3 fundamental problems in algorithm engineering

Benjamin Blankholm

E-mail: benner@cs.au.dk

Study nr. 201206087

April 4, 2016

List of Figures

1   Introduction

In this report 3 projects will be described, and analyzed. The projects are looking into optimizing algorithms, for search, matrix multiplication, and sorting. These three fields of algorithm engineering, are fundamental problems with wide spread use. This report looks into the three projects with a goal of improving the simpler and more theoretical algorithms, with respect to running time on real computers. This involves reasoning about the hardware architecture, and taking into account the evolution of modern systems.

2 Binary Search

In this project we have tried to optimize a regular binary search algorithm, by changing the split point and by changing the memory layout of the data searched. The project seeks to find a suitable data structure to support the *pred* operation.

$$Pred(x) = return \max\{y \in S | y \leq x\}$$

2.1 Modern hardware

It is a well known fact, that using asymmetrical splits in algorithms in some cases result in better performance. [1, 4] This is usually caused by the hardware architecture of modern computers. In modern CPUs long computation pipelines is used to compute multiple instructions in sequence. This means, that often it will be necessary for the CPU to try and predict which branch of instructions the execution of the program will take, and stick it in the computation pipeline. If this prediction turns out to be wrong, the computations of the instructions in the wrong branch is effectively wasted work. This is known as a branch misprediction. It turns out, that in some cases it can be beneficial to make these branching statements easier to predict, at the cost of having more comparisons [3]. This can be done by making the branching statement take the one path of execution more often than the other. Exactly how much bias is needed toward one of the branches, is not immediately clear.

An other known factor to influence running time, is cache and Translation Lookaside Buffer (TLB) misses. Because it is possible to make smaller memory sizes work faster, the memory architecture of modern computers is a hierarchy. This hierarchy contains in most desktop computers three layers of cache, Random Access Memory (RAM), and disk storage. Here the three layers of cache is known as L1, L2 and L3. Multicore CPUs are often structured such that each individual core has its own dedicated L1, and L2 caches, whereas the L3 cache is shared between the cores. Here the L1, is the smallest level of cache, but the fastest, whereas L3 is largest and slowest of the three L2 is somewhere in between. RAM is however a lot larger, than the small caches, so if the program running is using data that is too large to fit in the cache, some replacement of values in the caches must be performed during the execution of the program. When a program needs some resource stored in memory, it will look for it in the cache. If it is not present in the cache a cache miss occurs. This means that the resource must be found in a higher level of cache, og in main memory. If the resource is only located in main memory, a lookup in the TLB will be performed to get the address in physical memory.[1] If the address is present in the TLB a TLB hit occurs, and the physical address can be looked up almost instantly.

---

[1] In computer architecture one operates with mainly two kinds of memory, virtual and physical memory. Virtual memory, is used to hide the implications of having physical addresses. This means, that it seems like to the individual application, that it has the entire address space available. It also means, that a virtual address (the addresses the applications use) must be mapped to a physical address such that it can actually be stored.

If however the address is not pressent int the TLB a TLB miss occurs, and this causes a page walk, which requires several memory lookups [6, p. 450-452]. The TLB and the caches are ordered in such a way, that the resources are stored in sequential blocks. In the cache the data is loaded in cache lines, and the TLB is working with pages. This means that it is beneficial to access the resources used in a sequential manner whenever possible, because it will most often cause less pagewalks, and cache misses.

2.2 Improvements of binary search

In order to meet the complications that arises from modern hardware architecture discussed in section 2.1, some specific design choices has been made. One possible improvement is to try and limit the amount of branch mispredictions to gain better performance from less waisted computing time. This is done by trying to skew the split of the search space, in order to make it more plausible for the binary search algorithm to find the result in the one side rather than the other. If the algorithm is searching for a value on a randomly uniform data structure and the algorithm is splitting the data structure in half, one would expect that there is a 50/50 chance of finding the value in either of the two partitions. This would result in a large amount of branch mispredictions, and thus it might be better to make an other partitioning. On Figure 1 two different splits of an array is shown. In the tests of the algorithm a *skew* is used, to determine the split-point for the algorithm. This *skew* is a number between 0 and 1, where 0.5 would result in a 50/50 split, and 1 or 0 would result in the entire data structure being on the one side of the split, such that the binary search effectively would become a linear search.
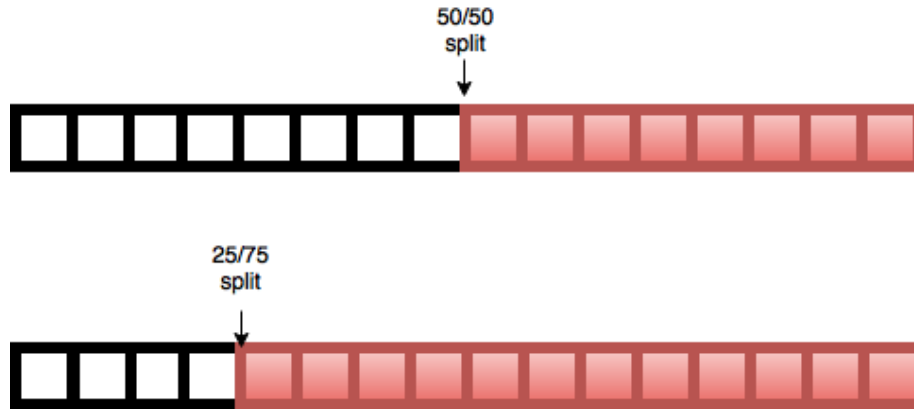

Figure 1: An array split 50/50, compared to an array split 25/75

Besides skewing the partitioning of the data structure, the data is stored as a Binary Search Tree (BST)[2, p. 286]. The simplest way of storing the data for a binary search would be using a sorted array. However this approach would

result in a large amount of cache misses and pagewalks, because the read from memory would be nothing like sequential. The recursive algorithm would take logarithmic steps in memory.

As opposed to the sorted array, three different memory layouts of a BST has been studied. These being a Breadth First Search (BFS) manor, a Depth First Search left (DFSl) manor, and a Depth First Search right (DFSr) manor. Figure 2 shows how the different layouts of BSTs are structured in memory. These layouts were chosen because of their simplicity, and their properties with respect to memory addresses when skewing the data structure. The hope for the DFSl layout is that it will perform well when the data structure is skewed to the left. The combination of high probability of visiting the left child of a node, and the fact that the left child is the next memory address, could potentially mean that there will be both less branch mispredictions and less cache misses and pagewalks. The similar mirrored argument holds for DFSr.

(a) BFS layout

(b) DFSl layout   (c) DFSr layout
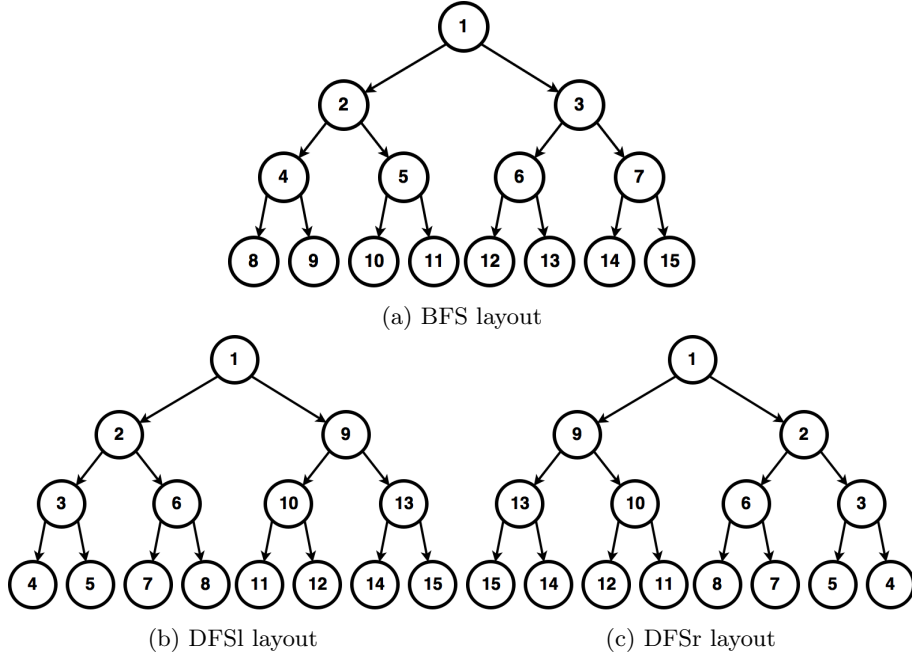
Figure 2: The BSTs with numbers representing the relative positioning of the nodes in memory, for each of the three types of layout.

2.3   Implementation

Because the tests performed are carried out on BSTs the binary search implementation looks as in Figure 3. The implementation takes the Node (which is shown in Figure 4) from the first index in the array. This is defined to be the root of the tree in all of the three memory layouts. And by the definition of the

BST this would be the first node to be checked by the binary search algorithm. From the root the algorithm simply follows the pointers according toward the better matching value, until it finds the best match for the search.

```cpp
int binary_search_on_bst(int s, Node array[]) {
  Node current_node = array[0];

  while(current_node.val != s){
    if(current_node.val > s && current_node.left != NULL) {
      current_node = *current_node.left;
    } else if(current_node.val < s
              && current_node.right != NULL
               && current_node.right->val < s) {
      current_node = *current_node.right;
    } else {
      break;
    }
  }

  return current_node.val;
};
```

Figure 3: Implementation of binary search on a BST in C++

```cpp
struct Node{
  int val;
  Node *left;
  Node *right;

  Node(){}

  Node(int v, Node *l, Node *r){
    this->val = v;
    this->left = l;
    this->right = r;
  }
};
```

Figure 4: The C++ representation of a BST node

To construct the skewed BST the algorithm in Figure 5 is used. This algorithm constructs the nodes and pointers, in such a way that the resulting BST becomes skewed with the provided fraction.

6

```
1  Node∗ build_BST(int in[], int L, int H, double skew){
2    if(L>=H) return NULL;
3
4    int i = L + floor((H−L)∗skew));
5
6    if(i == L){
7      return new Node(in[i],
8             NULL,
9             build_BST(in, L+1, H, skew));
10   }
11   if(i == H){
12     return new Node(in[i],
13             build_BST(in, L, H−1, skew),
14             NULL);
15   }
16
17   return new Node(in[i],
18           build_BST(in, L, i, skew),
19           build_BST(in, i+1, H, skew));
20 };
```

Figure 5: An algorithm for building a skewed BST from a sorted array

When the BST is constructed it can be placed in memory according to the layouts described in section 2.2. This is done by three different functions. (one for each layout) The algorithms are recursively laying out the nodes, ensuring the correct placement in memory. This is done by constructing an array of nodes of the same length as the size of the BST. The nodes from the original BST are then copied into the new array in such a way, that they will fulfill the constraints of the desired layout.The algorithm for laying out the BST in a DFSl manor can be found in Figure 6.

```
1  int tree_to_dfsl(int index, Node* current_node){
2    dfsl_array[index].val = current_node->val;
3
4    int next_index = index+1;
5    if(current_node->left != NULL){
6      dfsl_array[index].left = &dfsl_array[next_index];
7      next_index = tree_to_dfsl(next_index, current_node->left);
8    }
9
10   if (current_node->right != NULL) {
11     dfsl_array[index].right = &dfsl_array[next_index];
12     next_index = tree_to_dfsl(next_index, current_node->right);
13   }
14
15   return next_index;
16 };
```

Figure 6: The algorithm for laying out a BST in DFSl in memory

2.3.1 Implementation challenges

During the implementation of the algorithms for skewing and laying out the trees several challenges were found. The major problem was that the most obvious way to lay out a BST in a BFS manor, would not work for very skewed instances, because there would be large holes in the array, and the program would quickly run out of memory addresses[2]. To solve this problem, and give the BFS layout a fair chance in the tests, an other way of laying out the BST was designed. For this a new node type was used, which besides the two pointers and the value could also hold an index, which would represent the array index of the node. With this type of node it is possible to first build a BST from the new heavy nodes, and then run through the nodes to assign them indexes. When this is done the tree of heavy nodes is copied into an array of regular nodes, such that the extra index value in the struct would not cause any inconsistencies in the tests. This copying also makes sure, that each node will be placed in the array in accordance with its assigned index. The algorithm for assigning the indexes to the nodes can be found in Figure 7. This algorithm ensures that there will be no holes in the array, so all of the layouts would take up an equal amount of space in memory.

---

[2]In the straight forward BFS layout of a BST, the left child of a node would be found at the address of the node itself multiplied by two, and the right child would be found on the same address as the left plus one.

```
1  void bfs_index_heavy_tree(HeavyNode * root){
2    int index = 0;
3
4    queue<HeavyNode*> queue;
5    queue.push(root);
6    while(!queue.empty()){
7      HeavyNode * current_node = queue.front();
8      queue.pop();
9
10     current_node->index = index;
11
12     if(current_node->left != NULL){
13       queue.push(current_node->left);
14     }
15
16     if(current_node->right != NULL){
17       queue.push(current_node->right);
18     }
19     index++;
20   }
21   return;
22 };
```

Figure 7: The algorithm for assigning indexes to nodes in a BFS manor

2.4   Performed tests

All the tests in this project has been done on a MacBook Pro (15", ultimo 2011), the hardware specifications are as follows:

- CPU: Intel Core i7-2760QM (2,4 GHz)

  4 cores

  L1 cache: 32KB instruction/32kB data (per core)

  L2 cache: 256KB (per core)

  L3 cache: 6MB (shared)

- Memory: 8GB DDR3 RAM (1333 MHz)

Skews between 0.05 and 0.95 (0.05 size steps) has been tested using the different memory layouts. During the tests five different hardware counters has been monitored. These being the number of branch mispredictions, the number of times the hardware prefetchers causes a cache miss in L2 cache, the number of request for ownership (RFO) that misses L2 cache, the number of requests missing L3, and the number of misses in TLB that causes a pagewalk. Besides these hardware counters, the running time of the algorithm using the different skews has been measured as well. The algorithm has been tested on arrays of size 1,000,000 and 10,000,000 (all plots presented in the report is from tests run on BST size 1,000,000). The hope is to show a larger difference in TLB misses

for high and low *skews* using the larger dataset. During each test the *pred* query is run as many times as there is indices in the data structure. This means, that if the data structure contains 10,000,000 nodes, 10,000,000 lookups for random numbers will be performed.

From the tests one could expect to see that the curves for the running time would increase when gets closer to 0.5, and again when it is nearing the outer edges (0 and 1) where the algorithm will start to perform more like a linear search. Where usually binary search has a time complexity of $O(\log n)$ linear search has a time complexity of $O(n)$, which is the reason for the expected increase in running time toward the edges. On the other hand the findings from Brodal and Moruz [1] indicates that the running time could increase where a lot of branch mispredictions are likely to occur. With this put together one could expect to see a peak around a *skew* of 0.5, and again toward the extreme *skews* to either side. It is therefor expected to find the best performance where a slightly unbalanced BST is used. An other factor which should be taken into account is that the depth first layouts combined with a skew could result in a further increase in performance, because less cache faults and TLB misses are likely to occur. The reason is, that with a high skew (in the range $]0.5; 1]$) the majority of the nodes will be in the left subtree, and with a DFSl layout, they will furthermore be placed on sequential addresses in memory. Because of these factors a *skew* in the range $]0.5; 1]$ and a DFSl layout is expected to result in fewer branch mispredictions, and fewer cache faults which in turn could result in better performance.

2.5   Results

As expected the number of branch mispredictions, when running the tests for different *skews* decreases with a more skewed data structure. Figure 8 shows the trend of branch mispredictions decreasing as the skew moves away from 0.5. Very similar results were found for both DFSl and DFSr, which is as expected since the branching would not change because of changes in memory addresses for the nodes.
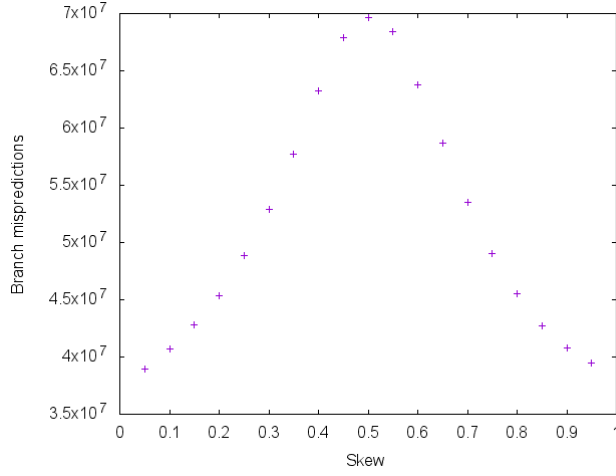
Figure 8: The number of branch mispredictions using BFS layout with different *skew*

From Figure 9 it can be seen that the number of TLB misses increases when the data structure is less balanced. A notable increase starts when the skew passes 0.2 and 0.8. Because a pagewalk is an expensive operation compared to clock cycles or cache requests, it may therefor be expected that the running time would also increase a fair amount when the skew passes these points. Likewise a similar trend can be seen for L2 cache misses in Figure 10.
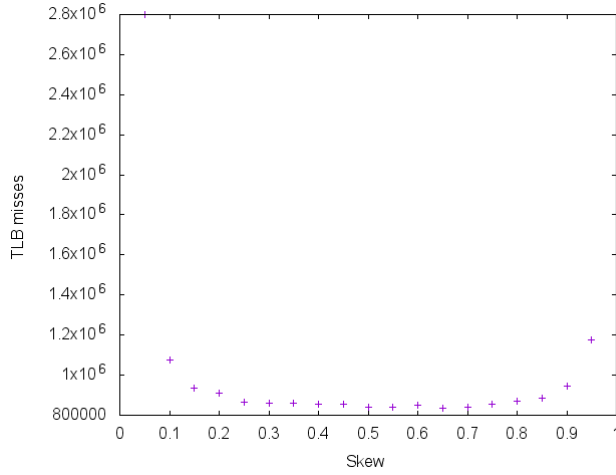

Figure 9: The number of TLB misses that causes a pagewalk when using BFS layout
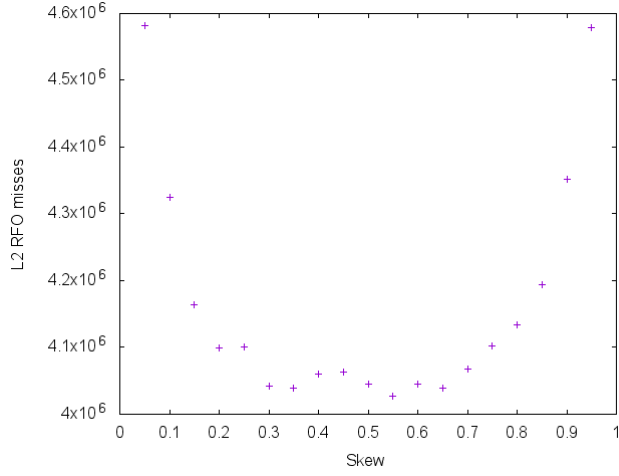
Figure 10: The number of cache faults caused by RFO requests, when using BFS layout

A quite different pattern can be seen in the counts for the L3 cache misses in Figure 11. Here the number of cache misses starts to decrease when the skew goes toward the two extremes (When the skew goes below 0.25 or when it exceeds 0.75). The reason for this could be that the data structure gets skewed enough for the sought after nodes to be placed close enough to each other in memory for the addresses to fit in the L3 cache. This would happen in cases where the tree is so unbalanced that several child nodes in sequence would have only one child. This would mean that the size of the layers in the tree would not increase all the way to the bottom layer, but rather at some point the size of the layers would start to decrease when moving down through the tree. This could result in an entire subtree being able to fit in a cache line of the L3 cache, and thus fewer misses would occur when moving through this sub tree.
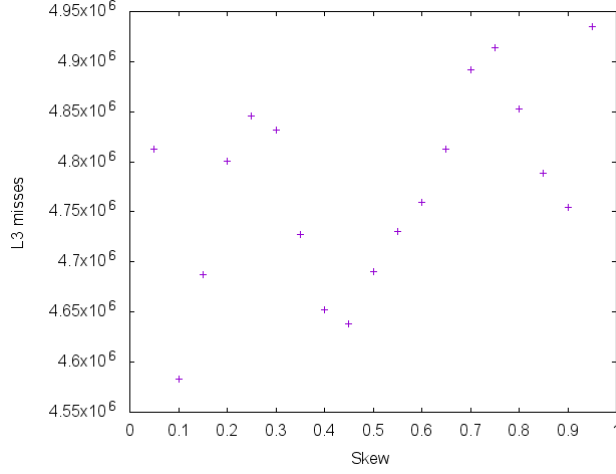
12

Figure 11: The number of L3 cache misses, when using BFS layout

In Figure 12 all the tests for all three memory layouts are put together. It is clear that the depth first memory layouts performs better in terms of cache misses and TLB misses, as expected. In all of the plots the BFS layout is causing around 20% more misses than the DFSl and DFSr layouts. It may also be seen that the plots for the L3 cache misses for DFSl and DFSr is more uniform than BFS (Figure 11 discussed in the previous paragraph). This can be explained from the fact that if the data structure is less balanced, there will be a larger chance of the node placed sequentially in memory being the next visited by the algorithm (when using DFS). And because memory is used in blocks, either as cache lines, or as pages, accessing the memory sequentially will in all likelihood mean that the next resource used is already loaded.

However when this all comes together, it is not immediately evident, that the depth first layouts actually perform better than BFS when looking at the running time. The curves found in Figure 13 seem to agree largely with the curvatures of the plots of the different cache misses and TLB misses. But the increase in cache misses from the depth first layouts to the BFS layout, does not seem to be enough of a change to make a notable difference on the running time. It could be however that the branch mispredictions is more of a dominating factor, which might explain why the running times around a *skew* of 0.5 is a bit larger than the running times for less balanced trees.
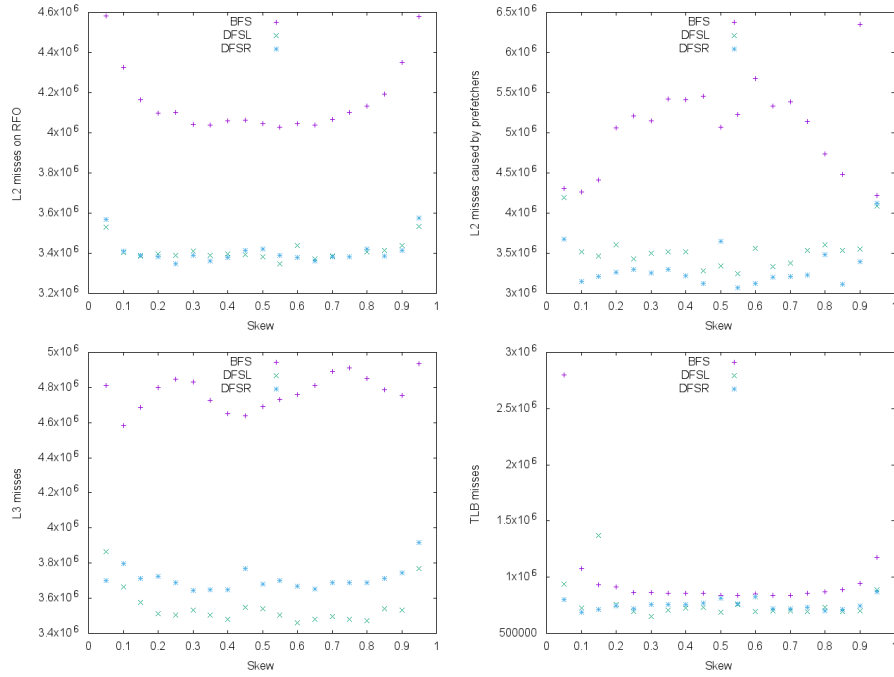
Figure 12: Combined plots of cache misses and TLB misses for the three different memory layouts
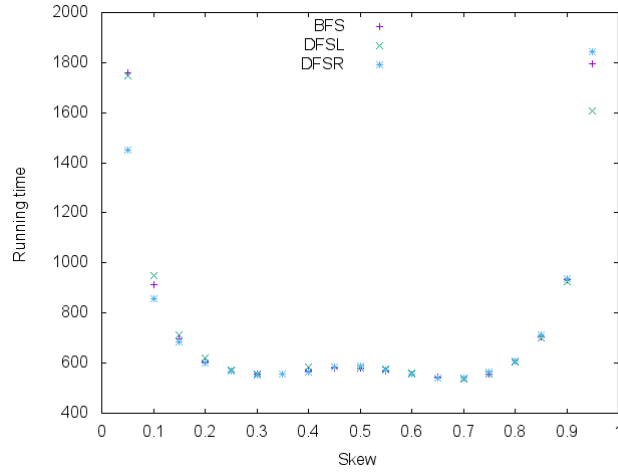

Figure 13: Combined plots of running times for the three different memory layouts

From Figure 13 it looks like a *skew* of 0.7 is preferable so a test of different

14

data sizes has been made to compare the running times using this *skew*. The result of this test can be found in Figure 14. In this figure it is again evident, that the different layouts does not seem to make much of a difference to the running time of the algorithm. One could expect that the DFSl layout would be superior because the BST would be unbalanced to the left when using a *skew* of 0.7. However et seems that if anything DFSr is actually performing a tiny bit better.
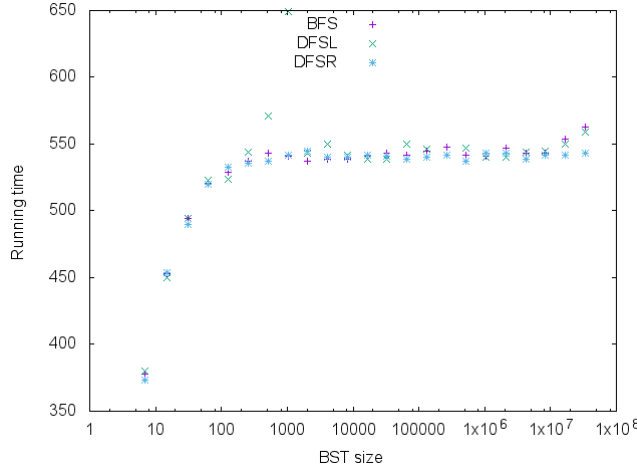

Figure 14: Running time for increasing sizes of the BST, using the three different memory layouts.

2.6 Conclusion

In this project it has been shown that using BSTs, the memory layout of the tree has a rather large impact on the amount of cache misses as well as TLB misses. BFS, DFSl and DFSr has been studied, and the depth first layouts has shown to cause a notable amount less cache misses and TLB misses. Along with branch mispredictions decreasing, with unbalancing the BST, it was expected that the DFS layouts would perform better than the BFS layout, but sadly the tests performed failed to show a notable difference in running time.

It may be that larger data sizes would need to be used in order to see a performance gain. If larger data sizes had been used more TLB misses would be expectable, and this may cause a notable difference to the running time. Unfortunately we have not been able to test this hypothesis because of lack of time.

2.7 How To run the code

The main code for this project can be found in `binary_search_BST.cpp`. The main function of the program takes as arguments the desired size of the BST

to be generated and tested on, as well as the desired *skew*. If the desired size of the BST is chosen to be $<= 0$ the program will do a test run, comparing the running time on the three different memory layouts on increasing sizes of BSTs.

3 Matrix Multiplication

In this project, we have tried to create an efficient matrix multiplication algorithm. We have used two approaches: traditional algorithm theory and cache-behaviour. We have used the article Cache-Oblivious Algorithms[5] as inspiration. We initially implemented very simple iterative algorithm without any considerations about cache and from there we used the approaches to change and improve the algorithm.

3.1 Improvements to matrix multiplication

The initial algorithm took two-dimensional integer arrays laid out intuitively in memory and calculated each cell for the result matrix. When storing a matrix, one can do it in a number of different memory layouts as seen in figure 15. The intuitive layout is row major for both factor matrices. This will result in a lot of cache hits for the first matrix, since it will load in the consecutive cells in each row needed to calculate a cell, but will result in a lot of cache misses for the second matrix, since that needs consecutive cells from each column. The first improvement, we made, was to change the layout of the second matrix to column major. This resulted in a big improvement, especially when multiplying large dimension matrices to other large dimension matrices. We also tried to improve a divide-and-conquer algorithm. The algorithm works by partitioning the matrices by the largest dimension and then doing the simple scalar function when it reaches 1 as the base case. This was a done with a row-column major layout as well, which should still work better than row-row, due to the way the partitions work. The result was a slow, but working algorithm. To improve this, we combined this with the iterative algorithm, where the base case would happen earlier and the multiply the using the iterative algorithm. This is in a way cache-oblivious, since we don't use any kind of explicit tuning, but how we define the base case condition could be seen as tuning. The algorithm could easily be expanded to be cache-aware by defining the base case condition depending on some argument. The general idea behind combining the two algorithms are to exploit the cache by using the iterative algorithm, when an entire row or column can be fit in a cache line.
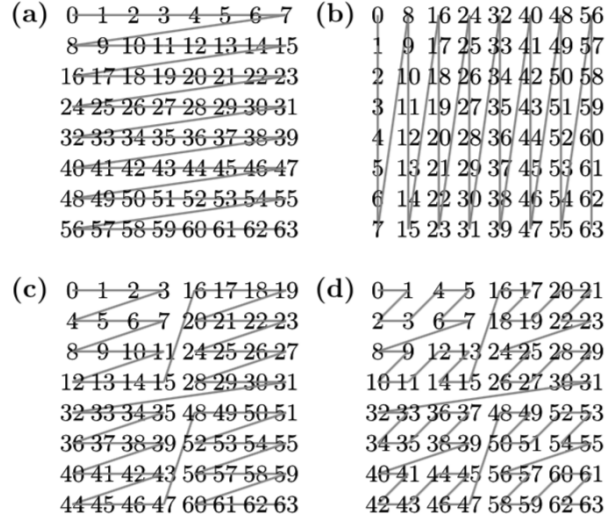
Figure 15: Layout of a 16x16 matrix in (a) row major, (b) column major, (c) 4x4-tiled, and (d) bit-interleaved layouts. Figure and caption borrowed from Frigo et al.[5, p. 3]

3.2 Implementation

Figure 16 shows the implementation of the row-column major layout iterative algorithm. It takes the two matrices as parameters. Then it sets up the result matrix by creating an array of MatrixARows pointers, which represents each row in the result matrix, and in each entry, it points to an integer pointer array, the size of MatrixBColumns. Then it runs through the row of matrix a and the column of matrix b and sums the products of the corresponding cells, which will be stored in the result matrix cell. Due to the column major layout of matrix b, the first dereference will give the respective column instead of the respective row, which is the case for matrix a. This is why the shared dimension (MatrixAColumns==MatrixBRows), i, is the second dereference for both matrices.

```
1  int** matrix_mult(int** ma, int** mb) {
2    int** res = new int*[MatrixARows];
3    for (int n=0; n<MatrixARows;n++) {
4      res[n] = new int[MatrixBColumns];
5      for (int j=0; j<MatrixBColumns; j++) {
6        int rescell = 0;
7        for (int i=0; i<MatrixAColumns; i++) {
8          rescell += ma[n][i]*mb[j][i];
9        }
10        res[n][j] = rescell;
11      }
12    }
13    return res;
14 }
```

Figure 16: The row-column major layout iterative algorithm for multiplying matrices

Figure 17 shows the implementation of the row-column major layout divide-and-conquer algorithm. It takes the result matrix mc as a parameter as well. It also takes the 'dimensions' of each of the three matrices as well. beg and end defines the partition of each matrix. ma is n x m, mb is m x p and mc is n x p. The first if-statement is the base case, which in this particular case is when each dimension of the partitions is less than 1000. matrix_mult is the iterative algorithm, but expanded with the dimension limits. The 3 else if's finds the largest dimension and partitions the matrices accordingly. The 1000 in the base case is one of the thresholds we tested. Others were 10 and 10000, so there's definitely room for more testing. Furthermore it could be changed to be defined by a parameter. We didn't really test 10000 though, since the programs started behaving strange with that large matrices.

```
1  int** rec_matrix_mult(int** ma, int** mb, int** mc, int nbeg, int↩
       nend, int mbeg, int mend, int pbeg, int pend) {
2    if (nend−nbeg < 1000 && mend−mbeg < 1000 && pend−pbeg < 1000) {
3      matrix_mult(ma, mb, mc, nbeg, nend, mbeg, mend, pbeg, pend);
4    } else if (nend−nbeg >= mend−mbeg && nend−nbeg >= pend−pbeg) {
5      int nmid = nbeg+(nend−nbeg)/2;
6      rec_matrix_mult(ma, mb, mc, nbeg, nmid, mbeg, mend, pbeg, ↩
           pend);
7      rec_matrix_mult(ma, mb, mc, nmid, nend, mbeg, mend, pbeg, ↩
           pend);
8    } else if (pend−pbeg >= mend−mbeg && pend−pbeg >= nend−nbeg) {
9      int pmid = pbeg+(pend−pbeg)/2;
10
11     rec_matrix_mult(ma, mb, mc, nbeg, nend, mbeg, mend, pbeg, ↩
           pmid);
12     rec_matrix_mult(ma, mb, mc, nbeg, nend, mbeg, mend, pmid, ↩
           pend);
13   } else if (mend−mbeg >= nend−nbeg && mend−mbeg >= pend−pbeg) {
14     int mmid = mbeg+(mend−mbeg)/2;
15
16     rec_matrix_mult(ma, mb, mc, nbeg, nend, mbeg, mmid, pbeg, ↩
           pend);
17     rec_matrix_mult(ma, mb, mc, nbeg, nend, mmid, mend, pbeg, ↩
           pend);
18   }
19   return mc;
20 }
```

Figure 17: The row-column major layout divide-and-conquer algorithm for multiplying matrices

3.3 Performed tests

3.4 Further improvements

3.5 How To run the code

4 Improving Javas Quick Sort

4.1 Performed tests

4.2 How To run the code

5 Bibliography

[1] Brodal, G. S. and Moruz, G. (2006). Skewed binary search trees. *In Proc. 14th Annual European Symposium on Algorithms (ESA), Lecture Notes in Computer Science*, 4168:708–719.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms.* MIT Press, Cambridge (Mass.).

[3] Gerth Stølting Brodal, G. M. (2005). Tradeoffs between branch mispredictions and comparisons for sorting algorithms. *Proc. 9th International Workshop on Algorithms and Data Structures (WADS), Lecture Notes in Computer Science,*, 3608:385–395.

[4] Kaligosi, K. and Sanders, P. (2006). How branch mispredictions affect quicksort. *Proc. 14th Annual European Symposium on Algorithms (ESA), Lecture Notes in Computer Science*, 4168:780–791.

[5] Matteo Frigo, Charles E. Leiserson, H. P. and Ramachandran, S. (2012). Cache-oblivious algorithms. *ACM Transactions on Algorithms*, page 22.

[6] Tanenbaum, A. (2006). *Structured Computer Organization.* Pearson, Prentice-Hall, Upper Saddle River (N.J.).