote this as $\varphi_t(a) \sqsubseteq a$, and in case of proper ancestors as $\varphi_t(a) \sqsubset a$. We say that a is a witness for $\varphi_t(a)$.

PROPOSITION A.6. *Since the partial bijection φ_t always maps nodes to ancestors, for every path of the translation tree,* RRMIN *always stores at least as many nodes as* MIN.

In order to prove the lemma A.4 we need to show how to preserve properties of the bijection φ_t and ISP. In accordance to the corollary A.2, MIN inserts a number of highest missing nodes in the regular phase, and uses round-robin approach on the remaining ones.

Let us first consider the case when MIN has only regular phase and inserts the complete path. In this case we substitute evictions and insertions of MIN with those described below.

Let MIN evict a node a. If $\varphi_t(a)$ has no descendants RRMIN evicts it. In the other case we find $\varphi_t(b)$ a descendant of $\varphi_t(a)$ with no descendants on his own.

[12] Round Robin MIN

[13] A partial bijection on a set is a bijection between two subsets of the set.

RRMIN evicts $\varphi_t(b)$, and we set $\varphi_{t+1}(b) \coloneqq \varphi_t(a)$. Clearly, we preserved properties of φ_{t+1}[14] and ISP holds.

Now let MIN insert a new node e. At this point we know that both RRMIN and MIN store all ancestors of e. If RRMIN did not store e yet, RRMIN inserts it and we set $\varphi_{t+1}(e) \coloneqq e$. If e is already stored, it means it has a witness $\varphi_t^{-1}(e)$ that is a proper descendant of e. We a find a sequence $e \sqsupset \varphi_t^{-1}(e) \sqsupset \varphi_t^{-2}(e) \sqsupset \ldots \sqsupset \varphi_t^{-k}(e) = g$, that ends with g RRMIN did not store yet. Such g exists as φ_t^{-1} is an injection on a finite set, and is undefined for e. We set $\varphi_{t+1}(h) \coloneqq h$ for all elements of the sequence except g. RRMIN inserts highest not stored ancestor f of g and we set $\varphi_{t+1}(g) \coloneqq f$. Note, that inserted node f might not be a required node. Properties of φ_t are preserved, and RRMIN did not disconnect the tree it stores. Also, RRMIN performed the same number of evictions and insertions as MIN. Note as well, that for all nodes on the translation path φ_t is identity. Finally, proposition A.6 guarantees that all access are safe to perform at the time they were scheduled.

Now let us consider case when MIN has both regular and round robin phase. Assume that the regular phase ends with the visit of node v. At this point, MIN stores the (nonempty for $W > d$ due to collorary A.1) initial segment p_v of the current path ending in v, it does not contain v's child on the current path, and it contains some number (maybe zero) of required nodes. Starting with v's child, MIN uses the round-robin strategy. Whenever, it has to insert a required node, it evicts its parent. Let ℓ_r and ℓ_{rr} be the number of evictions in the regular and round-robin phase, respectively.

RRMIN also proceeds in two phases. In the first phase, RRMIN simulates the regular phase as described above. RRMIN also performs ℓ_r evictions in the first phase and φ_t is the identity on p_v at the end of the first phase; this holds because φ_t maps nodes to ancestors, and since MIN contains p_v in its entirety at the end of the regular phase. Let d' be the number of nodes on the current path below v; MIN stores $d' - \ell_{\text{rr}}$ of them at the beginning of the round-robin phase, which it does not have to insert, and does not store ℓ_{rr} of them, which it has to insert. Since φ_t is the identity on p_v after phase 1 of the simulation and maps the $d' - \ell_{\text{rr}}$ required nodes stored by MIN to ancestors, RRMIN stores at least the next $d' - \ell_{\text{rr}}$ required nodes below v in the beginning of phase 2 of the simulation. In the round-robin phase RRMIN inserts the required nodes missing from the regular TC one after the other into rr disregarding what MIN does. Whenever MIN replaces a node a with its child g, in case of $W > d$ we fix φ_t by setting $\varphi_{t+1}(g) \coloneqq \varphi_t(a)$. By proposition A.6, RRMIN does no more evictions than MIN. Therefore, as it also preserves ISP in the regular TC the lemma A.4 holds.

A.8 Proof of Lemma A.5 In order to prove the lemma we will show how to use additional d regular cells in the TC to provide functionality of the special cell rr while preserving ISP in the whole TC. We run the RRMIN algorithm aside on a separate TC of size $W + 1$, and we introduce another replacement strategy we call LIS[15] on a TC of size $W + d$. LIS starts with an empty TC where d cells are marked. LIS preserves the following invariants.

1. Set of nodes stored in the unmarked cells by LIS is equal to set of nodes stored in the regular TC by RRMIN.
2. Set of nodes stored in the marked cells by LIS contains the node stored in the cell rr by RRMIN.
3. Exactly d cells are marked.
4. LIS has ISP.
5. No node is stored twice (once marked, once unmarked).

Whenever RRMIN can replicate evictions/insertions of LIS without violating the invariants, it does. Otherwise, we consider the following cases.

1. Let RRMIN in the regular phase evict a node a that has marked descendants in LIS. Then, LIS marks the cell containing a, and unmarks and evicts one of the marked nodes with no descendants that does not store the node stored in rr. Such a node exists, as the only other case is that the marked cells contain all nodes of some path excluding the root, and the leaf is stored in rr. Therefore, a is the root, but root is never evicted due to ISP.
2. In the regular phase, RRMIN inserts a node c to an empty cell while LIS already stores c in a marked cell. In this case LIS unmarks the cell with c, and marks the empty cell.
3. In the round robin phase, RRMIN replaces content of the cell rr, LIS (if needed) replaces the content of an arbitrary marked node with no descendants that is not on the current translation path. Since the root is always in the TC and there are d marked cells, such a cell always exists. ISP is preserved, as parent of this node is already in the TC.

[14] φ_{t+1} is equal to φ_t on all arguments not explicitly specified.

[15] Lazy strategy preserving the Initial Segments property

At this stage if we drop notions of φ_t and marked nodes, LIS becomes an eager replacement strategy on a standard TC. Therefore, we can use lemma A.2 to make it lazy. This concludes the proof of lemma A.5.

REMARK A.2. *We believe that requirement for d is essentially optimal. Consider the scenario when we access subsequent cells uniformly at random. Informally speaking,* MIN *will tend to permanently store first* $\log_K(W)$ *levels of the translation tree as they are frequently used, and will use a single cell to traverse the lower levels. In order to preserve ISP we need* $d-\log_K(W)+1$ *additional cells for storing the current path. Not uniform random patterns should lead to even higher requirements. This does not seam to give much more space for improvement.*

CONJECTURE A.1. *Strategy of storing higher nodes (lemma A.4) and using extra d cells to not evict nodes from the current translation path (lemma A.5) can be used to add ISP to any replacement strategy without efficiency loss.*