Cache-Conscious Runtime Optimization for Ranking Ensembles

Xun Tang, Xin Jin, Tao Yang
Department of Computer Science, University of California
Santa Barbara, CA 93106, USA
{xtang,xin_jin,tyang}@cs.ucsb.edu

ABSTRACT

Multi-tree ensemble models have been proven to be effective for document ranking. Using a large number of trees can improve accuracy, but it takes time to calculate ranking scores of matched documents. This paper investigates data traversal methods for fast score calculation with a large ensemble. We propose a 2D blocking scheme for better cache utilization with simpler code structure compared to previous work. The experiments with several benchmarks show significant acceleration in score calculation without loss of ranking accuracy.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Retrieval Models, Search Process

Keywords

Ensemble methods; query processing; cache locality

1. INTRODUCTION

Learning ensembles based on multiple trees are effective for web search and other complex data applications (e.g. [9, 8, 10]). It is not unusual that algorithm designers use thousands of trees to reach better accuracy and the number of trees becomes even larger with the integration of bagging. For example, winning teams in the Yahoo! learning-to-rank challenge [8] have all used boosted regression trees in one form or another and the total number of trees reported for scoring ranges from 3,000 to 20,000 [11, 6, 12], or even reaches 300,000 or more combined with bagging [13].

Computing scores from a large number of trees is time-consuming. Access of irregular document attributes along with dynamic tree branching impairs the effectiveness of CPU cache and instruction branch prediction. Compiler optimization [5] cannot handle complex code such as rank scoring very well. For example, processing a 8,051-tree ensemble can take up to 3.04 milliseconds for a document with 519 features on an AMD 3.1 GHz core. Thus the scoring time per query exceeds 6 seconds to rank the top-2,000 results. It takes more time proportionally to score more documents with larger trees or more trees and this is too slow for interactive query performance. Multi-tree calculation can be parallelized; however, query processing throughput is not increased because less queries are handled in parallel. Tradeoff between ranking accuracy and performance can be played by using earlier exit based on document-ordered traversal (DOT) or scorer-ordered traversal (SOT) [7], and by tree trimming [3]. The work in [4] proposes an architecture-conscious solution called VPred that converts control dependence of code to data dependence and employs loop unrolling with vectorization to reduce instruction branch misprediction and mask slow memory access latency. The weakness is that cache capacity is not fully exploited and maintaining the lengthy unrolled code is not convenient.

Unorchestrated slow memory access incurs significant costs since memory access latency can be up to 200 times slower than L1 cache latency. How can fast multi-tree ensemble ranking with simple code structure be accomplished via memory hierarchy optimization, without compromising ranking accuracy? This is the focus of this paper.

We propose a cache-conscious 2D blocking method to optimize data traversal for better temporal cache locality. Our experiments show that 2D blocking can be up to 620% faster than DOT, up to 245% faster than SOT, and up to 50% faster than VPred. After applying 2D blocking on top of VPred which shows advantage in reducing branch mis-prediction, the combined solution Block-VPred could be up to 100% faster than VPred. The proposed techniques are complementary to previous work and can be integrated with the tree trimming and early-exit approximation methods.

2. PROBLEM DEFINITION

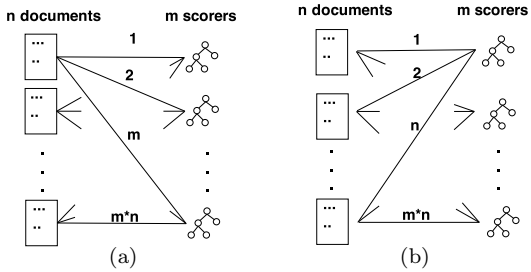
Given a query, there are n documents matching this query and the ensemble model contains m trees. Each tree is called a scorer and contributes a subscore to the overall score for a document. Following the notation in [7], Algorithm 1 shows the program of DOT. At each loop iteration i, all tress are calculated to gather subscores for a document before moving to another document. In implementation, each document is represented as a feature vector and each tree can be stored in a compact array-based format [4]. The time and space cost of updating the overall score with a subscore is relatively insignificant. The dominating cost is slow memory accesses during tree traversal based on document feature values. By exchanging loops i and j in Algorithm 1, DOT becomes SOT. Their key difference is the traversal order.

SIGIR'14, July 6–11, 2014, Gold Coast, Queensland, Australia.

ACM 978-1-4503-2257-7/14/07 ...$15.00. http://dx.doi.org/10.1145/2600428.2609525.

Algorithm 1: Ranking score calculation with DOT.

for $i = 1$ *to* n **do**
  for $j = 1$ *to* m **do**
    Compute a subscore for document i with tree j.
    Update document score with the above subscore.



(a) (b)

Figure 1: Data access order in DOT (a) and SOT (b).

Figure 1(a) shows the data access sequence in DOT, marked on edges between documents and tree-based scorers. These edges represent data interaction during ranking score calculation. DOT first accesses a document and the first tree (marked as Step 1); it then visits the same document and the second tree. All m trees are traversed before accessing the next document. As m becomes large, the capacity constraint of CPU cache such as L1, L2, or even L3 does not allow all m trees to be kept in the cache before the next document is accessed. The temporal locality of a document is exploited in DOT since the cached copy can be re-accessed many times before being flushed; however, there is no or minimal temporal locality exploited for trees. Similarly, Figure 1(b) marks data interaction edges and their access order in SOT. SOT traverses all documents for a tree before accessing the next tree. Temporal locality of a tree is exploited in SOT; however, there is no or minimal temporal locality exploited for documents when n is large.

VPred [4] converts if-then-else branches to dynamic data accesses by unrolling the tree depth loop. The execution still follows DOT order, but it overlaps the score computation of several documents to mask memory latency. Such vectorization technique also increases the chance of these documents staying in a cache when processing the next tree. However, it has not fully exploited cache capacity for better temporal locality. Another weakness is that the length of the unrolled code is quadratic to the maximum tree depth in a ensemble, and linear to the vectorization degree v. For example, the header file with maximum tree depth 51 and vectorization degree 16 requires 22,651 lines of code. Long code causes inconvenience in debugging and code extension. In comparison, our 2D blocking code has a header file of 159 lines.

3. 2D BLOCK ALGORITHM

Algorithm 2 is a 2D blocking approach that partitions the program in Algorithm 1 into four nested loops. The loop structure is named SDSD because the first (outer-most) and third levels iterate on tree-based Scorers while the second and fourth levels iterate on Documents. The inner two loops process d documents with s trees to compute subscores of these documents. We choose d and s values so that these d documents and s trees can be placed in the fast cache under its capacity constraint. To simplify the presentation of this paper, we assume $\frac{m}{s}$ and $\frac{n}{d}$ are integers. The hierarchical data access pattern is illustrated in Figure 2. The edges in the left portion of this figure represent the interaction among blocks of documents and blocks of trees with access sequence marked on edges. For each block-level edge, we demonstrate the data interaction inside blocks in the right portion of this figure. Note that there are other variations of 2D blocking structures: SDDS, DSDS and DSSD. Our evaluation finds that SDSD is the fastest for the tested benchmarks.

Algorithm 2: 2D blocking with SDSD structure.

for $j = 0$ *to* $\frac{m}{s} - 1$ **do**
  for $i = 0$ *to* $\frac{n}{d} - 1$ **do**
    for $jj = 1$ *to* s **do**
      for $ii = 1$ *to* d **do**
        Compute subscore for document $i \times d + ii$ with tree $j \times s + jj$.
        Update the score of this document.

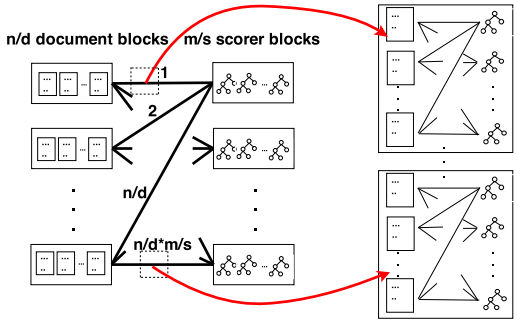


Figure 2: Data access order in the SDSD blocking scheme.

There are two to three levels of cache in modern AMD or Intel CPUs. For the tested datasets, L1 cache is typically too small to fit multiple trees and multiple document vectors for exploiting temporal locality. Thus L1 is used naturally for spatial locality and more attention is on L2 and L3 cache. 2D blocking design allows the selection of s and d values so that s trees and d documents fit in L2 cache.

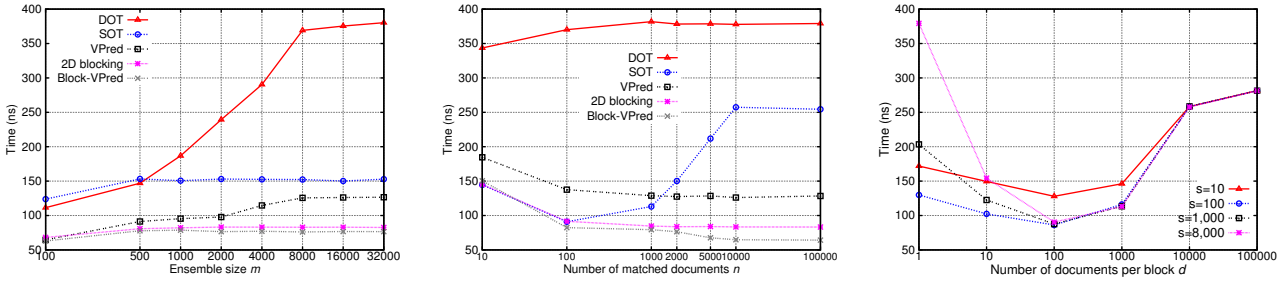
Detailed cache performance analysis requires a study of cache miss ratio estimation in multiple levels of cache. Due to the length restriction of this paper, we use a simplified cache-memory model to illustrate the benefits of the 2D blocking scheme. This model assumes there is one level of cache which can hold d document vectors and s tree-based scorers, i.e. space usage for s and d do not exceed cache capacity. Here we estimate the total slow memory accesses during score calculation using the *big O* notation. The inner-most loop ii in Algorithm 2 loads 1 tree and d document vectors. Then loop jj loads another tree and still accesses the same d document vectors. Thus there are a total of $O(s) + O(d)$ slow memory accesses for loops jj and ii. In loop level i, the s trees stay in the cache and every document block causes slow memory accesses, so memory access overhead is $O(s) + O(d) \times \frac{n}{d}$. Now looking at the the outer-most loop j, total memory access overhead per query is $\frac{m}{s}(O(s) + O(n)) = O(m + \frac{m \times n}{s})$.

From Figure 1, memory access overhead per query in DOT can be estimated as $O(m \times n + n)$ while it is $O(m \times n + m)$ for SOT. Since term $m \times n$ typically dominates, our 2D blocking algorithm incurs s times less overhead in loading data from slow memory to cache when compared with DOT or SOT.

Vectorization in VPred can be viewed as blocking a number of documents and the authors have reported [4] that a larger vectorization degree does not improve latency masking and for Yahoo! dataset, 16 or more degree performs about the same. The objective of 2D blocking scheme is to

Dataset	Leaves	m	n	DOT	SOT	VPred [v]	2D blocking [s, d]	Block-VPred [s, d, v]	Latency
Yahoo!	50	7,870	5,000	186.0	113.8	47.4 [8]	36.4 [300, 300]	36.7 [300, 320, 8]	1.43
	150	8,051	2,000	377.8	150.2	123.0 [8]	81.9 [100, 400]	76.1 [100, 480, 8]	1.23
	400	2,898	5,000	312.3	223.8	136.2 [8]	90.9 [100, 400]	86.0 [100, 400, 8]	1.25
MSLR-30K	50	1,647	5,000	88.3	41.4	32.6 [8]	26.6 [500, 1,000]	31.1 [500, 1,600, 8]	0.22
MQ2007	50	9,870	10,000	1.79	1.66	2.02 [8]	1.51 [300, 5,000]	1.94 [300, 5,000, 8]	0.15
	200	10,103	10,000	204.1	30.3	43.1 [32]	28.3 [100, 10,000]	26.2 [100, 5,000, 32]	2.65

Table 1: Scoring time per document per tree in nanoseconds for five algorithms. Last column shows the average scoring latency per query in seconds under the fastest algorithm marked in gray.



(a) (b) (c)

Figure 3: Scoring time per document per tree in nanoseconds when varying m (a) and n (b) for five algorithms, and varying s and d for 2D blocking (c). Benchmark used is Yahoo! dataset with a 150-leaf multi-tree ensemble.

fully exploit cache locality. We can apply 2D blocking on top of VPred to exploit more cache locality while inheriting the advantages of VPred. We call this approach Block-VPred. The code length of Block-VPred is about the same as VPred.

4. EVALUATIONS

2D block and Block-VPred methods are implemented in C and VPred code is from [4]. Code is compiled with GCC using optimization flag -O3. Experiments are conducted on a Linux server with 8 cores of 3.1GHz AMD Bulldozer FX8120 and 16GB memory. FX8120 has 16KB of L1 data cache per core, 2MB of L2 cache shared by two cores, 8MB of L3 cache shared by eight cores. The cache line is of size 64 bytes. Experiments are also conducted in Intel X5650 2.66GHz six-core dual processors and the conclusions are similar. This paper reports the results from AMD processors.

We use the following learning-to-rank datasets as the core test benchmarks. (1) Yahoo! dataset [8] with 709,877 documents and 519 features per document from its learning-to-rank challenge. (2) MSLR-30K dataset [2] with 3,771,125 documents and 136 features per document. (3) MQ2007 dataset [1] with 69,623 documents and 46 features per document. The tree ensembles are derived by the open-source jforests [10] package using LambdaMART [6]. To assess score computation in presence of a large number of trees, we have also used bagging methods to combine multiple ensembles and each ensemble contains additive boosting trees.

There are 23 to 120 documents per query labeled in these datasets. In practice, a search system with a large dataset ranks thousands or tens of thousands of top results after the preliminary selection. We synthetically generate more matched document vectors for each query. Among these synthetic vectors, we generate more vectors bear similarity to those with low labeled relevance scores, because typically the majority of matched results are less relevant.

Metrics. We mainly report the average time of computing a subscore for each matched document under one tree. This scoring time multiplied by n and m is the scoring latency per query for n matched documents ranked with an m-tree model. Each query is executed by a single core.

A comparison of scoring time. Table 1 lists scoring time under different settings. Column 2 is the maximum number of leaves per tree. Tuple [s,d,v] includes the parameters of 2D blocking and the vectorization degree of VPred that leads to the fastest scoring time. Choices of v for VPred are the best in the tested AMD architecture and are slightly different from the values reported in [4] with Intel processors. Last column is the average scoring latency per query in seconds after visiting all trees. For example, 2D blocking is 361% faster than DOT and is 50% faster than VPred for Row 3 with Yahoo! 150-leaf 8,051-tree benchmark. In this case, Block-VPred is 62% faster than VPred and each query takes 1.23 seconds to complete scoring with Block-VPred. For a smaller tree in Row 5 (MSLR-30K), Block-VPred is 17% slower than regular 2D blocking. In such cases, the benefit of converting control dependence as data dependence does not outweigh the overhead introduced.

Figure 3 shows the scoring time for Yahoo! dataset under different settings. In Figure 3(a), n is fixed as 2,000; DOT time rises dramatically when m increases because these trees do not fit in cache; SOT time keeps relatively flat as m increases. In Figure 3(b), m is fixed as 8,051 while n varies from 10 to 100,000. SOT time rises as n grows and 2D blocking is up to 245% faster. DOT time is relatively stable. 2D blocking time and its gap to VPred are barely affected by the change of m or n. Block-VPred is 90% faster than VPred when n=5,000, and 100% faster when n=100,000. Figure 3(c) shows the 2D blocking time when varying s and d. The lowest value is achieved with s=1,000 and d=100 when these trees and documents fit in L2 cache.

Cache behavior. Linux *perf* tool reports L1 and L3 cache miss ratios during execution. We observed no strong correlation between L1 miss ratio and scoring time. L1 cache allows program to exploit limited spatial locality, but is too small to exploit temporal locality in our problem context. L3 miss ratio does show a strong correlation with scoring time. In our design, 2D blocking sizes (s and d) are determined based on L2 cache size. Since L2 cache is about the same size as L3 per core in the tested AMD machine, reported L3 miss ratio reflects the characteristics of L2 miss ratio.

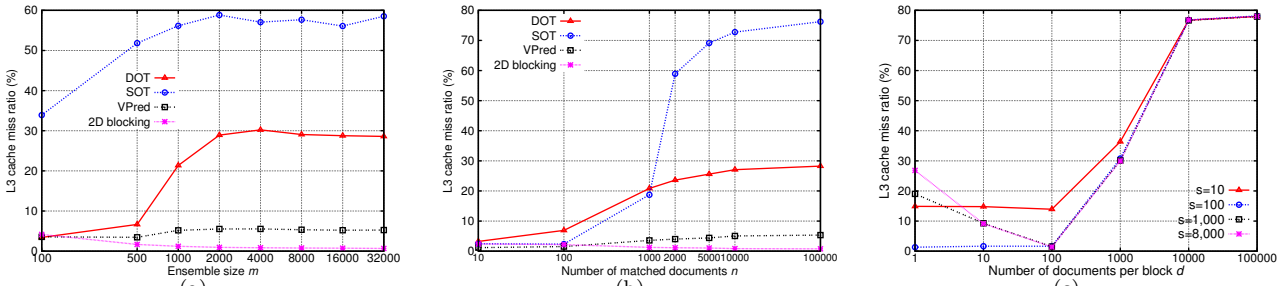


Figure 4: L3 miss ratio when varying n (a), varying m (b) for four algorithms, and when varying s and d for 2D blocking (c).

Figure 4 plots the L3 miss ratio under the same settings as Figure 3 for Yahoo! data. This ratio denotes among all the references to L3 cache, how many are missed and need to be fetched from memory. The ratios of Block-VPred, which are not listed, are very close to that of 2D blocking. In Figure 4(a) with n=2,000, SOT has a visibly higher miss ratio because it needs to bring back most of the documents from memory to L3 cache every time it evaluates them against a scorer; n is too big to fit all documents in cache. The miss ratio of DOT is low when all trees can be kept in L2 and L3 cache; this ratio grows dramatically after m=500. Figure 4(b) shows miss ratios when m=8,051 and n varies. The miss ratio of SOT is close to VPred and 2D blocking when n<100, but deteriorates significantly when n increases and these documents cannot fit in cache any more. The miss ratios of VPred in both Figure 4(a) and 4(b) are below 6% because vectorization improves cache hit ratio. Performance of 2D blocking is the best, maintaining miss ratio around 1% even when m or n is large.

Figure 4(c) plots L3 miss ratio of 2D blocking when varying s and d block sizes. The trends are strongly correlated with the scoring time curve in Figure 3(c). The optimal point is reached with s=1,000 and d=100 when these trees and documents fit in L2 cache. When s=1,000, miss ratio varies from 1.64% (d=100) to 78.1% (d=100,000). As a result, scoring time increases from 86.2ns to 281.5ns.

Branch mis-prediction. We have also collected instruction branch mis-prediction ratios during computation. For MQ2007 and 50-leaf trees, mis-prediction ratios of DOT, SOT, VPred, 2D blocking and Block-VPred are 1.9%, 3.0%, 1.1%, 2.9%, and 0.9% respectively. For 200-leaf trees, these ratios increase to 6.5%, 4.2%, 1.2%, 9.0%, and 1.1%. VPred's mis-prediction ratio is lower than 2D blocking while its scoring time is still longer, indicating the impact of cache locality on scoring time is bigger than branch mis-prediction. For smaller trees, mis-prediction ratios of 2D blocking and Block-VPred are close and this explains why Block-VPred does not outperform 2D blocking in Table 1 for 50-leaf trees. Adopting VPred's strategy of converting if-then-else instructions pays off for large trees. For such cases when n increases, Block-VPred outperforms 2D blocking with lower branch mis-prediction ratios. This is reflected in the Yahoo! 150-leaf 8,051-tree benchmark: mis-prediction ratios are 1.9%, 2.7%, 4.3%, and 6.1% for 2D blocking, 1.1%, 0.9%, 0.84%, and 0.44% for Block-VPred, corresponding to the cases of n=1,000, 5,000, 10,000 and 100,000 respectively.

5. CONCLUDING REMARKS

The main contribution of this work is cache-conscious design for computing ranking scores with a large number of trees and/or documents by exploiting memory hierarchy capacity for better temporal locality. Multi-tree score calculation of each query can be conducted in parallel on multiple cores to further reduce latency. Our experiments show that 2D blocking still maintains its advantage using multiple threads. In some applications, the number of top results (n) for each query is inherently small and can be much smaller than the optimal block size (d). In such cases, multiple queries could be combined and processed together to fully exploit cache capacity. Our experiments with Yahoo! dataset and 150-leaf 8,051-tree ensemble shows that combined processing could reduce scoring time per query by 12.0% when n=100, and by 48.7% when n=10.

Our 2D blocking technique is studied in the context of tree-based ranking ensembles and one of future work is to extend it for other types of ensembles by iteratively selecting a fixed number of the base rank models that can fit in the fast cache.

Acknowledgments. This work was supported in part by NSF IIS-1118106. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

6. REFERENCES

[1] Lector 4.0 datasets. http://research.microsoft.com/en-us/um/beijing/projects/letor/letor4dataset.aspx.

[2] Microsoft learning to rank datasets. http://research.microsoft.com/en-us/projects/mslr/.

[3] N. Asadi and J. Lin. Training Efficient Tree-Based Models for Document Ranking. In *ECIR*, pages 146–157, 2013.

[4] N. Asadi, J. Lin, and A. P. D. Vries. Runtime Optimizations for Tree-Based Machine Learning Models. *IEEE TKDE*, pages 1–13, 2013.

[5] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[6] C. J. C. Burges, K. M. Svore, P. N. Bennett, A. Pastusiak, and Q. Wu. Learning to rank using an ensemble of lambda-gradient models. In *J. of Machine Learning Research*, pages 25–35, 2011.

[7] B. B. Cambazoglu, H. Zaragoza, O. Chapelle, and J. Chen. Early Exit Optimizations for Additive Machine Learned Ranking Systems Ranking in Additive Ensembles. In *WSDM*, pages 411–420, 2010.

[8] O. Chapelle and Y. Chang. Yahoo! Learning to Rank Challenge Overview. *J. of Machine Learning Research*, pages 1–24, 2011.

[9] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[10] Y. Ganjisaffar, R. Caruana, and C. Lopes. Bagging Gradient-Boosted Trees for High Precision, Low Variance Ranking Models. In *SIGIR*, pages 85–94, 2011.

[11] P. Geurts and G. Louppe. Learning to rank with extremely randomized trees. *J. of Machine Learning Research*, 14:49–61, 2011.

[12] A. Gulin, I. Kuralenok, and D. Pavlov. Winning the transfer learning track of yahoo!'s learning to rank challenge with yetirank. *J. of Machine Learning Research*, 14:63–76, 2011.

[13] D. Y. Pavlov, A. Gorodilov, and C. A. Brunk. Bagboo: a scalable hybrid bagging-the-boosting model. In *CIKM*, pages 1897–1900, 2010.