Building Efficient Multi-Threaded Search Nodes

Carolina Bonacic [1] Carlos García [1] Mauricio Marin [2]
Manuel Prieto-Matias [1] Francisco Tirado [1]

[1]Universidad Complutense de Madrid, Spain
[2]Yahoo! Research Latin America, University of Santiago of Chile

ABSTRACT

Search nodes are single-purpose components of large Web search engines and their efficient implementation is critical to sustain thousands of queries per second and guarantee individual query response times within a fraction of a second. Current technology trends indicate that search nodes ought to be implemented as multi-threaded multi-core systems. The straightforward solution that system designers can apply in this case is simply to follow standard practice by deploying one asynchronous thread per active query in the node and attaching each thread to a different core. Each concurrent thread is responsible for sequentially processing a single query at a time. The only potential source of read/write conflicts among threads are the accesses to the different application caches present in the search node. However, new Web applications pose much more demanding requirements in terms of read/write conflicts than recent past applications since now data updates must take place concurrently with query processing. Insisting on the same paradigm of concurrent threads now augmented with a transaction concurrency control protocol is a feasible solution. In this paper we propose a more efficient and much simpler solution which has the additional advantage of enabling a very efficient administration of application caches. We propose performing relaxed bulk-synchronous parallelism at multi-core level.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

General Terms

Algorithms, Performance

Keywords

Web Search Engines, Query Processing, Multithreading

1. INTRODUCTION

Data centers of major Web search engines house hundreds of search nodes forming distributed memory clusters that must sustain thousands of queries per second. Currently, a wide variety of inverted file indexing and query/index caching strategies are used in search nodes to make them efficient in both query throughput and query response time.

Emerging applications of Web search engines require proper support for fast on-line updates of search node main memory contents and current trend in clusters of processors is towards multicore technology. Thus it is relevant to look at how to deal efficiently with the concurrency control problems arising when multiple threads are updating the inverted file index and query/index caches in multicore processors.

Fast concurrency control is a challenging task in applications that require real-time updates. We envisage at least four cases of applications with this requirement.

- In Q&A systems like Yahoo! Answers, thousands of users per second submit queries whose processing are speeded up by using an inverted file. Concurrently, other users submit small texts containing terms (words) that refer to those queries which can cause conflicting update operations on the inverted file. Certainly these users would like to see their texts become part of answers to related queries as soon as possible.

- In a photo sharing system like Flickr, users upload new pictures and label them with small texts and set pre-defined tags describing their contents. In a production system more than 10 millions pictures can be uploaded or updated per day. The labeling enables reader users to search for related pictures and writer users would like their pictures to become part of the results of those queries within a short period of time.

- Though still incipient, a challenging application is the case in which query ranking methods include the effect of previous user preferences to similar queries registered in the very last seconds. In this case, the clicks made by users on the answers to queries must be sent back to the search engine and indexed in an on-line manner. The objective is to use these clicks to refine the ranking of results for subsequent queries. Clicks can be indexed using an inverted file and now concurrency conflicts appear among the continuous stream of click updates over the index and the required operations needed to process time-consuming tasks such as determination of similar queries and related clicks on documents.

- Recently, major Web search engines have started to include in the results provided to user queries small documents retrieved from on-line multi-million user systems such as blogs and twitter. This implies reflecting in the inverted files of search nodes supporting this so-called “real-time search”, the effects of a continuous stream of new documents arriving to the search nodes.

CIKM'10, October 26–30, 2010, Toronto, Ontario, Canada.

We study the concurrency control issue at programming level where portability across different architectures is of paramount importance. In this case our only tools are the primitives available from the standard Posix threads library. We emphasize on variable number of threads as a firm requirement to our algorithmic design since in production systems it is usual to periodically run on designated cores administration software to collect state information and perform housekeeping.

Search nodes are single purpose systems in charge of a single task related to the processing of queries. We refer to tasks such as determination of the top-K documents that best match the query, construction of the answer Web page, advertising pertinent to query terms, spelling, suggestions, etc.. This paper focuses on the first task since it is the most demanding one in use of resources due to the huge sizes of the Web samples indexed by each search node.

While the details of the design of well-known search engines are kept confidential, current literature on search using inverted files and caching still assumes the classic single-CPU model and does not exploit the available thread level parallelism present in modern multicore architectures. Ignoring the issue of R/W conflicts on indexing and caching, a straightforward approach to transparently take advantage of such architectures is to rely on virtualization technology to increase the utilization of the multicore hardware. However, virtualization software can potentially lead to significant overheads [11] that can be detrimental to query throughput (e.g., Google claims not to use virtualization software in production search nodes [13]). Given the large number of servers involved in query processing, it is crucial to be as efficient as possible in search node realization to reduce the amount of hardware deployed in data centers. For the same reason, off-line query processing systems such as map-reduce computing upon distributed file systems [2] are certainly not suitable for our target of highly-optimized single-task-oriented search nodes.

The problem of efficient processing of concurrent R/W operations has deserved attention for several decades [7]. Currently, we can profit from a number of efficient solutions based on conservative and optimistic approaches to causal order of operations. The closest context to ours is transaction processing in databases. Below we compare our proposal against the approaches that are most suitable to our case. What previous work shows is that a good solution to concurrency control is critically dependent on the particular features of the application at hand and the target architecture. Our application poses difficulties on its own merits featured by differing degrees of causality constrains, block-wise indexing/caching/ranking strategies, and an extremely dynamic intensity of query traffic. We exploit these features to increase efficiency by performing parallel bulk processing and relaxed synchronization.

The remaining of the paper is structured as follows. Section 2 provides background and related work. Section 3 presents our proposal for multicore search nodes and Section 4 presents a comparative evaluation against alternative approaches. Section 5 presents concluding remarks.

2. BACKGROUND

Web search engines use inverted indexes to determine the top-K results for queries. Namely, the K documents that best match the query terms in accordance with a given document ranking method. The index is composed of a vocabulary table and a set of posting lists. The vocabulary table contains the set of relevant terms found in the text collection. Each of these terms is associated with a posting list which contains the document identifiers where the term appears in the collection along with additional data used for ranking purposes. To solve a query, it is necessary to get from the posting lists the set of documents associated with the query terms and then to perform a ranking of these documents in order to select the top-K documents as the query answer. A query receptionist machine, called the broker, sends queries for solution to a set of search nodes and blends their results to produce the global top-K ones.

A number of papers have been published on parallel query processing upon distributed inverted files (for recent work see [8, 12]). These strategies have been devised for distributed memory processors. To our knowledge, no work has been done in the subject of efficient query processing on modern multi-core architectures. A noticeable exception is the work presented in [3]. The authors study the use of the massive parallelism present in GPUs and conclude that significant gains in performance can be achieved by using lots of threads to process single queries. Concurrent write operations are not discussed.

Query throughput is further increased by using application caches (for recent work see [5, 6]). Usually the size of the index is huge and only the part that is most referenced by queries in a given period of time is maintained in main memory by means of a posting list cache. Also caches holding pre-computed results can be kept in each search node such as a top-K results cache for frequent queries which can be administered with the LRU policy. The posting lists cache can be implemented using Static-Dynamic Caching (SDC) [5]. SDC is composed of a static and a dynamic part. The former stores queries that are popular during long periods of time using LFU, whereas the dynamic part uses LRU to cache queries that become popular during short periods of time.

Bulk-synchronous parallelism. This is the main parallelization approach used in this paper. It refers to the BSP model of parallel computing proposed in [14] for a set of distributed memory processors (search nodes in our setting) that communicate each other via message-passing, and recently further refined for shared-memory multi-core processors in [15]. In generic terms the BSP model is defined as follows. Parallel computation is organized as a sequence of *supersteps*. During a superstep, the processors may perform computations on local data and/or send messages to other processors. The messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronization of the processors.

At macroscopic level, for the message-passing distributed memory model, it has been shown in [10] that BSP allows efficient concurrency control of read/write transactions across a set of P search nodes for high query traffic. It proposes a timestamped protocol for concurrency control. The BSP mode of parallel processing has also been shown efficient for high traffic for conventional query processing in [8] (i.e., only read transactions). At intermediate level, for the same read only setting, but assuming that each of the P search nodes is a multi-core machine, the use of bulk-synchronism at core level has been shown to be efficient in [1]. These results are limited to disjunctive queries and pruned document ranking.

The problem studied in the present paper is fairly different. It works at microscopic level, namely it proposes a multi-core bulk-synchronous strategy for efficient processing of read/write transactions confined to individual search nodes. The assumption here is the standard setting in which data center search nodes do not communicate each other to solve queries but they receive work to be done from a centralized broker machine and they respond with the results to the broker only. To achieve efficient performance we resort to a relaxed form of barrier synchronization of cores we have devised for this problem from PThreads Library constructs.

The parallel multi-core priority queue proposed in this paper resembles the one presented in [9]. The difference is that the proposal of this paper is tailored to multi-threaded cache management.

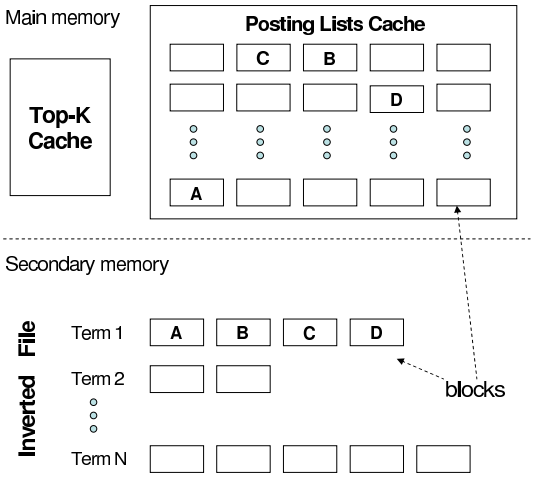


Figure 1: Search node caches.

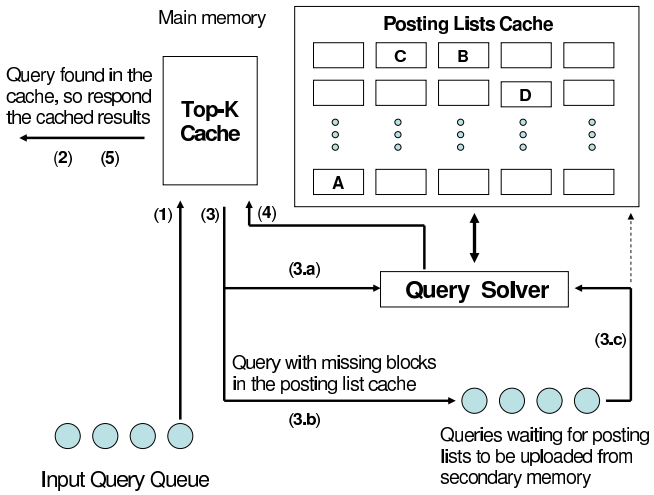


Figure 2: Paths followed by queries.

3. MULTI-THREADED SEARCH NODES

3.1 Problems to be solved

From the literature on indexing and caching we can learn of a number of alternative strategies for implementing high performance search nodes. Figure 1 illustrates a possible layout. The posting list cache consists of a large number of blocks which are used to store posting list items, namely pairs (doc_id, term_freq). Each posting list usually contains several blocks. The second cache holds the top-K results of the most frequent queries resolved by the search node. The whole inverted file is kept on local disk.

A feasible road map for the threads is illustrated in Figure 2. In this example, queries arrive at the input queue of the search node. A given number of threads are in charge of solving the queries. Every time a thread takes a new query from the input queue it checks whether the same query is already stored in the top-K cache (1). If there is a hit on this cache, the thread responds with the K document IDs stored in the respective entry (2). Otherwise, the thread verifies whether all of the blocks holding posting list items are already in the posting list cache (3). If so, the thread uses those blocks to solve the query by applying a document ranking algorithm (3.a). Once the thread gets the top-K document IDs, it applies the top-K cache admission and eviction policy to store the new entry in the cache (4), and responds with the calculated top-K results (5). If there are missing posting list blocks in the cache (3.b), the thread places the query in another queue of secondary memory requirements (a subordinate thread manages these transfers of new blocks) and verifies whether in this second queue there is another query for which the transferring of its blocks has finished to proceed with its solution (3.c), (4) and (5). Notice that we say "secondary memory" to refer to data stored on disk or data stored in compressed format in main memory.

The above multi-threaded query processing approach assumes the existence of a strategy able to properly deal with the concurrent R/W operations demanded by the threads on the caches and queues. As a result of the eviction policy, concurrent reads and writes can be performed on the same cache entries so threads must be synchronized to prevent from R/W conflicts. A textbook solution is to use mutual-exclusion locks on cache entries but notice the following difficulties that our proposal solves efficiently.

Upon calculation of the top-K results for a query, it is necessary to evict an entry in the top-K cache in order to store the new results. During the same time interval, other threads may be reading the top-K cache entries to decide whether or not to calculate the results for their respective queries. If concurrency control is performed at a very coarse level using a single or just a few locks to protect the whole cache, threads can be severely serialized. Locking at the entry level can provide better performance but still, locking each entry sequentially whilst scanning the cache can also generate significant overheads. As discussed in [5], this problem can be alleviated when using cache strategies such as the SDC where 80% of the entries are read-only and 20% of the entries are read/write LRU entries.

In the posting list cache, it is necessary to quickly select a sufficiently large number of cache blocks to be replaced by the blocks containing the items of the new posting list being uploaded from secondary memory. A priority queue data structure can be used for this purpose or alternatively a linked list queue with move-to-front heuristic for approximated best priority selection. For LRU, the move-to-front queue performs well, whereas for other cache policies, such as Landlord [6], the priority queue works efficiently.

If the blocks containing the posting list items are to be modified by writer threads as in our target applications of Web search, then the above concurrency control problem is exacerbated with the additional difficulty that locks must be consistent across cache entries and posting list updates. For instance, if a concurrency control strategy is based on the locking of the posting lists associated with the query terms, then this should also cause the locking (or hiding from the cache eviction algorithm) of all of the cache blocks holding the respective posting list items.

In addition, whilst a thread is reading or updating a posting list ℓ, the cache management algorithm should not evict from the cache the blocks assigned to keep in main memory the posting list items of ℓ. As in the previous case, a straightforward solution would be to touch and temporarily hide from the cache eviction algorithm all of the blocks of ℓ before a thread uses them. This process also requires proper synchronization of threads and thereby this causes further overheads in the search node.

Posting list writing also leads to the problem of consistency between the new version of a posting list and the possible entries for the same term already stored in the top-K cache. If the newly inserted or updated posting list item has a frequency which is high enough or larger than the previous version respectively, then it is likely that the associated document might be also present in the respective top-K entry. In this case it is more practical to invalidate

all of the cache entries where the terms appears in the respective query since the only way to be sure is by re-computing the ranking for the query which is too expensive to do at this stage. This causes an additional concurrency check in a multi-threaded strategy.

3.2 Proposed solution

In this paper we propose a simple but very efficient strategy to solve the above described synchronization problems and others of this nature we might have missed in the example of Figure 2. To be precise, we do not actually solve them but avoid them by logically treating the steps described in Figure 2 as if they were executed by a single sequential process. In each step, this process resorts to the available intra-query parallelism to exploit the thread-level parallelism in the multicore processors. We efficiently parallelize the query solver by using N_t threads so that one thread runs in a different core. We apply a similar approach in the cache algorithms. Notice that our strategy can be easily adapted to work in a setting in which the number of cores available for processing may change dynamically. Namely, at each step of Figure 2, the number of available cores may change due to the execution of other service processes in the search node.

At any time instant, the search node focuses on the parallel processing of a single query. In the example of Figure 2, the most relevant operations to be parallelized are the ones related to the query solver and caches. The granularity of the other operations in terms of cost is very small and can be executed using a single core. Synchronization barriers are employed to group operations and amortize overheads.

3.2.1 Query Solver

The data structure organization is presented in Figure 3. To solve a given query it is necessary to traverse the posting lists of all of the query terms. The items in each posting list are grouped in blocks, and the query solver traverses them sequentially. When using an early termination document ranking method and after traversing a given number of blocks, the query solver may decide that it is not necessary to go further on the list of blocks to obtain the top-K results for the query.

The idea is that each time the query solver processes a block, it uses N_t threads to rank documents from the posting list items assigned to each thread. For instance, Figure 3 shows this situation for the first block of term *term_0* where $N_t = 4$ threads evenly process the block in parallel.

Since the performance of multi-core processors is highly dependent on data locality, whilst processing a query all threads should make most memory accesses to local data (i.e., data stored in the private caches of the respective cores). To this end, each thread holds what we call a *fast_track* and queries are processed iteratively using two major steps:

- **Fetching**. The first step consists on fetching from main memory a K-sized piece of each posting list associated with the query terms and storing a different piece of size K/N_t in the *fast_track* memory of each of the N_t participating threads.
- **Ranking**. In the second step, the N_t threads perform in parallel the actual ranking of documents and, if necessary, they ask for additional K-sized pieces of the posting lists in order to produce the K best ranked documents for the query. This may involve the determination of the documents that contain all query terms, which implies performing an intersection operation among the involved posting lists.

Thus the ranking process can take one or more iterations to be completed.

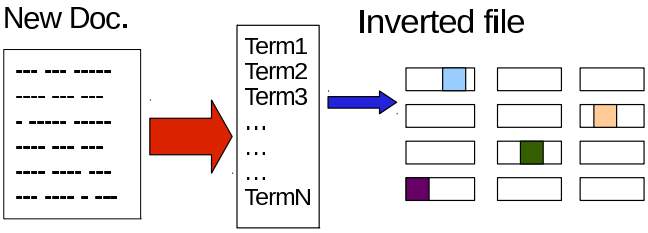


Figure 4: Document insertion/update.

On the other hand, in the input query queue of the search node we can have write operations that represent the case in which a new document needs to be inserted in the index or a new version of the document must be properly reflected in the index. Figure 4 illustrates the case in which a new document is inserted in the index and its effects in a determined set of terms and determined sections of their respective posting lists. A write operation can hit any of the blocks and they can be already in the posting list cache or in secondary memory. We evenly distribute the terms on the available threads and each thread modifies the respective posting lists in parallel.

For disjunctive queries the posting list are kept sorted by term frequency whereas for conjunctive queries the lists are kept sorted by document ID. The later requires intersecting the posting lists. In this case, it is useful to keep lists sorted by document IDs since intersection operations can be made in linear time on the lengths of the participating lists. Insertions of new documents can be made at the end of the respective posting lists. However, if the document IDs are associated with a global document rank value, then insertions on the index are not possible in such an efficient way and it may be more efficient to delay insertions and perform them in bulk to amortize overheads (these problems are out of scope in this paper). To support insertions in disjunctive queries, we maintain empty slots in the posting list blocks and apply a B-Tree like strategy by creating a new block moving items from two consecutive blocks in the posting list.

The BP Algorithm. A key feature of the proposed strategy is that it relies on bulk processing (BP) at thread level to achieve efficient performance whereas barriers are used to prevent from R/W conflicts. In essence, threads are barrier synchronized at the end of a sequence of iterations for queries and also after each document update/insertion operation. The details of the proposed strategy are presented in Algorithm 1 which represents a case where the query solver has in its input queue both query and document write/update operations, and it processes all of them in bulk before delivering the global top-K results for the queries encountered in the queue.

Despite its simplicity, barriers can introduce overheads that may limit scalability when operating under high query traffic. We rely on our own *oblivious* barrier implementation. A standard barrier synchronization primitive on shared memory architectures uses a single conditional wait to which all participating threads sign up to wait for all others. Instead, we keep both a conditional wait and a lock-protected counter for each participating thread. A given thread wakes up when all others have increased its respective counter. We do this to provide a setting in which threads can be master of one operation and slaves of other operations for which the other threads are masters. A given thread executing the role of master must wait for their slaves to complete the job whereas in its role of slave it does not wait for others at the synchronization point and just increments the master counter to signal that it has finished the job for the master.

In Algorithm 1 threads become master when they execute the

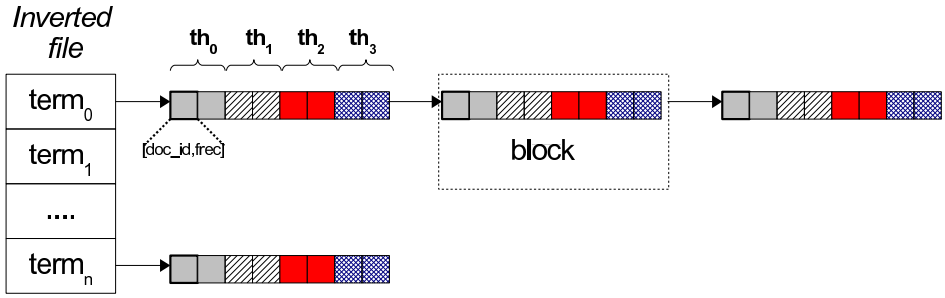


Figure 3: Inverted file organization where each posting list is stored in a number of blocks and where each block is logically divided in chunks that are assigned to threads. Each chunk is composed of a number of posting list items and each item is a pair (doc_id, freq). In this example, the first block of *term_0* is processed in parallel by threads *th0*, *th1*, *th2* and *th3*.

function *getGlobalTopK*(). The master-less write operations do not affect the scheme provided that the value of *num_Operations* is an integer multiplicative factor of the total number of threads (*nThreads*) participating in the query solver. If this is not possible due to a low traffic of query and document operations, we use the standard barrier primitive provided by the Posix threads library (as argued below, in low traffic it is not relevant being very efficient).

The details of the oblivious barrier synchronization routine are given in Algorithm 2. The *condWait*(a, b) and *condSignal*(a) operations have the same semantics as their Posix counterparts, i.e. the former releases the lock b and puts the calling thread to sleep on the condition variable a and the later wakes up the thread waiting on a.

Dealing with peaks in query traffic. Search nodes must ensure that the response time of individual queries is not above a given upper bound, even under a situation of high query traffic. In this scenario, it is sensible to delay document writing until periods of moderate or low traffic.

The proposed BP strategy is flexible enough in this case since the query solver can impose an upper limit to the number of blocks traversed per query and maintain a round-robin queue of queries under processing. The upper limit can be adjusted depending on the number of different queries being processed by the query solver. This is important because the length of posting lists follows the Zipf Law, namely a reduced number of them can have huge sizes whereas the remaining ones have relatively small sizes. So the search node has to ensure an upper bound for the response time of individual queries even at the expense of producing an approximated answer for large queries currently being solved and delaying document insertion/update operations.

Under a situation of high query traffic, the BP strategy can allow the query solver to perform round-robin so that fairly small queries can be solved in an exact manner within the upper bound restriction for the response time. Processing one or more blocks per query can be considered as quanta of processing assigned to each query in a round-robin fashion. This allows queries to evenly share the hardware resources which in our case are composed of application caches and cores. In addition, the oblivious synchronization algorithm enables more parallelism among queries which further increases query throughput.

3.2.2 Administering Application Caches

Notice that independently of the admission and eviction policy applied in the application caches, the associated selection abstract data type must efficiently support the retrieval of the n_E items with the lowest priorities and the insertion of n_I items with arbitrary priorities, where we should expect $n_E \gg 1$ and $n_I \gg 1$. If a priority queue is to be used for this purpose, then this defines the problem as one of parallel priority queues from which we can benefit from a number of solutions proposed in previous literature (e.g., [4]). Nevertheless, in our setting it is crucial that list blocks be stored in consecutive regions of main memory. This to increase the speed of data transfer between disk and cache. In the following we propose a priority queue design that serves well our purpose.

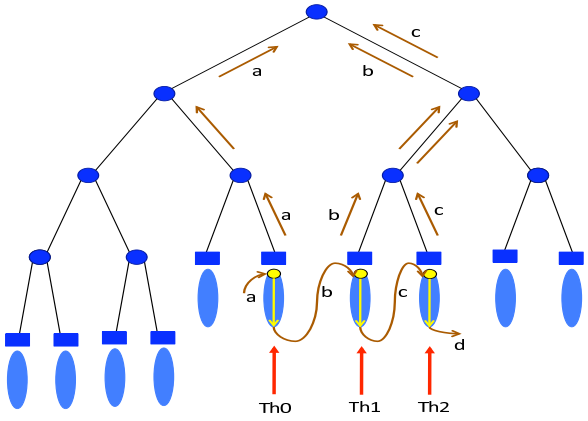


Figure 5: The local minima priority queue.

The data structure is illustrated in Figure 5 (caches are of fixed size). Fat nodes (vertical ovals in the figure) are all of the same size and contain n items where each item is associated with a single and fixed block in the cache. They span contiguous regions of main memory. Posting list blocks are associated with these items in a dynamic manner in accordance with the eviction policy of the cache.

The complete binary tree (CBT) on top of the fat nodes is used to select the fat node containing the item with the lowest priority. Each fat node has a representative which is the item with the lowest priority in the node (local minimum). The CBT maintains a tournament among all of the local minima so that the root of the tree contains the node ID owning the global minimum. As shown in the figure, each leaf of the CBT is associated with a unique fat node. Tournaments are made along the paths to the root by writing in each internal node the node ID of the winner between two nodes. The winner is the local minimum with lower priority.

The CBT is easily implemented by using an array of integers organized with the implicit heap invariant that each father i has its

Algorithm 1 The **BP** algorithm. Steps executed by a single thread running in parallel with other $nThreads - 1$ threads.

```
Bulk_Processing( num_Operations )
  private operation, next, term, pid, merger, i, t, block;
  shared solver_input_queue, inverted_file, results, out;
  pid = getThreadID();
  for ( next=0; next < num_Operations; next++ ) do
    operation = solver_input_queue[next];
    merger = next%nThreads;
    if (operation.type == QUERY) then
      for (t=0; t < operation.nterms; t++) do
        term= operation.term[t];
        iterations= ceil( List_size[term]/block_size );
        for (i=0; i < iterations; i++) do
          block= getNextBlock( inverted_file[term], pid );
          copy( fast_track, block ); // fetch
          runRanking( fast_track, doc_scores );
        end for
      end for
      results[merger, pid]= getTopK( soc_scores );
    else
      // operation.type == DOC_WRITE
      for (t=0; t < operation.nterms; t++) do
        term = operation.term[t];
        if (term%nThreads == pid) then
          // The threads evenly process different terms
          write( inverted_file, term, pid,
                 operation.freq[t], operation.doc_id);
        end if
      end for
    end if
    Oblivious_Barrier( merger, pid );
    if (operation.type == QUERY) then
      out[next]= getGlobalTopK( merger, results );
    end if
  end for
End_Bulk_Processing
```

children at positions $2i$ and $2i+1$. The advantage over the classical heap is that each update step in the data structure requires just one comparison between two priority values. This makes the update operation very fast which is convenient for us since each update on the CBT can be protected with a mutual-exclusion lock without compromising performance of multi-threading. Notice that not all updates are required to reach the root of the tree and the granularity of fat node administration is much larger than an update along the path to the root (granularity is proportional to the size n of the fat node).

As shown in Figure 5, parallelism is present across the fat nodes. The figure shows the traversal a, b, c, d of three fat nodes performed by a single thread which corresponds to the case in which a posting list with all of its blocks residing in the three nodes is hit by a query. Here it is necessary to update as fast as possible all the priority values assigned to the respective cache blocks. So the same can be done in parallel by using three different threads (one per fat node) to speed up this process (threads Th0, Th1 and Th2).

Algorithm 2 Oblivious barrier synchronization of threads.

```
Oblivious_Barrier( merger, pid )
  shared array_counter, array_mutex, array_cond;
  setLock( array_mutex[pid] );
  array_counter[pid]++;
  if ( merger == pid ) then
    if ( array_counter[pid] < nThreads ) then
      condWait( array_cond[pid], array_mutex[pid] );
    else
      array_counter[pid]= 0;
    end if
  else
    if ( array_counter[pid] == nThreads ) then
      array_counter[pid]= 0;
      condSignal( array_cond[pid] );
    end if
  end if
  unSetLock( array_mutex[pid] );
End_Oblivious_Barrier
```

The key fact that enables the local minima strategy to be suitable for this job is that potentially all of the priority values for the blocks of a posting lists are the same. This drastically reduces the amount of update operations executed on the CBT to keep the priority queue invariant.

After each update, the CBT root indicates the fat node containing the item with the lowest priority in the whole set of nodes. This is necessary to make space in the cache upon the arrival of a new posting list or whilst using a posting list and one or more of its blocks are not present in the cache. Because of the fact that all contiguous items in the selected n-sized fat node are likely to contain the same priority value, one should expect to consult the CBT root every n item insertions. The effect is that the same node can be used to host several blocks of a given posting list, which reinforces that related blocks become stored in contiguous memory.

There is a trade-off among the size of the fat nodes, the amount of fragmentation that one is willing to tolerate and the amount of updates executed on the CBT per posting list hit or admission. The worst case is similar to using a classical heap. Namely, the CBT root produces a different fat node ID for each posting list block insertion.

4. COMPARATIVE EVALUATION

Precisely comparing the performance of a bulk-synchronous multi-core multi-threaded system as the proposed in this paper against an alternative fully-asynchronous multi-core multi-threaded one, each containing a query solver, index updater and application caches operating at variable rates of query traffic, is a complex problem. To ease this difficulty we proceeded as follows. We first performed a demanding set of experiments excluding the effects of administering application caches. That is, in these experiments we considered that the whole set of posting lists hit by read/write transactions is always available in main memory (currently there are production systems that operate in this all-in-main-memory mode). We only assume high query traffic situations which, as explained in the previous section, is the relevant case for our setting. Notice that query resolution or index updating does not take place until the involved posting list items are properly cached in main memory. Otherwise

the respective transaction is enqueued to wait for the required items. This means that caching efficiency can be evaluated in a separate experiment wherein the relevant component to look at is the priority queue realization (searching for a cache hit is $O(1)$ due to the use of hashing on the posting list terms). For this case we performed a different set of experiments devised to evaluate the running time cost of the proposed priority queue by considering different cache hit ratios.

Another relevant motivation for separate experimentation is the following. If the BP strategy is found to beat state of the art asynchronous alternatives in the all-in-main-memory scenario, then this is a clear indication that it is more efficient to process each transaction one by one, each time by using all of the available cores in the machine at that particular moment. In this case, priority queue administration is also executed for individual transactions in parallel and thereby its operations can be optimized by following an approach similar to the BP strategy.

Baseline concurrency control strategies. To compare the proposed BP strategy against alternative strategies, we have taken from the literature a number of concurrency control strategies for read-/write transactions and adapted them to our context. They use either locks or barriers as synchronization mechanisms (i.e., they are Posix threads feasible) and their main characteristics are summarized in Table 1. The first column indicates whether the strategy introduces serializable scheduling of user queries and index updates, the second column indicates the source of parallelism and the third column indicates the synchronization primitive used to prevent from R/W conflicts. We say read transaction to mean the processing of a (read-only) query whereas write transaction to mean the insertion or update of a document.

The most restrictive strategies guarantee scheduling of transactions that are serializable (i.e., they maintain the illusion of serial execution). The other strategies can be applied in scenarios where serializability is not enforced, which potentially may deliver better performance. Notice that, like the proposed BP strategy, all of the strategies work using the same scheme of fetching and ranking over the core-cache friendly fast-track data structure. Also all strategies get read/write transactions from a shared input queue IQ where a basic step towards serializability (BP excluded) is to first lock the IQ in exclusive access mode to acquire a transaction and then release the lock on IQ as soon as possible.

The "*Concurrent-Reads (CR)*" strategy is similar to the BP strategy in terms of exclusive parallel write operations. But it allows the overlapping of read transactions along time by assigning one different thread to each query. Each thread entirely solves its assigned query. Before a write transaction takes place, the CR strategy waits for all of the current read transactions being processed to end. In this way, the CR strategy exploits the parallelism available from all of the active user (read-only) queries in a given period of time. Since some threads may start to idle before the beginning of the next write transaction, we tested a version of CR where the idle threads help others to complete their read transactions. However, we did not observe improvements in performance due to the overheads associated with the required job stealing and thread scheduling procedures.

The "*Term-Level-Parallelism (TLP)*" strategy allows the overlapping of both read and write transactions in time as long as they involve different query terms. Each transaction is executed by a different thread and mutual-exclusion locks are used to protect the involved posting lists. Any given thread does not release the IQ lock to proceed with the new transaction until it acquires all of the respective list locks. Subsequently each list lock is released as soon as the respective list has been processed. We tested two versions of this two-phase locking approach. The first one always allow concurrent read transactions (*TLP1*), no matter if they share query terms, whereas the second one, which is much easier to implement, does not overlap read transactions that have one or more terms in common (*TLP2*). We also tested a non-serializable version of TLP2 in which the thread releases the IQ lock as soon as the next transaction in the IQ is assigned to the thread to let it request the list locks concurrently with others threads. This prevents idle threads from delay when a thread owning the IQ lock has to wait for list locks currently assigned to working threads. In this case, list locks are requested in term ID order to prevent from deadlocks. Performance did not improve which shows that IQ locking does not cause a bottleneck.

	Serial	Parallelism	Primitive
BP	Yes	PRT and PWT	Thread Barrier
CR	Yes	CRT and PWT	Thread Barrier
TLP1	Yes	CRT (term sharing) and CWT	List Locking
TLP2	Yes	CRT (no sharing) and CWT	List Locking
RTLP	No	non-atomic: CRT and CWT	List Locking
RBLP	No	non-atomic: CRT and CWT	Block Locking

CRT= concurrent read transactions PRT= read transaction in parallel
CWT= concurrent write transactions PWT= write transaction in parallel

Table 1: Strategies compared against the BP strategy.

We also explored a couple of strategies which relax serializability requirements. The first one, which is denoted as "*Relaxed-Term-Level-Parallelism (RTLP)*", is similar to the TLP strategy but it does not force threads to acquire all list locks before processing a transaction. This removes the atomic commitment of transactions and thereby increases concurrency. A posting list is processed as soon as the respective lock is granted and it is immediately released after processing the list. Our second approach, which is denoted as "*Relaxed-Block-Level-Parallelism (RBLP)*", uses a similar strategy to RTLP but it goes further on concurrency boosting by performing locking at posting list block level. The rationale of this approach comes from the fact that posting lists can significantly differ from each other in size and thereby it is desirable to let two or more threads work on different blocks of the same (large) list concurrently.

Notice that the total number of terms (posting lists) is of the order of tens of millions and the total number of posting list blocks is much more than that. Declaring an array of locks in each case is a waste of resources that may lead to runtime overheads in the Posix threads library. During the lifetime of a query the number of concurrent queries can be of the order of thousands and so is the number of different terms. Thus we use a hash table to map either term IDs or block IDs to a reduced number of lock variables, where the table size is large enough to minimize entry collisions.

To investigate the best that optimistic protocols or strategies based on the use of transactional memory can do we performed the following idealized experiment. We removed from our `C++` code all lock instructions and executed only read transactions. The results were similar (at most 4% faster) to those achieved by the RTLP strategy showing that the differences in performance comes from issues such as processor cache locality and load balance. Notice that all strategies use the same `C++` base code where the only differences among them are related to the specific points in which we introduce the respective locks or barriers.

Experimental setting. The experiments were performed in a dedicated cluster node equipped with two *Intel's Quad-Xeon* processors which provide a total of 8 cores. We assume that all transactions find all posting lists cached in main memory and query/doc-

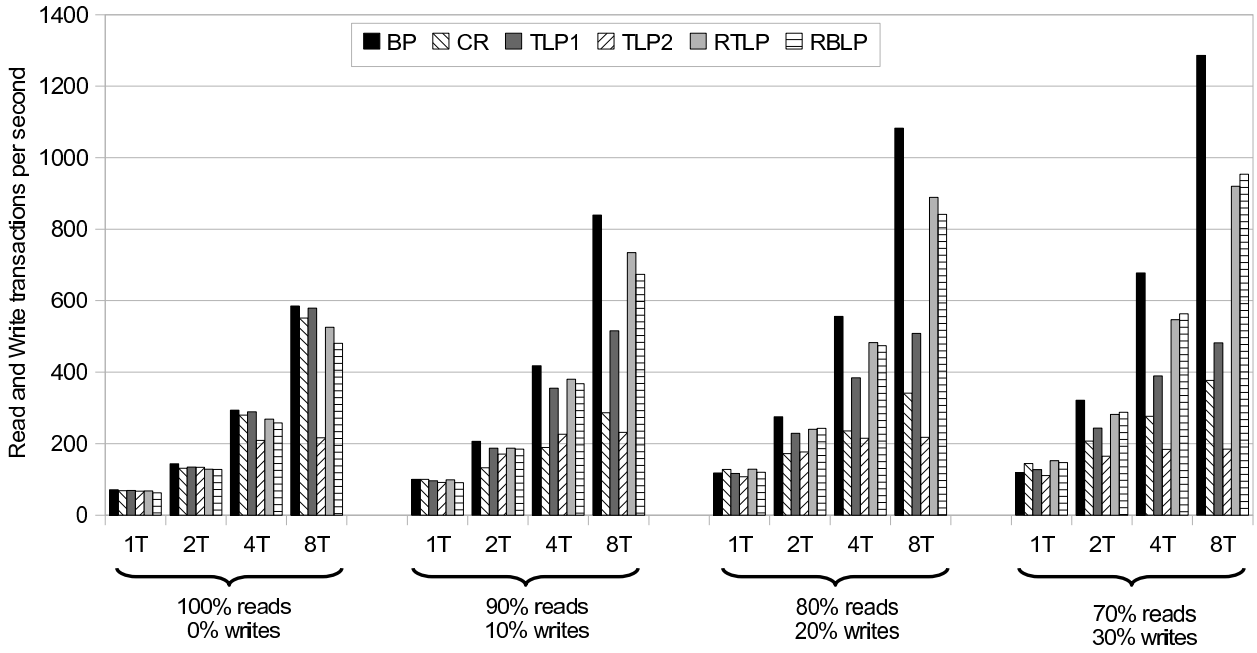


Figure 6: Throughput (R/W transactions per second) achieved by the strategies for different rates of index updates.

ument traffic is so high that cores are busy all the time. Thus below we show peak throughput results which is the relevant issue to study in our application domain. When traffic is low enough, studying comparative performance is not relevant provided that strategy response time for each transaction is below a given upper limit.

The posting lists were constructed from a large sample of the UK Web, and read transactions were generated from a 16M query log containing queries submitted by actual users of the Yahoo! UK search engine. Each query has, on the average, 2.37 terms. This is our baseline workload. To support write transactions, small documents were extracted from the Web sample. To explore different scenarios we constructed workloads where adjacent (in time) transactions have higher chances of accessing the same posting lists.

Intuitively, the larger the block size, the coarser the parallelism for strategies like BP, but smaller blocks tend to improve data locality in processor caches so a trade-off is in place. In the experiments reported below we used for each strategy the block size that delivered the best performance.

Throughput of read/write transactions. Figure 6 shows the throughput achieved when the workload contain read and write transactions. These are results for disjunctive queries. Each set of bars is for executions using 1, 2, 4 and 8 threads (1T, ..., 8T). In each case, we injected a different proportion of read and write transactions whilst keeping constant the total number of processed transactions. Also for write transactions we randomly injected document update and insert operations. Overall, the figure shows that the higher the rate of write transactions, the higher the throughput. Even though write transaction needs to update hundreds or even thousands of posting lists, their overall cost is smaller than the cost of processing queries.

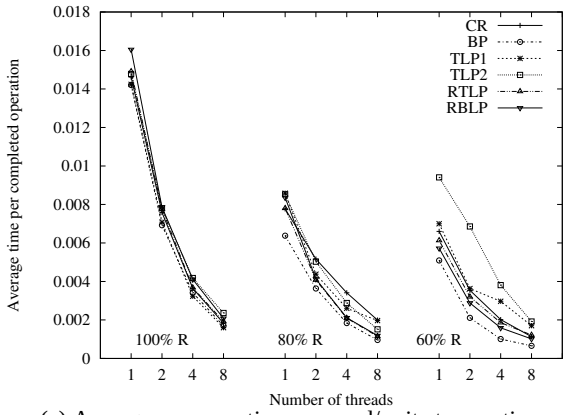
The results of Figure 6 show two relevant facts for the BP strategy. First, the figure shows that the BP strategy scales up efficiently with the number of threads (throughput doubles as the number of threads is doubled) independently of the rate of write transactions injected in the experiments. BP outperforms all others for large number of threads. Only RTLP and RBLP achieve competitive performance but it comes at the expense of destroying transaction atomicity and serializability. Secondly, BP achieves a fairly similar performance to the CR strategy for the case of read only transactions (100% reads). This represents conventional search engines that do not update their indexes in an on-line manner and assign one concurrent thread per active query. The results indicate that either strategy can be used for this case but the key advantage of using BP is that the application cache administration is more efficient since the type of computation performed on the data structure is similar to the one generated by the read/write transactions. Basically, in steady state, one big read that is immediately followed by one big write which, as the figure shows, are better served by employing relaxed bulk-synchronous parallelism.

CR and TLP1 only provide satisfactory results for low write transaction rates since, only in this case, threads are able to find enough read transactions to process them concurrently. TLP2 does not performs well since queries that include popular terms occasionally cause substantial delays to subsequent queries containing the same terms (a popular term has a posting list of large size which leads to a large query processing time). We also experimented with traces producing a very large degree of R/W conflicts and the overall trend remained the same showing that the BP strategy is robust.

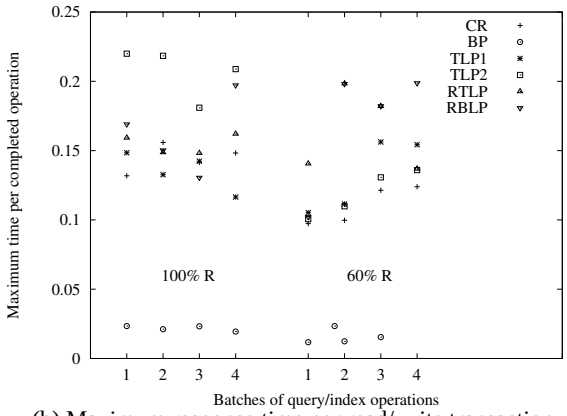
Similar trend to Figure 6, though with a better performance of BP over the other strategies for high write rates, was observed for read transactions composed of conjunctive queries. We observed an average improvement of 30% which is expected since these queries are faster than disjunctive ones as ranking is performed upon shorter lists resulting from posting list intersections.

Response times of individual transactions. Figure 7.a shows the average response time per read/write transaction whereas Figure 7.b shows the maximum response time observed every 1000 processed read/write transactions. The BP strategy achieves the smallest times in each case.

Evaluating the priority queue. In the following we present performance results for the local minima priority queue. In Table 2 we show the results in two main sections. The first one contains data for a case where the average hit ratio in the cache is 70% and the second section is a case where the ratio decreases to 30%. The ta-



(a) Average response time per read/write transaction.



(b) Maximum response time per read/write transaction observed in batches of 1,000 transactions.

Figure 7: Read/write transaction times in seconds.

ble columns are as follows. R represents the number of posting list block IDs stored in fat nodes. T is the sequential time we obtained for the different values of R. Columns $S2$, $S4$, $S8$ are the speed-ups obtained with 2, 4 and 8 threads respectively. Column C contains the fraction of updates performed on the CBT over the total number of accesses to fat nodes. The columns $C4$, $C5$ and $C6$ represent the fraction of times the sequential algorithm, whilst traversing a posting list for a given term, (**i**) finds all of the blocks stored in the cache ($C4$), (**ii**) finds some of the blocks in the cache ($C5$) and (**iii**) finds no blocks ($C6$).

The results show that there is an optimal size of fat node that leads to a small running time. This depends on the ratio of cache size to inverted index size. The speed-ups achieved with 2, 4 and 8 threads are reasonable whereas the values of C indicate that the updates on the tree are not frequent which has a positive effect in thread contention for accessing the tree. The columns $C4$, $C5$ and $C6$ show that this is because most posting lists are allocated in contiguous regions of memory, namely they are orderly placed in fat nodes using the full space provided by these nodes.

Speed-ups with a click-through work-load. To further illustrate the potential use of the BP strategy in a less conventional setting we implemented a click-through search node. Our clicks engine has an inverted file that indexes URLs clicked by users in previous actual queries submitted to the Web search engine. The vocabulary table of that inverted file is formed by the query terms and the associated postings lists contain references to the URLs clicked by users together with other data used for ranking. Furthermore, we use double inverted indexing so that from terms we can

70% cache hits

R	T	$S2$	$S4$	$S8$	C	$C4$	$C5$	$C6$
8	110	1.05	1.61	2.43	0.12	0.63	0.01	0.34
16	91	1.35	2.32	3.94	0.06	0.63	0.01	0.34
32	119	1.71	3.11	4.89	0.03	0.64	0.01	0.34
64	250	1.94	3.06	5.84	0.01	0.64	0.01	0.34

30% cache hits

R	T	$S2$	$S4$	$S8$	C	$C4$	$C5$	$C6$
8	94	1.00	1.46	2.20	0.12	0.29	0.07	0.63
16	96	1.39	2.33	4.10	0.06	0.30	0.01	0.68
32	80	1.64	2.60	4.69	0.03	0.32	0.03	0.63
64	131	1.43	2.93	5.58	0.02	0.33	0.02	0.63

Table 2: Results for the priority queue

reach clicked URLs and vice-versa. A key operation is to start from the query terms to reach a set of clicked URLs, then these URLs are used to get a new set of terms from the second inverted file and from these terms get more URLs. The resulting sets of terms and URLs are then operated each other to generate a list of ranked URLs.

Read/write conflicts take place when it is necessary to include in the inverted indexes the effects of clicked URLs made by recent past users. As new clicked URLs and their respective queries arrive to the search node, it is necessary to detect if the clicked URLs are already being indexed. If so, the clicks count of the respective URLs must be increased and the item promoted to the front of the posting lists associated with the query terms.

Query traces have been built using our UK query-log which has been used to build synthetic click-through traces using different user behavior models. Obviously, users do not click on links at random, but make an informed choice that depends on many factors, but for the focus of this paper, we believe that a random choice model is enough to obtain useful insights on the comparative performance of the strategies tested.

We define speed-up as the ratio total running time with a single thread where all thread synchronization primitives have been removed from the code, to total running time achieved with more than one thread. Figure 8 shows the speed-ups of the different strategies. Again, BP is the strategy that scales the best in this work-load.

Finally, Figure 9 shows running times of individual query and index-update operations. First, the curves in Figure 9 [top] show that average running time per operation tends to be very similar each other across strategies. They are smaller for 5 clicks than for 1 click per user search indicating that index-update operations are faster than query operations, and the trace executed in that case is populated by more of these faster operations. Except for the BP strategy, all other strategies basically assign one thread to process sequentially the query/index-update operation. One would expect BP to outperform all others in this case because it uses all threads to process each query/index-update operation. However, BP has the burden of barrier synchronizing the threads in order to start with the next operation, and the results show that this cost is significant. On the other hand, the points in Figure 9 [bottom] clearly show that all other strategies tend to consume a significant amount of time for some operations which is an indication that they suffer, from time to time, from long delays due to lock contention among the active threads. The BP strategy does not suffer from this problem because it simply processes one operation at a time and, while it uses locks for implementing the oblivious barrier synchronization, the threads do not have to compete each other to acquire locks at the end of each operation. This explains the better performance of BP with respect to the relevant performance metric for our case, namely query/index-update throughput.

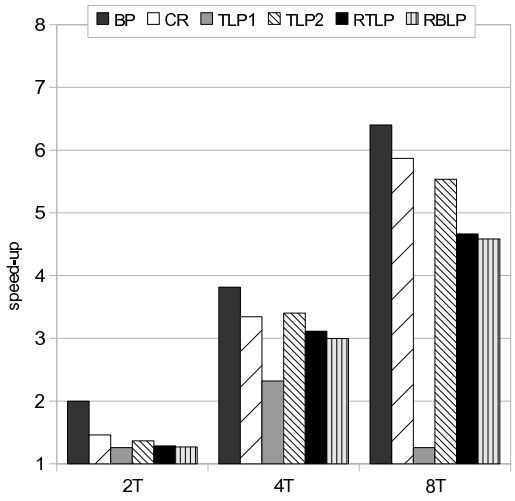


Figure 8: Speed-ups of the different approaches for the click-through workload and assuming that users click on 5 links per search on average. Results for 2, 4 and 8 threads (T).

5. CONCLUDING REMARKS

We have proposed a BP strategy to process read/write transactions in Web search nodes operating at high query traffic on multi-core processors. It outperforms alternative strategies based on previous approaches to concurrency control and it can be accommodated to different design goals. For instance, on demand, the BP strategy can easily apply round-robin processing and approximated answers to queries in situations of drastic peaks in query traffic. Here document insertion/update operations may be delayed until the peak vanishes away. The comparative evaluation shows that the BP strategy achieves even better performance than the strategies that renounce to transaction atomicity and serializability (i.e., RTLP and RBLP). A salient feature of the BP strategy is the use of an efficient oblivious barrier synchronization operation we have devised to prevent from R/W conflicts.

We have also proposed a priority queue (PQ) which is useful to efficiently administer cached posting list blocks. Unlike standard concurrent PQs and convenient for efficient performance, the design of the proposed PQ promotes large data transfers from disk to contiguous regions of main memory and causes very low contention of threads. This is key to efficient transaction processing. The integration into the BP strategy is straightforward as its design is based on the same BP concept of single operations performed in parallel by using all of the currently available threads.

Acknowledgments. This work has been supported by Spanish projects CICYT-TIN 2005/5619 and Ingenio 2010 Consolider CSD-00C-2007-20811, and Chilean project FONDECYT 1060776.

6. REFERENCES

[1] C. Bonacic, M. Marin, C. Garcia, M. Prieto and F. Tirado. Exploiting Hybrid Parallelism in Web Search Engines. In *Euro-Par*, 2008.

[2] Hadoop Web site. http://hadoop.apache.org.

[3] S. Ding, J. He, H. Yan and T. Suel. Using graphics processors for IR query processing. In *WWW*, 2009.

[4] K. Dragicevic and D. Bauer. A survey of concurrent priority queue algorithms. In *IPDPS*, 2008.

[5] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of Web search engines: caching and prefetching query results by exploiting historical usage data. ACM TOIS, 24(1):51-78, 2006.

[6] Q. Gan, T. Suel. Improved techniques for result caching in Web search engines. In *WWW*, 2009.

[7] E. Koskinen, M. Parkinson and M. Herlihy. Coarse-Grained Transactions. In *POPL*, 2010.

[8] M. Marin, G.V. Costa, C. Bonacic, R. Baeza-Yates. Sync/Async parallel search for the efficient design and construction of Web search engines. In *Parallel Computing* 36(4): 153-168, 2010.

[9] M. Marin, R. Paredes and C. Bonacic, High-performance priority queues for parallel crawlers, In *WIDM*, 2008.

[10] M. Marin, C. Bonacic, V. Gil-Costa, C. Gomez. A Search Engine Accepting On-Line Updates. In *Euro-Par*, 2007.

[11] U. F. Minhas, J. Yadav, A. Aboulnaga and K. Salem. Database systems on virtual machines: How Much do you lose?. In *SMDB*, 2008.

[12] W. Moffat, J. Webber, Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, Aug. 2007.

[13] B. Weihl. Computing at scale: challenges and opportunities. In *Google Faculty Summit*, 2008 (available from YouTube).

[14] L. G. Valiant. A Bridging Model for Parallel Computation *Comm. ACM*, 33(8): 103-111, 1990

[15] L. G. Valiant. A Bridging Model for Multi-core Computing. In *ESA*, 2008.

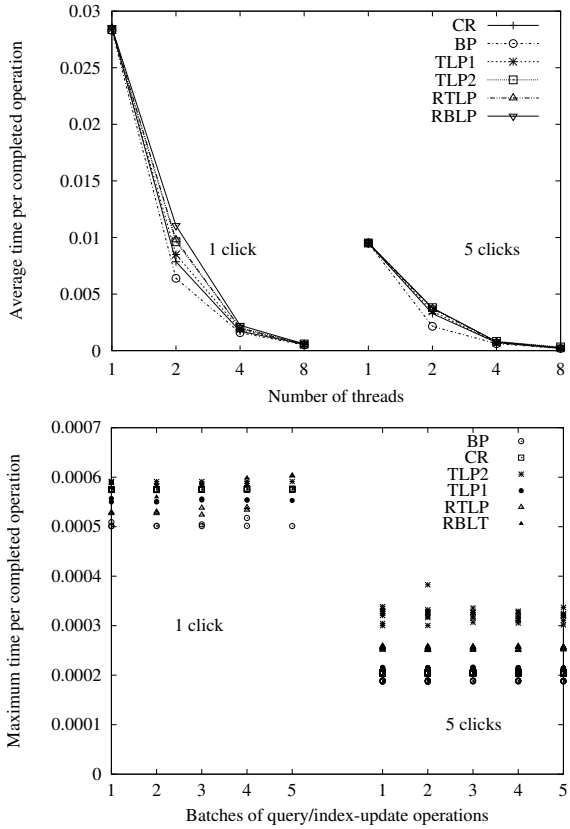


Figure 9: Individual query times. The figure at the top shows results for average running time per query/update-index operation for 1, 2, 4 and 8 threads. The figure at the bottom shows the maximum running time observed every 1,000 read/write transactions for 8 threads.