Accelerating Search and Recognition Workloads with SSE 4.2 String and Text Processing Instructions

Guangyu Shi
gshi2@wisc.edu

Min Li
mli46@wisc.edu

Mikko Lipasti
mikko@engr.wisc.edu

University of Wisconsin-Madison

Abstract

Today's information is increasing rapidly, doubling every three years. Consequently, the search and recognition stages in computer applications will consume a growing portion of the total CPU time. The SSE 4.2 instruction set, first implemented in Intel's Core i7, provides string and text processing instructions (STTNI) that utilize SIMD operations for processing character data. Though originally conceived for accelerating string, text, and XML processing, the powerful new capabilities of these instructions are useful outside of these domains, and it is worth revisiting the search and recognition stages of numerous applications to utilize STTNI to improve performance. In this paper, we explored the feasibility and potential benefit of using STTNI to improve the CPU and memory performance of search-and-recognition applications. We optimized four benchmark applications – cache simulation, B+tree search algorithm, template matching, Basic Local Alignment Search Tool (BLAST) – with STTNI, and the new applications outperform their respective original implementations by a factor of 1.4x to 13x.

1 Introduction

Today's information is increasing rapidly. According to UC-Berkley's project in 2003[1], the data in the world are doubling every three years and currently it is measured in exabytes a billion billion bytes. As a result, people's expectation on search and recognition (SR) application keeps getting higher. Accuracy and speed are two basic metrics of SR applications. High accuracy guarantees that the applications retrieve useful information while filter out redundant information, and high speed guarantees that the information is retrieved in time. As the technology scaling reaches the fundamental limit, performance benefit from frequency scaling will diminish in future. Therefore, we need to find a solution to meet future computational needs of SR applications.

Intel SSE 4.2 instructions, first implemented in Intel's Core i7 processors, complete the SSE 4 instruction set with 7 new instructions. Among them, four string and text processing instructions (STTNI) offer the capabilities to simultaneously operate on 16 bytes in two data arrays. Four instructions has the same format:

Opcode operand1, operand2, MODE

Operand 1 and 2 are SIMD registers holding 128-bit value. The third operand MODE is a constant specifying the type of comparison of the instruction. Table I summarizes the four instructions.

Table 1: STTNI Instructions In SSE4.2 [2]

Instruction	Description
PCMPESTRI	Packed compare explicit length strings, return Index
PCMPESTRM	Packed compare explicit length strings, return Mask
PCMPISTRI	Packed compare implicit length strings, return Index
PCMPISTRM	Packed compare implicit length strings, return Mask

Table 2: STTNI Compare Mode Options [2]

Mode	Return value is true if
equal any	Element i in fragment2 matches any element j in fragment1
ranges	Element i in fragment2 is within any range pairs specified in fragment1
equal each	Element i in fragment2 matches element i in fragment1
equal ordered	Element i and subsequent, consecutive valid elements in fragment2 match fully or partially with fragment1 starting from element 0

STTNI takes two 128-bit values from SIMD registers as source and target, and a third immediate value as MODE. STTNI can divide a 128-bit value into 8x16-bit elements or 16x8-bit elements depending on the precision requirement. Table 2 illustrates four MODE options for STTNI [2]. Each compare mode is useful in specific applications. For example, Equal Any and Ranges mode can be used in XML parsers, and Equal Ordered mode can be selected for substring searches.

The original purpose of developing these instructions is to accelerate string, text and XML processing. Intel

Researchers have used STTNI in XML parsing [3] and schema validation [4]. According to the performance report from Intel Corporation, each application achieved a speedup of 70% and 20%, respectively. To study the behavior of STTNI, we compared the STTNI-based string functions with the ones in string.h to measure the performance improvement when we replace the original sequential comparison with the STTNI 16-byte parallel comparison. Figure 1 shows the normalized speedup of the STTNI-version string functions with respect to the original implementation in string.h. From the result we can see that with the growth of the length of the strings, the speedup that STTNI-version string functions achieve increases from 1.2x to 4.7x.

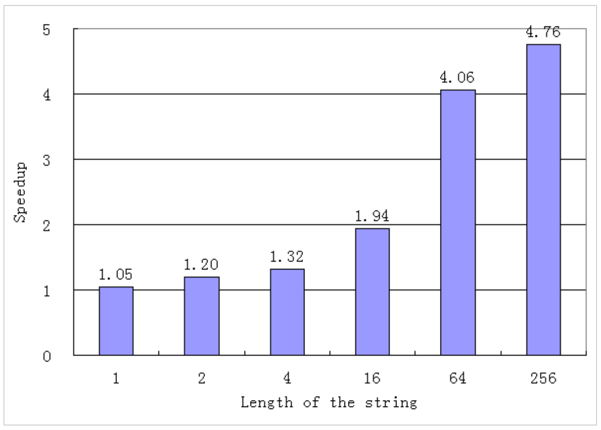


Figure 1: Speedup of cstring functions modified by STTNI with respect to the cstring functions in string.h.

The performance improvement in string functions points us to a novel and promising solution for improving the performance of SR workloads. In this paper, we explore the possibility of revising the SR stages of applications to improve the performance without sacrificing their accuracy. We also analyze when and how to use STTNI in such applications in order to achieve the highest performance improvement.

This paper makes the following contributions:

(1) We classify the SR applications by the data structures, and proposed optimization technique for each class.

(2) We apply our STTNI-based optimization on 4 benchmark applications - cache simulation, B+tree search algorithm, template matching, Basic Local Alignment Search Tool (BLAST) - and analyzed the CPU performance and cache performance of each application. Experimental results show that our proposed implementation achieves performance improvement of 1.4x to 13x.

(3) We analyzed different types of overhead when using STTNI and discussed how to minimize or avoid them.

This paper is organized as follows: Section 2 illustrates the general optimization technique of our approach. Section 3 specifies the hardware configuration and other details of our experimental environment. Section 4 describes our proposed implementations of four benchmark applications. For each of them, we will first summarize the related work in improving the performance of each application. Then we describe the STTNI-based implementation as well as the baseline implementation. After that we show the CPU performance and memory performance of the new design in our experiment compared to the baseline. Section 5 we conclude our work and discuss future works.

2 Optimization Technique

One attribute of STTNI is that the word length for comparing can be either 8 bits or 16 bits long, depending on whether the compare mode is BYTE or WORD in the third argument of STTNI. We define STTNI-word as the element that STTNI compares between two sequences (in order to distinguish the word in query and reference sequences, the length of which are not necessarily 8 bits or 16 bits).

2.1 Nature of SR Applications

SR applications compare query sequences with reference sequences. It can compare for either equality or inequality. Note that in some applications query is compared against reference, but in other ones the relation is reversed. In STTNI, the first operand is the reference and the second operand is the incoming query.

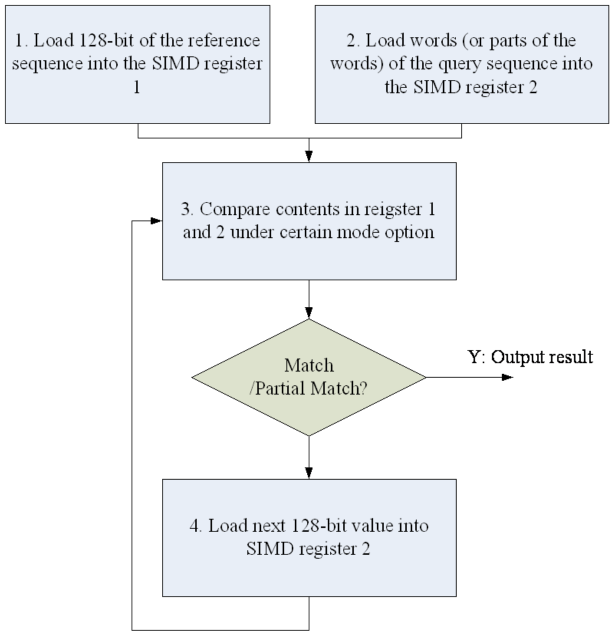
The most simple SR algorithm (such as in strcmp) compares sequences sequentially. These applications are useful when sequences are relatively short. For long sequence compare, they become impractical due to the slow speed of sequential compare.

To achieve a better performance, optimization algorithms and data structures are developed. Hash table and search tree structures are most commonly used. Hash tables are widely used when compare for equality is needed. It minimizes the number of necessary comparisons by creating a table structure and assigning key words values which are used as index of the hash table. Ideally, finding desired elements in the hash table requires no compare if the hash table has entries for every possible element. In that case, a translation by the hash function is enough to find the element, and reference is not needed. In many cases, however, hash collisions the cases when different key words have the same hash value cannot be avoided and must be handled in some way. As a result, hash collision resolving is followed after hash translation. Most of the collision resolving method requires compare between incoming keys and the keys stored in the hash buckets. Designers balance the hash table size and the average number of collisions to achieve the best overall performance.

Search trees, on the other hand, are widely used when compare for inequality is needed. Tree data structure arranges key words in a specific order, and searching for key words in the tree becomes traversing from the root to the leaves. The number of comparisons required is proportional to the distance from the root to the destination node and the number of key words in each node. Tree structures do not have as significant memory space overhead as a hash table. As a result, they need more compares.

Almost all the SR applications can be categorized into the three types discussed above. By choosing strategies with different mode option, STTNI can be used to accelerate all three kinds of SR applications.

2.2 Design Strategies



1. Load 128-bit of the reference sequence into the SIMD register 1
2. Load words (or parts of the words) of the query sequence into the SIMD register 2
3. Compare contents in reigster 1 and 2 under certain mode option
Match /Partial Match?
Y: Output result
4. Load next 128-bit value into SIMD register 2

Figure 2: Block diagram of STTNI usage in general.

Figure 2 shows the block diagram of how STTNI is used in general. First, it loads at most 128-bit values into the SIMD register. If the size of a word in the sequences exceeds the size of an STTNI-word, a part of the word should be loaded in the SIMD register. The corresponding part of a word in the query sequence should be loaded into another SIMD register. Then the application compares the contents of the registers. If one or more match is found, the matching position will be recorded so that the SR stage will start from a filtered search space. If no match is found, or if the application requires finding all matches, the next round of STTNI comparisons will follow up. In either case, the compare stage will proceed in parallel within SIMD capability, which is much faster than a regular compare method. The steps in figure 2 are shared by all kinds of STTNI-based SR applications. For the rest of the part, depending on the nature of the applications (as described in 2.1), usage of the STTNI is different correspondingly. They are illustrated in figure 3. The baseline algorithm (without using STTNI) is shown in the upper-half of the figures, and the STTNI-optimized algorithm is shown in the lower-half of the figures.

1) Sequential Compare.

Figure 3 (a) shows the STTNI optimization technique for sequential compare. If the query sequence is longer than 16 bytes (cannot fit in a SIMD register), the first 16-byte word is compared first. If there is a match, the next 16-byte word is compared, and so forth. Although it looks quite straightforward STTNI compares 16 bytes at a time, designers need to select a proper mode option in order to achieve the best performance.

If there are multiple reference sequences, we compare the query with 8 (16) references at a time and each reference is equal or less than 2 (1) bytes. If each reference sequence is longer than 2 (1) bytes, then load 2 (1) bytes in one at a time, and repetitively compare when matches are found.

2) Search Tree

Figure 3 (b) illustrates the way of optimizing applications which uses search tree structure. In a search tree, comparisons between query and the keys take place on each node until the user finds the desired element. The approach we use here is similar to the optimization on sequential compare. But instead of comparing two sequences directly, we compare the query key with keys in each node. When the keys in each node are arranged in order, STTNI Range mode is very useful in comparing the key word and finding the next branch.

3) Hash Table

For hash tables with collision resolving, STTNI can be used for comparing the incoming key against keys in each bucket, which is similar to the optimization in sequential compare.

We know that hash collision is introduced in the purpose of decreasing the number of hash table entries (increase the utilization of the hash table) and thus reduce the memory space overhead. Based on this observation, we can use STTNI to introduce more collisions in each bucket, and reduce the number of entries further. As a result, the collision resolving step requires more comparisons between the query key and keys in the buckets. However, compare steps can be handled by STTNI, therefore the overall performance will not suffer significantly from the extra times of compare. Therefore, besides a promising performance improvement, using STTNI can also balance the stress between CPU and memory and relax pressure on the most critical resource. Designers are able to rebalance the number of hash table entries with the number of collisions to achieve a better overall performance. Figure 3 (c) illustrates the way of optimizing hash table-based applications. This method is applicable to hash table algorithms with or without collision resolving step.

The STTNI-based optimization technique for all three types of SR applications is generally applicable. Besides, it does not conflict with most of the existing optimization techniques. Therefore, performance improvement can be accumulated.

2.3 Minimizing the Overhead

In spite of the powerful SIMD capability, using STTNI is not free. Several issues will cause the STTNI compare to be slower, sometimes even worse than regular compare.

1) Initialization of STTNI. As [3] mentioned, the initialization of STTNI can cause performance overhead. This

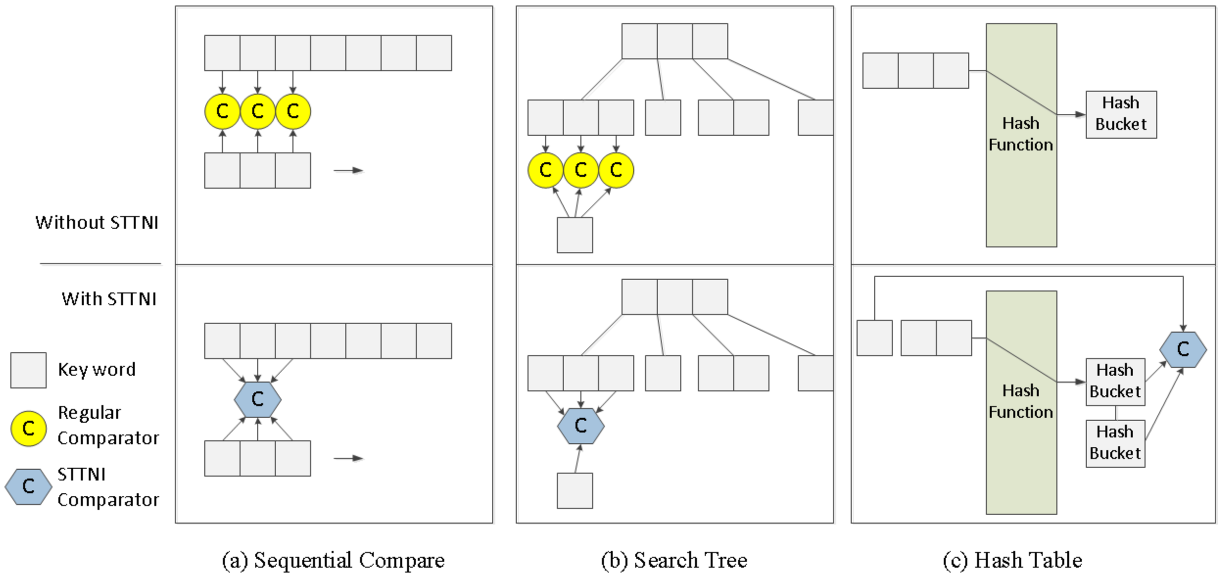


Figure 3: STTNI optimization scheme for 3 different types of SR applications.

overhead is negligible when the query sequence or the reference sequence is much longer than 128 bits, which in most cases is true.

2) Utilization of the SIMD register. Because STTNI can only compare in BYTE mode (each word occupies 8-bit) or WORD mode (each word occupies 16-bit), an arbitrary word length can possibly leave part of the SIMD register empty, and eventually causes performance degradation. Algorithms must guarantee the utilization of both SIMD registers has a lower bound of a certain threshold to ensure STTNI will improve the performance.

3) Overhead of loading 128-bit value. When loading 128-bit value from the reference sequence to the SIMD register, the overhead is minimal if the value is a continuous array (stores linearly in the memory). The worst case is that the program needs to load each word into the register individually. From our experiments, STTNI will not provide any benefit in that case. To avoid the SIMD loading overhead, we need to rearrange the data of the application in the memory, if necessary, to ensure that each 128-bit value can be loaded into the SIMD register at one time. If each word in the reference is longer than an STTNI-word, we need to truncate the word into a partial word and make use of the STTNI compare.

From the discussion above we know that, in order to take as much advantage of the STTNI compare as possible, one should 1) use STTNI to compare the sequences as long as possible, 2) use STTNI to compare the sequences which the length of a word is equal or greater than one STTNI-word (8-bit or 16-bit) in STTNI, and 3) arrange the reference sequence to be placed in a continuous memory space.

3 Experimental Configurations

All of our applications run on an Intel Core i7 Model 30 processor with 8 cores; each core operates at 2.80 GHz. The private L1 instruction cache and data cache are both 32KB for each core. Private L2 cache size is 256KB for each core, and L3 cache is 8MB shared across all the cores. We are running 2.6.31 version of Ubuntu 9.10 Linux and compile all the applications with gcc version 4.4.1 at optimization level -O3. Performance results were collected using the built-in performance counters, accessed via the Performance Analysis Programming Interface (PAPI) version 4.0.0.0 [5]. All applications were measured repeatedly and the average execution time is reported.

4 Optimization on Benchmark Applications

We choose the benchmark applications from a diverse set of case studies from multiple disciplines: computer simulation, image processing, database and life science. In terms of performance, our baseline reflects the state of the art. Cache simulation and template matching we used here are considered sequential comparing SR application (first category in 2.2). B+tree algorithm is an SR application based on search tree structure (second category in 2.2). BLAST belongs to hash-table based SR applications (third category in 2.2). We will illustrate how we apply STTNI-based optimization differently and achieve certain performance benefits from them. We primarily focus on CPU performance (speedup). We also discuss the memory performance by presenting the data of cache misses, which is an important factor of CPU performance degradation.

4.1 Cache Simulators

Cache simulators are used for evaluating program behavior, studying cache related issues such as replacement policy, coherence and so forth [6]. Mark Hill in University of Wisconsin - Madison has been working on projects of cache simulators including Tycho, Dinero III and Dinero IV [7]. For architecture level simulation, the cache simulation time is always a major concern. Every generation of cache simulators the developers put significant effort in decreasing the simulation times.

Like real caches, on each reference to a memory address, the tag bits of the corresponding set in the cache will be checked against the higher bits to see if this address hits in the cache or misses. When simulating a set-associative cache or a fully-associative cache, the simulator need to compare high address bits against tag bits of each cache line in the set. This is a slow sequential process and costs most of the simulation time.

Program 1 Pseudo-code for STTNI-based cache simulator.

```
// Pseudo code for cache simulation:
load 16 subtags into SIMD regsiter
set compare mode as Equal Any;
while (i<associativity)
begin
  compares subtags with reference;
  if(match) begin
    compares full tag with reference;
    if(hit)
      return hit location;
  end
  i += 16;
  load next 16 subtags into SIMD register;
end
return miss;
```

Program 1 is the pseudo code of the tag comparing stage of the STTNI-based cache simulator. We improved it by putting the least-significant bits as sub-tags into an additional directory. The number of bits is 8 or 16 depending on the associativity. When doing a comparison, we load at most 16 sub-tags into the SIMD register and compare them with the corresponding bits in the address. This step filters out the unmatched lines. A regular sequential comparison between the high address bits and the entire tag bits of the lines returned from STTNI comparison is followed to confirm whether a hit takes place.

Our aggressive baseline cache simulator has two speedup features. The first one is an address hashing mechanism (not hash table) that start comparing the tag bits from the most-recently used line of this set, rather than sequentially scanning from set 0 to set n-1. The other is a fast return-on-misses mechanism. When there is no hit in the entire set (indicated by a hash table miss), the simulator will return a miss signal fast rather than after checking tag bits of each way in the set. The STTNI scheme is build on top of these optimizations.

Figure 4 shows the speedup of the STTNI-based set associative cache simulation with different associativity. Other parameters such as total cache size or line size do not affect the result significantly as we expect. From the experimental result we can see that the speedup is smaller than 1 when the associativity is smaller than 4. With the associativity increases, the speedup of STTNI-based implementation increases to up to 8.3x when associativity is 64. This is because the number of comparison needed is proportional to the associativity. Although it is also true with STTNI-based implementation, the subtag filtering step helps avoid unnecessary comparisons and thus accelerates the simulation process.

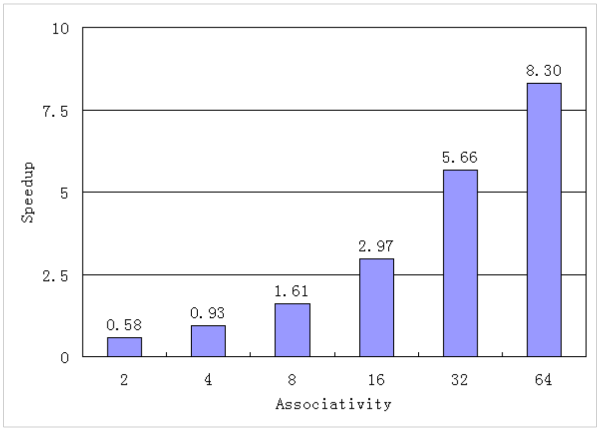


Figure 4: Speedup of STTNI-based cache simulators with respect to the baseline implementation.

L1 Miss	L2 Miss	L3 Miss
124%	136%	100%

Table 3: Cache performance of STTNI-based cache simulators with respect to the baseline implementation.

Table 3 shows the cache performance of the STTNI-based cache simulator when associativity is 8. The numbers of accesses and misses are normalized to the baseline design. From here we can see that by applying STTNI to the cache simulator, the numbers of L1 and L2 misses are increased due to an additional directory we used to store the sub-tags. The L3 misses, however, does not have significant change because the additional directory have a moderate size that can fit in L3 cache well. The memory overhead is insignificant compared to the CPU performance improvement.

4.2 Template Matching

Template matching [10] is a technique to detect similar areas of the template image within a reference image. Basically, the present detection can be attributed to two methods: one is the feature based and the other is the template-based approach. In our experiments, we choose to modify the template-based approach to adapt the advantage of STTNI.

In 1995, J. P. Lewis raises the idea of computing normalized cross correlation in transformation domain to quicken the template matching process [10]. In 2002, H. Schweitzer proves the polynomial based template matching approach can accelerate the normalized correlation approach by a factor of 100 [11]. Shinichiro Omachi and Masako Omachi put forward the Algebraic Template Matching (ATM, [12]) method, which achieves a speedup of 8x 9x comparing to [11]. In our experiment, the baseline algorithm uses direct compare method. Therefore we categorize it as sequential compare SR application.

In our approach, each pixel is defined as 1 byte. We mask the lower bits of each pixel value and use the uppermost bits to compare between the template and the reference images. There are two steps in our approach: for the first round, we find all the possible matching points by comparing the first 16 pixel values (each pixel value is 8 bits and the SIMD register can hold 128 bits) of the template image with the whole reference image pixels; then at the second round, we start from every possible matching point and compare the following 16 pixels in the reference image with related template pixels values. Once no exact match found, we remove the possible matching points and move on to the next. If the whole template area is compared and they are all matched, the matching area is said to be found; otherwise a no matching status is returned.

In our experiment, we use the Open Source Computer Vision Library (OpenCV) to load in pictures and convert its pixel information into a one-dimensional unsigned char array. STTNI Equal Ordered mode is used. The baseline has no STTNI acceleration. The size of the template image is 400*400 pixels and the size of reference images range from 400*400 to 2000*2000.

Figure 5 shows the execution time (in clock cycles) of the STTNI implementation of template matching algorithm and the baseline implementation. We can conclude that when the difference between sizes of reference and template gets bigger, the speed-up will become more apparent. In the best case here, STTNI-based template matching achieves a speedup of 13.8x.

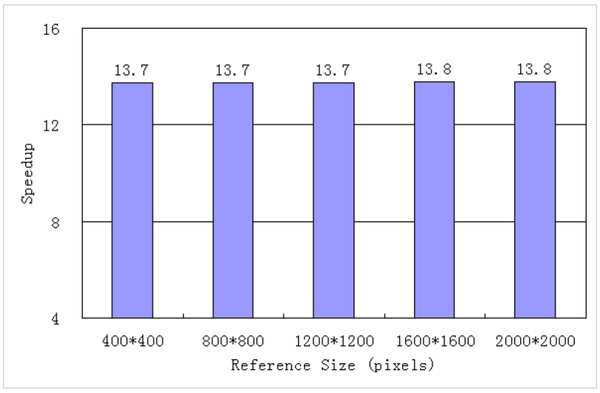


Figure 5: Speedup of STTNI-based template matching with respect to the baseline implementation.

L1 Miss	L2 Miss	L3 Miss
147%	112%	108%

Table 4: Cache performance of STTNI-based template matching with respect to the baseline implementation.

Table 4 shows the cache performance of the STTNI-based template matching algorithm. All values are normalized to the values we obtain from the baseline implementation. As we use an additional data structure to store the truncated search space, the number of L1 misses increased by 47%. This data structure occupies relatively small memory space compared to the reference image. Therefore the numbers of L2 and L3 misses does not increase as much as L1 misses, and therefore does not cause degradation of overall performance.

4.3 B+Tree Algorithm

B+ tree [8] is one kind of tree data structure that allows search, insert and delete operations to finish in logarithmic amortized time. B+ tree is widely used in databases and file systems due to its advantage in I/O operations over other tree structures. In a B+ tree, each node is constituted by an array of words. The words not only store the data, but also indicate the sub-tree to search if the query word is not found in current node. For every operation, B+ tree always needs to search to the correct position for the query word first. B-tree search can be CPU-intensive when the number of nodes or the number of words in a node is very large. Therefore B+ tree search time has always been a critical issue for the developers.

Program 2 Pseudo-code for STTNI-based B-tree search.

```
// Pseudo code for B-tree search:
load 4 pairs of words;
set compare mode as Ranges;
while (i<number of words in current node)
begin
  compare query word with 4 word pairs;
  if (query word within ranges) begin
    compare query word sequentially;
    return hit location;
  end
  else
    shift SIMD register content by 8 bits;
    compare query word with the rest 3 word pairs;
    if (query word within ranges) begin
      compare query word sequentially;
      return hit location;
    end
  else
    i += 8;
    load next 4 pairs of words;
  end
end
return miss;
```

We use STTNI to optimize the B-tree search within a node. Program 2 shows the pseudo code of the STTNI-based B-tree search within a node. In order to utilize STTNI, we define the word as 16-bits each. For longer words, our approach is also applicable because we can compare the least significant 16 bits of the word before a full-word compare. STTNI Range mode is chosen. For each iteration, we load at most 8 words into the mm register. 8 words can define 7 ranges of words, but STTNI define ranges by pairs of words in the SIMD register, therefore 4 ranges are defined. Then we do STTNI comparison between the query word and the 4 ranges to see if the query word falls into one of the ranges. In order to cover the rest 3 ranges, we right shift the 128-bit value by 16 bits to constitute new ranges in the register. Another STTNI comparison is followed if the first one returns no match. We do the sequential comparison only when a match is found by STTNI. Therefore the numbers of iterations needed is decreased by 4 times, approximately.

Our implementation of B+ tree is based on the algorithm described in [9]. The baseline design does sequential comparisons within a node. From the result in figure 6 we observe that once the number of words in a node increases, the STTNI-based implementation outperforms the baseline. In the best case the speed up is approximately 2.32x. A log2-based search within a node can improve the performance by always comparing the word in the middle of the range with the query word. However, STTNI acceleration phase can be built along with the log2-based search after proper justifications of the data structure. Table 5 shows the cache performance of STTNI-based B-tree search algorithm with the maximum number of words. All values are normalized to the values of the baseline implementation. From the data we can see the revised SR stage have little impact on cache performance. Since we do no have additional data structure to support the SIMD compare, there is no additional memory pressure.

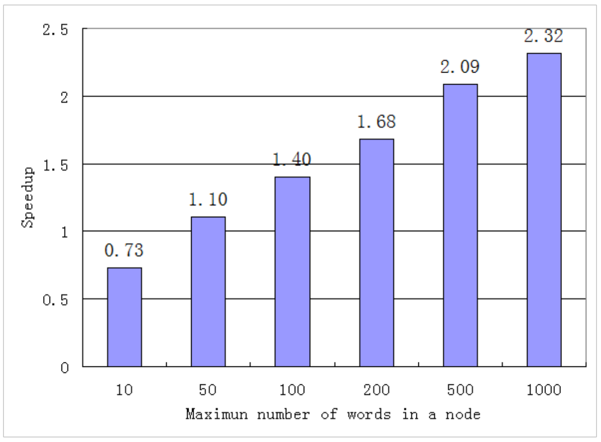


Figure 6: Speedup of STTNI-based B+ tree algorithm with respect to the baseline implementation. Total number of nodes in the B+ tree is 10^6.

L1 Miss	L2 Miss	L3 Miss
104%	97%	102%

Table 5: Cache performance of B+ tree algorithm with respect to the baseline implementation.

4.4 BLAST

Basic Local Alignment Search Tool [13], or BLAST, is an application that scans a biological query sequence, such as the amino-acid sequence of different proteins or nucleotides of DNA sequences, and finds the similarity between it and the subject sequences in the database. In the preprocessing stage of BLAST, every word appearing in the query sequence is stored in a word list. Usually it is a data structure that is efficient for searching. When scanning the subject sequences in the database, each word is checked against the word list of query sequence, and all matches will return both positions of this word in query sequence and in database to the program. Because of the huge volume of the database, usually the subject sequence scanning stage costs about 65% of the CPU time of the entire BLAST program.

Although BLAST is developed as a replacement of more time-consuming sequence alignment algorithms such as Smith-Waterman process, scientists and researchers still put considerable attentions on accelerating it. Different versions of BLAST toolboxes including NCBI BLAST [14], WU-BLAST (now acquired by AB-BLAST [15]) were developed. Different program of BLAST is developed for improving the performance under different applications such as [16] [17]. Some other local alignment algorithms such as RPAlign [18] were developed as fast and accurate alternatives.

We choose NCBI BLAST [14], one of the most widely used BLAST applications in the world, as our baseline design. In NCBI BLAST, a hash table is created, and the hash function encodes each word into a unique value. As the advantage of hash tables, the SR algorithm access the hash table once and retrieve the number of occurrence of each word as well as the offset of each word in the query sequence, instead of comparing subject sequences with the query sequence repeatedly. The disadvantage of this approach is the huge size of the hash table, which is proportional to the number of bits in an encoded word. In DNA sequences, typically a word contains 11 residues and each residue needs 2 bits. Therefore, to create a hash table for all possible 11-letter words, the entries in the hash table will be $(2 * 2)^{11}$=4M. Because of the significant size of the hash table, cache performance becomes a critical issue to the overall performance.

However, usually not all the words exist in the query sequence, which means that many of the entries in the hash table are empty. NCBI BLAST developers have realized the cache performance issue and optimize the program by two means. First, it estimates the number of entries needed and decreases the size of the hash table if it is possible. Therefore even if the hash table word length is smaller than the actual word size, there will be enough entries for each word so the accuracy is not sacrificed too much. Second, it allocates a smaller PV array to record if a word has a hit in the query or not. When scanning the subject sequence, the program will first access the PV array. If there is no hit in the query sequence, the program will skip accessing the hash table in order to save CPU time.

We apply our optimization scheme discussed in 2.2 3) (figure 3(c)) here. 4 letters from LSB skip the hash function and used to compare directly with keys stored in the entries of a hash bucket. The rest part of the word is hashed in order to find the bucket. Initially, 16 entries are allocated in each bucket. Whenever there are more than 16 entries needed, the hash table size is doubled and the number of compared bits is increase by one. In the best case, the hash table is 16 times smaller than the baseline hash table. Although the number of comparisons in the SR stage is increased, STTNI compares 16 tags a time and thus minimizes the overhead.

We choose the "nt" database from NCBI in our experiment. It is a nucleotide sequence database with entries from all traditional divisions of GenBank, EMBL, and DDBJ. The size of this database is 10.5 GB in total. For this experiment, in order to study sensitivity of performance to word length, we modified original program so that the hash table word length is set manually instead of being determined automatically by the program. We run the BLAST program with query sequences of different length from NCBI benchmark. Figure 7 shows the speedup of the STTNI-version subject sequence scanning stage of BLAST compared with the baseline. The sequences are DNA sequences and one letter consumes two bits. The range of the hash table word length is from 8 to 12. 1 letter increase in the hash table word length will increase the number of entries in the hash table to four times larger. Since we are not accelerating the comparison directly, the speedup is not as much as that in previous applications. In the best case, STTNI-based implementation achieves a speedup of 1.47x.

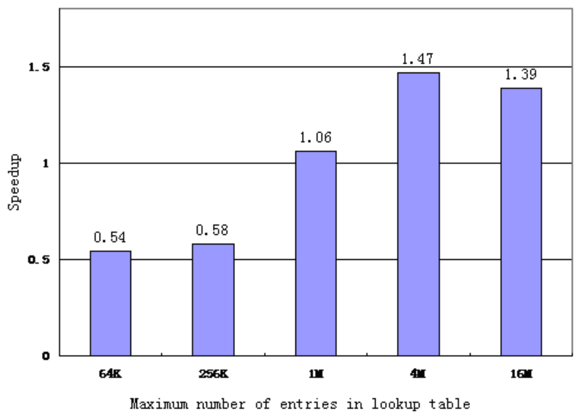


Figure 7: Speed up of the STTNI-based BLAST subject sequence scanning stage with respect to the baseline implementation.

Table 6 shows the cache performance of BLAST with STTNI-based optimization when the maximum number of entries. All values are normalized to the values we obtain from the baseline implementation. The increase in L1 and L2 misses (36% and 37%, respectively) are due to additional computation needed to calculate the new index value. The L3 miss decreases by 63%. This observation is consistent with our expectation. Since we reduced the index bits by adding one compare in the SR stage of BLAST, the new hash table size is decreased and a larger portion of it can fit in the L3 cache. From the cache performance result we can conclude that the speedup in figure 12 comes from the decreased L3 miss penalties.

L1 Miss	L2 Miss	L3 Miss
136%	137%	37%

Table 6: Cache performance of the STTNI-based BLAST subject sequence scanning stage with respect to the baseline implementation.

4.5 Reasons of Diverse Performance Improvement

One may notice that the range of speedup is relatively large (from 1.4x to 13x). We discuss the most significant factors here.

1) Percentage of parallel computing in the overall execution. Amdahl's law parallelization 1/(1-P+P/N) can illustrate the influence of parallel portion in the program. Here N is the parallelism factor provided by STTNI. Note that N can be greater than 16 because the STTNI actually compare each word in one sequence with every word in another sequence, and the result is obtained from a 16*16 matrix. Therefore, the maximum value of N is 256.

2) Different natures of SR application. As illustrated in section 2.2, sequential compare requires the least extra computation, and thus it achieves the speedup from STTNI directly. In our case, template matching and cache simulation achieves the highest speedup. Hash tables are highly tuned for performance requirements. STTNI is used to reduce the memory overhead, and the performance gain comes indirectly. Therefore, BLAST has the least performance improvement.

3) Mode option in STTNI. As illustrated in [2], although the STTNI generates a 16*16 matrix for each operation, the return value may not use all of them depending on in which mode it is comparing. For example, Equal Each mode will only return elements on the diagonal, and Equal Order mode will only return elements at row i column j, when i¿=j. As a result, the achievable parallelism is different for each mode.

4) Overhead discussed in section 2.3. Initialization overhead is similar for most benchmarks. SIMD register utilization and memory loading overhead are very different among SR applications. For example, B+ tree algorithm rearranging the words into an array spends more CPU time rearranging the words into an array than in cache simulation and template matching. That is the reason why B+ tree search achieve less speedup than cache simulation and template matching even if more than a hundred keys are stored in each node.

5 Conclusion

We propose an optimization technique based on STTNI for SR applications, and propose the STTNI implementation of four diverse applications: cache simulation, B-tree search, template matching and BLAST. We use different strategies in each application in order to improve the performance as much as possible. In most applications, the speedup of STTNI-based implementations increases with the volume of the workload. Our proposed implementation can be adapted to many other applications with SR workloads. STTNI will not conflict with existing optimizations. We can expect an accumulated performance improvement. The design effort of each application is different depending on the nature of the application itself. In our case, cache simulation requires the least and BLAST requires the most.

In future, we are looking forward to revising more applications with STTNI. In addition to the programmer revising approach, we want to investigate automatic code generation. We also want to revise more applications and their optimized versions so that we can combine the performance improvement of the optimization with the benefit from STTNI. We are going to revise the applications in a finer granularity, not only the search and compare stage but also the whole application to cooperate with the new feature of STTNI. We also plan to further evaluate the memory behavior of STTNI-based applications, including cache performance and memory space consumption. Finally, we want to study the power overhead of these applications.

References

[1] P. Lymen and H. Varian, *How Much Information*, UC-Berkeley Project, 2003. Available: http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/

[2] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, 2009

[3] Z. Lei, "XML Parsing Accelerator with Intel Streaming SIMD Extensions 4 (Intel SSE4)", Intel Corporation, 2008. Available: http://software.intel.com/en-us/articles/xml-parsing-accelerator-with-intel-streaming-simd-extensions-4-intel-sse4/

[4] Y. Le, "Schema Validation with Intel Streaming SIMD Extensions 4 (Intel SSE4)", Intel Corporation, 2008. Available: http://software.intel.com/en-us/articles/schema-validation-with-intel-streaming-simd-extensions-4-intel-sse4/

[5] Performance Application Programming Interface. Available: http://icl.cs.utk.edu/papi/

[6] H.G. Rotithor, "On the effective use of a cache memory simulator in a computer architecture course", *IEEE Trans. Education*, vol. 38(4), pp. 357360, 1995.

[7] M.D. Hill, Dinero Project. Available: http://pages.cs.wisc.edu/ markhill/DineroIV/

[8] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes", *Acta Informatica* vol. 1 (3), 1972, pp. 173189

[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest and Clifford Stein, *Introduction to Algorithms*, 2nd ed., MIT Press and McGraw-Hill. ISBN 0-262-53196-8, 2001.

[10] J.P. Lewis, "Fast Template Matching", *Vision Interface*, 1995

[11] H. Schweitzer, J. W. Bell, and F. Wu, "Very fast template matching", in *Proc. 7th Eur. Conf. Computer Vision IV*, 2002, pp. 358372.

[12] Shinichiro Omachi, Masako Omachi, "Fast Template Matching With Polynomials", *IEEE Trans Image Processing*, vol. 16 (8), 2007

[13] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, D.J. Lipman, "Basic local alignment search tool". *J Mol Biol*, vol. 215 (3), pp. 403410, 1990.

[14] National Center for Biotechnology Information. Available: http://blast.ncbi.nlm.nih.gov/

[15] Advanced Biocomputing, LLC, Available: http://www.advbiocomp.com/blast.html

[16] S. F. Altschul, T. L.Madden, A. A. Schaffer, J. Zhang, Z. Zhang,W. Miller, and D. J. Lipman, "Gapped BLAST and PSI-BLAST: A new generation of protein database search programs, *Nucleic Acids Res*, vol. 25, pp. 33893402, 1997.

[17] Z. Zhang, S. Schwartz, L.Wagner, andW.Miller, "A greedy algorithm for aligning DNA sequences", *J. Comput. Biol.*, vol. 7, pp. 203214, 2000.

[18] S. Bandyopadhyay and R.Mitra, "A Parallel Pairwise Local Sequence Alignment Algorithm", *International Conference on Advanced Communication Technology*, vol.3 pp. 2317 2320, 2009.