

ACache: Using Caching to Improve the Performance of Multiple Sequence Alignments

Xun Tu, Kajal T. Claypool and Cindy X. Chen
Computer Science Department, University of Massachusetts Lowell
One University Ave.
Lowell MA 01854
{xtu|kajal|cchen}@cs.uml.edu

Abstract

Multiple sequence alignment represents a class of powerful bioinformatics tools with many uses in computational biology ranging from discovery of characteristic motifs and conserved regions in protein families to improved prediction of secondary and tertiary structure. Today, with rapidly growing data repositories offering scientists significantly more data with which to make better decisions, it is increasingly important to run these multiple alignment calculations as rapidly as possible. However, while several multiple alignment algorithms have been developed, these algorithms remain computationally expensive taking as long as 2 to 3 days for some queries. In this paper, we propose a new caching technique to improve the performance of multiple sequence alignment algorithms. In particular, we propose a nested two level cache hierarchy that provides caching of pairwise alignment results – a computationally expensive subcomponent of the multiple sequence alignment algorithms. A key contribution of our work is the development of two novel cache replacement policies that closely track the scientist’s query patterns over time. We present experimental results that validate the benefits of caching over the repeated computation of the alignments, provide heuristics for determining which alignments would benefit from the caching, and show the effectiveness of the developed cache replacement policies.

1 Introduction

Multiple alignment of protein sequences has become an essential tool for molecular biology. It has traditionally been used to find characteristic motifs and conserved regions in protein families, and in the determination of secondary and tertiary structures of the proteins. With the rapid increase in the number of protein sequences, notably

from the genome sequencing projects, automatic methods of searching protein databases for homologous sequences, followed by the multiple alignment of the top scoring hits is becoming standard practice. These automatic methods involve the alignment of large numbers of potentially divergent sequences from multi-domain proteins. The development of accurate, reliable multiple sequence alignment programs capable of handling these divergent sets is, therefore, of utmost importance and has resulted in numerous strategies ranging from dynamic programming algorithms that provide a mathematically optimal alignment for short sequences; to progressive alignment algorithms, such as ClustalW [26] where the multiple alignments are built up gradually by first aligning the closest sequences and then successively adding the more distant ones.

It is generally accepted that ClustalW and its GUI-based counterpart ClustalX provide the most accurate and reliable results, albeit at the cost of performance taking on the order of hours for the alignment of relatively small number of sequences (30 sequences). To counter this poor performance, numerous techniques ranging from the application of heuristics to parallel implementations to caching have been proposed to speed up the execution time for ClustalW [26, 1, 6, 24, 3, 17, 18, 22, 20, 4]. Of these approaches, *caching* provides the biggest bang for the buck – large speedups in the most cost-effective way. Catalyurek et al. [4] were among the first to present a strategy for caching the alignments produced by ClustalW [26].

While the work by Catalyurek et al. [4] is an important first step, their approach presents a static cache structure that does not adapt to the changing needs of its scientist community. In our work, we address this drawback and present a *dynamic* cache structure that can acclimate to both slight modifications in the query pattern (alignment of an orphan sequence within the same family) as well as to large changes in the query pattern (alignment of divergent sequences that straddle multiple families or patterns that shift from one family to another). In this paper, we

present *ACache*— a nested two-level B+ tree based caching structure that aligns itself to the natural hierarchy of families and superfamilies in the molecular biology domain, and capitalizes on the inherent query patterns¹ in this domain.

A key contribution of our work is the development of two cache replacement policies, *hot key* and *hot key-pair* that track typical query patterns centered around a particular family or superfamily of current interest to the scientist community. We present experimental results that (1) validate the benefits of caching over the repeated computation of the alignments; (2) provide heuristics for determining which alignments would benefit the most from the caching; and (3) show that we can maintain the cache hit ratio between 40% to 60% when using the hot key and hot key-pair replacement policies. It should be noted that our results and approach are easily transferable to other multiple alignment algorithms, as well as to the caching of pairwise alignments in a multi-user environment.

Roadmap: The rest of the paper is organized as follows. Section 2 describes our overall approach, together with the implementation details of our cache hierarchy structure. Section 3 discusses the cache replacement policies, while Section 4 presents an experimental evaluation of our caching approach. Section 5 discusses related work and Section 6 concludes the work.

2 Approach - The *ACache* System

2.1 Overview

The fundamental idea behind any progressive multiple alignment, such as the ClustalW [26] is to generate an initial phylogenetic tree to incrementally construct the multiple sequence alignments. ClustalW [26] (and other progressive alignment algorithms), has three computation stages:

Pairwise Alignment: This step computes a distance matrix based on the alignments for all pairs of sequences. For q sequences, this step requires the computation of $q(q-1)/2$ pairwise alignments. As a result, the execution complexity of this step is $o(q^2l^2)$ for q sequences, each of which has an average length of l ;

Computation of the Guide Tree: This step calculates a phylogenetic tree from the distance matrix. Different techniques can be used to compute this phylogenetic tree. ClustalW [26], for example, uses the neighbor-joining method [25] – a popular method for constructing phylogenetic trees. The computational complexity of this step is $o(q^3)$;

Progressive Alignment: The last “progressive” step computes a series of pairwise alignments using dynamic programming. The goal is to align larger and larger groups of sequences, and the branching order in the guide tree deter-

mines the ordering of the sequence alignments. The computational complexity of this step is $o(ql^2)$.

The computational complexity of the pairwise alignment step is the dominant factor in the total execution time of the progressive alignment method [3, 17, 4]. This was also validated by our preliminary experiments that showed that an upward of 90% of the total execution time of ClustalW [26] was spent in computing the pairwise alignments.

We propose to cache the pairwise alignments computed in the first step of multiple sequence alignment (MSA) to improve the overall performance of the ClustalW [26]. This choice is based on two observations. First, the pairwise alignment is the dominant factor in the overall query execution time. This, minimizing this execution time can result in significant improvements to the overall execution time. Second, the guide tree and the progressively aligned sequences generated in the last two steps are highly dependent on the relationships between the sequences in the query. These intermediate results can be reused if and only if the two queries are exactly the same or have a high degree of similarity. Each pairwise alignment, however, is computed independently and its contribution to the distance matrix is independent of the other pairwise alignments. Thus, pairwise alignments generated by one query have greater potential for reuse than the guide tree and the groups of multiple alignment.

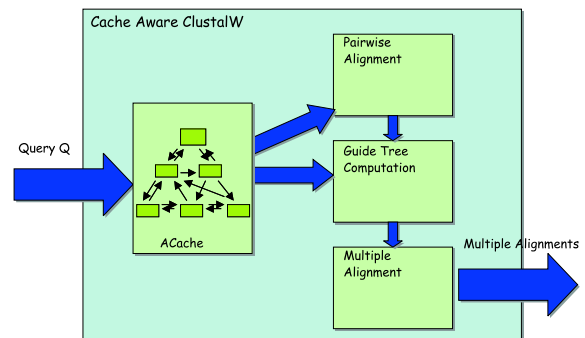


Figure 1. Architecture of a Cache Aware ClustalW.

In our approach, we now propose a *cache-aware* ClustalW [26] (see Figure 1). Here, when a query is received from a user *ACache* is searched to find the pairwise alignments of all sequence pairs specified in the user query. For all sequence pairs found in the *ACache* (a cache hit), the alignments are passed to the second phase of the ClustalW [26] algorithm. For all other sequence pairs, the pairwise alignment is computed, inserted into the *ACache* (replacing other alignments in the cache if the cache is full), and then passed as input to the second phase of MSA. In the following sections, we present the design considerations for

¹Alignment of an orphan sequence to a family of sequences is a typical query pattern for most scientists in this molecular biology domain.

ACache and detail its structure.

2.2 Design Considerations

There are two primary factors that must be considered when constructing a caching structure for pairwise alignments. First, an alignment between two sequences is commutative, that is $\text{align}(S1, S2) = \text{align}(S2, S1)$. For both space and efficiency considerations, duplicate alignments should not be cached. That is, both $\text{align}(S1, S2)$ and $\text{align}(S2, S1)$ should refer to the same cached alignment. Second, sequence alignment can be computed between any two sequences. However, the lengths of these sequences can vary greatly, ranging, for example in Swisprot [2], from the shortest length of 2 amino acids to the longest sequence length of 8797 amino acids. For space and performance considerations, the *keys* used in the cache structure should have the same length, irrespective of the actual lengths of the sequences.

2.3 The *ACache* Structure

The *ACache* structure scales gracefully to a large number of keys, requires no duplicate storage of alignments, and is insensitive to the lengths of the sequences. Furthermore, it is particularly well-suited for the molecular biology domain where sequences naturally fall into a hierarchical family and superfamily structure; and where query patterns typically demonstrate a concentration on the alignment of orphan (or unknown) sequences to sequences within one family.

Figure 2 shows the two-level nested hierarchical tree structure of *ACache*. The first level of the *ACache* hierarchy is a regular B+ tree whose leaf pages contain pointers to the roots of second-level trees that cache the actual alignment data. The second level tree is represented by a novel *B-Link* tree structure specifically designed for facilitating fast cache replacement, to represent the set of second-level trees. A B-Link tree is an extended B+ tree that has an additional *parent* pointer at every node (intermediate and leaf), and a *right sibling* pointer for each intermediate node (see Figure 2). The leaf pages of the second-level B-Link tree contain the actual alignments, where each alignment is stored as a pair of strings that represent the aligned sequences in the FASTA format [13], and an integer score that represents the overall score of the pairwise alignment. This string based storage structure at the leaf level is sufficient for caching the intermediate results for the multiple sequence alignment algorithms, but can be enhanced in the future to provide more efficient querying over the aligned sequences themselves.

Unique sequence identifiers are used as keys for the *ACache* rather than the sequences themselves. This is motivated by two primary reasons. First, while B+ trees provide efficient search and insert operations, they can not easily handle unbounded-length strings such as the variable

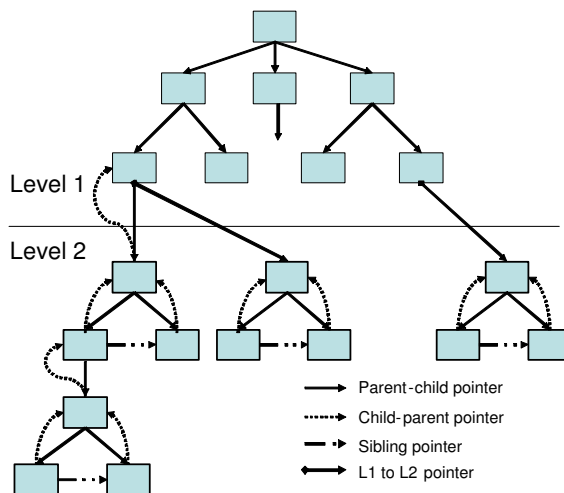


Figure 2. The Two-Level Hierarchical Structure of the *ACache*.

length sequences [10]. Second, the memory requirements for storing these sequences, while not prohibitive, is significant enough to impede the performance of the search operation. In our work, we use the MD5 hash algorithm to generate unique sequence identifiers. The MD5 algorithm takes as input arbitrary length data and produces as output a 128-bit “fingerprint” or “message digest”. For a given sequence, the derived 128-bit fingerprint is used as the unique identifier for that sequence, and hence as the key in *ACache*.

Given a sequence pair, $S1$ and $S2$, the first level B+ tree is keyed on the smaller sequence identifier, while the second level B-Link tree is keyed on the larger of the two sequence identifiers – ensuring that no duplicate alignments are stored in *ACache*. Hence, a B-Link tree for a sequence $S1$ will contain results from the alignment of $S1$ with some other sequences S_i , such that the sequence identifiers of all S_i are greater than the sequence identifier of $S1$.

2.4 Searching

The search algorithm employed to search and retrieve the alignment between two given sequences, $S1$ and $S2$, begins with a pre-processing step that computes the MD5 keys for the two given sequences, $S1$ and $S2$. The smaller of the two sequence identifiers, say $S1$, is used as the search key for the first-level B+ tree, and the search follows the standard B+ tree search algorithm. The first phase of the search algorithm terminates with the identification of the root of the second-level B-Link tree that contains the cached alignments between $S1$ and all other sequences whose identifiers are greater than that of $S1$. The search process terminates if no B-Link tree root is found, implying that there are no cached alignments for sequence $S1$.

In the second phase, the search algorithm traverses through the B-Link tree using the second sequence identifier, S_2 , as the key. The search algorithm terminates (1) when the right leaf page, that is the leaf page that contains the alignment of S_1 with S_2 , is identified; or (2) when the sequence identifier of S_2 is not found in the B-Link tree, indicating that there is no cached alignment for the sequence pair S_1 and S_2 .

2.5 Updates

Updates, insertions and deletions, like searches, can be performed efficiently in *ACache*. An update is a key deletion followed by a key insertion. Inserting an alignment into the *ACache* involves either an insertion into the second-level tree alone, or an insertion into both the first- and the second-level trees. Insertion into the first-level tree follows the insertion algorithm of B+ tree.

Insertions into the second-level B-Link tree that do not require splitting of the leaf or intermediate nodes also follow the insert algorithm of the B+ tree. The difference between the two algorithms (B+ and B-Link) arises when the insertion of a new pairwise alignment into the B-Link tree requires a split of either a leaf or an intermediate node. In the case of these splits, the parent and sibling pointers of the B-Link tree need to be set up to ensure fail-safe page replacement at cache replacement time. In the event of leaf node split, that is split of leaf node L_1 , the parent pointer of the new leaf node L_2 must be set to the same value as the parent of the node L_1 (see Figure 3). This is in addition to the standard B+ tree child reference pointer from the parent to the new leaf node L_2 , and is maintained to ensure that there are no dangling child pointers as a result of page replacement. In the event of an intermediate node split, both the parent and the sibling pointers of the new node must be set. The parent pointer is set similar to the leaf node parent pointer. The sibling pointer of the split node is set to the newly created node as shown in Figure 3. Once again this is to ensure that the right parent can be found with minimal number of traversals at page replacement time. Deletes are implemented without merging as is often the case for many implementations of B+ trees.

3 Cache Replacement

3.1 Hot Key and Hot Key Pair Policies

Swissprot [2], the largest resource for protein sequences, in its current version contains over 170,000 sequences. Caching of all possible pairwise alignments between these sequences (a total of 14,449,915,000 pairwise alignments) is not practical and the overhead of such a large cache would potentially cancel out its benefits. By the same token, a static cache structure that stores some but not all the

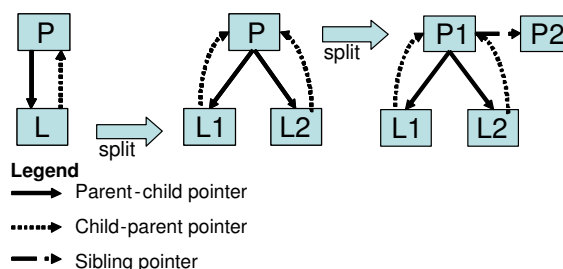


Figure 3. Example Showing the Split of a Leaf and an Intermediate Node.

alignments of interest at any given time cannot fully maximize the performance benefits of caching. In this paper, we propose two cache replacement policies, *hot key pair* and *hot key* corresponding to the two-level cache hierarchy of *ACache*. These policies leverage the domain knowledge of the query patterns typical in molecular biology.

The first step for most molecular biology analysis begins with the discovery of sequences similar to either (1) an unknown (orphan) sequence to initiate the process of identifying the structure and function of the unknown sequence; or (2) a known sequence to initiate the isolation of residues responsible for certain structural or functional characteristics. Although not true for highly divergent sequences, it is likely that the discovered similar sequences are from the same family. The second step of the process involves building multiple alignment from these similar sequences. Thus, a common query pattern for biologists is repeated multiple alignments of a given sequence S_1 to a set of sequences within a specific family. The interest of the biologist can often be tracked by a shift in the query pattern to a comparison of S_1 with other families of sequences, or by a complete abandonment of the sequence S_1 . The *hot key pair* and *hot key* replacement policies trace these gradual as well as sudden changes in the query patterns.

Central to both of these policies, is the notion of a “hot key”. A hot key is defined at the first level B+ tree and represents the *most frequently used* sequence within a given time window. Degrading *hotness* of the sequences thus tracks the interest of the scientist in a given sequence within a time frame. Based on this notion of a hot key, we now define the hot key and the hot key pair replacement policies.

Hot Key Replacement Policy.

The *hot key* policy is a coarse-grain replacement policy targeted towards tracking large shifts in sequences of interest, and effectively replaces the entire B-Link tree for a given sequence. Thus, it ensures that all alignments for a “hot” sequence (been queried recently) are kept in cache, while the entire B-Link tree for a less frequently or not referenced

sequence can be tossed out by the replacement algorithm.

Hot Key Pair Replacement Policy.

The *hot key pair* policy, targeted towards tracking gradual shifts in the query pattern, is a replacement policy that works in conjunction with the idea of a “hot key” to provide a hierarchy of candidate B-Link leaf pages for replacement. Entire B-Link trees are ranked by the hotness of their key, while leaf pages within each individual B-Link tree follow a least recently used replacement policy. Thus, a page is a candidate for replacement if (a) its first level tree key is the least “hot key”; and (b) it is the least recently used page in its B-Link tree.

Consider the *ACache* structure shown in Figure 4. Here, sequence S_1 is aligned with sequences S_2 , S_3 and S_4 , and each alignment ($\langle S_1, S_2 \rangle$, $\langle S_1, S_3 \rangle$ and $\langle S_1, S_4 \rangle$) is cached in the B-Link tree occupying one leaf page per alignment. In addition, *ACache* has a second B-Link tree that accommodates the alignments of sequence S_x with sequences S_a and S_b . For simplicity of the example, we assume that each alignment occupies a separate leaf page, and the cache is full. Query 1 in Figure 4 gives a sample query pattern that accesses pages L_1 and L_2 . Now consider a shift in the query pattern to the pattern Query 3 shown in Figure 4. Here, a new alignment between $\langle S_1, S_5 \rangle$ must be constructed and inserted into the cache.

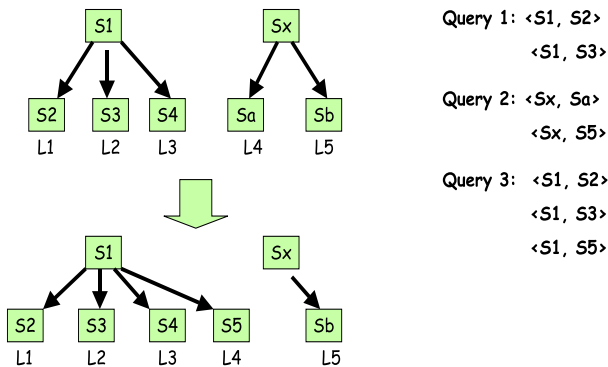


Figure 4. An Example to Explicate the Hot Key Pair Replacement Policy.

Let us assume that the leaf pages L_3 , L_4 , and L_5 are ranked by their access order – thus L_3 is the least recently used and L_5 the most recently used among the 3 leaf pages; and the sequence S_1 is the “hot” sequence. Based on these assumptions, the hot key pair policy will choose page L_4 for replacement as opposed to page L_3 – the likely candidate for a LRU policy. This is in keeping with the intuition that alignments on page L_3 – representing alignments to

sequence S_1 – are more likely to be referenced in the near future and hence should not be removed.

The Top-k Hot Keys. While the hot key pair policy is in keeping with the basic assumption that query patterns typically follow the alignment of one sequence with a set of sequences, it is possible that a page such as L_3 may in actuality not be used again. In this case, the hot key pair policy will only replace L_3 when S_1 is no longer a hot key, while a LRU policy would result in a quicker replacement of the page. To compensate for this, we limit the “hot key” selection to the *top-k hot keys* – that is at any given time only the top-k first level sequence keys are regarded as hot. For all other B-Link trees the hot key pair degrades to LRU.

3.2 Implementing the Cache Replacement Policy

The hot key and hot key pair policies are implemented via two separate circular buffers – the hot key buffer and the hot key pair buffer respectively. The hot key buffer maintains the root page identifiers (*rootIds*) of the B-Link trees corresponding to the first level keys, a la the hot keys. The hot key pair buffer, on the other hand, maintains the identifiers of the leaf pages (*leafId*) across the different second level B-Link trees together with the root identifiers of its B-Link tree: $\langle \text{leafId}, \text{rootId} \rangle$.

Whenever there is a cache miss that results in the insertion of a new pairwise alignment, the cache is examined for free space. If the required space cannot be allocated, the hot key pair replacement algorithm selects the record $\langle \text{leafId}, \text{rootId} \rangle$ from the tail of the hot key pair buffer for replacement. The *rootId* of the selected record is compared to the top-k *rootIds* at the head of the hot key buffer. If there is a match of the *rootId*, that is the *rootId* is in the top-k identifiers, the algorithm places the record at the head of the hot key pair buffer, selects another record from its tail, and repeats the comparison with the hot key buffer. If no match of the *rootId* is found, that is the *rootId* is not in the top-k identifiers, the leaf page (*leafId*) is removed from the parent’s set of children, and then reallocated to a different parent as needed.

Figure 5 gives the pseudo-code for the hot key pair replacement algorithm. Here, the parent and the sibling pointers are used in conjunction to identify the right parent node for a given leaf node ².

4 Experimental Evaluation

The *ACache* system was implemented in Java (SDK 2.0) and evaluated on a 933 MHz Pentium 4 machine with 2GB

²It is possible that after a split (of the parent node) a leaf node may no longer be pointing to its parent node. In this case, the parent node’s sibling pointer is traversed to retrieve the right parent node.

```

LeafPage replace (pageId id)
{
    CircularBuffer hotKey;
    CircularBuffer hotKeyPair;
    LeafPage lp = null;
    while (lp == null) {
        // select record from hotKeyPair
        Record record = hotKeyPair.getTail ();
        RootId rootId = record.getRootId ();
        if (! hotKey.existsTopK(rootId))
            lp = GetPage (record.getLeafId ());
        hotKeyPair.head(record);
    }
    // select leafPage's parent
    Node parent = lp.GetParent ();
    // find the child node in the parent
    while (!(parent.getChild(lp))) {
        parent = parent.getSibling ();
    }
    // reset the child reference in parent
    return lp;
}

```

Figure 5. The Replacement Algorithm for a Leaf Page.

of memory running Debian Linux kernel 2.2.19. Given our focus on caching the pairwise alignments, our experiments were conducted using the Needleman-Wunsch [19] global pairwise sequence alignment, and the Smith-Waterman local pairwise alignment algorithms. Both algorithms were implemented in Java (SDK 2.0) and their running times evaluated on the same machine. All sequences used in the experiments were obtained from Swissprot³. The selected subsets of sequences approximated the sequence length distribution of the Swissprot data. For all experiments, k (top- k) was fixed to 0.

4.1 Results and Analysis

Algorithm Profiling. The first set of experiments performed a time profile analysis of Needleman-Wunsch - the popular global pairwise alignment algorithm [19]. The complexity of this algorithm is $O(nm)$, where n and m are the lengths of the two sequences. For this experiment, a set of 100 sequences was randomly selected from the Swissprot data set [2], with verification that the selected sequences approximated the data distribution of the complete Swissprot data set. For this set, all possible pairs of alignments were

³The entire Swissprot dataset can be downloaded in FASTA format from <http://www.ebi.ac.uk/uniprot/database/download.html>.

computed. Figure 6 shows a scatter plot of the alignment time on the y-axis versus the product of the two lengths on the x-axis. The graph shows that there is a strong linear relationship between the product of lengths of the two sequences and the alignment time, with a coefficient of determination of 0.98. These linear relationships between alignment times and sequence length products are useful for determining which sequence alignments should be cached.

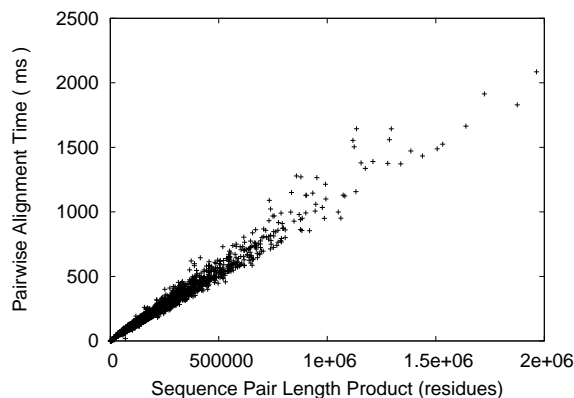


Figure 6. Scatter Plot of Alignment Time versus Product of Sequence Lengths for the Needleman-Wunsch Algorithm.

Cache Search Time. The first set of experiments evaluated cache performance in terms of cache search time for different cache characteristics, such as the cache size, number of items in the cache and the average length of sequences. To approximate the interests of different scientist groups, sequences were randomly selected from a set of superfamilies in the Swissprot data set to warm-up the cache under the different size constraints. For the first experiment (see Figure 7), we ensured that the data distribution maintained a balance between the B-Link trees (all B-Link trees were approximately of the same size) for the average case scenario, and produced an imbalance (one B-Link tree was much larger than the other B-Link trees) to measure the search time in the worst case scenario.

Figure 7 shows the relationship between the average cache size and the average cache search time. The x-axis is the cache size in KBytes, and the y-axis is the average cache search time in milliseconds. The graph depicts both the average case search time obtained when there is a balance between all B-Link trees, and the worst case search time when one B-Link tree is significantly larger than the other B-Link trees. For the worst case, the search time reported is the time taken to search for an alignment in the large B-Link tree. It can be noted that the worst case search times are significantly higher, nearly double that of the av-

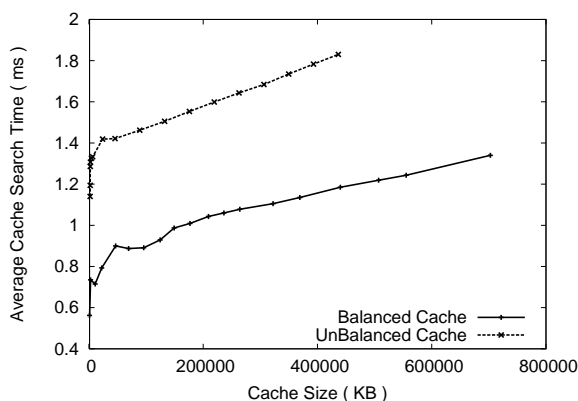


Figure 7. Effect of Cache Size on Average Search Time.

verage case. Moreover, while in the average case there is a slight increase in the average search times with an increase in cache size, in the worst case there is a sharper increase (steeper slope) in the search time as the cache size grows. This is due to the fact that in the average case, the overall *ACache* structure is balanced, that is, all the second level B-Link trees are of approximately the same size. Any increase in the cache size is amortized over all the B-Link trees. However, in the worst case a large majority of the total alignments are stored under one B-Link tree creating an imbalance in the overall *ACache* structure – one B-Link tree is much larger in terms of number of keys, height of tree and number of leaf pages than the others, accounting for the sharper increase in the search times.

Effects of Cache on Alignment Computation. The next set of experiments examined the effects of caching on the overall pairwise alignment computation time. We used two disjoint sets *S1* and *S2* with 200 sequences randomly selected from different superfamilies in Swissprot.

Figure 8 shows the total computation time of a single query containing 50 sequences for varying cache hit-ratio. The x-axis shows the cache hit-ratio, while the y-axis gives the total computation time for the query in milliseconds. For this experiment, the cache was pre-loaded with 19900 pairwise alignments computed from the two sequence sets *S1* and *S2*. The query sequences used in the experiment were selected at random from the second set *S2*. At a 0% hit ratio, 1225 total pairwise alignments were computed taking approximately 117,500 milliseconds. A slight overhead was incurred in checking the cache, and this accounts for the thin bar at the top. As the cache hit-ratio increased, the total query time dropped to almost 7 milliseconds with a 100% cache hit ratio. From a practical standpoint, a 40% to 60% cache hit ratio provides significant performance gains – decreasing the computation time by nearly half.

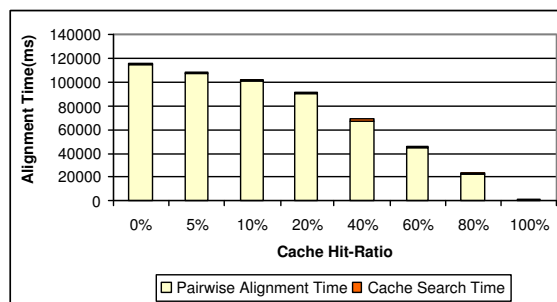


Figure 8. Effect of Cache Hit Ratio on Total Computation Time for Query with 50 Sequences.

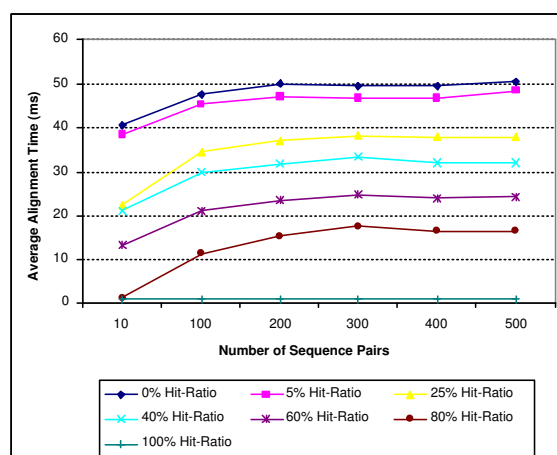


Figure 9. Effect of Cache Hit Ratio on Average Computation Time for Different Queries.

Figure 9 shows the average computation time for an increasing number of sequence pairs in the query. The x-axis is the number of sequence pairs in the query, while the y-axis is the average computation time for the given query for different hit ratios. The average computation time is the mean of the time taken to perform the cache search (retrieve the alignments) and the time taken to compute the alignment for the remaining sequence pairs. At 100% cache hit-ratio, the average computation time reflects the cache search time (about 2 milliseconds) for the sequence pairs. The alignment time for all sequence pairs, in this case, is 0 (the alignments for all sequence pairs are retrieved from the cache, and hence not computed). For all other cache hit ratios, there is a sub-linear increase in the average computation time with increase in sequence pairs. This increase in the average computation time is proportional to the increase in average alignment time contributed by the sequence pairs for which the alignment is not found in the cache. A trend similar to that in Figure 8 can be observed in terms of the

cache hit ratio. As the cache hit-ratio increases a drop in the average computation time can be seen with an average computation time of about 2 milliseconds for a 100% cache hit-ratio.

Cache Replacement Policy. The goal of this set of experiments was (1) to examine the effectiveness of the hot key pair policy in maintaining the average cache hit ratio between 40% and 60% under variable query load; and (2) to compare the cache hit ratio maintained by the hot key pair policy with the cache hit ratio of the static cache for a single repetitive query pattern exercising the “hot key” factor.

Figure 10 shows the cache hit ratio maintained by the hot key pair policy as well as the static cache for a varying query load. In the absence of a real query load, we selected a set of 14 queries to provide a close approximation. Each query contained 50 sequences selected from a superfamily in the Swissprot data set. Nine out of these 14 queries were unique, in that they had no overlapping sequences and were used to simulate the diversity of scientist interest. The remaining 5 queries, used to simulate some commonality between user interests as well as modified queries by the same user, were composed out of sequences from the 9 unique queries. We use the following numbering scheme to clarify these data sets in the graph (Figure 10): The convention of `query` followed by a single integer is used to represent the unique queries, while the `query` followed by a set of integers represents a query composed from sequences in the other queries. Thus, `query12` is a query that contains 25 sequences from `query1` and 25 sequences from `query2`. The x-axis shows these different queries, while the y-axis depicts the cache hit ratio for each of the queries. The measurements reported are for both the hot key pair cache as well the static cache. We used warm caches – that is both caches (hot key pair and static) were pre-loaded with 300 items randomly selected from the Swissprot data set. For queries `query0` and `query1` (see Figure 10) the cache hit ratio for both the static and hot key pair cache are at almost 0%. This was an expected result as `query0` and `query1` are non-overlapping queries. For the query `query01`, a combination of sequences from `query0` and `query1`, the hot key pair provided a 100% cache hit ratio. For this experiment, the hot key pair was able to maintain a hit ratio of about 40% for a variable query load. The experiment was repeated with variations in the execution order of the queries. The hot key pair maintained cache hit ratio between 33% and 58% for different query execution orders.

We also ran a second set of experiments designed primarily to evaluate the “hot key” factor for repeated query patterns. For this experiment, we used a single query comprised of 498 pairs of sequences that were taken from the AAA superfamily. The experiment was run with a fully loaded warm cache – the cache contained 5000 records randomly selected from the Swissprot dataset. While both the

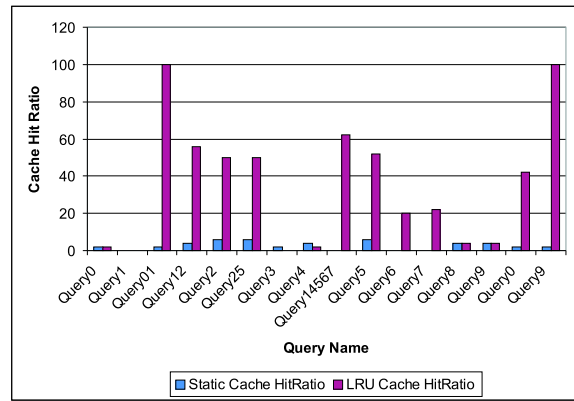


Figure 10. Effects of Hot Key Pair Replacement Policy on the Cache Hit Ratio of Multiple Queries

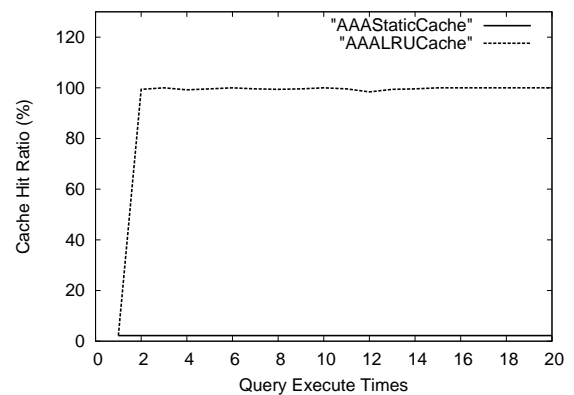


Figure 11. Effect of Hot Key Pair Replacement Policy on Single Query – Evaluating the “hot” factor.

static and hot key pair cache were evaluated and compared, the hot key pair cache was dynamically adjusted by first executing the query, second replacing 1000 records in the cache using the hot key pair policy, and then re-executing the query. Twenty such iterations were performed. Figure 11 shows the cache hit ratio maintained by the static cache as well as the hot key pair cache. The x-axis gives the query iteration number, while the y-axis denotes the cache hit ratio. As can be observed from the graph in Figure 11, the cache hit ratio for the hot key pair cache rises to 100% after the first iteration and is then maintained at nearly 100% for the remaining iterations. The static cache hit ratio is maintained at 5%, its initial cache hit ratio.

Cache Admission. The last set of experiments focuses on obtaining insights into the characteristics of the sequences that would benefit from caching – enabling us to make “good” caching decisions. Figure 12 shows the cumulative

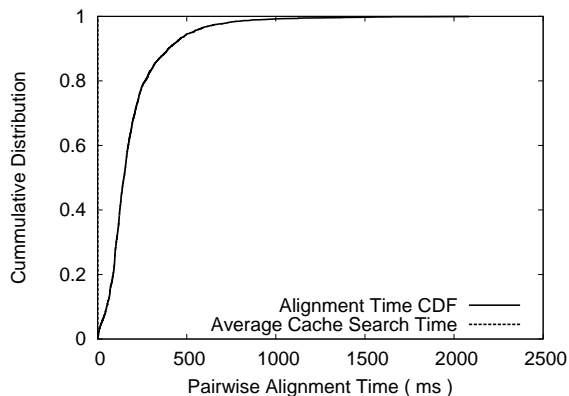


Figure 12. Cumulative Density Function for Cache Alignment Times Compared to Average Cache Search Time for the Needleman-Wunsch Algorithm.

density function for the alignment times of 100 sequences. These sequences were selected from Swissprot with a data distribution that closely approximates the Swissprot data distribution. The x-axis plots the alignment times in milliseconds and the y-axis gives the cumulative distribution. For over 90% of the sequences the cache search time is less than the alignment computation time, suggesting that over 90% of the sequences would benefit from caching. Combining the results of this graph with the graph in Figure 6, all sequence pairs with product of length greater than 300 would benefit from caching. Similar results were obtained for the Smith-Waterman algorithms (see [27]).

5 Related Work

Caching has been a well-studied topic in database literature. ADMS [5, 23] caches the results of sub-query expressions corresponding to join nodes in the evaluation tree of each user query. Subsequent queries are optimized by using previously cached views. Query matching plays an important role here in identifying subsets of data that can be used to answer an incoming query. They use multiple cache replacement policies including LRU, LFU and LCS (Largest Space Required), and show that LCS consistently outperforms the other two strategies. A key similarity between this and our work is the granularity of replacement. In ADMS, cache replacement is performed by tossing out entire views. Similarly, in *ACache* cache replacement is performed by throwing out entire sets of alignments. Other works [7, 15, 8] have examined the caching of results based on projection [7] and predicate matching [15, 8].

Cache structures have also been used for pre-fetching

of data primarily in the context of object-oriented databases [11, 9]. DeWitt et al. examined the trade-offs between three architectural possibilities, page-level, object-level and file-level, for cache structures. Object caching works at the level of individual objects, while page caching works at the granularity of disk pages. File caching, a simplification of the page caching approach, handles caching of a collection of pages. In our work, while we primarily use page-level caching (the granularity of replacement is a page), file-level caching can also be utilized. In file-level caching, an entire B-link tree is replaced corresponding to a shorter sequence that has little or no references. There is some level of object replacement in the first level B+ trees.

Performance improvements for multiple alignment algorithms has been a key issue over the recent years [14, 16, 18, 12, 22, 20, 4]. Most of these algorithms rely on progressive alignment as a fundamental technique to improve the performance. Parmentier et al. [21] have proposed *PhylTree* (PT) and a parallel version of the same. Their work investigates caching of the partial alignment guide trees to eliminate redundancy of computations in a parallel environment. Catalyurek et al. [3, 4] have developed a component-based implementation for running multiple sequence alignment on a SMP machine. Their work is the most closely related to ours, in that they also examine the effects of caching pairwise alignments. However, our work differs in two significant ways. First, their focus is on an SMP based implementation of ClustalW, and hence they provide a read-only static cache. The *ACache* system provides not only a dynamically updatable cache (inserts and deletes), but also builds in heuristics (based on our analysis presented here) to affirm the sequence alignments that should be cached. Second, as a result of implementing a read-only cache, their work does not discuss cache replacement policies. We propose multi-level cache replacement policies, and also examine the data structure necessary to accomplish this efficiently.

6 Conclusions and Future Work

In this paper, we propose *ACache*— a two-level structure for caching previously computed pairwise alignments. A unique feature of our *ACache* structure is the B-link tree, an extension of the B+ tree. Traditional B+ trees are designed to be unbounded (in size) indexing structures. However, in our “caching environment” it was essential to bound the size of the B+ trees and introduce mechanisms for replacing entire pages of the B+ tree in a fail safe manner when needed. The B-link trees introduced in this paper, provide this functionality via the additional parent and sibling pointers. The parent and sibling pointer aware update algorithm ensures fail-safe updates. Our experimental results show that we are able to achieve speedups of up to 2.5 for a cache hit-ratio between 40% and 60%, and a nearly 5 time speedup for a

90% to 100% cache hit ratio. Our experimental results have also helped classify the sequences in terms of their lengths, that would most benefit from the caching. Any sequence pair whose product of sequence length is greater than 300 residues, would in general benefit from caching. However, for practical reasons cache admission policy could be set to cache only those sequences whose sequence length product is equal to or greater than 600 residues. This would still accommodate the average sequence lengths in Swissprot [2].

Future Work. There are a number of directions in which this work can be extended. An immediate direction is a more extensive evaluation of the hot key pair cache for realistic query workloads. Another direction is to comprehensively study the impact of the *top-k* on the cache hit ratio. Our initial investigations have shown that while a *top-0* degrades the hot key pair to LRU, a too high 'k' value virtually results in a static cache. The optimal k value is dependent on the query workloads and may need to be self-tuning to reflect query workload patterns.

References

- [1] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Efficient Execution of Multiple Query Workloads in Data Analysis Applications. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing(CDROM)*, pages 53–53, 2001.
- [2] A. Bairoch and R. Apweiler. The SWISS-PROT Protein Sequence Databank and its Supplement TrEMBL. *Nucleic Acid Res.*, 1(28):45–48, 2000.
- [3] U. Catalyurek, M. Gray, T. Kurc, J. Saltz, E. Stahlberg, and R. Ferreira. A Component-Based Implementation of Multiple Sequence Alignments. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 122–126, 2003.
- [4] U. Catalyurek, E. Stahlberg, R. Ferreira, T. Kurc, and J. Saltz. Improving Performance of Multiple Sequence Alignment Analysis in Multi-Client Environments, 2002.
- [5] C. Chen and R. N. Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *Int. Conference on Extending Database Technology (EDBT)*, 1994.
- [6] F.-C. F. Chen and M. H. Dunham. Common Subexpression Processing in Multiple-Query Processing. *Transactions on Knowledge and Data Engineering*, 10(3):493–499, 1998.
- [7] G. Copeland, S. Khosafian, M. Smith, and P. Valdureiz. Buffering Schemes for Permanent Data. In *IEEE Int. Conf. on Data Engineering (ICDE)*, 1986.
- [8] S. Dar, M. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Int. Conference on Very Large Data Bases*, 1996.
- [9] D. DeWitt, P. Futersack, D. Maier, and F. Velez. A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems. In *Int. Conference on Very Large Data Bases*, 1990.
- [10] C. Faloutsos, R. Snodgrass, V. Subrahmanian, S. Zaniolo, C. Ceri, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.
- [11] M. Franklin. Client Data Caching: A Foundation for High Performance Object Database Systems. In *Kluwer*, 1996.
- [12] D. Higgins, J. Thompson, and T. Gibson. CLUSTALW: Improving the Sensitivity of Progressive Multiple Sequence Alignment Through Sequence Weighting, Position-specific Gap Penalties and Weight Matrix Choice. *Nucleic Acid Research*, 22(1), 1994.
- [13] E. B. Institute. FASTA Format Description.
- [14] K. Katoh, K. Misawa, and T. Miyata. MAFFT: A Novel Method for Rapid Multiple Sequence Alignment Based on Fast Fourier Transform. *Nucleic Acid Research*, 30(1), 2002.
- [15] A. Keller and B. J. A Predicate Based Caching Scheme for Client-Server Database Architectures. *VLDB Journal*, 5(1), 1996.
- [16] C. Lee, C. Grasso, and M. Sharlow. Multiple Sequence Alignment Using Partial Order Graphs. *Bioinformatics*, 18(1), 2002.
- [17] D. Mikhailov, H. Cofer, and R. Goperts. Performance optimization of Clustal W: Parallel Clustal W, HT Clustal, and MULTICLUSTAL. Technical report, SGI Life and Chemical Science, 1999.
- [18] B. Morgenstern. DIALIGN2: Improvement of the Segment-to-Segment Approach to Multiple Sequence Alignment. *Bioinformatics*, 15(1), 1999.
- [19] S. B. Needleman and C. D. Wunsch. A General Method Application to the Search of Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology*, 48:443–453, 1970.
- [20] C. Notredame and J. Henriga. T-Coffee: A Novel Method for Fast and Accurate Multiple Sequence Alignment. *Journal of Molecular Biology*, 302(1), 2000.
- [21] G. Parmentier, D. Trystram, and J. Zola. Cache-based Parallelization of Multiple Sequence Alignments. *Spring-Verlag*, 3149:1005–1012, 2004.
- [22] J. Pei, R. Sadreyev, and N. Grishin. PCMA: Fast and Accurate Multiple Alignment Based on Profile Consistency. *Bioinformatics*, 19(1), 2003.
- [23] N. e. a. Roussopoulos. The ADMS Project: Views “R” Us. In *IEEE Data Engineering Bulletin*, June 1995.
- [24] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithm for Multiple Query Optimization. In *Proceedings of the 2000 ACM-SIGMOD conference*, pages 249–260, 2000.
- [25] N. Saitou and M. Nei. The Neighbour-Joining Method: A New Method for Reconstructing Phylogenetic Trees. *Molecular Biology Evolution*, 4:406–425, 1987.
- [26] J. D. Thompson, D. G. Higgins, and T. J. Gibson. Clustal W: Improving The Sensitivity Of Progressive Multiple Sequence Alignment Through Sequence Weighting, Position-Specific Gap Penalties And Weight Matrix Choice. *Nucleic Acids Research*, 22(22):4673–4680, 1994.
- [27] X. Tu and K. Claypool. Acache: A caching scheme to improve the performance of clustalw. Technical Report UML-TR05-10, University of Massachusetts Lowell, May 2005.