Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 1. pp. 361–374.

Concurrent implementation of caches

Antal Tátrai, Balázs Dezső, István Fekete

Eötvös Loránd University, Faculty of Informatics
e-mail:{tatraian,deba,fekete}@inf.elte.hu

Abstract

Caches are typically used in the practice for making the respond time faster when of accessing remote data. Every case when caches are used, local copies of the slow data are created, and if we want to access them again, it is just enough to find in the local copy. A typical field of the application of caches is the domain name resolution. The user usually only knows the name of a server, but the internet protocol uses the IP number for identification. So the user has to ask the IP address of the desired server. He/she will get the answer mostly from a local cache which stores the data from a remote server.

Because the cache only contains a copy, it can differ from the original data. Time limited cache resolve this problem with a **time to live** (**TTL**) value attached to the original data records. When an element's TTL expired, it has to be updated or removed from the cache.

Traditional solutions consist a data structure (e.g. hash tables, search trees) and a read/write lock for mutual exclusion in concurrent way. Several thread can read the cache in parallel, but just one thread can write into it. Accordingly when a thread writes into the cache no other thread can read or write. Because the time-limited caches' records have TTL value, periodically has to clear or update the expired elements. These actions are performed by a dedicated thread.

In the paper the authors show caches improved with several locks and additional structures for helping the cleaning action. Several possible concurrent implementations will be described, and compared by their theoretical and experimental running times, furthermore in following sections will be described solutions based on chained hash tables. Some parts of our improvements are reusable in the case of other data structures.

Keywords: time limited cache, chained hash table, concurrent algorithm

1. Traditional solutions

There are several solutions for time-limited caches. In this section we give a possible concurrent cache implementation and mention some other potential so-

lution. Furthermore we describe a special kind of threads use the cache, which maintains expired elements.

Each cache has a database which is stored either on the disk or in the memory. When the data are stored on disk, usually a database system is used for searching. Since the speed of this solution greatly depends on the implementation of the database system, these caches are not examine in the present paper. In certain cases, the database is stored in the memory. In this case, some data can be written on the disk, but it seems to be practical to store the key values in the memory for faster search. If the requested record is on the disk, it can be read, and this solution does not make the search slower, but the getting of proper element. The last solution is the one where the whole database is in the memory. Subsequently, these two solutions are merged into one where the whole database is in the memory. If the database is in the memory more data structures can be used: search trees, chained or open addressing hash tables. In this paper we use the chained hash table ([1], [2]) for data structure and suppose that the whole database is in memory.

A simply way of implementing a concurrent cache is based on threads and read/write lock. Threads can read or write the database and read/write lock is responsible for mutual exclusion.

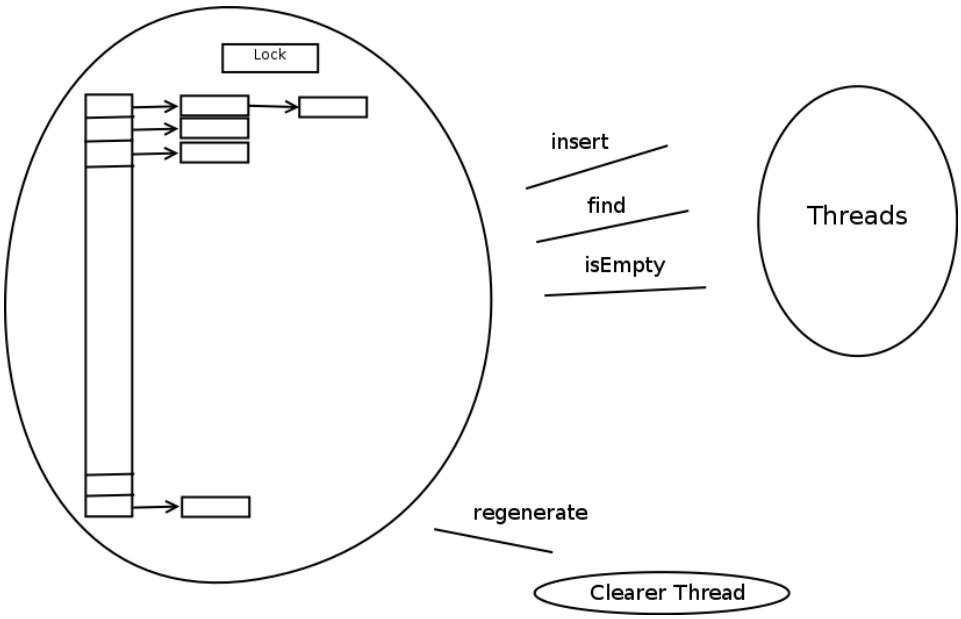

Figure 1: A cache implementation

Read/write lock is a gate for the data structure, threads can access the structure by closing this lock in read or in write mode. If a thread closes the lock for reading no other thread can secure this lock for writing but several threads can get read

access for the structure in parallel. If a thread tries to close the lock for writing it has to wait the reader threads leaving the critical (mutual excluded) area (open the lock), and after that it can use the structure and during its action no other thread can read or write the cache. The thread implementations use queues for performing this functionality.

It is a special thread that maintains the data as it cleans up or updates expired records. This thread will be called: **cleaner thread**. This thread periodically reads the database, and removes or refreshes the records. During action only this thread can reach the data so all read and write accesses have to wait.

It is hard to estimate the exact running times of the data structure, because sometimes only one thread can run and every other should wait. This explains why the waiting time can be much longer than the searching time.

2. Improvements for concurrent caches

Several parts of caches can be improved: faster data structures can be created, a better locking mechanism can be figured out or the clearer thread can be made faster. This paper will describe the last two improvements.

This paper we will use the term **primary structure** when it refers to the data structure of the cache, and will call **secondary structure** the other structures, which are responsible for efficient data updating. When these solutions were implemented, the hashed value of the key was also in the record. The corresponding bucket was calculated by a modulation with the size of the backbone of the hash table. This is useful in resizing, because the new index can be recalculated with just a modulation during each reinsertion.

2.1. Locking mechanism

In this subsection the working mechanism of a chained hash table with read/write locks, designed for concurrent applications will be examined, and yield a better performance but larger memory requirement version.

In a chained hash table only three kinds of operation may occur: reading a record from a bucket (represented by linked lists); insertion of a new record into the hash table when it fits in the proper list; and the reorganisation of the data structure (resizing the backbone of the hash table, and reinserting the records) when the length of the list (where record was inserted) gets too long or other criteria came true. The cleaner thread never reorganises the cache, so its functionality is the same as in the second type of operation above (namely just erase from a list, and do not reorganise the structure).

In order to make write access parallel, we introduce additional locks for the hash table. The original lock is kept, and it is called **resize lock** from now on. Furthermore, extra locks are added for the buckets of the chained hash table. Now let us examine, how the basic algorithms work. When a thread tries to access the hash table for reading, it calculates the hash value, and closes the resize lock and the

corresponding list's lock for reading. If the thread found the corresponding record, or it scanned all records in the bucket, it frees the locks in reverse order. In the case of writing this algorithm becomes more complicated. When a writing occurs the thread secures the resize lock for reading and closes the corresponding list's lock for writing. Consequently, the other threads can read or write in the other buckets of the hash table. The unlocking mechanism works as above, in reverse order. If an inserting occurs in a bucket that has too many elements, (or there is any other reason to resize the hash table), an additional step will be executed as follows: after the writing frees its locks, it blocks the whole hash table by closing the resize lock for writing. In this case, no other thread can reach the data structure and the resizer thread can make its actions undisturbed. After that, it unlocks the whole structure.

This solution needs more memory than the simple one, but enables concurrent reading and writing actions or parallel writings in the cache. Anotherdrawback of this version is that the whole hash table implementation has to be reimplemented instead of combining a built-in hash table and lock solution.

2.2. Cleaning thread improvement

The main approach of clearing improvments is the arrangement of data by two values: the key and the expiration time of the data. The idea of multiple indexes is widespread in informatics. In this case, we have simpler data structure, because we do not lookup records simultaneously by both keys. Each thread except the cleaner thread only uses the original key for searching, and the cleaner thread looks up elements in the cache by the expiration time.

The authors redesigned the hash table and added a new data structure to it. This **secondary structure** works like an index structure to the records. It simply stores the expiration times and pointers to the original records in the hash table. This secondary structure can be implemented in many different ways. These will be described in the following sections. It is important to note that insertion into the cache takes more time and a new lock is added to the secondary structure, because during the insertion a new item is added to it, too. This lock will be used only during writing, but it will not affect reading.

2.2.1. Binary heap

The secondary structure is responsible for the efficient retrieval of expired items in the primary structure. The first obvious observation is that an arbitrary priority queue implementation could be the base of the secondary data structure. Items should be stored with the expiration time as priority and at each cleaning, each item with lower priority value than the current time, should be erased.

The authors studied the well-known **binary heap** as one of secondary structure implementations. The worst case time complexity of the insertion into and erasure from the heap is $O(\log(n))$ where n is the number of the items in the heap, but cleaner thread can check the existing of expired element in $\Theta(1)$ time, furthermore

can find the expired elements in $O(en \log(n))$ time, where en is the number of expired elements.

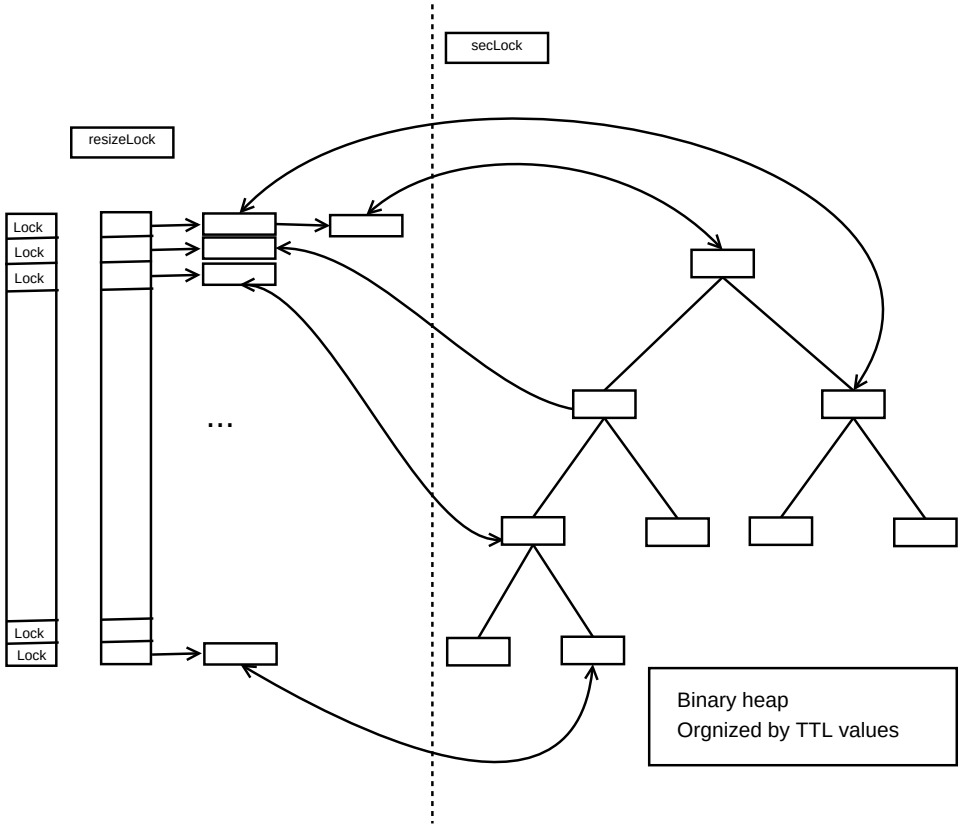

Figure 2: A cache with binary heap as secondary structure

2.2.2. Bucket list

Sometimes it is advantageous if the TTL values of the new elements are mostly constant. In this case can be used a special data structure, which will be called **bucket list**. This list contains time buckets depending on the clearing period of the cleaner thread. The time is divided into periods, and each bucket represents a period. When a new element arrives, it will be inserted into the proper bucket. For example, if a new element's TTL is 5 seconds, it is put into the bucket containing the elements with TTL values that supply the next assumption: $ct_k < et \leqslant ct_{k+1}$, where ct_i is the time of the i^{th} clearing and et is the expiration time of the element (in this case $et = now + 5\ seconds$). During an insertion, usually the new element

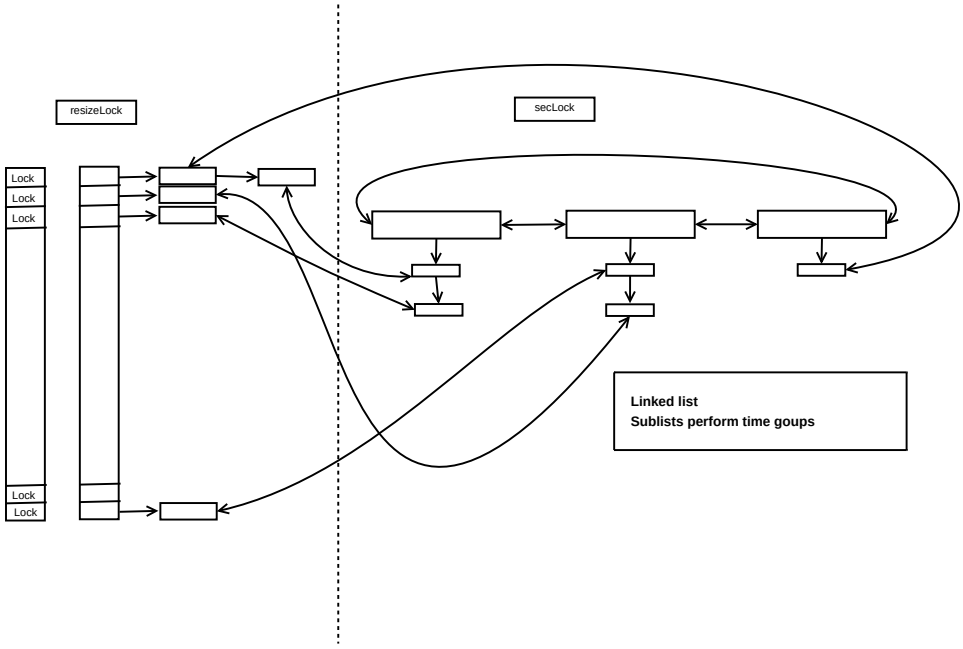

Figure 3: A cache with bucket list as secondary structure

has to be inserted into the last bucket, if we assume that the TTL values are mostly constant. In this case, the elements that will expire in the next clearing, are all in one bucket at the begining of the list. As a result the cleaning thread just easily reads the first bucket's border time, and if it is less than the current time, it updates or removes the referenced elements. Note that we do not need to store the empty buckets.

The running time of this structure depends on the assumption. As you can see in Figure 3, the buckets are represented by a simple one directional linked list. In this case, insertion into a bucket is a $\Theta(1)$ time algorithm. If the TTL values are mostly constant, the insertion to it is $\Omega(1)$. In the worst case this action take $\Theta(tb)$ asymptotic running time, where tb is the number of non-empty time buckets.

The clearing always has $\Theta(en)$ running cost, where en is the number of expired elements, because the clearer thread can find the expired elements by walking through the last bucket.

2.2.3. Unbalanced binary tree

This data structure is an **unbalanced binary tree** with a pointer to the smallest and the biggest element. The proper place to insert a new node is always searched from the biggest item in the tree. The searching is performed as a

walk-down to the first smaller node on the path from the biggest to the root, and afterwards a regular binary tree searching is done from the found node. It should be voted that if the current item is bigger than the previous biggest value in the tree, then the searching is performed in constant time, while an other hand the expected value of the insertion time for a random sequence is $O(\log(n))$. Erasure has amortized constant time complexity, but it could be time consuming in particular cases, because it should search the new smallest item in the heap.

The main motivation of this data structure is to combine the advantages of the previous two data structures. The bucket list gives $O(1)$ time complexity for the insertion in the best case but in the worst case it could be $O(tb) = O(n)$. Erasure from the binary heap is $O(en \log(n))$. The unbalanced binary tree implementation gives $O(en \log(n))$ average time complexity for a random permutation of expiration times and $O(1)$ for sorted expiration times.

2.2.4. $(2, 3)$-tree

This data structure provides as worst case performance the result of the previous data structure's average case behaviour, at least in amortised manner. The authors can generalise an adaptive sorting mechanism to get better upper bound on the time complexity of the insertion and erasure. First of all, the inversion number of k item should be defined in a sequence s as $I_k^s = \sum \chi(s_i > k)$. Now the performance of data structure operations: the amortised complexity of the insertion is $O(\log(I_k^s))$ and the erasure is $O(1)$ where k is the currently inserted item's expiration time and s is the sequence of the currently existing expiration times in the cache.

It is a well-known result that if we just insert into a $(2, 3)$-tree and we do not erase [3], then the number of the node-cuts will be $O(n)$. Consequently, a sequence can be sorted $O(n \log(\frac{\sum I_k}{n}))$ time with the following algorithm: Insert the items of the sequence into a $(2, 3)$-tree, always perform the search from the greatest node and then iterate on the leaf nodes. It will be proved that similar performance could be achived for the secondary structure problem.

Let us denote the path from the root to the smallest item as P-nodes and the remaining nodes as T-nodes (see Figure 4). The number of the 2-degree P-nodes plus the number of 3-degree T-nodes is used as amortization function. The insertion first searches the proper place in $O(\log(I_k^s))$ time, if the node-cuts of insertion do not reach the P-nodes then each node-cut will decrease the amortization function, since it splits a 3-degree T-node into two 2-degree T-nodes. Alternatively, if a node-cut reaches a P-node, then the following holds for the inversion number of the new node: $\log(I_k^s) = O(\log(n))$, consequently the node-cuts and the change of the amortization value, which could be bound with $O(\log(n))$, is appropiate for the $O(\log(I_k^s))$ insertion cost.

The minimum erasure's time complexity could be similarly bound with $O(1)$ const. The erasure makes node-merges and each node-merge decrease the amortization value by one, since it decreases the number of 2-degree P-nodes. In addition, it does not increase the number of 3-degree T-nodes, thus the total amortized cost is $O(1)$. Of course, the pointer to the minimum and maximum item could be held

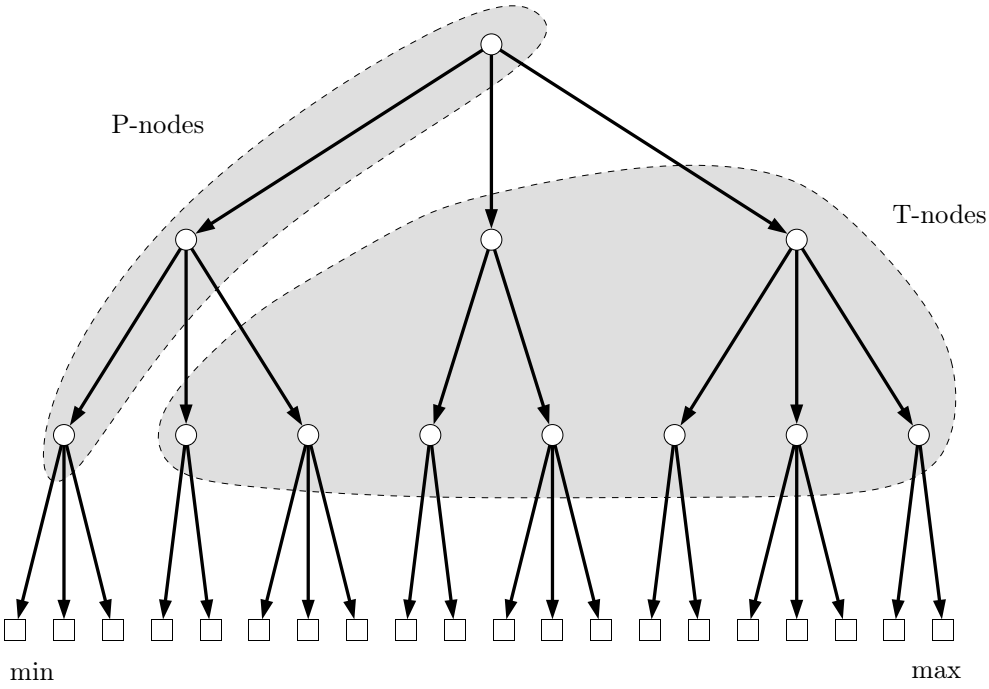

Figure 4: The $(2, 3)$-tree based secondary structure

in constant time for both operation.

The current data structure gives better asymptotical theoretical time complexity than the previous implementations, but it has quite complicated structure and worse performance in real world cache implementations.

3. Measure

3.1. Methodology

For trying out the proposition the structures above were implemented. A chained hash table was written, and this basic structure was used for all cache types. The `PThread` library and it is default implementation was used on the target architecture for locking and starting threads. Furthermore an additional type was implemented that just consists of the hash table and the locks for each bucket. This solution is between the simple and the two-structures version. In the Result section the following terms will be used: **Simple** will be the name of the simple solution, we will name **Multi** the solution described before and each two structures cache type name will be the name of it is second structure.

Metrics that are able to measure the performance of implemented caches had to be made. The successful readings, writings and the ended clearing periods were counted. The test programs earn these data from every thread including the clearer thread. Finally all test programs make a summary of the successful readings and writings that were performed in the system during the test.

Because we have five different cache types (Simple, Multi, Bucket List, Binary Search Tree and Binary Heap), each type has its own test program. During a measure all test programs were run in sequential order, and this was performed five times. This results were collected, and the average values were counted. Until these tests terminated no other program run on the test machines.

Each test program made the same functionality, described bellow:
Test program algorithm:

1. Read the input file
2. Create an empty cache and start the clearer thread
3. Start the threads
4. Join with the threads
5. Collect data from threads (including clearer thread)
6. Make summary from collected data.

Algorithm of a thread:

1. While the running time is not over
2. Generate random number and choose the action (reading or writing)
3. Perform the chosen action
4. Increase the relevant counter (the counter is the object of a thread,so no mutual exclusion is needed).
5. End while

Several parameters for the tests can be set, such as

- size of input file
- running time of a test (in seconds)
- the cleaning thread running period (in seconds)
- number of the threads
- rate of readings versus writings

- `TTL values of a record. It can be a constant value, or a random number with poisson distribution.`

The measure script was run on different machines and operating systems, because the speed of caches hardly depends on the thread implementation. After all we had 4 machines, 3 different architectures and 3 different operating systems which are the next:

- MacBook, **Processor**: Intel Core Duo 2GHz, **Memory**: 1Gb, **Operating system**: Darwin 8, **Compiler**: gcc 4.0.1
- iMac, **Processor**: Intel Core Duo 1.83GHz, **Memory**: 1Gb, **Operating system**: Darwin 8, **Compiler**: gcc 4.0.1
- Pongrac, **Processor**: Intel Pentium 4 3GHz, **Memory**: 1Gb, **Operating system**: FreeBSD 6.1, **Compiler**: gcc 3.4.4
- Lime, **Processor**: 4 pices AMD Dual Core Opteron 2.2GHz **Memory**: 16Gb, **Operating system**: Linux version 2.6.16.27-0.9-smp (SuSE), **Compiler**: gcc 4.1.0

3.2. Results

Aim of our measuring was to collect data about the different data structures performance during scaling the number of input elements, threads or read/write rate. We counted the read and write accesses together, it will call transaction.

As you can see in Figure 5 in the case of Pongrac, the **Simple** was the fastest data structure. The other four implemented version was nearly two tierce slower. On the Lime the **Simple** version was just a bit slower than the others were. Finally on the two Macintosh the **Simple** was much slower then the other four versions.

In the case of Pongrac, the answer is very simply for this result: it has just only one processor, and it is not a Dual Core, it has just a Hiper Threading extension. The other three result present, that the performance of the caches hardly depends on the implementation of the operating system's kernel, and `PThread` library. `Darwin` is based on the `FreeBSD` kernel, but its thread realisation is much faster (its data structures are embedded in char* arrays, no copy during creations, etc.). In the future the authors are going to show the result of iMac. The behaviour of the caches is similar in the other concurrent case too, but much less significant.

In Figure 6, it can be seen that the concurrent versions are not as sensitive to the scaling of input data as the simple solution. Serious answer to this question cannot be given, but maybe the running time of the clearer thread is responsible for the lower transaction number of the simple solution.

It was also measured how the caches work when is increased the number of threads. The result is shown in Figure 7. The number of the successful transactions decreases if the number of threads increases. When a writing (or clearing) occurs in a **Simpe** cache, it blocks all the other threads. If it is slow, the kernel will block it, and other thread can run. However this other thread cannot do anything else

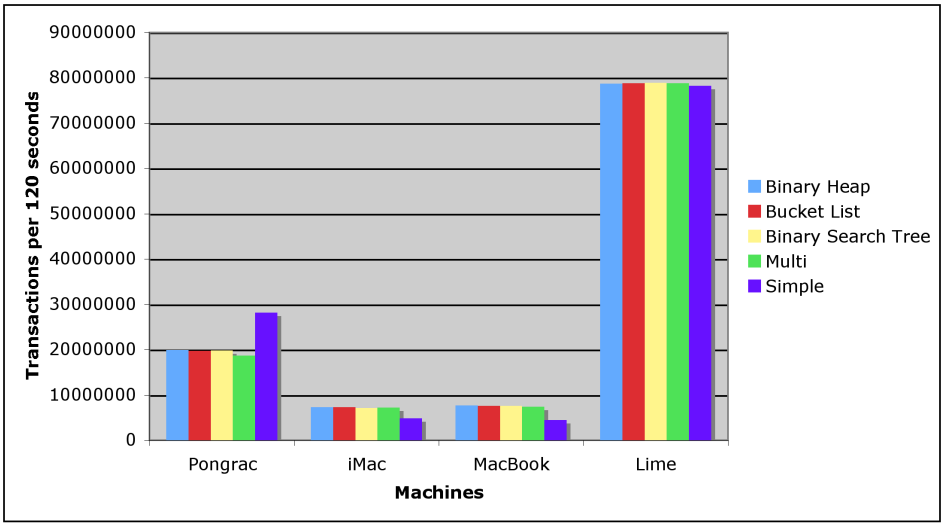

Figure 5: Performaces

just wait for the lock, therefore it will go to sleep, and an other thread will be waked up. The cost of this empty waking is higher if there are more threads. The concurrent version will be faster because it has less writing that locks the whole data structures.

Figure 8 shows that by increasing the rate of writing/reading, the concurrent versions essentially produce the same number of transactions contrast to the simple version which will be slower. This is a logical consequence of the thread switching discussed above. But two other interesting details come from this diagram. The **secondary lock** (the lock of the secondary structure), is not a bottleneck of the performance in the concurrent cases, furthermore the secondary structures does not make significant speed up.

4. Conclusion

This paper gave a review of the simple or traditional solution of the time-limited caches that use only one lock for mutual exclusion. The authors introduced more locks, one for each bucket of a chained hash table and one more for the secondary structures. Additional data structures for increasing the performance of the cleaner thread were described. Four kinds of cache were introduce that have secondary structure (namely: Binary heap, Binary Search Tree, 2-3 Tree and Bucket List), furthermore in the Measuring section a special 5th kind of cache was described which has not secondary structure but it has locks for each bucket of its hash table.

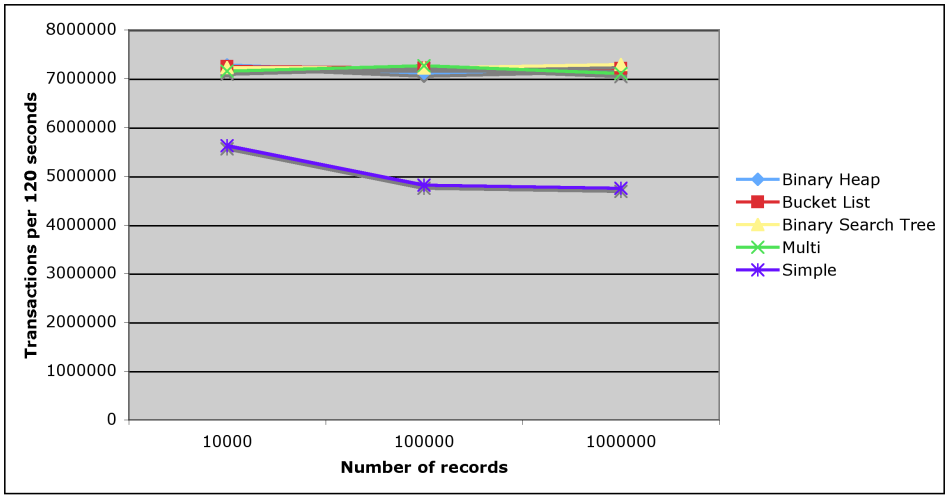

Figure 6: Transactions vs. Size of Cache (iMac)

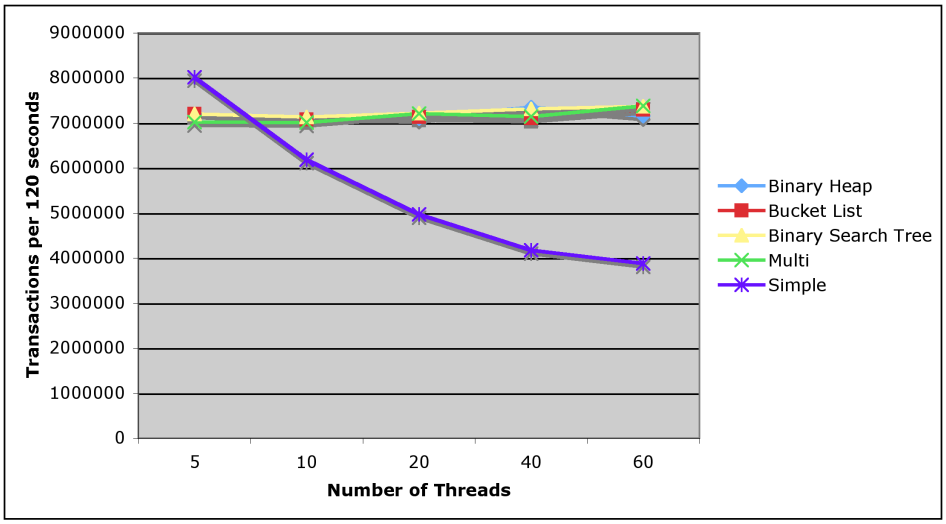

Figure 7: Transactions vs. Number of Threads (iMac)

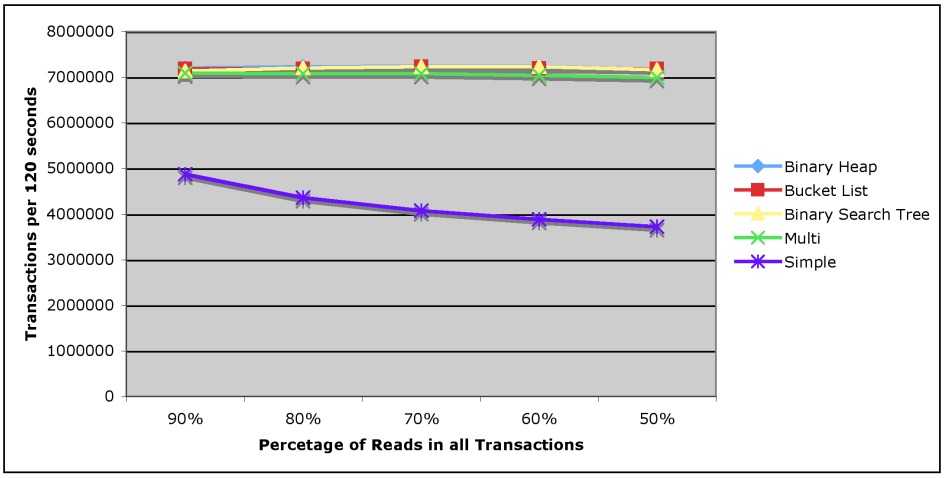

Figure 8: Transactions vs. Read/Write Rate (iMac)

Five type of caches were implemented (Binary Heap, Binary Search Tree, Bucket List, Multi and Simple) with the same hash table implementation as well as the the test programs that measure its performance. These programs were running on four computers.

The results shows that the concurrent versions work only in more than one logical processor systems and the performance firmly depends on thread implementation of the kernel and `PThread` library. Additionally the diagrams showed that concurrent caches are less reactive to the scaling of the number of records, threads and writings' rate, on the order hand secondary structures does not make significant speed up as compared to **Multi** version.

It should be noted that the concurrent versions have much larger memory requirement than the simple solution. In `Darwin` a lock object's size is about 200 bytes. Furthermore this ideas could be reusable in the case of open addressed hashes, but in that case the memory requirement will be larger.

The implementation of the caches and the test programs can be downloaded form the next URL:
`http://people.inf.elte.hu/tatraian/data/concurrent-caches.zip`

Acknowledgements. Thanks to the **Lemon Project** (`https://lemon.cs.elte.hu/`) for the testing opportunity on Lime. Thanks to **Ittzés Péter** for the test opportunity on iMac.

References

[1] Tamassia, R., Cantrill, B., Basic Data Structures, In M. J. Attalah, editor, *Algorithms and Theory of Computation Handbook*, chapter 4, CRC Press, (1998).

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., Introduction to Algorithms (2nd edition), *The MIT Press,* (2001).

[3] Brown, M. R., Tarjan, R. E., Design and analysis of a data structure for representing sorted lists, *Technical Report CS-TR-78-709,* Stanford University, (1978).

Antal Tátrai, Balázs Dezső, István Fekete
Eötvös Loránd University, Faculty of Informatics
H-1117 Budapest
Pázmány Péter sétány 1/c.
Hungary