Biosequence Similarity Search on the *Mercury* System

Praveen Krishnamurthy, Jeremy Buhler, Roger Chamberlain, Mark Franklin, Kwame Gyang, and Joseph Lancaster

Praveen Krishnamurthy, Jeremy Buhler, Roger Chamberlain, Mark Franklin, Kwame Gyang, and Joseph Lancaster, "Biosequence Similarity Search on the *Mercury* System," in *Proc. of the IEEE 15th International Conference on Application-specific Systems, Architectures and Processors*, September 2004, pp. 365-375.

Washington University
Dept. of Computer Science and Engineering
Campus Box 1045
One Brookings Dr.
St. Louis, MO 63130

Biosequence Similarity Search on the *Mercury* System

Praveen Krishnamurthy, Jeremy Buhler, Roger Chamberlain, Mark Franklin, Kwame Gyang, and Joseph Lancaster
Department of Computer Science and Engineering
Washington University in St. Louis
{praveenk, jbuhler, roger, jbf, kg2, jmlancas}@cse.wustl.edu

Abstract

Biosequence similarity search is an important application in modern molecular biology. Search algorithms aim to identify sets of sequences whose extensional similarity suggests a common evolutionary origin or function. The most widely used similarity search tool for biosequences is BLAST, a program designed to compare query sequences to a database. Here, we present the design of BLASTN, the version of BLAST that searches DNA sequences, on the Mercury system, an architecture that supports high-volume, high-throughput data movement off a data store and into reconfigurable hardware. An important component of application deployment on the Mercury system is the functional decomposition of the application onto both the reconfigurable hardware and the traditional processor. Both the Mercury BLASTN application design and its performance analysis are described.

1: Introduction

Computational search through large databases of DNA and protein sequence is a fundamental tool of modern molecular biology. Rapid advances in the speed and cost-effectiveness of DNA sequencing have led to an explosion in the rate at which new sequences, including entire mammalian genomes [16], are being generated. To understand the function and evolutionary history of an organism, biologists now seek to identify discrete biologically meaningful features in its genome sequence. A powerful approach to identify such features is *comparative annotation*, in which a *query sequence*, such as new genome, is compared to a large database of known biosequences. Database sequences exhibiting high similarity to the query, as measured by string edit distance [14], are hypothesized to derive from the same ancestral sequence as the query and in many cases to have the same biological function.

BLAST, the **B**asic **L**ocal **A**lignment **S**earch **T**ool [1], is the most widely used software for rapidly comparing a query sequence to a biosequence database. Although BLAST's algorithms are highly optimized for efficient similarity search, growth in the databases it uses is outpacing speed improvements in general-purpose computing hardware. For example, the National Center for Biological Information (NCBI) Genbank database grew exponentially between 1992 and 2003 with a doubling time of 12–16 months [10]. The problem is particularly acute for BLASTN, the BLAST variant used to compare DNA sequences, because each new genome sequenced from animals or higher plants produces between 10^8 and 10^{10} bytes of new DNA sequence.

One response to runaway growth in biosequence databases has been to distribute BLAST searches across multiple computers, each responsible for searching only part of a database.

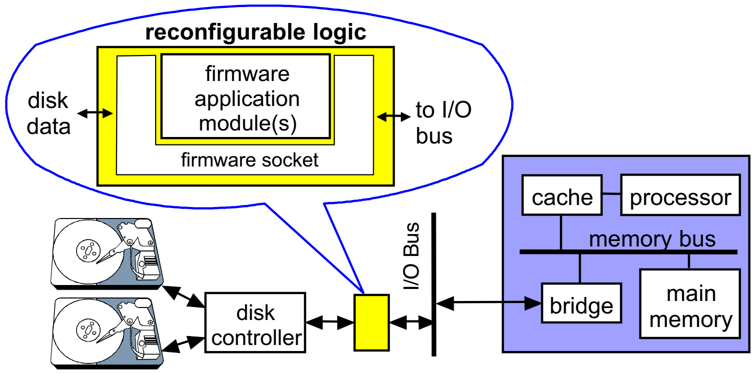

Figure 1. ***Mercury*** **system architecture**

This approach requires both a substantial hardware investment and the ability to coordinate a search across processors. An alternate approach which makes more parsimonious use of hardware is to build a specialized BLAST accelerator. By using an application-specific architecture and exploiting the high I/O bandwidth of modern storage systems, an accelerator can execute the BLAST algorithms much faster than a general-purpose CPU.

The *Mercury* system [3] is a prototype architecture that supports disk-based computation at very high data rates using reconfigurable hardware. Computing applications historically have been coded using the following paradigm: read input data into main memory with explicit I/O calls, compute on that data writing results back to main memory, and send the output from main memory with explicit I/O calls. In contrast, the *Mercury* system is built around the concept of continuous data flow. Data from disk(s) flow into the computational resource(s); one or more functions (often physically pipelined) are performed on the data; and the results flow to the intended destination. As the computational resources include reconfigurable hardware, application deployment requires hardware/software codesign. The *Mercury* system builds upon the work of Reidel [13] (active disks), Dally [4] (stream processors), and a host of work developed in the reconfigurable computing community.

This paper describes the re-engineering of the original BLASTN application for effective deployment on the *Mercury* system. We examine the existing application to explore its performance properties, propose a novel algorithmic optimization, prototype a number of critical components of the application, and evaluate the performance potential of the overall application running on the *Mercury* system.

2: System Architecture

The *Mercury* system (Figure 1) is reconfigurable logic, associated with the disk controller, that provides computing capability in close proximity to the data flowing off the disk drive(s). It operates independently and uses no I/O bus, memory bus, or processor cache cycles. Instead, initial processing of the data occurs locally at the disk. The reconfigurable logic is implemented via a Field-Programmable Gate Array (FPGA).

Application functionality is divided into two parts executing on the FPGA and the main processor, respectively. Application deployment therefore has the classic components of a hardware/software codesign problem, with the need to map application elements to multiple computational resources (i.e., FPGA and processor). A unique aspect of the *Mercury*

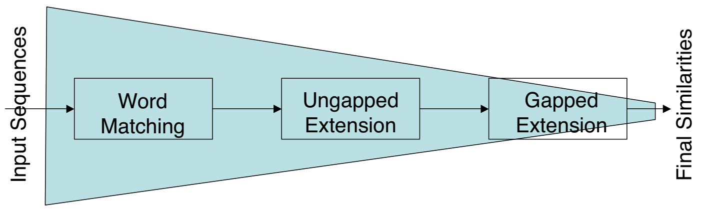

Figure 2. Pipeline stages of NCBI BLAST algorithm

system is that it was designed specifically to work well with high-volume data applications. The computational resource that is best suited to simpler, repetitive operations on a large data set is positioned closer to the data, while the resource best suited to more complex operations on smaller data volumes is (logically) farther away from the data.

The application set that is well matched to the *Mercury* system architecture is a pipeline that consumes a high data volume at its input, reduces that data volume to a smaller set, and performs higher-level processing on this smaller set. Our previous work has illustrated the use of the system for a number of text search applications [17, 19]. BLASTN has properties that fit well with the *Mercury* system's capabilities.

While Figure 1 illustrates our vision of the system architecture, our prototyping work has so far been limited to a series of implementations that are progressively closer to, but do not yet exactly match, the architecture depicted in the figure. Our earliest prototypes used ATA drives [17, 19] and were severely speed-limited by the disks. Our most recent prototypes are built using a set of 15,000 rpm Ultra320 SCSI drives organized in a RAID-0 configuration. On this configuration, we have demonstrated sustained read performance of over 800 MB/sec for continuous 500 GB reads. The prototype FPGA infrastructure is currently parallel to the disk controller on the I/O bus, which limits throughput into the FPGA. We have, however, demonstrated sustained data throughput of over 400 MB/sec from the disk array into the FPGA [6].

In what follows, we refer to computations deployed in the FPGA as *firmware* and computations deployed on the processor as *software*. To facilitate the deployment of applications on the FPGA, we have developed a firmware socket interface that provides a consistent environment for the development of firmware application modules. Data from the disk array is delivered to the FPGA via the firmware socket, while outbound data from the reconfigurable logic is delivered into the main memory of the processor for access by software.

3: Description of NCBI BLASTN

This section describes the open-source version of BLASTN distributed by the National Center for Biological Information (NCBI) and used by numerous biological research labs. As shown in Figure 2, BLASTN is functionally organized as a pipeline with three stages: word matching, ungapped extension, and gapped extension. The inputs to this pipeline are a query sequence and a database, each consisting of a string of DNA *bases*. A base is typically one of $\{A, C, G, T\}$, but other characters (a total of 15) are used to denote uncertainty about or special properties of certain bases. DNA sequences, including these special characters, can be represented using four bits per base; however, to minimize storage and I/O bandwidth, NCBI BLASTN stores its database using only two bits per base.

Each stage of BLASTN's pipeline implements progressively more sophisticated and more expensive computations to identify biologically meaningful similarities between query and database. In stage 1, BLASTN discovers *word matches* between query and database. A word match is a string of some fixed length w (hereafter called a "w-mer") that occurs in both query and database. Significantly similar sequences frequently share a w-mer match for $w \approx 10$, though such matches also occur frequently by chance between unrelated sequences. Each word match is therefore filtered through stage 2, which tries to extend it into an *ungapped alignment* between query and database. An ungapped alignment may contain mismatched bases but consists primarily of matching base pairs. Ungapped alignments with too few matching base pairs are discarded, while the remainder are further filtered through stage 3, which extends them into *gapped alignments* that permit both mismatches and localized insertion or deletion of bases. In the final operation following the end of stage 3, gapped alignments with sufficiently many matching base pairs are reported to the user.

Although each stage of BLASTN is more compute-intensive than the last, each stage also discards a substantial fraction of its inputs. The volume of data that is processed at each stage therefore gradually decreases. Table 1 quantifies the data reduction at each stage of the pipeline[1]. The match rate, p_i, represents the probability that an output from stage i is generated from an individual input to that stage. For stage 1, p_1 measures the number of matches per DNA base read from the database. Stages 1 and 2 are highly effective at reducing the data volume to the next stage. Note that, as the query length increases, the rate at which matches are output from stage 1 into stage 2 also increases, raising the workload for stage 2.

In the performance predictions that follow, we will consider the throughput of individual stages of the pipeline as well as the throughput of the entire pipeline. To make throughputs comparable, they are normalized to be in units of input bases per second from the database. When executing on a single computational resource (i.e., software running on a single processor), the average compute time per input base can be expressed as $t_1 + p_1 t_2 + p_1 p_2 t_3$, where t_i is the compute time for stage i for each input item (base, match, or alignment) to stage i. The normalized throughput is then $Tput = 1/(t_1 + p_1 t_2 + p_1 p_2 t_3)$.

Table 1. Match rates p across pipeline stages

Query Size (bases)	**Stage 1** (p_1)	**Stage 2** (p_2)	**Stage 3** (p_3)
10 K	0.00858	0.0000550	0.320
100 K	0.0841	0.0000174	0.175
1 M	0.837	0.0000175	0.117

3.1: Details of BLASTN Stage 1

To facilitate later comparison with our firmware design, we now briefly describe the implementation of NCBI BLASTN's stage 1. This implementation uses a default match length $w = 11$. Due to the speed advantages of comparing complete bytes at a time, discovery of 11-mer matches is implemented in two phases. BLASTN first checks two complete bytes of the database, containing 8 bases, against a lookup table constructed from the query. Only two-byte words occurring on full byte boundaries are checked. If the query contains the same 8-base word, BLASTN tries to extend this 8-base match to 11 bases by seeking additional matching residues on either side.

[1]Reduction measurements for NCBI BLASTN were taken in the same experiments used to generate the timings of Section 3.2.

Two 11-mer matches that occur close to each other in both query and database are likely to have arisen from the same underlying biological similarity. To avoid having later stages expend the effort to discover this similarity twice, NCBI BLASTN implements a *redundancy elimination* filter at the end of stage 1. The filter checks whether each new 11-mer match overlaps or is close to a previously observed match. If so, the new match is suppressed, since it would likely lead only to rediscovery of any feature found by the previous match.

3.2: Performance of NCBI BLASTN

To quantify the performance of NCBI BLASTN on a general-purpose CPU, we measured its execution time with default parameters on a 2.8 GHz Pentium 4 PC, with an L2 cache size of 512 KB and 1 GB of RAM, running Linux. We compared a database containing the mouse genome (1.16 Gbases after removing repetitive sequence) to queries of various lengths selected at random from the human genome. CPU time was measured separately for each of the three pipeline stages.

The length of a typical query sequence in BLASTN is application-dependent. For example, a short DNA sequence obtained in a single lab experiment may be only a few kilobases, while in genome-to-genome comparison, a query (one of the genomes) may be billions of bases long. A BLAST implementation should support the largest computationally feasible query length, both to accommodate long individual queries and to support the optimization of "query packing," in which multiple short queries are concatenated and processed in a single pass over the database. Conversely, queries longer than the maximum feasible length may be broken into pieces, each of which is processed in a separate pass.

In our experiments, we tested queries of 10 Kbases, 100 Kbases, and 1 Mbase, both to simulate different applications of BLASTN and to assess the impact of query length on the performance of our firmware implementation. One megabase is a reasonable upper bound on query size for NCBI BLASTN with standard parameters, since it generates 11-mer word matches by chance alone at a rate approaching one match for every base read from the database. Timings were averaged over at least 20 queries for each length, and each query's running time was averaged over three identical runs of BLASTN.

Table 2 gives the distribution of times spent in each stage of NCBI BLASTN for various query sizes. Times are given with 95% confidence intervals. Time spent in stage 1 dominated that spent in later pipeline stages, while time spent in stage 3 was almost negligible. Although later stages are computationally more intensive, each stage is such an efficient filter that it discards most of its input, leaving later stages with comparatively little work.

Table 2. Percentage of pipeline time spent in each stage of NCBI BLASTN

Query Size (bases)	**Stage 1**	**Stage 2**	**Stage 3**
10 K	86.53±1.51%	13.24±1.99%	0.23±0.017%
100 K	83.35±1.28%	16.57±2.17%	0.08±0.007%
1 M	85.29±2.40%	14.68±3.70%	0.03±0.002%

From the measured running times of our experiments and the size of the mouse genome database, we computed the throughput (in Mbases from the database per second) achieved by NCBI BLASTN's pipeline for varying query sizes. The results are shown in the first row of Table 3. Throughput depends strongly on query length. To explain this observation, we used the predicted filtering efficiencies p_i for each pipeline stage and the distribution of

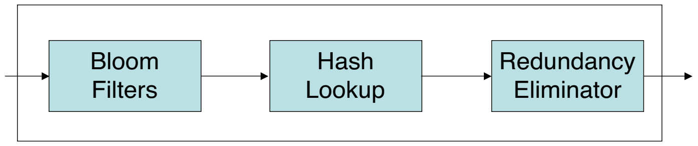

Figure 3. Division of BLAST stage 1 (word matching) into 3 substages

running times by stage to estimate the average time spent to process each base in stage 1, each word match in stage 2, and each ungapped alignment in stage 3. These results are shown in the remaining rows of the table. While the overhead per input remains constant for stage 2 and actually decreases for stage 3, the cost per base in stage 1 grows linearly with query length. This cost growth derives from the linear increase in the expected number of matches per base that occur purely by chance, in the absence of any meaningful similarity.

Table 3. Summary of performance results for software runs of NCBI BLASTN

Query Size	10 Kbases	100 Kbases	1 Mbase	Units
Throughput	66.9	8.76	0.657	Mbases/sec
Stage 1 (time per base, t_1)	0.0129	0.0951	1.30	μsec/base
Stage 2 (time per match, t_2)	0.231	0.225	0.267	μsec/match
Stage 3 (t_3)	71.3	58.9	34.4	μsec/alignment

The empirical performance of NCBI BLASTN's pipeline demonstrates that stage 1 is a performance bottleneck and therefore the first target for speedup in firmware.

4: Firmware Implementation of Stage 1

Our firmware implementation of stage 1 reflects the overall functionality of stage 1 in NCBI BLASTN but makes no attempt to implement this functionality using the same mechanisms. Our design decomposes stage 1 into 3 substages (Figure 3). The initial substage implements a prefilter using Bloom filters; the middle substage determines the query position of w-mers in the database that successfully pass through the Bloom filters (using hashing); and the final substage performs redundancy elimination.

4.1: Prefiltering using Bloom Filters

A Bloom filter [2] is a probabilistic algorithm to quickly test membership in a large set using multiple hash functions into a single array of bits. Bloom filters find many uses in networking and other applications [5]. Figure 4 illustrates programming a w-mer and querying for membership. Programming the filter amounts to setting the bits of the memory location obtained by the hash functions. Querying the Bloom filter yields a match when all the memory locations in the vector obtained from hashing the query contain '1'.

A Bloom filter yields no false negatives but does yield false positives at a rate f determined by the number of w-mers programmed into it and the length of its memory vector. The rate f can be modeled as $f = (1 - e^{-nk/m})^k$, where n is the number of w-mers programmed into the filter, m is the filter memory size in bits, and k is the number of hash functions. This rate is shown as a function of memory size and query string length in Figure 5. The false positive rate decreases exponentially with linearly increasing memory size; given a few hundred Kbytes of memory, the filter incurs few false positives.

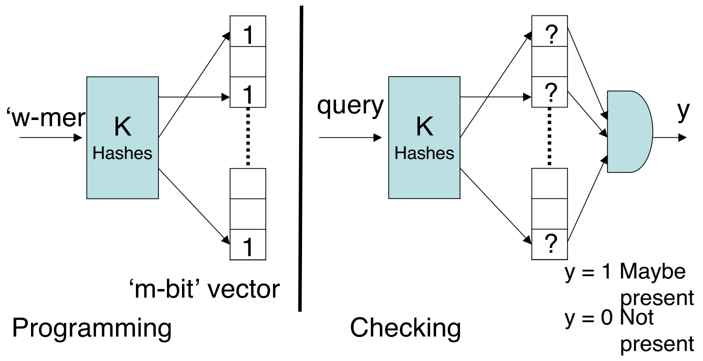

Figure 4. Bloom filter: Operation detail

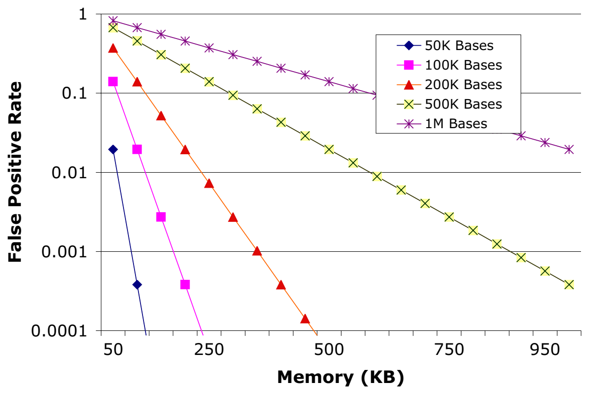

Figure 5. Bloom filter false positive rate vs. memory size

Bloom filters are more efficiently implemented in firmware than in software, as we can store the memory vectors on-chip (using block RAMs), calculate the hash functions in parallel, and look up the locations of the memory vector in parallel.

4.2: Hash Lookup

For our implementation of word matching in firmware, we employ an open-addressed double hashing scheme. Note here that we do not employ a comparison only on byte boundaries as is the case with NCBI BLASTN. The hash function is used not only to obtain the positions of valid w-mers in the query but also to eliminate any false positives from the prefilter. Each occupied slot in the hash table contains the w-mer that hashed there, the number of times that w-mer appears in the query, and a list of its locations in the query. The table is stored in SRAM, from which we can read a random location in a single clock cycle. We pipeline hash lookups to obtain an effective rate of 1 match/clk.

4.3: Redundancy Filter

We have modified NCBI BLASTN's filter for redundant word matches to work efficiently in firmware. Let the *diagonal* of match be the difference $j - i$ between its starting offsets j in the database and i in the query sequence. Every word match is contained in some diagonal. To remove redundant matches, the filter keeps track of the most recent word match (counting by its position in the database) on each diagonal. If a new match overlaps or is close to a previous match on the same diagonal, it is discarded.

Our redundancy filter, which is entirely contained in stage 1, is more efficiently implementable in hardware than the NCBI BLASTN filter, which requires feedback from stage 2. The filter can be implemented in space proportional to the query as described in [12].

5: Performance Analysis

We assess the performance of our design in three phases. First, we assess the performance gain relative to software of stage 1 alone. Second, we assess the overall performance of a design that exploits the firmware implementation of stage 1 and continues to use software to implement stages 2 and 3. Finally, we discuss the benefit that can be gained from a (future) firmware implementation of stage 2 and provide performance targets for that design.

5.1: Word Matching (Stage 1) in Firmware

Our firmware stage 1 design may be characterized as follows. We target a false positive rate of 10^{-4} per base for the prefilter, which is significantly less than its rate of true positives per base. The load on the hash lookups is therefore reduced by a factor of $\frac{1}{p_1+f}$, where p_1 is the output probability from stage 1 (Table 1). We have built a firmware prototype of these prefilters that can consume 16 bases/clk at a clock rate of 80 MHz. The prototype is limited to a query size of 100 Kbases, since the on-chip memory requirements are too large for a 1 Mbase query. To perform 1 Mbase query, we pass the database through the firmware stage 1 design 10 times, each pass consuming 10% of the query. This results in an effective throughput for 1 Mbase queries that is 1/10th that for 100 Kbase queries.

The stage 1 hash table processes both true positives and any false positives from the prefilter. Because this table is stored in off-chip SRAM with abundant memory, there are few collisions; hence, a lookup takes on average only one clock cycle. Our hash table and redundancy filter prototypes can sustain an average input rate of 1 match/clk (1 lookup/clk) at 80 MHz. Although the hash table and redundancy filter operate at less than the full input rate, the prefilter enables them to be slower by a factor of $\frac{1}{p_1+f}$.

The raw throughput supported by our stage 1 implementation is $16 \times 80 = 1.28$ Gbases/sec. Using the *Mercury* system infrastructure, which is currently limited to 400 MBytes/sec input bandwidth, we can support a throughput of 800 Mbases/sec into stage 1. Note that we use 4 bits per base, thereby eliminating potentially significant post-processing of masked sequences arising from NCBI BLASTN's use of 2 bits/base. Table 4 compares the performance of stage 1 in firmware to the software BLASTN.

Table 4. Firmware vs. software stage 1 (throughput and speedup)

Query Size	10 Kbases	100 Kbases	1 Mbases	Units
NCBI BLASTN Stage 1 ($Tput_1$)	77.4	10.5	0.771	Mbases/sec
Mercury BLASTN Stage 1($Tput_1$)	800	800	80	Mbases/sec
Speedup (S_1)	10.3	76.1	104	

5.2: Overall Performance of BLASTN on the *Mercury* System

We now consider *overall* pipeline performance. As the component pieces described above have yet to be integrated into a functioning whole, the performance numbers that follow are model-based predictions.

When executing the application across multiple resources, the overall throughput is determined by the minimum throughput achieved on any one resource. Here, stage 1 executes in firmware, while stages 2 and 3 execute in software. The throughput is therefore

$$Tput_{overall} = \min\left(Tput_1, \frac{1}{p_1(t_2 + p_2 t_3)}\right),$$

where $Tput_1$, stage 1 throughput, is from Table 4; t_i, the time to perform stage i in software, is from Table 3; and p_i, the probability of an output from stage i, is from Table 1.

Table 5 compares the overall performance of *Mercury*-based BLASTN with that of NCBI BLASTN. Though we have shown significant speedup for stage 1 in firmware (refer to Table 4), the overall speedup is limited to a factor of 6 to 8. Overall performance is now limited by the software-based stage 2. Hence, though we successfully deployed stage 1 in firmware with high throughput, the overall application still suffers from limitations imposed by the remaining pipeline stages.

Table 5. Overall performance (throughput and speedup)

Query Size	**10 Kbases**	**100 Kbases**	**1 Mbase**	**Units**
NCBI BLASTN	67.0	8.76	0.657	Mbases/sec
Mercury BLASTN	497	52.6	4.47	Mbases/sec
Speedup	7.42	6.01	6.80	

If t_3 is the software compute time per match from stage 3 (from Table 3), the maximum pipeline throughput that can be sustained by stage 3 is $Tput_3 = 1/p_1 p_2 t_3$ (as above, normalized to be in units of input bases per second from the database). For 1 Mbase query sizes, this rate is approximately 2 Gbases/sec, which exceeds the input rate supported by the firmware stage 1. Hence, stage 3 is unlikely to be a bottleneck to overall performance.

We next consider the overall performance impact of accelerating stage 2. This impact can be modeled as $Tput_{overall} = \min(Tput_1, Tput_2, Tput_3)$, where $Tput_1$ and $Tput_3$ are as above and $Tput_2$ is now $S_2/p_1 t_2$. S_2 is a model input representing the speedup of a hypothetical firmware stage 2 implementation. This model determines the performance required of the stage 2 firmware in order to achieve a given overall pipeline throughput.

Figure 6(A) plots the throughput of the overall application, as a function of the stage 2 speedup S_2, for various query sizes. By increasing the performance of the bottleneck stage 2, overall performance improves until the throughput reaches the limit imposed by stage 1, at which point it saturates.

Figure 6(B) plots the speedup, relative to a pure software implementation, of the entire application as a function of stage 2 speedup, again for various query sizes. If, as seems likely, we can achieve even modest speedup in a firmware stage 2, we predict that overall performance of *Mercury* vs. NCBI BLASTN will improve by two orders of magnitude.

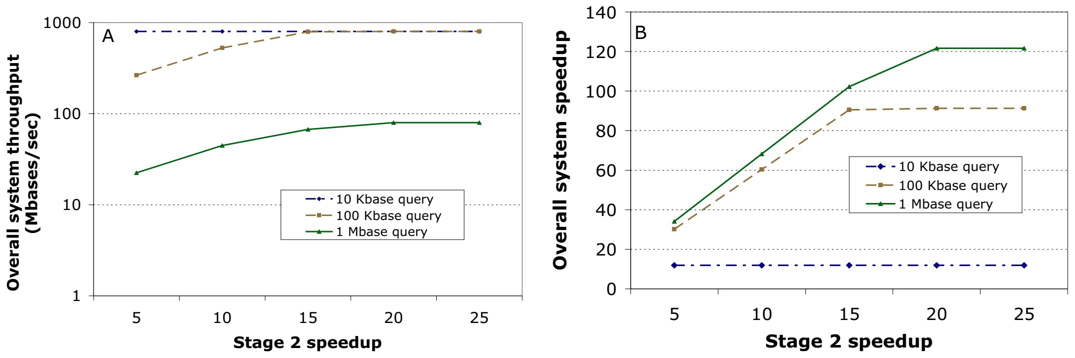

Figure 6. Performance of *Mercury* BLASTN with improved stage 2

6: Related Work

Biosequence similarity search is a fundamental task of modern biology. Several research groups have therefore implemented systems to accelerate similarity search in hardware.

Hardware implementations of the Smith-Waterman dynamic programming algorithm have been reported in the literature, using both non-reconfigurable ASIC logic [7] and reconfigurable logic [8, 18]. These enhancements focus on gapped alignment, which is more heavily loaded in proteomic BLAST comparisons. However, our analysis of the BLASTN pipeline shows that there is a significant reduction in data before reaching gapped extension in stage 3. Hence, these solutions do not greatly accelerate BLASTN.

High-end commercial systems have been developed to accelerate or replace BLAST [11, 15]. The Paracel GeneMatcher^{TM} [11] relies on non-reconfigurable ASIC logic, which is inflexible in its application and cannot easily be updated to exploit technology improvements. In contrast, FPGA-based systems can be reprogrammed to tackle diverse applications and can be redeployed on newer, faster FPGAs with minimal additional design work. RDisk [9] is one such FPGA-based approach which claims a 60 Mbases/sec throughput for stage 1 of BLAST using a single disk.

Two commercial products that do not rely on ASIC technology are BLASTMachine2^{TM} from Paracel [11] and DeCypherBLAST^{TM} from TimeLogic [15]. The highest-end 32-CPU Linux cluster BLASTMachine2^{TM} performs BLASTN with a throughput of 2.93 Mbases/sec for a 2.8 Mbase query. *Mercury* BLASTN with only stage 1 implemented in firmware has a predicted throughput of 4.47 Mbases/sec for a 1 Mbase query. Hence, BLASTMachine2^{TM} (with 32 nodes) has roughly twice the throughput of *Mercury* BLASTN (with 1 node).

The DeCypherBLAST^{TM}solution uses an FPGA-based approach to improve the performance of BLASTN. This solution has throughput rate of 213 Kbases/sec for a 16-Mbase query, which is comparable to that of *Mercury* BLASTN with only stage 1 in firmware, processing a query length of 1 Mbase.

7: Conclusions and Future Work

This paper presents the design of BLASTN, an important biosequence search application, for the *Mercury* system, an architecture that provides both FPGA and traditional processor computing resources and is optimized for disk-based, data-intensive applications. We constructed prototype application components for a firmware (FPGA-based) stage 1 of the BLASTN pipeline, including the addition of a Bloom filter-based prefilter, a firmware hash table, and a match redundancy eliminator.

We compared the performance of our firmware stage 1 implementation to that of NCBI BLASTN's software stage 1 implementation. We also estimated overall performance of *Mercury* BLASTN, both for the current version with only stage 1 in firmware and for a future version that will also deploy a firmware stage 2.

Because of the strong predicted impact of stage 2 speedups on overall application performance, we are proceeding with a firmware implementation of stage 2, to be followed by a full end-to-end deployment of BLASTN on the *Mercury* prototype.

8: Acknowledgments

This work was supported by NSF Career grant DBI–0237902, and NSF grants ITR–0313203 and CCR–0217334.

References

[1] S. F. Altschul et al. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–402, 1997.

[2] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, May 1970.

[3] R. D. Chamberlain et al. The *Mercury* system: Exploiting truly fast hardware for data search. In *Proc. of Int'l Workshop on Storage Network Architecture and Parallel I/Os*, pages 65–72, September 2003.

[4] W. J. Dally et al. Merrimac: Supercomputing with streams. In *Proc. of Supercomputing Conf.*, November 2003.

[5] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1):52–61, 2004.

[6] M. Franklin et al. An architecture for fast processing of large unstructured data sets. In *Int'l Conference on Computer Design*, October 2004. To appear.

[7] J. D. Hirschberg, R. Hughley, and K. Karplus. Kestrel: a programmable array for sequence analysis. In *Proc. of IEEE Int'l Conf. on Application-specific Systems, Architecture, and Processors*, 1996.

[8] D. T. Hoang. Searching genetic databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–91, 1995.

[9] D. Lavenier, S. Guytant, S. Derrien, and S. Rubin. A reconfigurable parallel disk system for filtering genomic banks. In *ERSA'03, Engineering of Reconfigurable Systems and Algorithms*, 2003.

[10] National Center for Biological Information. Growth of GenBank, 2002. http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.

[11] Paracel, Inc. http://www.paracel.com.

[12] P. A. Pevzner and M. S. Waterman. Multiple filtration and approximate pattern matching. *Algorithmica*, 13(1/2):135–154, 1995.

[13] E. Reidel, C. Faloutsos, G. Gibson, and D. Nagle. Active disks for large-scale data processing. *IEEE Computer*, 34(6):68–74, June 2001.

[14] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–97, March 1981.

[15] TimeLogic Corporation. http://www.timelogic.com.

[16] R. H. Waterston et al. Initial sequencing and comparative analysis of the mouse genome. *Nature*, 420:520–562, 2002.

[17] B. West et al. An FPGA-based search engine for unstructured database. In *Proc. of 2nd Workshop on Application Specific Processors*, pages 25–32, December 2003.

[18] Y. Yamaguchi, T. Maruyama, and A. Konagaya. High speed homology search with FPGAs. In *Pacific Symposium on Biocomputing*, pages 271–282, 2002.

[19] Q. Zhang et al. Massively parallel data mining using reconfigurable hardware: Approximate string matching. In *Proc. of Workshop on Massively Parallel Processing*, April 2004.