Algorithm Engineering

Kenneth S. Bøgh, Manuel R. Ciosici, Mathias Rav

{ksb,manuel,rav}@cs.au.dk

March 17, 2015

Contents

1 Introduction

In this report, we present an empirical analysis of our implementations of binary search, radix sort and matrix multiplication. We focus on how each implementation performs in practice, but our report will include theoretical calculations on the cache performance of our implementations whenever helps explain our observations.

In Section 1.1, we present the experimental setup. Sections 2, 3 and 4 cover the three different projects, and each section is split into introduction and experiments, with two pages set aside for plots and 3-4 pages of text.

1.1 Experimental setup

All experiments in this report are performed on the computer `beast` in Nygaard-346. The specifications of the system are outlined in Figure 1. Since our algorithms work with 32-bit integers, throughout this report, $B = 16$ is the cache line size (64 Bytes), $B' = 1024$ is the page size (4096 Bytes). We refer to the size of a cache or the TLB by M whenever we analyze theoretical cache performance. Hyper-threading is enabled in all experiments, meaning parallel algorithms can run two threads on one core. It should be noted that the machine has a relatively slow clock rate, more cores and a larger L3 cache when compared to most desktops and laptops. This makes the number of instructions executed during an experiment relatively more important than it would be on a machine with a higher frequency.

All the algorithms are implemented in C++11 and compiled using GCC version 4.8.3 with the optimization flags `-O3 -mavx -march=native`.

We use the `clock_gettime` system call with the `CLOCK_MONOTONIC` flag to measure the running time of algorithms.

Furthermore, we use the PAPI (Performance Application Programming Interface) library version 5.4 for measuring hardware utilization as part of our analysis. Since there is a limit to the available hardware counters, we have executed each experiment with the four different sets of hardware counters listed in Figure 2. Each measurement with a set of hardware counters is repeated 10 times to minimize the impact of hardware interrupts and other processes running on the machine.

Therefore, all plots in this report show averages of 10 measurements, as well as minimum and maximum measurements as errorbars (if applicable) to provide a clear picture of the stability of the measurements. The elapsed time is measured simultaneous with the four sets of PAPI counters, so plots of elapsed time show average, minimum and maximum of 40 measurements. Our plots do not show any measurements for preprocessing or data generation.

For all of our experiments, we generate data and queries using the pseudo-random number generator (PRNG) described by Matsumoto and Nishimura [4] and implemented as `std::mt19937` in C++11.

Instead of actual appendices, we have used the `attachfile` LaTeX package to include the data files for each plot. Clicking on plots will open up the tab-separated values in a text editor or spreadsheet application. The code used for all experiments is attached in the following PDF annotation:

General information	
Workstation	Fujitsu CELSIUS M720
CPU	Intel Xeon E5-2650
Micro-architecture	Sandy Bridge
Linux kernel version	3.2.65

Performance	
Number of cores	8
Number of threads	16
Processor frequency	2.0 GHz
SIMD register width	256 bits

Cache, TLB and memory	
Cache line size	64 B
L1 data cache	32 KB per core (8-way)
L2 cache	256 KB per core (8-way)
L3 cache	20 MB shared (20-way)
RAM	32 GB
Page size	4096 B
L1 data TLB	64 entries per core (4-way)
L2 shared TLB	512 entries per core (4-way)

Figure 1: Hardware layout. TLB information from the Intel Software Developer's Manual, Volume 3A, Table 11-1.

Instructions

PAPI_STL_ICY: Idle cycles
PAPI_TOT_INS: Instructions executed
PAPI_TOT_CYC: Total cycles

Cache

PAPI_L2_DCA: L2 data cache accesses
PAPI_L2_DCM: L2 data cache misses
PAPI_L3_TCA: L3 total cache accesses
PAPI_L3_TCM: L3 total cache misses

Branches

PAPI_BR_PRC: Branches predicted
PAPI_BR_MSP: Branches mispredicted

TLB

PAPI_TLB_DM: Data TLB misses
PAPI_TLB_IM: Instruction TLB misses

Figure 2: The four sets of PAPI counters used in our experiments

2 Binary search

In this section we provide a detailed comparative performance analysis of four different memory layouts for binary search trees. This work is based on the results presented by Brodal et al. [1]. We assume familiarity with binary search trees.

2.1 Introduction

The problem we address in this section is the implicit predecessor data structure problem, which can be stated as follows: Preprocess a totally-ordered set S of N elements into an array of N cells such that given an input x, we can efficiently find the *predecessor* of x in S, that is, the largest element in the set $\{y \in S \mid y \leq x\}$, or report that x is less than all elements in S.

It is easy to show that at least $\lceil \log_2 N \rceil$ elements of S must be read in the worst case for any data structure, and this is easily achieved using binary search. In practice, the memory layout and the complexity of the query algorithm have a significant impact on performance beyond the $\geq \lceil \log_2 N \rceil$ reads.

Another way to view the problem is that we build a balanced binary search tree (BST) on the elements in S and perform predecessor queries by reading the root and recursing either into the left subtree or the right subtree. We work on the *implicit* variant of the problem, in which no additional information may be stored. In all of our implementations, we store a permutation of S in an array along with its size, N; the binary search tree is *not* represented by pointers and nodes explicitly.

We study the following four memory layouts for binary search trees.

An obvious memory layout is the **in-order** layout which stores data in sorted order in the array. A query proceeds by maintaining a half-open interval $[l, r)$, indicating that the predecessor of x is in that range, where initially $l = 0$ and $r = N$. Any subtree of the BST is laid out contiguously in the array, with the left subtree before the root and the right subtree after the root as illustrated in Figure 3(a).

Another well-known layout is the **breadth-first** (BFS) layout of binary heaps. The root is stored in the first cell (cell 0), and the children of cell i are cells $2i + 1$ and $2i + 2$. Each layer of the BST is laid out contiguously in memory, illustrated in Figure 3(c). The BST is *heap-shaped*, meaning that all levels of the tree except the last are fully filled, and the last level is filled from left to right.

Like BFS, the **depth-first** (DFS) layout has the root in the first cell. Any subtree is laid out contiguously, with the root followed by the left subtree, which is then followed by the right subtree, illustrated in Figure 3(b). The BST is heap-shaped.

For the **van Emde Boas** layout, it still holds that subtrees are laid out contiguously. Instead of specifying where the root is laid out, we specify that the *top tree*, that is, the first h layers of the tree of height $2h$ or $2h + 1$, is recursively laid out in the beginning of the array. From this, we derive that the root is indeed the first element of the array. The top tree is followed by the 2^h *bottom trees* of height h or $h + 1$. This is illustrated in Figure 3(d). The BST is heap-shaped.

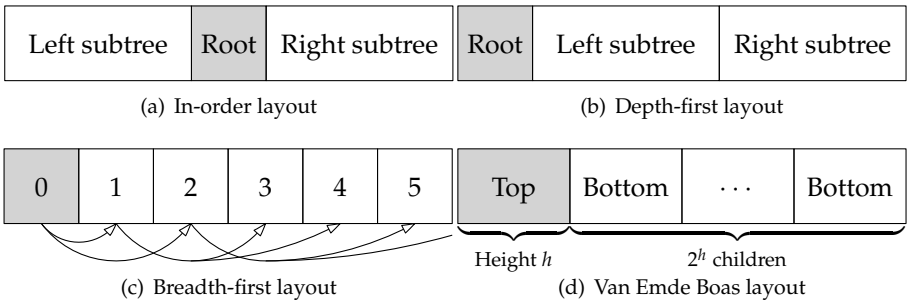


Figure 3: The four memory layouts we consider. See also Figure 3 in Brodal et al. [1] for an illustration of the indices assigned by the layouts to nodes of a complete binary tree of height 4.

2.2 Experiments

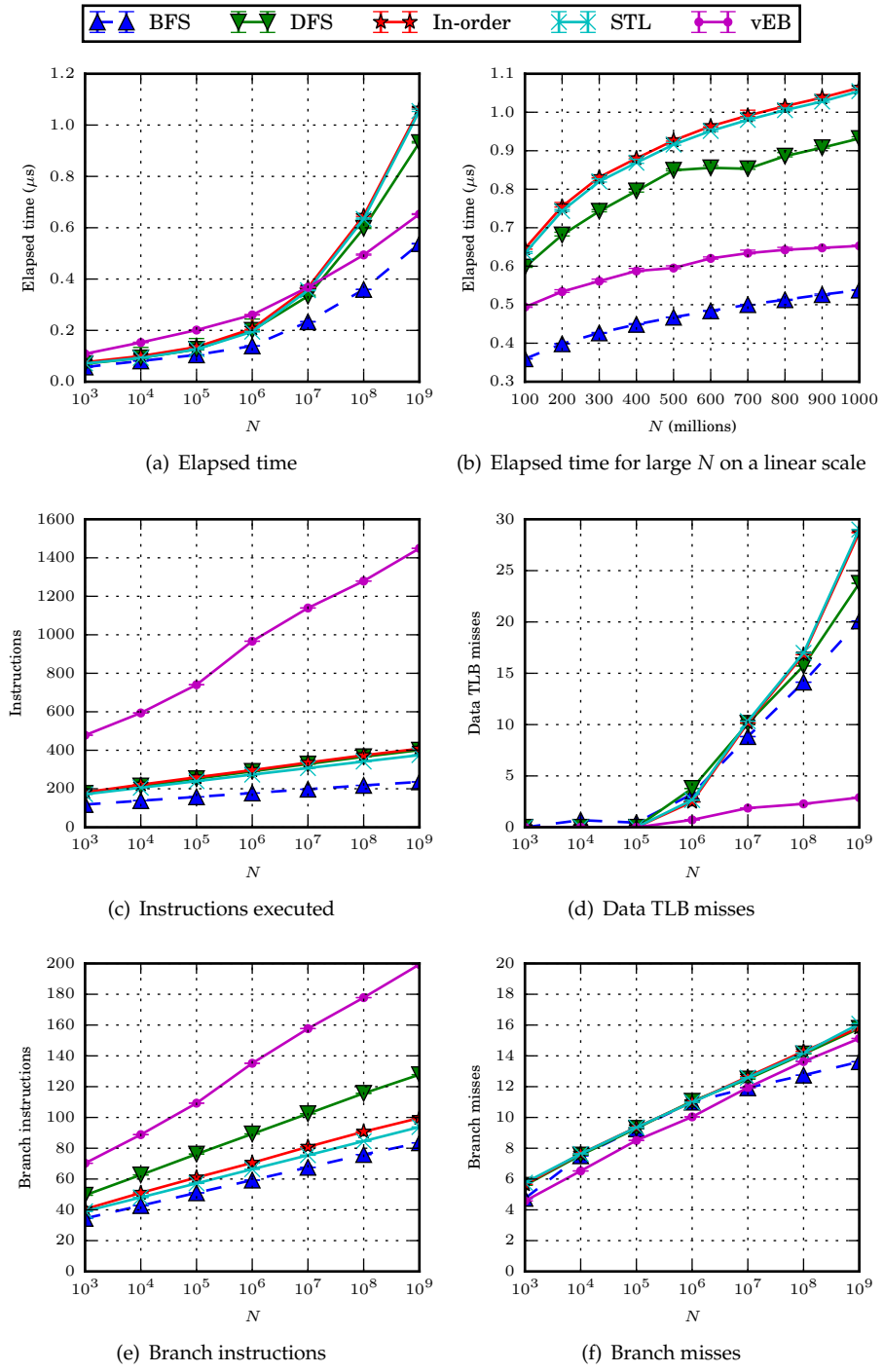
In our experiments, we have implemented two variations of in-order layout (that is, traditional binary search): an implementation using `std::lower_bound` from the C++ standard library (STL), and our own implementation using a half-open interval $[l, r)$ as described in the previous section (in-order).

The DFS layout and vEB layout compute the height of the tree using an MSB instruction on N for every query, since we need to know the tree height to find the children of the root.

We have run the five different BST memory layouts on data sets of size $N = 10^3, 10^4, 10^5$ to see the effect of the data structure fitting or not fitting into the L1 and L2 caches, $N = 1\text{M}, 2\text{M}, 3\text{M}, \ldots, 10\text{M}$ to see the effect of the data structure no longer fitting into the L3 cache, and $N = 100\text{M}, 200\text{M}, 300\text{M}, \ldots, 1000\text{M}$ to see the effect of the data structure well exceeding the size of the L3 cache. For each data structure and choice of N, we run 10 measurements and take the average. Each measurement is on a loop of 10^7 queries (when $N \leq 10\text{M}$) or 10^6 queries (when $N \geq 100\text{M}$).

The state of the pseudo-random number generator (PRNG) is captured after input generation, and this state is used to initialize the PRNG generating the random queries. This means that we average 10 measurements on the same random queries, and that different data structures see the same input and the same queries for the same input size.

In-order. The STL implementation of in-order is only slightly faster than our own in-order implementation as seen in Figures 4(a) and 4(b). STL performs less branch instructions in Figure 4(e), but incurs the same number of branch mispredictions in Figure 4(f) and cache misses on all levels in Figures 5(c), 5(d) and 5(f) as our in-order implementation. The reason that STL has less branch instructions is that it does not have a special case for when x is equal to the current node; instead, it just sees that x is less or equal to the current node and recurses on the left subtree. This means that more cells are read than necessary in some cases, but this penalty is offset by the reduced number of instructions.



(a) Elapsed time

(b) Elapsed time for large N on a linear scale

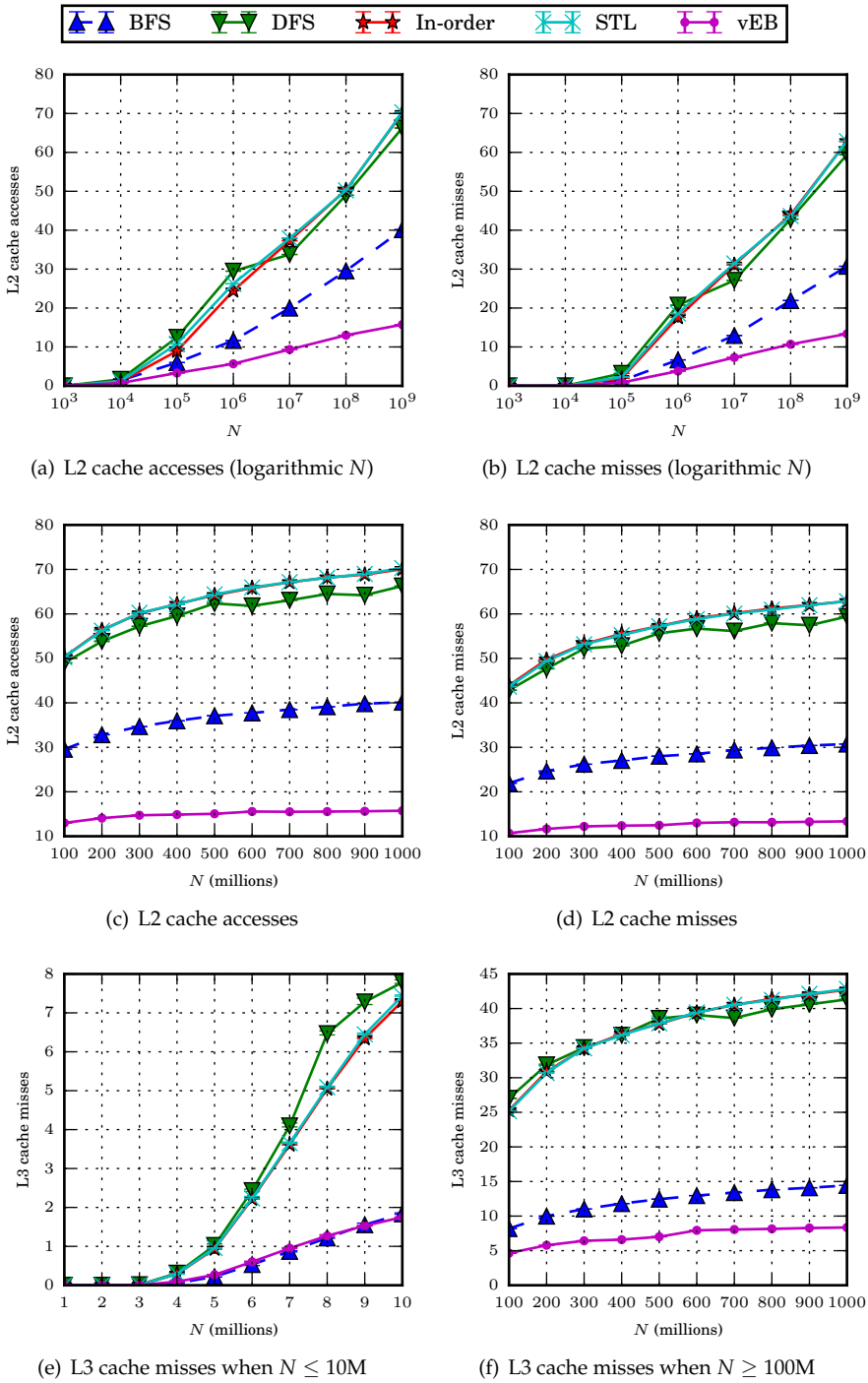
(c) Instructions executed

(d) Data TLB misses

(e) Branch instructions

(f) Branch misses

Figure 4: Plots showing the computational performance of our five predecessor data structures, per query.



(a) L2 cache accesses (logarithmic N)

(b) L2 cache misses (logarithmic N)

(c) L2 cache accesses

(d) L2 cache misses

(e) L3 cache misses when $N \leq 10$M

(f) L3 cache misses when $N \geq 100$M

Figure 5: Plots showing the memory-wise performance of our five predecessor data structures, per query.

BFS and DFS. BFS is by far the shortest implementation of all in terms of instructions executed in the query loop, since each iteration in the while-loop requires just a single comparison with N and a simple multiply-and-add instruction (opcode `LEA`), so it only executes two thirds of the instructions of its nearest competitor (STL) in Figure 4(c). DFS requires an extra branch instruction per level in the search tree compared to BFS in Figure 4(e) because it has to figure out the height of the subtree we recurse into. This does not lead to a much larger number of branch misses than BFS as shown in Figure 4(f), indicating that this extra branch is often predicted correctly.

An optimal cache strategy for BFS keeps the first M elements of the search tree in cache, meaning it should incur approximately $\log_2 \frac{N}{M}$ cache faults per query. On the other hand, DFS will incur a cache fault roughly every time it recurses into the right subtree, which is expected to happen half of the time, so it should incur approximately $\frac{1}{2} \log_2 \frac{N}{B}$ cache faults before the subtree is small enough to fit in a cache line. This means that as long as $N \leq M^2$, we expect BFS to incur fewer cache faults than DFS; since the size of the L3 cache is several megabytes, we need N to be on the order of terabytes to have $N > M^2$ in the case of L3 cache. Our L2 cache is 256 KB which corresponds to 2^{16} elements, meaning we need to have at least 2^{32} elements in the input before DFS could hope to perform better than BFS. Unfortunately, our largest experiment is $N = 10^9 < 2^{30}$. On the other hand, having more than 2^{32} 32-bit integers in our input is futile; an alternative in this case is direct addressing into an array of size 2^{32} which we would expect to be more efficient than both BFS and DFS. When $N > 2^{32}$, direct addressing would use *less* space than the memory layouts for binary search trees we have implemented.

FiXme Note: Actually $N \leq M^2/B$; the rest of the section needs to be rewritten

Van Emde Boas layout. We have seen in the lecture that the van Emde Boas layout incurs roughly 1 cache fault per $\frac{1}{2} \log B$ levels in the search tree, or $\mathcal{O}(\log_B N)$ cache faults in total, for any cache line size B, and from our measurements of TLB misses in Figure 4(d), for which the block size is $B' = 1024$, it seems that van Emde Boas has the least number of TLB misses as N becomes large. Regarding memory caches, for which the block size is only $B = 16$, van Emde Boas is outperformed by BFS and DFS on both L2 accesses, L2 misses and L3 misses in Figures 5(c), 5(d) and 5(f). Comparing the relative order of L2 cache faults to the order of TLB faults, we see that the increased block size of the TLB gives an advantage to the van Emde Boas layout.

Overall, the reason why van Emde Boas is *not* a clear winner is due to the computational overhead involved at every level of the recursion. It needs to maintain several pieces of information for each node while it recurses, and this leads to the great increase in number of branch instructions in Figure 4(e) (more cases to consider) and in instructions in Figure 4(c) (more information to maintain). This is particularly bad on the CPU with low clock frequency on which we ran the experiments.

3 Matrix multiplication

In this section we provide a detailed comparative performance analysis of four different matrix multiplication algorithms. This work is partially based on the results presented by Frigo et al. [3]. We assume familiarity with matrix operations.

3.1 Introduction

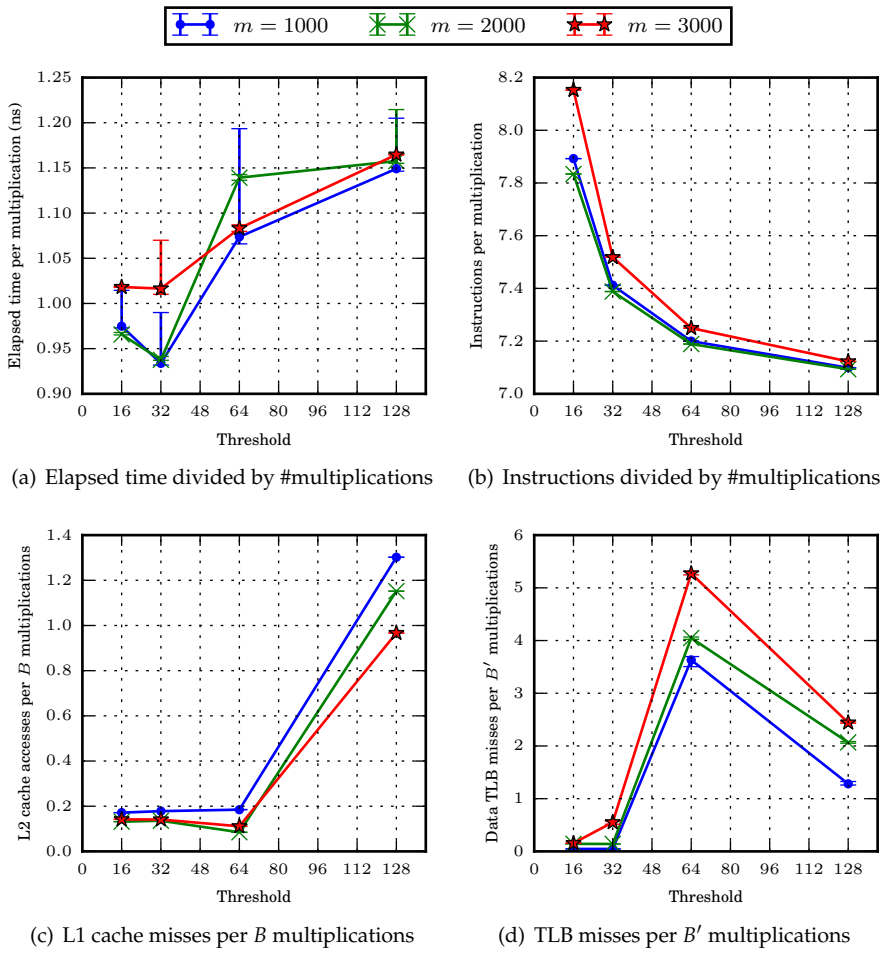
In our implementations of four different algorithms for matrix multiplication, we implement matrix multiplication in terms of *matrix slices*, which target a rectangle of one of the input or output matrices. A matrix slice consists of a pointer to the top left element along with a number of rows, columns, and a *stride* which specifies the distance from the start of one row to the start of the next row. For a slice corresponding to a full input or output matrix, the stride is equal to the number of columns. We can efficiently point to any element inside a matrix slice using a simple multiply-and-add instruction (opcode `LEA`), and we can efficiently compute a representation of a subslice of a matrix slice by keeping the same stride and changing the pointer and the number of rows and columns.

All our experiments for matrix multiplications use single-precision floating point numbers, meaning each element takes up 32 bits. All four algorithms perform exactly mnp scalar multiplications when multiplying matrix A (of size $n \times m$) with matrix B (of size $m \times p$), so when we later refer to cache misses per B multiplications, for example, it means the measurement divided by $\frac{mnp}{B}$. This gives a more precise picture of the performance, since it shows the relationship between what we measure and what we would measure in an ideal world in which each multiplication incurred $\frac{1}{B}$ cache faults amortized. The four matrix multiplication algorithms are briefly described below.

The **naive algorithm** is a direct implementation of the multiplication operation definition for matrices: $(\mathrm{AB})_{ij} = \sum_{k=0}^{m} a_{ik} b_{kj}$. In order to multiply A with B, each cell in the output is produced one at a time by fixing a row of A and traversing the columns of B one after another.

The **recursive algorithm** is from the article on cache oblivious algorithms by Frigo et al. [3]. It multiplies A with B by recursing on matrix slices of half the size of the input matrices depending on which of the three dimensions is the largest. When all of m, n and p are less than some constant threshold, the recursion stops and the algorithm switches to multiplying the matrices using the naive algorithm described above. The idea behind using the naive algorithm is that once the matrices reach a certain size, they fit in cache, so the naive algorithm does not incur any cache faults.

The **transpose-ij algorithm** multiplies A with B by first computing $\mathrm{B}^\intercal$ (the transpose of B: $(\mathrm{B}^\intercal)_{ij} = \mathrm{B}_{ji}$) using a straightforward recursive cache-efficient procedure. After that, it generates each output entry (i, j) one after another similar to the naive algorithm. In other words, it fixes one row of the A matrix at a time, while iterating through rows of $\mathrm{B}^\intercal$ (which correspond to columns in the original B matrix). Practically, the algorithm computes the contribution of a row in the A matrix to the result matrix before moving on to the next row. The transposition ensures that B is read sequentially m times.



(a) Elapsed time divided by #multiplications

(b) Instructions divided by #multiplications

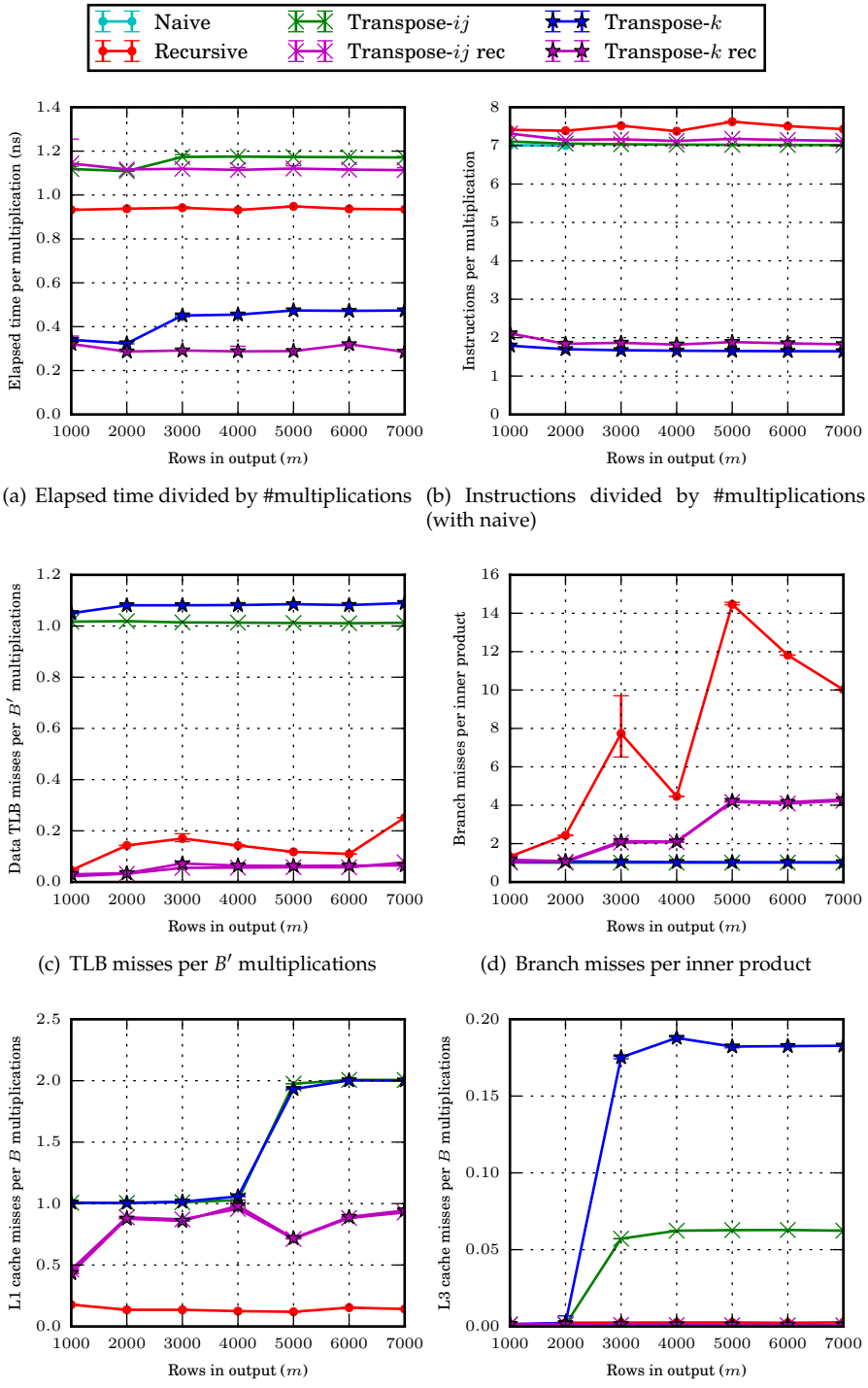
(c) L1 cache misses per B multiplications

(d) TLB misses per B' multiplications

Figure 6: Various plots showing the effect of changing the base case threshold of recursive.

The **transpose-k algorithm** multiplies A with B by first computing $\mathrm{A}^\intercal$. After that, it performs n passes over the output AB, in pass k adding $(\mathrm{A}^\intercal)_{ki}\mathrm{B}_{kj}$ to entry (i, j) in the output, row by row. In other words, the algorithm computes the outer product of column k of A and row k of B, adding the result to the result matrix.

Measurements for the transpose-ij and transpose-k algorithms include the computation of the matrix transpose, since this preprocessing step is not needed for the other two algorithms.



(a) Elapsed time divided by #multiplications (b) Instructions divided by #multiplications (with naive)

(c) TLB misses per B' multiplications (d) Branch misses per inner product

(e) L1 cache misses per B multiplications (f) L3 cache misses per B multiplications

Figure 7: Various plots showing the performance of matrix multiplication.

3.2 Experiments

In all our experiments, we fix $n = m - 100$ and $p = m - 200$ and set m to be a multiple of 1000. The naive algorithm is more than 9 times slower than the rest when $m = 2000$ since it incurs more than 1000 times more TLB misses, and so we did not run experiments with naive for larger m, and it is excluded from most of the plots. However, the data for the naive algorithm is included in the data files annotating each plot.

Threshold selection. In order to establish the best constant threshold value to use in the recursive matrix multiplication algorithm, we ran the algorithm with threshold values $16, 32, 64, 128$ and with the three input sizes $m = 1000$, $m = 2000$ and $m = 3000$ as shown in Figure 6. In Figure 6(a), we see that the fastest time for the matrix multiplication is achieved when the threshold is set to 32 regardless of input size, even though the number of instructions displayed in Figure 6(b) decreases since the naive algorithm is much simpler than the recursive, and the number of L1 cache misses displayed in Figure 6(c) does not increase until the threshold is 128.

The bottleneck at a threshold of 64 seems to be the TLB misses shown in Figure 6(d). For the three input sizes, the picture is the same: The number of TLB misses at threshold 64 is much larger than at threshold 32 and 128. We explain it as follows. When $m = 1000$, $n = 900$, $p = 800$, at a recursion depth of 12, we have $m/2^4 = 62.5$, $n/2^4 = 56.25$, $p/2^4 = 50$, meaning that the algorithm has ended up with two inputs that are smaller than the threshold. Further, the number of columns in B is 800 and the stride is 3200 bytes which is just less than B', the size of a page. When the naive algorithm, as part of the recursive algorithm, multiplies a 62×56 matrix slice with a 56×50 matrix slice, it goes through the columns of the right hand matrix one after another for a total of 62 times. However, in the common case, the 56 rows reside in different pages, meaning that 56 pages must fit in the TLB to prevent TLB faults. The L1 TLB only has 64 entries, so it is likely that some of these pages will be evicted while each column of the right hand side is being traversed. If we are unlucky, a least-recently-used policy will determine that the pages mapping the first rows should be evicted at the time when we look up pages for the last rows.

All experiments on the recursive algorithm in Figure 7 are run with the threshold set to 32.

Translation lookaside buffer misses. The transpose-ij algorithm reads the $B^\intercal$ matrix a number of m times. Optimally, all the pages containing the $B^\intercal$ matrix would be stored in the L1 TLB, which consists of 64 entries. For example, when $m = 1000$, $B^\intercal$ is of size 800×900, uses 2.74 MB of memory and spans $np/B' = 704$ pages, which exceeds both L1 and L2 TLB. Similarly, A spans $mn/B' = 879$ pages, and the result matrix spans $mp/B' = 783$ pages.

In order to compute the result matrix, we incur at least the following number of TLB faults: scanning the result once incurs mp/B' faults, scanning A once incurs mn/B' faults, and scanning B m times incurs mnp/B' faults. When we measure the number of TLB faults divided by mnp/B', we see the values in Figure 7(c). Since the number of TLB misses is dominated by the number of TLB faults for the $B^\intercal$ matrix, if we increase the number of elements in the $B^\intercal$ matrix by a constant, we will also increase the number of TLB misses by the same constant, resulting in the same value of TLB misses per B' multiplications. The same effect can be observed for the transpose-k algorithm.

L1 cache misses. The optimal cache eviction strategy for transpose-ij keeps a row of A and a cache line of B and the result matrix in cache ($2B + n$ cells), leading to a total of $(mnp + mn + mp)/B$ cache faults for scanning B m times, A once and the output once, and indeed we see roughly mnp/B L1 cache faults in Figure 7(e) when up to and including $m = 4000$, for which $2B + n = 15\,728$ bytes, less than half of the L1 cache. Going to $m = 5000$, we see that the number of L1 misses doubles, even though our cache requirement, $2B + n = 19\,728$ bytes, is only a little more than half of the L1 cache. We can explain this by the following observation: Computing the inner product between two rows of length n requires touching $2n$ memory cells, so if $2n \leq M$, a cache eviction strategy such as LRU will perform optimally. However, when $m = 5000$, $2n$ is *greater than* the size of the L1 cache, so the L1 cache starts evicting the beginning of the rows while computing the inner product. When the next inner product has to be computed, then instead of only reading a row from $\mathrm{B}^\intercal$ into L1, we read both a row of $\mathrm{B}^\intercal$ and a row from A into L1, which explains why the number of L1 misses doubles when $n \geq 5000$.

L3 cache misses. The recursive algorithm only touches the memory of the matrices when it calls the naive algorithm, and it only does so when the inputs are less than 32 in all dimensions, which is small enough for both inputs and the output to fit in L3 cache. Therefore, it is not surprising that the recursive algorithm incurs very few L3 cache misses as seen in Figure 7(f): around one per $400B$ scalar multiplications.

The optimal cache requirement of the transpose-k algorithm requires the L3 cache to store the entire result matrix, one row of the B matrix and one cache line from the A matrix. Similarly, the transpose-ij algorithm requires the L3 cache to store the entire result matrix, one row of the B matrix and one entire row of the A matrix. When $m = 2000$, the L3 cache storage requirements for transpose-k is 13.8 MB, and for transpose-ij 13.88 MB, which fit well within the L3 cache, so we measure very few L3 cache misses. When $m = 3000$, the cache storage requirements for transpose-k grow to 32.37 MB and for transpose-ij to 32.38 MB, which is well above the size of the L3 cache. This explains the increase in L3 cache misses for transpose-k and transpose-ij shown in Figure 7(f). The effect of increased L3 cache misses on transpose-k and transpose-ij can also be seen in Figure 7(a) where the elapsed time increases considerably when m is increased from 2000 to 3000.

Instruction count. Since the transpose-k algorithm fixes one cell in $\mathrm{A}^\intercal$ at a time and multiplies that value with an entire row in B, the GCC compiler vectorizes the code in order to perform one scalar-to-vector multiplication utilizing AVX, meaning that it reduces the number of instructions executed by roughly a factor of 4 compared to transpose-ij as can be seen in Figure 7(b). Although `beast` is equipped with SIMD registers of 256 bits and thus is able to multiply 8 32-bit numbers at once, GCC cannot distinguish between CPUs with 256-bit SIMD registers and CPUs with 128-bit SIMD registers.

Branch mispredictions. Our non-recursive matrix multiplication algorithms compute mp inner products in some order, and we expect each inner product computation to use a loop of n iterations. We expect this loop to be easily predictable, meaning we expect mp branch predictions. When we plot the number of branch predictions divided by mp in Figure 7(d), we see that this is the case for naive, transpose-ij and transpose-k, which all incur roughly mp branch mispredictions. The recursive algorithm, however, uses branches

in order to decide which way to split the matrices and when to call the naive algorithm. To decide which order to split, we need to know which of m, n and p is largest, which is a difficult branching to predict. Because of this, the recursive algorithm incurs a large number of branch mispredictions.

It is in fact the combination of large number of instructions and branch mispredictions that makes the recursive algorithm lag behind transpose-ij. The machine `beast` has a relatively slow CPU clock, but a large cache, so again computational bottlenecks are more pronounced in our experiments.

4 Radix sort

In this section we provide a detailed comparative performance analysis of four different algorithms for sorting 32-bit integers: One implementation of standard radix sort, two variations of radix sort as presented by Wassenberg and Sanders [5], and introsort as implemented in GNU `std::sort`. We assume familiarity with radix sort.

4.1 Introduction

Introsort. In the C++ standard library, according to the specification, `std::sort` may be any comparison-based sorting algorithm that performs $\mathcal{O}(n \log n)$ comparisons in the worst case. In the GNU C++ compiler, the standard library implements a worst-case efficient variant of deterministic quicksort known as introsort.

In introsort, quicksort is run recursively by partitioning on the middle element in the input until $n \leq 16$ or the algorithm has recursed more than $2\lfloor \log_2(n) \rfloor$ times. If the recursion limit is reached, heap sort is used to sort the rest, which ensures $\mathcal{O}(n \log n)$ comparisons in the worst case, and after the quick sort recursion has completed, insertion sort is called to sort the remaining unsorted sublists of size ≤ 16 using $\leq 16n$ comparisons in total.

Since each cache line holds 16 integers, the unsorted sublists span at most 2 cache lines. Our input is pseudo-randomly sampled from a uniform distribution, so we do not expect the heap sort case to be called on any large sublists, and as such we expect in practice that this deterministic quick sort algorithm will behave as nicely as randomized quick sort.

Naive radix sort. This is an implementation of the radix sort algorithm as described by Cormen, Leiserson, Rivest, and Stein [2] in which $b = 32$ and $d = 4$. This means that the algorithm splits 32-bit integers into four *digits* of $k = \frac{b}{d} = 8$ bits each. Since our architecture is little-endian, we can treat each 32-bit integer in memory as an array of four 8-bit digits, where the least significant digit appears as the first byte. This simplifies the implementation since we do not need to perform bit shifting and masking to retrieve the digits in a word.

The algorithm starts by computing a histogram of the least significant digit (LSD) on the input of size N, that is, an array C of 2^k entries summing to N, such that $C[x]$ is the number of input elements for which the LSD is x. We then produce an array of offset indices, B, by summing up all buckets to the left of each bucket in the histogram, such that $B[x] = \sum_{y=0}^{x-1} C[y]$. Using this offset array, we can now distribute the input into an order sorted by the LSD, while

at the same time computing the histogram on the next digit. The algorithm proceeds in a similar manner on the next three digits until the input has been sorted. For each of the four digits, the distribution is performed sequentially from an input buffer of size N into 2^k output streams in an output buffer of size N, which means that the algorithm requires two large buffers of size N that alternate between the roles of input and output.

Virtual memory radix sort. Like the naive radix sort described above, the VM (virtual memory) radix sort algorithm proposed by Wassenberg and Sanders [5] splits each b-bit integer into d digits of $k = \frac{b}{d}$ bits. The algorithm starts by distributing the numbers into buckets by the most significant digit (MSD) so that the numbers are in sorted order when we only consider the MSD. After this distribution, each bucket can be sorted on the remainder of the digits in parallel using up to 2^k threads.

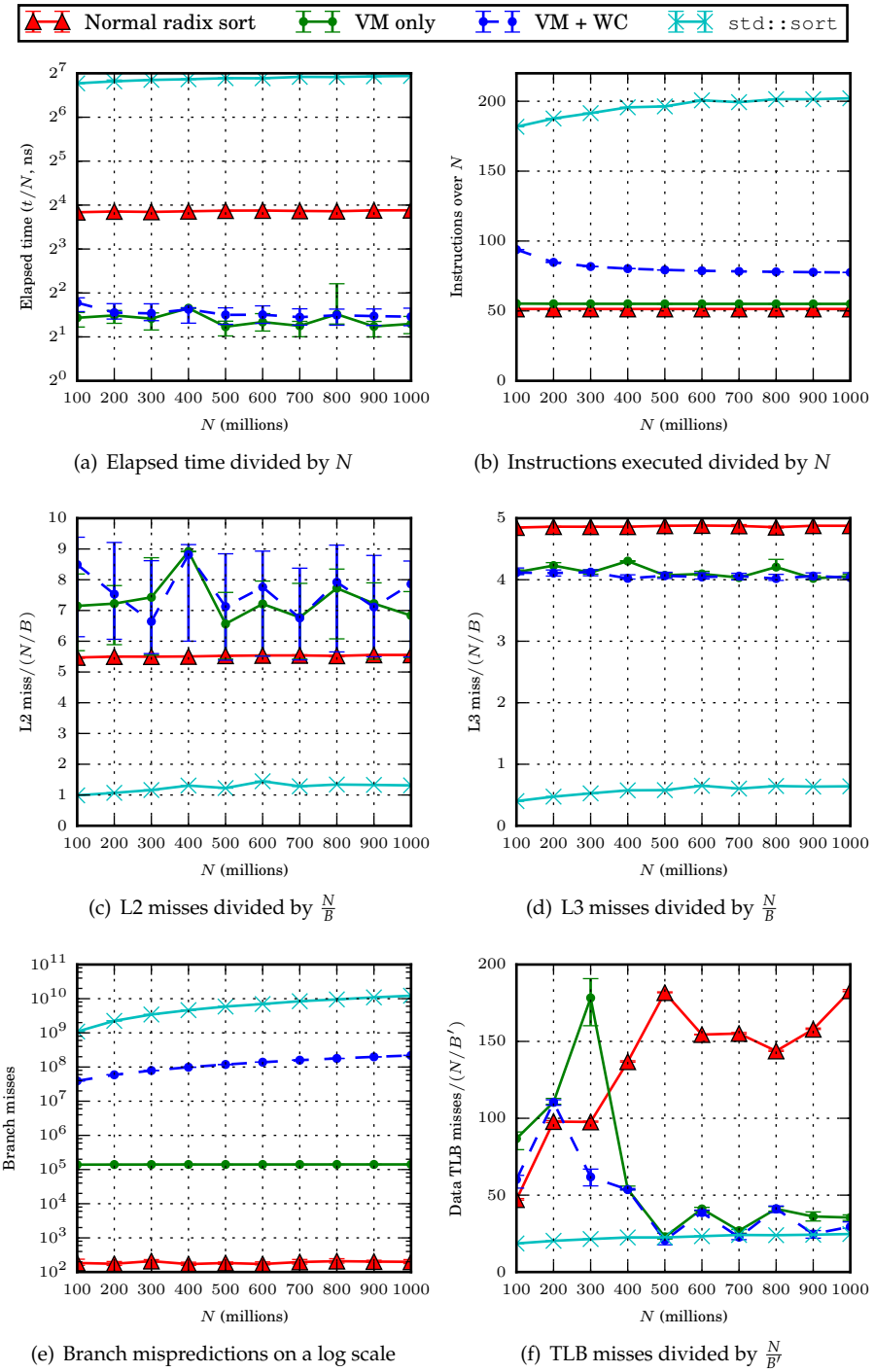
In the remainder of this section and in our implementation and experiments, without loss of generality, we assume that $b = 32$, $d = 4$ and $k = 8$ as with the naive radix sort we implemented. We split each 32-bit integer x into four digits $d_3d_2d_1d_0$ of 8 bits each, where d_3 is the most significant digit and d_0 is the least significant digit.

When distributing by the MSD, we do not know the sizes of the MSD buckets ahead of time. We know that each bucket holds at most N elements, so we round N up to the nearest multiple of the page size B' (4096 bytes) and allocate 2^kN cells of virtual memory to hold the buckets. Thanks to the virtual memory system of the OS, we do not require 2^kN cells of RAM, but only $\leq N + 2^kB'$ cells, which in practice means that we require up to 1 MB of extra memory. Furthermore, in order to parallelize the distribution by MSD, we split the input into P equally sized chunks and allocate 2^kN cells of virtual memory to each of the P threads, resulting in a practical overhead of physical memory of $P \cdot 1\text{ MB} \leq 16\text{ MB}$.

After distributing by the MSD, we produce offset indices based on the histogram of the MSD in the same manner as in naive radix sort, and then we handle each MSD bucket individually in parallel as follows. First, we distribute by d_0 into 2^k virtual memory d_0 buckets of capacity N. Since P threads distributed the input by MSD into each their own buffers in the first phase, we have P input sequences which we have to distribute by d_0.

Next, each of the 2^k d_0 buckets is distributed into 2^k different d_1 buckets. While we scan the input to perform the d_1 distribution, we compute a histogram of d_2, which in turn is used to generate output offset indices for d_2, as described above in naive radix sort. Finally we handle each d_1 bucket one after another, distributing each element into its correct position in the output, by combining the MSD offset indices and the d_2 offset indices.

To support the described approach we allocate a large amount of virtual memory (1 petabyte $= 2^{50}$ bytes) at the beginning of the algorithm, of which only the pages that are accessed gets mapped to actual memory. The algorithm described so far is referred to in all plots as *VM only*.



(a) Elapsed time divided by N

(b) Instructions executed divided by N

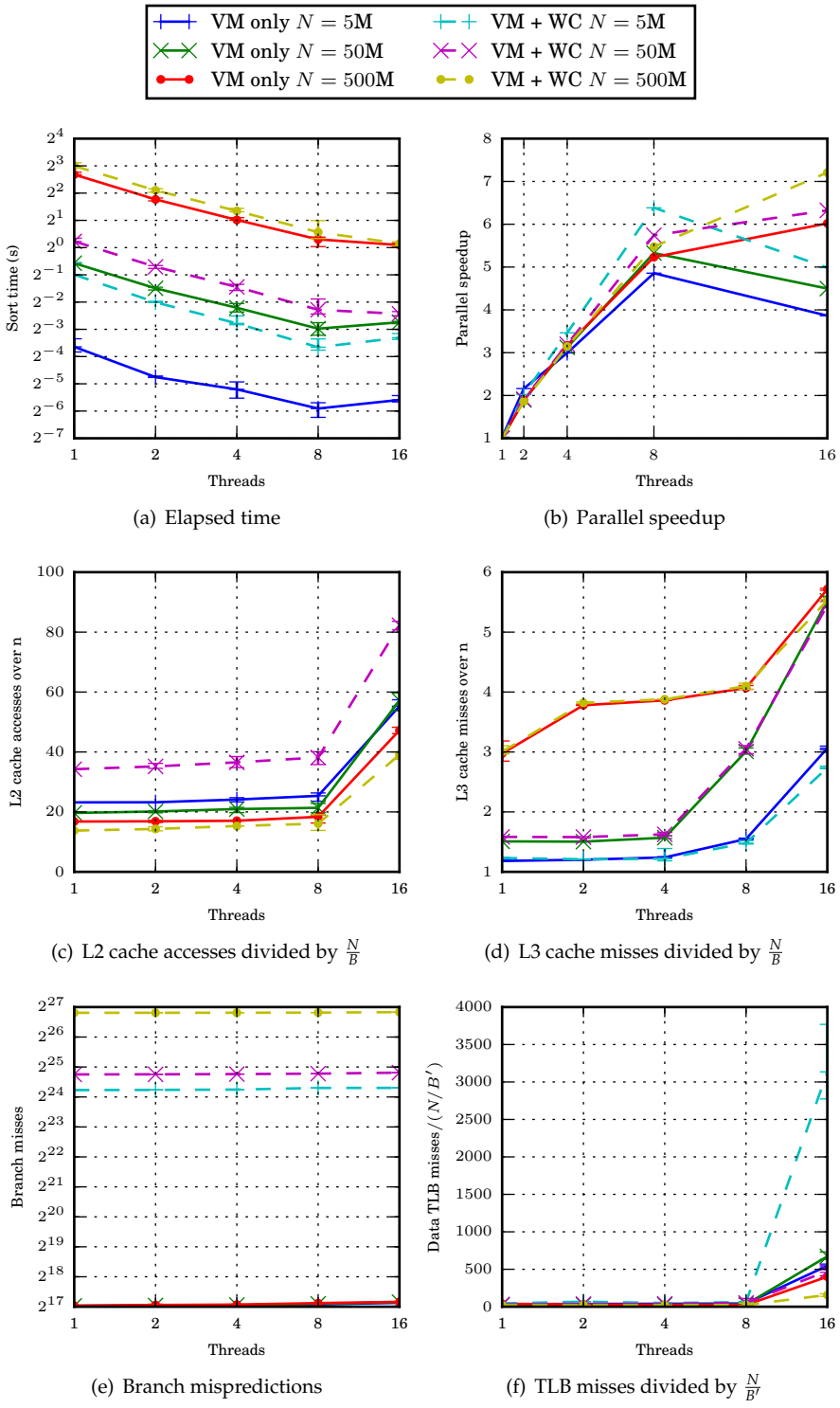
(c) L2 misses divided by $\frac{N}{B}$

(d) L3 misses divided by $\frac{N}{B}$

(e) Branch mispredictions on a log scale

(f) TLB misses divided by $\frac{N}{B'}$

Figure 8: Various plots showing the performance of normal radix sort, two variations of the VM radix sort with 16 threads, and `std::sort` (an implementation of introsort).



(a) Elapsed time

(b) Parallel speedup

(c) L2 cache accesses divided by $\frac{N}{B}$

(d) L3 cache misses divided by $\frac{N}{B}$

(e) Branch mispredictions

(f) TLB misses divided by $\frac{N}{B'}$

Figure 9: Various plots showing the performance of VM radix sort as we vary the number of threads on an 8-core CPU with hyper-threading.

The **VM + WC** (virtual memory with write combining) version of the algorithm adds software write combining (WC). Whenever the VM radix sort algorithm distributes elements into buckets, instead of writing to each output bucket directly, an array of $2^k B$ consecutive cells is used to store one cache line of the output per bucket, and when a buffer for a bucket is full, the entire cache line is copied to its position in the bucket.

Some instruction sets have support for *non-temporal writes*, meaning writes to memory locations that are not going to be read soon after the write. This is a hint to the memory cache that the destination should not be stored in caches, but that the cache line for the destination instead can be freely evicted without incurring performance penalties. Our implementation does not use special instructions for non-temporal writes, but instead we use `memcpy`, which simply copies the data using ordinary move instructions. Our initial investigations during implementation showed that even implementing write combining using plain `memcpy` led to a speedup over the VM only algorithm, even though it increases the number of instructions and memory cell writes. We thus leave it as future work to investigate thoroughly whether it is beneficial to use specialized instructions for non-temporal writes.

4.2 Experiments

To evaluate the performance of the different sorting algorithms, we perform two sets of experiments. First, we run all algorithms on data sizes $N = 100\text{M}, 200\text{M}, 300\text{M}, \ldots, 1000\text{M}$ to see how they scale with data size, using 8 threads (one thread per core) for the two parallel algorithms. Next, we only run the parallel algorithms, instead varying the the amount of threads used. We do this on three different dataset sizes, namely $N = 5\text{M}, 50\text{M}$ and 500M, to see the parallel speedup on data sets with different size. As mentioned in the previous section, we run these experiments with $w = 32$, $d = 8$ (four digits of 8 bits each). In the remainder of this section we perform a detailed analysis of the results obtained from these experiments.

Throughput. We first consider the running time of the algorithms expressed as time per input element in Figure 8(a). We observe that the radix variants are generally a factor 5 faster than `std::sort`, and VM radix sort is faster than normal radix sort.

`std::sort`. On our particular test case of 32-bit integers generated by a PRNG, `std::sort` is by far the slowest algorithm. It generally has quite good cache and TLB performance as seen in Figures 8(c), 8(d) and 8(f), but it performs up to 4 times as many instructions as seen in Figure 8(b) and several orders of magnitude more branch mispredictions as seen in Figure 8(e). This is unsurprising, since `std::sort` is the only comparison-based sorting algorithm in our experiments, so we would naturally expect it to execute more instructions and see many more branch misses.

Hyper-threading and TLB misses. Next we consider the effect of hyper-threading in the experiments. Considering Figure 9(f), it is interesting to see that when we go from 8 to 16 threads, or 2 threads per core, the TLB miss rate jumps dramatically by a factor 17 (VM only) and by a factor 6.7 (VM + WC) on the largest data set, $N = 500\text{M}$.

Without write combining, we require $2^k = 256$ entries in the TLB per thread to avoid TLB misses during output. Since the L2 TLB has only 512 entries, we see a dramatic increase in the number of TLB misses as seen in Figure 9(f) when going from 8 threads to 16 threads, meaning that the 2 threads per core are fighting for the same 512 entries in the L2 TLB. In the worst case, this could lead to a TLB miss for every output element instead of every $B' = 1024$ output elements, but in the experiment, we only see a factor 17 increase. The L2 TLB evictions mean that the page table needs to be consulted more often, in turn leading to many more cache misses as seen in Figures 9(c) and 9(d).

With write combining, we require $2^k B$ cells (where $B = 16$) of output buffers, which corresponds to $4B'$, or rather, four pages in the TLB, but every time we have written B elements to an output buffer, we copy it to the page in which it belongs. With hyper-threading, we do not have room in the TLB for all the destination pages, so instead of incurring a TLB fault every B' elements as we would optimally, we incur a TLB fault every B elements, or 64 times more often.

Write combining. When we add write combining to VM radix sort, the number of cell writes roughly doubles, since the data that would previously be written directly to buckets is now being temporarily written to a buffer. This also explains the increase in number of instructions caused by write combining seen in Figure 8(b).

When we implemented VM radix sort on our office workstations running fast 3.4 GHz CPUs, we saw that the immense increase in number of instructions added by write combining did not make computation the bottleneck, and VM + WC was significantly faster than VM only.

Similarly, the experiments of Wassenberg and Sanders [5] run on a dual Intel W5580 system. Each CPU has 4 cores of 3.2 GHz and only 8 MB of L3 cache shared among the 4 cores. With the engineering effort of the authors, write combining makes a huge difference on such a system.

However, on `beast`, we see in Figure 9(a) that write combining performs consistently worse than VM only, because the CPU is slower than the CPU used by Wassenberg and Sanders [5] (2.0 GHz), although it has more cores and a larger cache. On `beast`, the increased number of instructions and branch mispredictions together make write combining a bad deal.

Cache readahead/prefetching. Our implementation of normal radix sort touches $9N$ contiguous memory cells: Reading the input to compute the histogram on the first digit (N reads), and then running counting sort on each of the four digits (N reads and N writes) while at the same time computing the histogram of the next digit (included in the N reads). It requires $2^k = 256$ cells to store a histogram and another $2^k = 256$ cells to store a vector of current output indices for each bucket. However, Figure 8(c) shows that only $\approx 5.5\frac{N}{B}$ cache line operations incur an L2 fault in normal radix sort, which is much less than the $9\frac{N}{B}$ faults predicted by the analysis above. From this, we predict that the CPU in a non-deterministic manner prefetches the data we need into the L2 cache, decreasing the number of faults we actually measure. This would also explain the great fluctuations we see in the number of L2 faults for the VM radix sort algorithms.

Bibliography

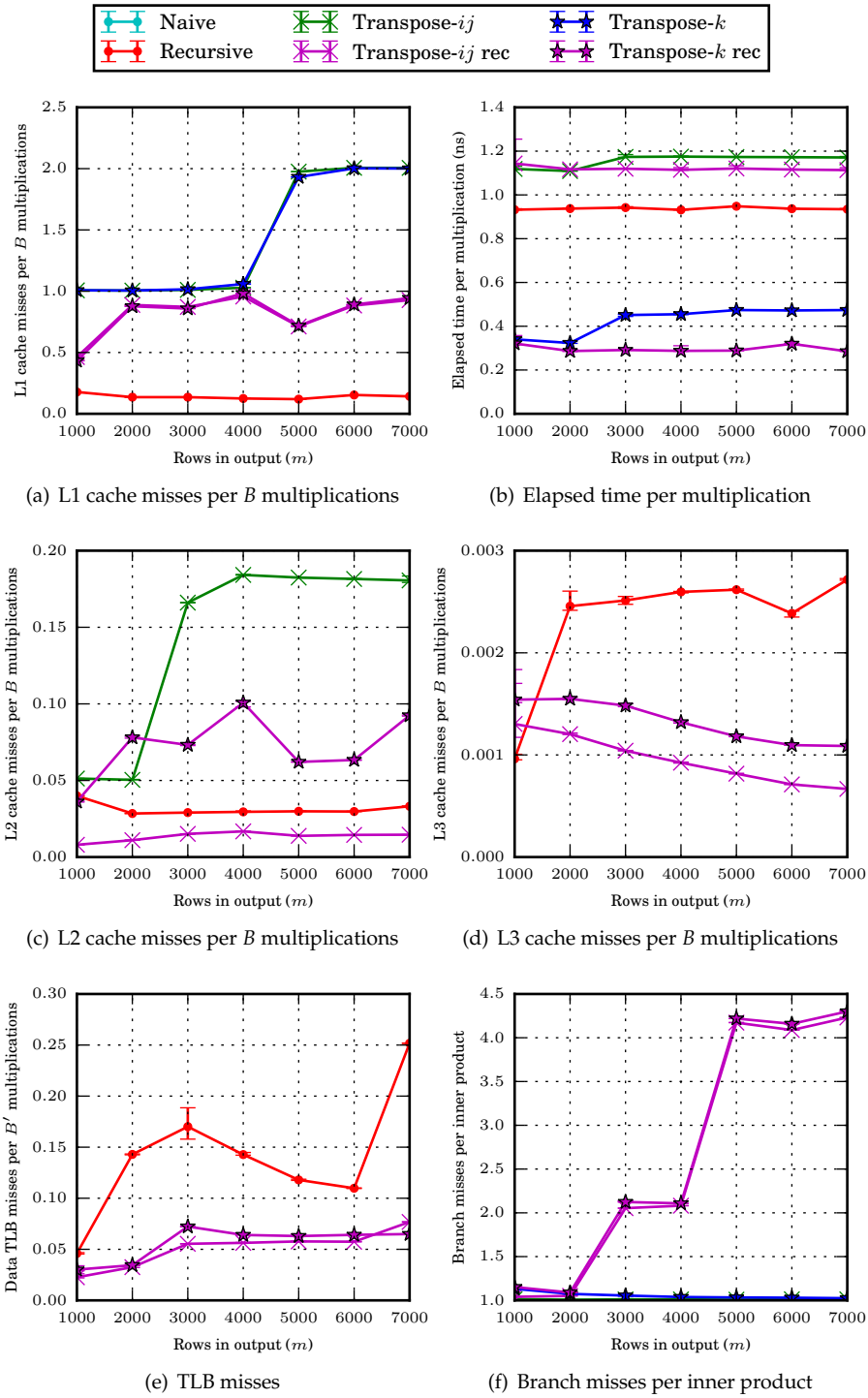
[1] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 39–48, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. ISBN 0-89871-513-X. URL `http://dl.acm.org/citation.cfm?id=545381.545386`.

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms third edition, 2009.

[3] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. *ACM Transactions on Algorithms*, 8(1): 1–22, January 2012. ISSN 15496325. doi: 10.1145/2071379.2071383. URL `http://dl.acm.org/citation.cfm?id=2071379.2071383`.

[4] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1): 3–30, 1998.

[5] Jan Wassenberg and Peter Sanders. *Engineering a Multi-core Radix Sort*, volume 6853 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-23396-8. doi: 10.1007/978-3-642-23397-5. URL `http://link.springer.com/10.1007/978-3-642-23397-5`.



(a) L1 cache misses per B multiplications

(b) Elapsed time per multiplication

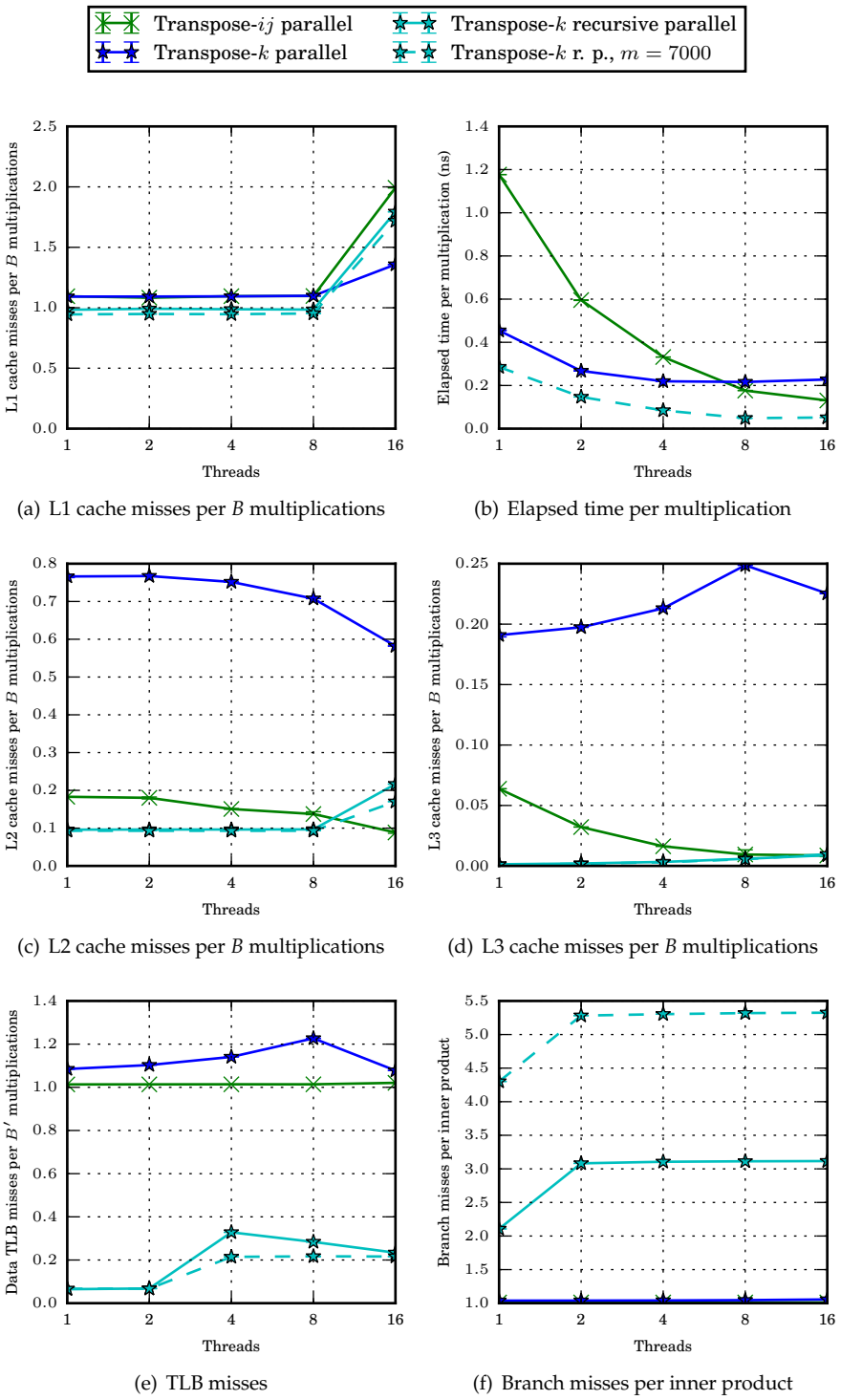
(c) L2 cache misses per B multiplications

(d) L3 cache misses per B multiplications

(e) TLB misses

(f) Branch misses per inner product

Figure 10: Recursive variants of transpose-ij and transpose-k implemented after the report deadline. Only 5 or 20 measurements per data point as opposed to 10 or 40.



(a) L1 cache misses per B multiplications

(b) Elapsed time per multiplication

(c) L2 cache misses per B multiplications

(d) L3 cache misses per B multiplications

(e) TLB misses

(f) Branch misses per inner product

Figure 11: Parallel versions of the algorithms transpose-ij, transpose-k and transpose-k recursive, implemented after the report deadline, at $m = 4000$ unless otherwise noted. Only 5 or 20 measurements per data point as opposed to 10 or 40.