

Microarchitecture Characteristics and Implications of Alignment of Multiple Bioinformatics Sequences

Yue Li

Department of ECE
University of Florida
yli@ece.ufl.edu

Lan Luo

Department of CISE
University of Florida
luolan@ufl.edu

Tao Li

Department of ECE
University of Florida
taoli@ece.ufl.edu

Tamer Kahveci

Department of CISE
University of Florida
tamer@cise.ufl.edu

Abstract

With the growth of bioinformatics and computational biology industry, multiple sequence alignment (MSA) applications have become an important emerging workload. In spite of the large amount of recent attention given to the MSA software design, there has been little quantitative understanding of the performance of such applications on modern microprocessors and systems. In this paper, we analyze performance and characteristics of MSA software from the perspective of processor microarchitecture. We use twelve popular MSA programs employing a wide variety of alignment approaches. The basic workload characteristics and the efficiencies of various microarchitecture features, such as trace cache, out-of-order execution, caching, branch prediction, speculative execution and phase behavior are examined and analyzed on the Pentium 4 microarchitecture.

Our major observations of this work are: (1) Instruction footprints of MSA programs are typically small and can fit in the L1 instruction cache. The trace cache shows high percentage of time in deliver mode. (2) Loads and stores account for the 60% of dynamic instructions executed. This indicates that further improving memory bandwidth will be beneficial to the performance of MSA software. (3) Prefetching and large L2 cache can efficiently handle the working sets of a majority of the studied benchmarks. Nevertheless, MSA software using exact alignment method yields poor cache performance. The data TLB behavior largely depends on the alignment methods used. (4) The studied MSA applications show large variation in dynamic branch frequency and mix. The indirect branches, calls, and returns can be predicted with high accuracy. The overall branch misprediction rates exceed 5% on half of the examined benchmarks. (5) The IPC of the studied benchmarks ranges from 0.15 to 0.93. Overall, the processor speculatively executes 27% more instructions than it retires.

1. Introduction

In the last few decades, advances in molecular biology and laboratory equipments have allowed the increasingly rapid obtaining of enormous amount of genomic and proteomic data [1]. Bioinformatics explores computational methods to allow researchers to sift through this massive biological data in order to provide useful information. Bioinformatics applications are widely used in many areas of life science, such as drug design, human therapeutics, forensics, and homeland security. A number of recent market research reports estimate the size of the bioinformatics market is projected to grow to \$176 billion by 2005, and \$243 billion by 2010 [2].

Multiple sequence alignment (MSA), which lines up multiple genomes, is one of the most important applications in the bioinformatics [3]. MSA plays a vital role in analyzing genomic data and understanding the biological significance and functionality of genes and proteins. Predicting the structure and functions of proteins, classification of proteins, and phylogenetic analysis are a few examples to countless applications that use MSA.

Finding the optimal MSA for a given set of genomes is an NP-complete problem [4]. A significant body of work has been done to find heuristic solutions to the MSA problem [5]. However, there is little quantitative understanding of the performance of these MSA methods on modern microprocessor and memory architecture. To ensure good hardware performance, a detailed characterization of how the MSA software uses various microarchitectural features provided by the contemporary microprocessor is needed. Adhering to this philosophy, this paper studies the performance and characteristics of twelve widely used MSA programs on Intel Pentium-4 microarchitecture [6]. We examine basic workload characteristics and efficiencies of caching, TLBs, out-of-order execution, branch prediction and speculative execution.

We chose Pentium 4 architecture due to its advanced design and popularity. Since MSA benchmarks are not well-known from the architecture perspective, we believe that an in-depth analysis of a wide variety of MSA software on the representative architecture is crucial in understanding the implications of bioinformatics multiple sequence alignment tools on today's market. Although the characteristics of MSA applications may vary for different architectures, we believe that our experiments are broad enough from the perspective of bioinformatics market needs. Note that our goal in this paper is not to develop new MSA software, but to explore how advanced microarchitecture behaves for existing MSA applications.

The rest of the paper is organized as follows. Section 2 provides a background of MSA and describes the selected programs. Section 3 describes the experimental methodology. Section 4 presents the detailed characterization of MSA applications and their architectural implications. Section 5 summarizes the major findings of this work and concludes the paper.

2. Background: Multiple Sequence Alignment (MSA)

Studying evolutionary relationships between sequences is one of the main goals of bioinformatics. The majority of

biological sequences are DNA and protein sequences. A DNA sequence is made from an alphabet of four elements, namely A, T, C, and G, called nucleotides. A protein can be regarded as a sequence of amino acids. There are twenty distinct amino acids. Thus, a protein can be regarded as a sequence defined on an alphabet of size twenty.

MSA is the process of aligning three or more sequences with each other so as to match as many residues (nucleotides or amino acids) as possible. Alignment of multiple sequences involves placing the residues that derive from a common ancestor to the same column. This is achieved by introducing gaps (which represent insertions or deletions) into sequences. Thus, an alignment is a hypothetical model of mutations (substitutions, insertions, and deletions) that occurred during sequence evolution. The best alignment will be the one that represents the most likely evolutionary scenario. Sometimes, the evolution history of the sequences can not be determined precisely. In such cases, usually a computable measure, such as sum-of-pairs score is used to determine the quality of the multiple alignments. Sum-of-pairs score is defined as the sum of the scores of the underlying alignments of all pairs of sequences in the resulting multiple alignment, where a score is computed for a pair of sequences based on the matching and mismatching characters. Figure 1 shows a multiple alignment among the DNA sequences A = "AGGTCAGTCTAGGAC", B="GGACTGAGGTC", and C="GAGGACTGGCTACGGAC".

```

Sequence A  - AGGTCAGTCTA - GGAC
Sequence B  - - GGACTGA - - - - GGTC
Sequence C  GAGGACTGGCTACGGAC

```

Figure 1. An example of MSA (The aligned DNA sequences match in seven positions).

The number of multiple sequence alignment methods has been increased steadily. Most MSA algorithms can be classified as one of the following categories: exact, progressive, iterative, anchor-based and probabilistic methods. Given a set of sequences, exact methods deliver an alignment optimal with respect to a computable objective function, such as sum-of-pairs score, through exhaustive search. Progressive methods find a multiple alignment by iteratively picking two sequences from this set and replacing them with their alignment (i.e., consensus sequence) until all sequences are aligned into a single consensus sequence. Thus, progressive methods guarantee that never more than two sequences are simultaneously aligned. The choice of sequence pairs is the main difference among various progressive methods. Iterative methods start with an initial alignment; they then repeatedly refine this alignment through a series of iterations until no more improvements can be made. Depending on the strategy used to improve the alignment, iterative methods can be deterministic or stochastic. Anchor-based methods use local motifs (short common subsequences) as anchors. Later, the unaligned regions between consecutive anchors are aligned using other techniques. Probabilistic methods pre-compute the substitution probabilities by analyzing known multiple

alignments. They use these probabilities to maximize the substitution probabilities for a given set of sequences.

Of the many algorithms, we selected a subset of 12 programs based on their popularity, availability, and how representative they were of aligning multiple sequences in general. We briefly describe the MSA tools we selected.

Msa [7] uses high-dimensional dynamic programming (DP) to exhaustively produce all possible alignments of the input sequences. The number of dimensions is equal to the number of sequences compared. It then chooses the alignment with the highest sum-of-pairs score. It uses the distances between pairs of sequences to eliminate unpromising alignments to improve efficiency.

Clustal w [8] first finds a phylogenetic tree for the multiple sequences to be aligned. Phylogenetic tree shows the ancestral relationships among sequences. If two sequences are derived from the same ancestor, then they are located in a subtree rooted at their parent. *Clustal w* progressively aligns pairs of sequences that are siblings on this tree starting from the leaf nodes until all sequences are aligned.

Trealign [9] is similar to *Clustal w*. It builds a phylogenetic tree with minimum parsimony on the input sequences. It then aligns pairs of these sequences using dynamic programming starting from the tips of the phylogenetic tree.

T-coffee [10] computes the distance between every pair of sequences. It then computes a phylogenetic tree from these distances using neighbor joining method. It uses this tree as a guide tree to align sequences progressively.

Poa [11] progressively aligns pairs of sequences. Unlike *Clustal w*, *Poa* represents each sequence or the alignment of multiple sequences using graphs. Every node of this graph corresponds to a nucleotide. *Poa* aligns such graphs, instead of sequences, at every step until all sequences are aligned.

Probcons [12] uses a Hidden Markov Model (HMM) to compute the posterior probability of aligning every pair of letters. It then builds a guide tree (similar to phylogenetic tree) for the given sequences using these probability values. Finally, it aligns the sequences progressively by following the guide tree.

SAGA [13] employs genetic algorithm to optimize the sum-of-pairs score of the multiple alignment. It first finds a population of possible solutions. These solutions are then updated with random mutations iteratively to find better alignments.

Muscle [14] computes a k-mer (subsequence of length k) distance for every pair of sequences. Next, it builds a guide tree using these distances. It progressively aligns sequences with the help of this guide tree. Later, it iteratively computes the Kimura distance between aligned nucleotides and realigns the sequences.

Mavid [16] finds common subsequences with the help of a suffix tree. It then chooses such subsequences to align

sequences at these positions. *Mavid* uses anchor-based method for alignments of large numbers of DNA sequences.

Mafft [17] converts nucleotide sequences to sequences of real numbers by storing the volume and polarity of each nucleotide. It assumes that two subsequences are similar if they have similar volume and polarities. *Mafft* uses fast Fourier transformation of these sequences to find the positions where these sequences have similar volumes and polarities. These positions are used as anchors and the nucleotides at these positions are aligned together.

Dialign [18] aligns pairs of sequences to find long gap-free similar subsequences using dynamic programming. Later, it greedily chooses the subsequence pairs with largest similarity score and anchors two sequences at that location until all similar subsequences are exhausted. If the position of such a subsequence conflicts with an existing anchor, then that subsequence is discarded.

Hmmer [19] employs Hidden Markov Models (profile HMMs) for aligning multiple sequences. Profile HMMs are statistical models of multiple sequence alignments. They capture position-specific information about how conserved each column of the alignment is, and which residues are likely.

Table 1 summarizes the selected MSA programs and their algorithm categories.

Table 1. Five main classes of MSA and the selected programs for each class.

Type	Tool
Exact	<i>Msa</i>
Progressive	<i>Clustal w, Treealign, Poa, Probcons, Muscle, T-coffee</i>
Iterative	<i>SAGA, Muscle</i>
Anchor-based	<i>Mafft, Dialign, Mavid</i>
Probabilistic	<i>SAGA, Hmmer, Probcons, Muscle</i>

3. Methodology

To observe the architectural characteristics of MSA algorithms and how they utilize various microarchitecture features, we conducted our experiments using hardware performance counters. This section describes our experimental setup.

3.1. System Configuration

All experiments were run on a 3GHz Pentium 4 (Prescott) processor [6] with 1GB of DRAM running RedHat 9.0 Linux kernel version 2.4.26. All MSA benchmarks were compiled using Intel’s C/C++ Linux compilers with the maximum level of optimizations. The input datasets for the MSA benchmarks were chosen from a highly popular biological database – the National Center for Biotechnology Information (NCBI) [20] *Bacteria genomes* databases. In this study, the 317 *Ureaplasma’s* gene sequences [21] were used as the inputs for

all the MSA benchmarks. All MSA benchmarks were executed to completion.

3.2. Pentium 4 Microarchitecture

The front end of the Prescott micro-architecture fetches and decodes x86 instructions. It builds the decoded instruction into sequences of μ ops called traces, which are stored in the execution trace cache. The Pentium 4 processors have two areas where branch predictions are performed - in the front-end of the pipeline, and at the execution trace cache (the trace cache uses branch prediction when it builds a trace). The pipeline in Prescott has 31 stages, so a pipeline flush due to poor branch prediction can result in a much larger clock cycle penalty. The front-end BTB (Branch Target Buffer, 4K entries) is accessed on a trace cache miss and smaller Trace-cache BTB (2K entries) is used to detect next trace line. The trace cache BTB, together with the front-end BTB, uses a highly advanced branch prediction algorithm. Static branch prediction will occur at decode time if the front-end BTB has no dynamic branch prediction data for a particular branch. Dynamic branch prediction accuracy is also enhanced by adding an indirect branch predictor. The out-of-order execution engine, which consists of the allocation, renaming, and scheduling functions, can issue three μ ops per cycle to the next pipeline stage. To exploit the instruction level parallelism (ILP) in the programs, the Prescott microarchitecture provides a very large window of instructions (up to 126) from which the execution units can choose.

Prescott memory subsystem contains an 8-way, 16KB L1 data cache and an 8-way, 1MB, write-back L2 unified cache with 128 bytes/cache line. The levels in the cache hierarchy are not inclusive. All caches use a pseudo-LRU (least recently used) replacement algorithm. The Pentium 4 microarchitecture supports both hardware and software controlled prefetching mechanisms.

3.3. Pentium 4 Hardware Counters

We used the Pentium 4 hardware counters to measure various architectural events [22]. The Pentium 4 performance counting hardware includes 18 hardware counters that can count 18 different events simultaneously in parallel with pipeline execution. The 18 Counter Configuration Control Registers (CCCRs), each associated with a unique counter, configure the counters for specific counting schemes such as event filtering and interrupt generation. The 45 Event Selection Control Registers (ESCRs) specify the hardware events to be counted and some additional Model Specific Registers (MSRs) for special mechanisms like replay tagging [23]. These counters collect various statistics including the number and type of retired instructions, mispredicted branches, cache misses etc. We used a total of 59 event types for the data presented in this paper.

4. Workload Characteristics

This section provides a detailed workload characterization of MSA benchmarks on the studied microarchitecture. The examined architectural features include instruction

distribution, out of order execution, cache and TLB performance, branch and efficiency of branch prediction.

4.1. Instruction Characteristics

The total number of instructions executed on the studied MSA workloads ranges from hundreds of billion to thousands of billion. This indicates that the computation requirement to align a large set of DNA/protein sequences is non-trivial. The using of performance counters (instead of simulation) allows us to examine the entire program characteristics running on the realistic and meaningful data sets.

Figure 2 presents the dynamic instruction profile of the MSA programs. The dynamic instructions are broken down into five categories: load, store, branch, floating point (FP) and integer. As it can be seen, the most frequently executed instructions are loads. This is because all these tools need to read data from the dynamic programming matrix and write the results back onto the same matrix many times. The percentage of loads is significantly more than that of store in all the programs since the dynamic programming (DP) algorithm has to read multiple entries from the DP matrix to update a single entry. As a whole, memory operations occupy a significant share of the total instructions mix, which is 63 percent on average. Therefore, MSA workloads are data-centric in nature. This indicates that MSA applications can benefit from techniques to improve memory bandwidth in general.

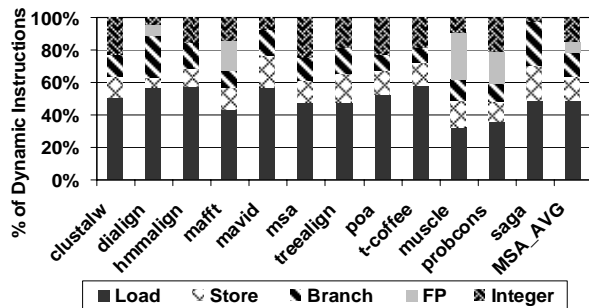


Figure 2. Dynamic operations profile.

Branch instructions exhibit significant differences from algorithm to algorithm. For example, 27% of dynamic instructions in benchmarks *Dialign* and *SAGA* are branches. This can be explained as follows. *Dialign* usually generates a large candidate set of anchors which then needs to be analyzed to find a set of non-conflicting anchors. This analysis involves a large number of comparisons among candidate anchors. *SAGA* evaluates and compares all the members of the solution population at per iteration. As the population of solutions and the number of iterations increases, the number of comparisons also increases. A more detailed analysis on the branches and branch prediction can be found in the Section 4.6. The majority of MSA workloads contain few floating point operations. Only methods that calculate statistics and likelihood values or phylogenetic trees in their algorithms use floating point instructions. For example, *Mafft* computes the Fourier transformations of the volumes and polarities of the amino acids for different combinations of sequences. *Muscle* incurs floating point operations during the computation of

Kimura distances. *Probcons* computes the posterior probability of aligning every pair of letters.

4.2. IPC and μ PC

Using the events that count the number of cycles and number of instructions retired during the program execution, we computed the IPC (instruction-per-cycle) of the studied MSA benchmarks. On the high-performance processors such as Pentium 4, the IPC metric indicates how efficiently the microprocessors exploit instruction level parallelism (ILP). In order to improve the efficiency of superscalar execution and the parallelism of programs, each x86 instruction is further translated into one or more μ ops inside the Pentium 4 processor. Typically, a simple instruction is translated into around one to three μ ops. The results of the measured IPC and μ ops per cycle (μ PC) on the benchmarks are shown in Figure 3.

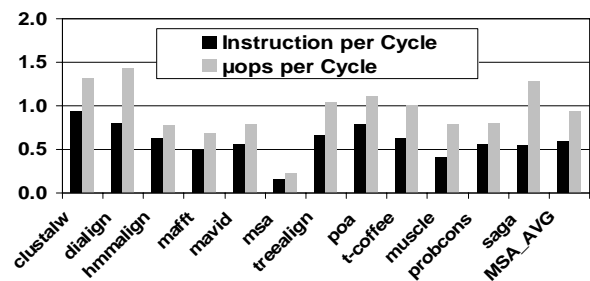


Figure 3. IPC vs. μ PC.

The greatest IPC values come from *Clustal w*, *Dialign* and *Poa*. The lowest IPC values are *Msa* and *Muscle*. The IPC ranges from 0.15 to 0.93, with an average around 0.60. A lower IPC can be caused by an increase in cache misses, branch mispredictions, or pipeline stalls in the CPU. For example, MSA methods (*Mafft* and *Muscle*) extensively using floating point instructions yield lower IPCs due to the pipeline stalls on the long latency floating point operations. The IPC is remarkably low on benchmark *Msa* due to the excessive data cache misses. These cache misses are incurred since the exhaustive search strategy of *Msa* reads and writes large amounts of data. The μ PC ranges from 0.23 to 1.32, with an average around 0.94. Only six benchmarks (*Clustal w*, *Dialign*, *Treealign*, *Poa*, *T-coffee* and *SAGA*) achieve more than one μ ops per cycle. This implies that, for majority of MSA applications, the available ILP that can be exploited by the Pentium 4 microarchitecture is limited.

4.3. Trace Cache

As the front end, the Prescott trace cache sends up to 3 μ ops per cycle directly to the out-of-order execution engine, without the need for them to pass through the decoding logic. Only when there is a trace cache miss does the front-end fetches x86 instructions from the L2 cache. There are some exceedingly long x86 instructions (e.g., the string manipulation instructions) that decode into hundreds of μ ops. For these long instructions, the Prescott fetches μ ops from a special μ ops ROM that stores the canned μ ops sequence.

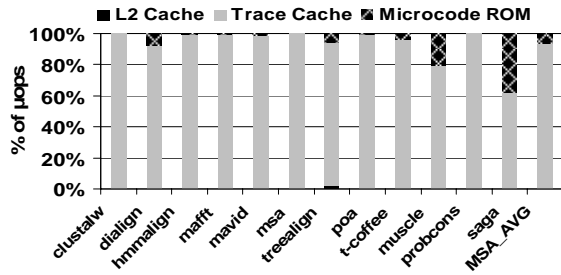


Figure 4. Source of the μops.

Figure 4 shows the proportion of the μops fetched from the L2 cache, the trace cache, and the μops ROM respectively. As can be seen, a dominant fraction (93%) of the μops is supplied by the trace cache. On benchmarks *Dialign*, *Mavid*, *Treealign* and *T-coffee*, around 2%-8% of the μops come from the μops ROM, implying that these workloads use x86 complex instructions more frequently. The μops ROM contributes 20% and 39% of the dynamically executed μops on benchmark *Muscle* and *SAGA*. This is because these two programs excessively use the string manipulation instructions to handle biological sequences. For example, *SAGA* repeatedly mutates existing population of alignments which involves costly string operations. Moreover, *SAGA* uses x86 FSQRT (floating point square root) instructions. The L2 cache contributes to less than 1% of the μops on most of the benchmarks (except *Treealign*). The instruction footprint generated by benchmark *Treealign* yields more trace cache misses. A closer investigation shows that this benchmark performs operations on both graphs and phylogenetic trees alternately. The codes performing these two operations conflict with each other in the trace cache. Nevertheless, on the majority of benchmarks, the Prescott trace cache is highly efficient in providing the μops to the rest of the pipeline. This indicates that the instruction footprints of MSA applications are small and cache misses due to instruction fetches are negligible.

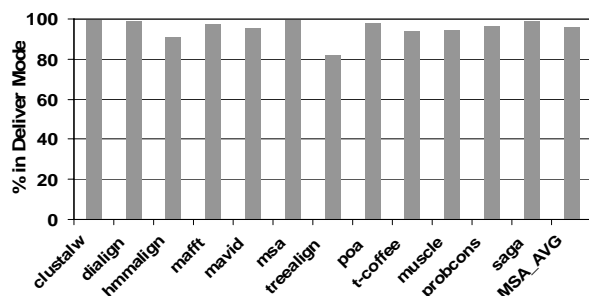


Figure 5. % of TC deliver mode.

The trace cache operates in two modes: deliver mode and build mode. The deliver mode is the mode in which the trace cache is feeding stored traces to the execution logic to be executed. This is the mode that the trace cache normally runs in. When there is a trace cache miss, the trace cache goes into build mode. In this mode, the front end fetches x86 instructions from the L2 cache, translates into μops, builds a trace segment with it, and loads that segment into the trace

cache to be executed. Figure 5 shows the percentage of non-sleep cycles that the trace cache is delivering μops vs. decoding and building traces. Overall, the utilization of the trace cache is extremely high except on the benchmarks *Treealign*.

4.4. Cache Misses

Figure 6 presents the counts of cache misses per 1000 instruction retired. We see that instruction related cache misses are nearly fully satisfied by the trace cache. Data cache miss ratios are higher because the data footprint is much larger than the instruction footprint. For example, *Msa* can cause more than 40 L1 data cache misses on every thousand instructions executed. This can be explained as follows. Unlike other methods, *Msa* fills a multi-dimensional dynamic programming matrix. As the number of dimensions (i.e., the number of sequences compared) increases, the number of matrix entries needed to compute a single DP entry increases exponentially. Thus, these entries do not fit into cache resulting in cache misses every time a new value is computed. On the average, the studied bioinformatics applications generate 11 L1 cache misses per thousand retired instructions.

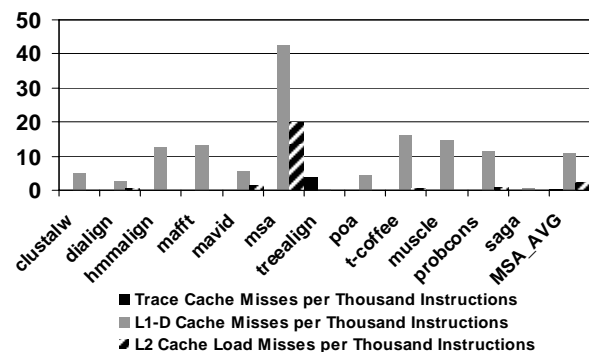


Figure 6. Cache miss rates.

We found that the L1 data cache misses on most of the benchmarks can be nearly fully satisfied by the L2 cache. The Pentium 4 processors use automatic hardware prefetch to bring cache lines into the unified L2 cache based on prior reference patterns. Prefetching is beneficial because many accesses to the biological sequences are sequential, and thus, predictable.

Interestingly, the benchmarks (*Msa*) with the highest L1 data cache misses also have the highest L2 misses, implying their poor data locality. This is because, in order to compute the alignment score for an entry of the DP matrix, *Msa* needs to access the information in all neighboring entries of that entry. As the dimensionality of the DP matrix (i.e., number of sequences) increases, the location of these entries gets exponentially far away from each other causing poor data locality. Figures 3 and 6 show a fairly strong correlation between the L2 misses and IPC, which indicates that the L2 miss latency is more difficult to be completely overlapped by out-of-order execution. We observed that overall prefetching and L2 cache can efficiently handle the working sets of MSA applications.

4.5. TLB Misses

The Pentium 4 processor uses separate TLB (Translation Lookaside Buffer) to translate the virtual address into physical address for instruction and data accesses. Prescott has a 128-entry, fully-associative instruction TLB (ITLB) and a 64-entry, fully associative data TLB (DTLB). Figure 7 presents the ITLB and DTLB miss rates across the studied benchmarks. The ITLB miss rates are well below 1% on most benchmarks. Figure 7 also shows that most of the DTLB accesses can be handled very well by the Pentium 4 processor. Nevertheless, *msa* yields high (16%) DTLB miss rates due to its large data memory footprint. This is mainly caused by the high dimensional DP matrix that *Msa* uses. Since all MSA software run the same input data set, it is clear that the internal data structures created by the algorithms largely affect the DTLB behavior.

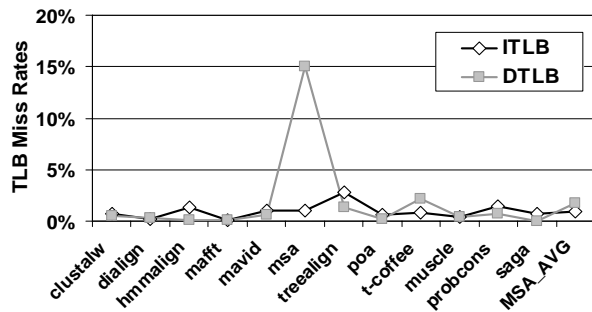


Figure 7. TLB miss rates.

4.6. Branches and Branch Prediction

Figure 8 presents the fraction of branches that belong to conditional branches, indirect branches, calls and returns. Conditional branches, ranging from 54% (*Muscle*) to 99% (*Hmalign*) of the dynamic branches, dominate the control flow transfers in the MSA applications. Indirect branches account for more than 10% of the dynamic branches on benchmarks *Treealign*, *Muscle* and *SAGA*. We further examined the source code and found that the percentage of indirect branches is caused by the software programming style and they are not an inherent part of the algorithm. For example, the benchmarks *Treealign*, *T-coffee* and *SAGA* embed the case-switch statements in various loops to determine sequence format, or to select one operation from all possible choices to process the sequence elements. The benchmark *Muscle*, programmed with C++, uses additional virtual functions to implement the algorithm. On the average, conditional branches, indirect branches, call, and return contribute to the 81%, 8%, 6% and 6% of the total dynamic branches respectively.

Figure 9 shows the branch misprediction rates on the MSA applications. The overall branch misprediction rates exceed 5% on 6 out of the 12 benchmarks. The misprediction rates on the indirect branches are less than 2% on all studied benchmarks. Typically, the targets of indirect branches are difficult to be predicted accurately using conventional Branch Target Buffer (BTB). The results show that the advanced

indirect branch prediction mechanism used in the Pentium 4 processor works well on the MSA software. Figure 9 also shows that calls and returns can also be predicted accurately with the 16-entry return address stack. To further improve branch prediction accuracy of MSA software, efforts should focus on the conditional branch prediction.

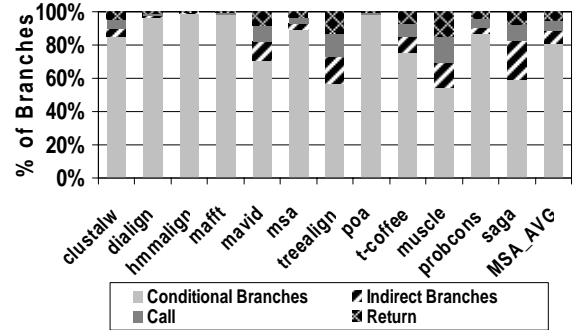


Figure 8. Dynamic branch mix.

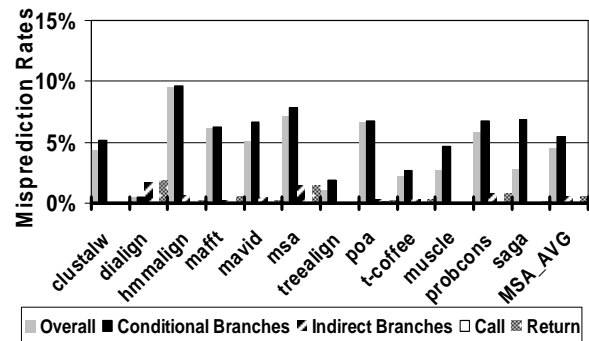


Figure 9. Misprediction rates.

4.7. Speculative Execution

To reach high performance, the Pentium 4 machine fetches and executes instructions along the predicted path until the branch is resolved. In case there is a branch misprediction, the speculatively executed instructions along the mispredicted path are flushed. The speculative execution factor or the ratio of the total number of instructions decoded to the total number of instructions retired quantitatively captures how aggressively the processor executes the speculated instructions.

Figure 10 shows the speculative execution factors for instructions and μ ops on the MSA software. On the average, the processor decodes 27% more instructions than it retires. Note that there is a fairly strong correlation between the branch prediction accuracy and the speculative execution factor on these programs. Due to the use of deeply pipelined design (31 stages behind the trace cache) to reach high operation clock frequency, the accuracy of branch prediction plays an important role on Prescott pipeline performance. MSA benchmarks with higher mispredicted branches per instruction have higher speculated instructions, indicating these applications can further benefit from more accurate branch prediction.

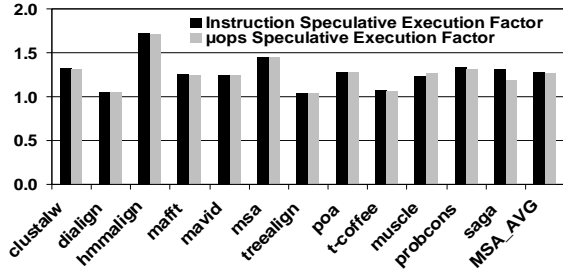


Figure 10. Speculation factor.

4.8. Phase Behavior

Recent computer architecture research has shown that program execution exhibits phase behavior, and these

behaviors can be seen even on the largest of scales [27]. Program phases can be exploited to design adaptive microarchitecture, guide feedback compiler optimization and reduce simulation time. To reveal the phase behavior of MSA applications, we sampled performance counters at a time interval of 0.1 second. Figure 11 shows the sampled IPC of six MSA applications. As can be seen, the studied MSA applications show heterogeneous phase behavior. For example, benchmark *T-coffee* shows periodic spikes where program execution yields high IPC. The phase behavior of benchmarks *Msa* and *Trealign* is highly predictable for the entire program execution. Benchmarks *Muscle*, *Clustal w* and *T-coffee* exhibit irregular and unpredictable phase behavior during the program execution.

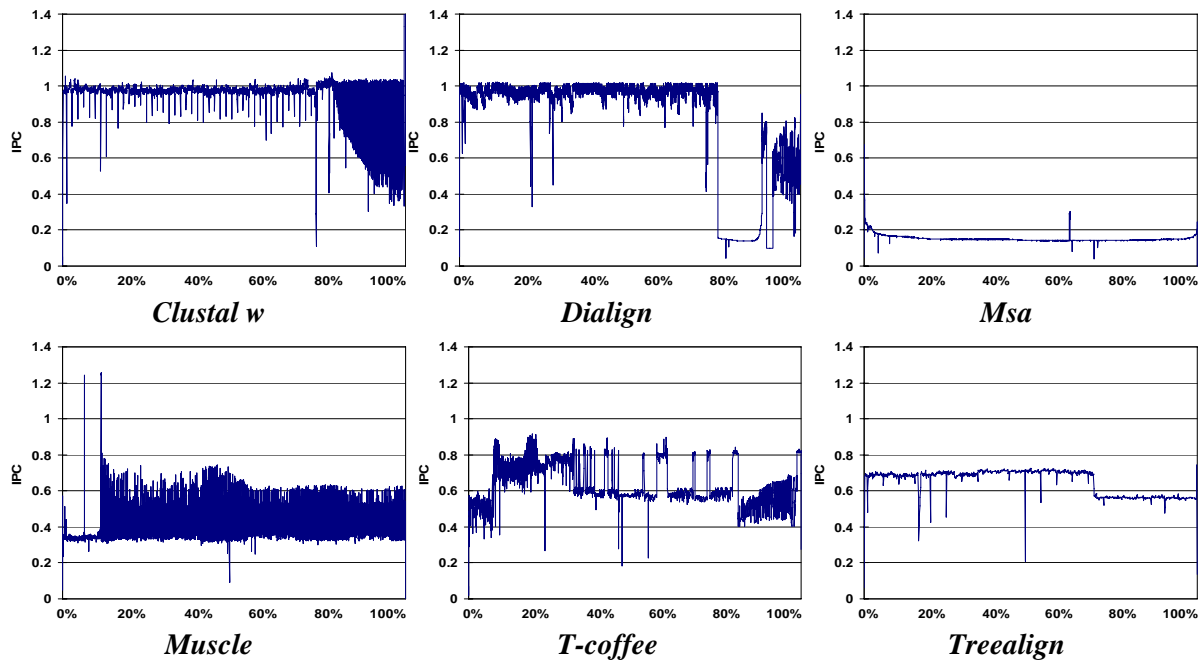


Figure 11. Phase behavior of MSA workloads.

5. Conclusions

As requirements for the processing of biological data grow, bioinformatics becomes an important type of application domain. The assembly of a multiple sequence alignment (MSA) has become one of the most common tasks in bioinformatics. Despite the amount of attention dedicated to the MSA problem, it is largely unknown how various MSA methods use the advanced microarchitectural features provided by modern processors. Our work studies architectural properties for several widespread multiple sequence alignment algorithms on an actual Intel Pentium 4 processor.

We found that bioinformatics multiple sequence alignment workloads benefit from many advanced Pentium 4 microarchitecture features, such as trace cache, prefetching

and large size L2 cache, and advanced indirect branch predictor. We believe that several observations we made in this study can be useful for performance optimization of MSA workloads from the architectural point of view. For example, MSA workloads intensively access (i.e., read) memory and the access patterns can be captured by hardware prefetcher. Thus, a smaller L1 data cache with multiple read ports and prefetching can provide higher memory bandwidth while reducing cache hit latency. We also observed that despite the relatively good behavior on cache and branch prediction, the IPC performance of MSA workloads is still poor. To fully utilize the superscalar capability provided by the advanced microarchitecture, we believe that additional techniques, such as value prediction [24] and more aggressive compiler optimizations (e.g., superblock [25] and hyperblock [26]), should be used.

The results obtained in this paper open up new avenues for future MSA algorithms. For example, to reduce excessive amount of loads and stores, heuristics methods can be applied to MSA algorithms to further reduce the amount of search space explored. To effectively reduce branch misprediction rates and pipeline flushes, new MSA algorithms should explore the search space more deterministically. That is, unpromising alignments need to be eliminated preemptively using better strategies. These improvements can be obtained by summarizing and indexing the search space and statistically analyzing the sequences. In the future work, we will explore the hardware and software techniques to optimize the performance of MSA tools.

References

- [1] <http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html>
- [2] Bioinformation Market Study for Washington Technology Center, Alta Biomedical Group LLC, www.altabiomedical.com, June 2003.
- [3] C. Notredame, Recent Progress in Multiple Sequence Alignment: a Survey, *Pharmacogenomics*, Jan. 3(1):131-44, 2002.
- [4] L Wang and T. Jiang, On the Complexity of Multiple Sequence Alignment, *Journal of Computational Biology*, Vol. 1, N. 4, pages 337-348, 1994.
- [5] David W. Mount, *Bioinformatics: Sequence and Genome Analysis*, ISBN: 0879696087, Cold Spring Harbor Laboratory Press, 2001.
- [6] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, The Microarchitecture of the Pentium 4 Processor, *Intel Technology Journal*, 1st quarter 2001.
- [7] D. Lipman, S. Altschul and J. Kececioglu, A Tool for Multiple Sequence Alignment, *Proc. Natl. Acad. Sci. USA* 86, 4412-4415, 1989
- [8] J. D. Thompson, D.G. Higgins, and T.J. Gibson, Clustal W: Improving the Sensitivity of Progressive Multiple Sequence Alignment through Sequence Weighting, Positions-specific Gap Penalties and Weight Matrix Choice, *Nucleic Acids Research*, vol. 22, no. 22, pages 4673-4680, 1994.
- [9] J. J. Hein, *TreeAlign*, in *Computer Analysis of Sequence Data*, edited by A. M. Griffin and H. G. Griffin. Humana Press, Tolowa, New Jersey, page 349-346, 1994.
- [10] C. Notredame, D. Higgins, J. Heringa, T-Coffee: A Novel Method for Multiple Sequence Alignments, *Journal of Molecular Biology*, 302, 205-217, 2000.
- [11] C. Lee, C. Grasso and M. Sharlow, Multiple Sequence Alignment Using Partial Order Graphs, *Bioinformatics* 18: 452-464, 2002.
- [12] C. B. Do, M. Brudno and S. Batzoglou, ProbCons: Probabilistic Consistency-based Multiple Alignment of Amino Acid Sequences, *ISMB* 2004.
- [13] C. Notredame and D.G. Higgins, SAGA: Sequence Alignment by Genetic Algorithm, *Nucleic Acid Research*, Vol. 24, 1515-1524, 1996.
- [14] R. C. Edgar, MUSCLE: Multiple Sequence Alignment with High Accuracy and High Throughput, *Nucleic Acids Research* 32(5), 1792-97, 2004.
- [15] M Kimura, A Simple Method for Estimating Evolutionary Rates of Base Substitutions through Comparative Studies of Nucleotide Sequences, *J. Mol. Evol.*, 16:111-120, 1980.
- [16] Bray N and Pachter L, MAVID: Constrained Ancestral Alignment of Multiple Sequences, *Genome Research*, 14:693-699, 2004.
- [17] K. Katoh, K. Misawa, K. Kuma and T. Miyata, MAFFT: a Novel Method for Rapid Multiple Sequence Alignment based on Fast Fourier Transform. *Nucleic Acid Res.*, 30:3059-3066, 2002.
- [18] B. Morgenstern, DIALIGN 2: Improvement of the Segment-to-Segment Approach to Multiple Sequence Alignment, *Bioinformatics* 15, 211 - 218, 1999
- [19] S. R. Eddy, Profile Hidden Markov Models, *Bioinformatics Review*, vol. 14, no. 9, page 755-763, 1998.
- [20] NCBI, <http://www.ncbi.nlm.nih.gov/>
- [21] The NCBI Bacteria Genomes Database, <ftp://ftp.ncbi.nih.gov/genomes/Bacteria/>
- [22] B. Sprunt, The Basics of Performance Monitoring Hardware, *IEEE Micro*, July-August, page 64-71, 2002.
- [23] Intel Pentium 4 Processor Optimization, Reference Manual, Intel Corporation, 2001.
- [24] M. H. Lipasti, C. B. Wilkerson and John Paul Shen, Value Locality and Load Value Prediction, In *Proceedings of the International Symposium on Computer Architecture*, 1996.
- [25] W.W. Hwu, Scott A. Mahlke, The Superblock: An Effective Technique for VLIW and Superscalar Compilation, In the *Journal of Supercomputing*, page 224-233, May 1993.
- [26] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, Roger A. Bringmann, Effective Compiler Support for Predicated Execution Using the Hyperblock, In the *International Symposium on Microarchitecture*, 1994.
- [27] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder, Automatically Characterizing Large Scale Program Behavior, In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.