Biosequence Similarity Search on the *Mercury* System

Praveen Krishnamurthy, Jeremy Buhler, Roger Chamberlain, Mark Franklin,
Kwame Gyang, Arpith Jacob, and Joseph Lancaster
Department of Computer Science and Engineering
Washington University in St. Louis
{praveenk, jbuhler, roger, jbf, kg2, jarpith, 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 [26], 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 [22], 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 [17]. 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. 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 [7] 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 [21] (active disks), Dally [9] (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

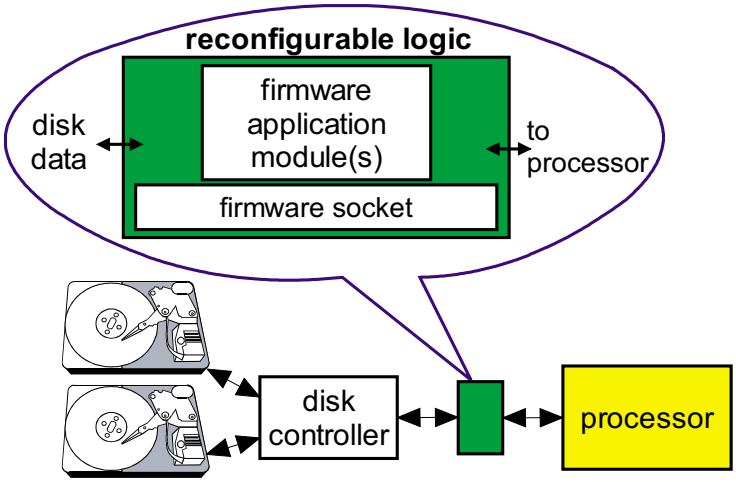

Figure 1. Mercury system architecture

application running on the Mercury system.

2: System Architecture

The Mercury system (Figure 1) contains reconfigurable logic, associated with the disk controller, that provides computing capability in close proximity to the data flowing off the disk drive(s). Initial processing of the data occurs locally at the disk, prior to delivery to the processor. 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 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 [4, 5, 6, 12, 27, 29]. 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 [27, 29] 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 700 MB/sec from the disk array into the FPGA [5, 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.

The current prototype system uses a Xilinx Virtex-II 6000 series FPGA, which provides 8,448 Configurable Logic Blocks (CLBs), 144 18 Kbit Block RAMs (BRAMs), 144 18×18 bit multipliers, and 1104 I/O pins. Each CLB is comprised of 4 slices and each slice provides two 4 input lookup tables (LUTs) for a total of 67,584 LUTs on chip.

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

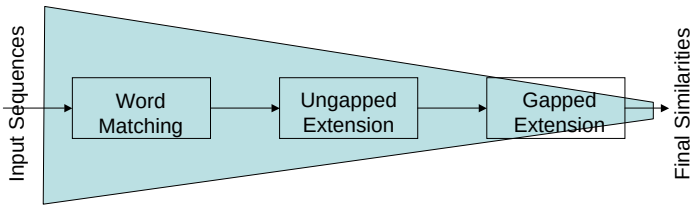

Figure 2. Pipeline stages of NCBI BLAST algorithm

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

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

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
25 K	0.0205	0.0000619	0.141
50 K	0.0411	0.0000189	0.194
100 K	0.0841	0.0000174	0.175
1 M	0.851	0.0000172	0.096

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.

Two 11-mer matches that occur close to each other in both the 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 2. 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, 25 Kbases, 50 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±1.33%	13.24±1.99%	0.23±0.02%
25 K	83.89±2.56%	15.88±4.40%	0.22±0.04%
50 K	82.63±2.94%	17.28±4.96%	0.09±0.01%
100 K	83.35±1.28%	16.58±2.17%	0.08±0.01%
1 M	85.39±3.34%	14.68±5.24%	0.03±0.01%

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

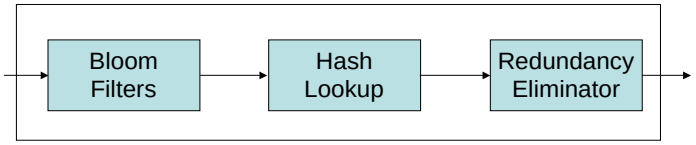

Figure 3. Division of BLAST stage 1 (word matching) into 3 substages (1a: Bloom Filters, 1b: Hash Lookup, and 1c: Redundancy Eliminator)

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 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**	**25 Kbases**	**50 Kbases**	**100 Kbases**	**1 Mbase**	**Units**
Throughput	67.0	29.2	14.9	8.76	0.648	Mbases/sec
Stage 1 (time per base, t_1)	0.0129	0.0287	0.0553	0.0951	1.32	μsec/base
Stage 2 (time per match, t_2)	0.231	0.265	0.281	0.225	0.264	μsec/match
Stage 3 (t_3)	71.3	60.4	81.8	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.

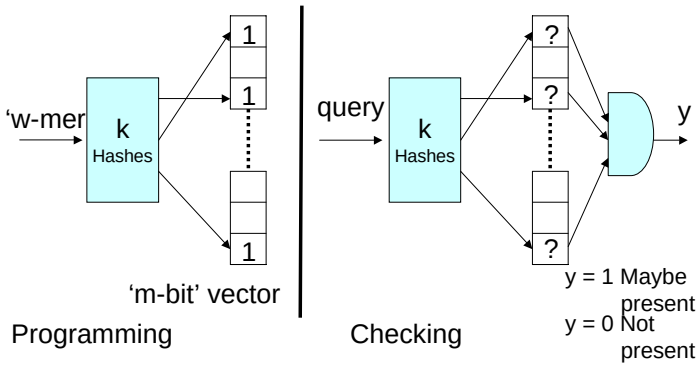

Figure 4. Typical Bloom filter functional diagram

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 [10]. Figure 4 illustrates a typical Bloom filter datapath. Programming the filter amounts to setting to '1' each of the bits of the memory locations 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 entries programmed into the filter, m is the filter memory size in bits, and k is the number of hash functions.

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. However, as the number of ports on these block RAMs are finite, the hash functions are restricted to address only specific block RAMs.

4.2 Architecture of Bloom filters

In our implementation each memory vector made using block RAMs is addresses using a unique hash function (see Figure 5). The false positive rate of Bloom filters in this configuration is given by $f = (1 - (1 - 1/m)^N)^k$ where m is the address range of each hash

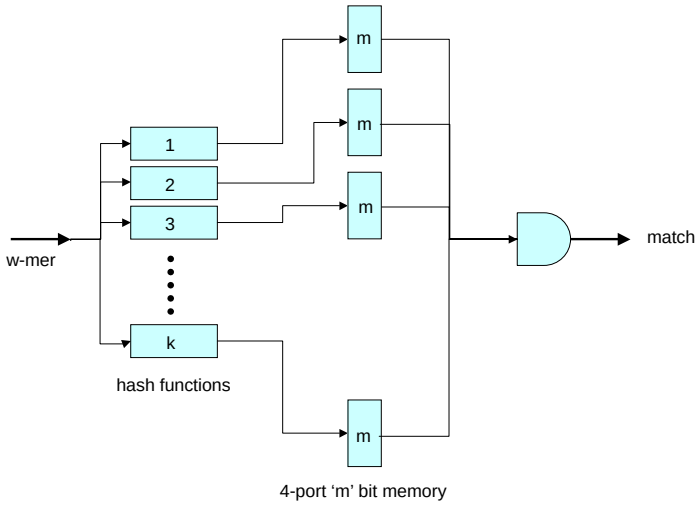

Figure 5. Firmware Implementation of Bloom filter using block RAMs

function, N is the length of the query, and k is the number of hash functions used.

This being the primary block RAM intensive stage in our design, we dedicate 96 (about two-thirds) of the 144 block RAMs available on-chip to the Bloom filter implementation. The stage is designed to consume 16 bases every clock cycle, and operate at 133 MHz (yielding a potential throughput of 2 Gbases/s). To sustain this rate, we must process 16 w-mers every clock cycle, and hence require 16 identical copies of the Bloom filter.

The memory requirements are reduced by half by using both of the ports of the block RAMs. Furthermore, using the clock management circuitry on the FPGA, we can double clock the block RAMs and in effect make each block RAM quad-ported. This is illustrated in Figure 6. Thus 4 copies of the 4-way parallel Bloom filters are sufficient to process all 16 w-mers (as illustrated in Figure 7).

This doubling of effective memory helps us to decrease the false positive rate of the queries (by dedicating more memory for each Bloom filter) or process larger queries at manageable false positive rates. These tradeoffs are illustrated in Figure 8, which shows the expected true match rate (solid line) and the overall match rate (including false positives) of stage 1a (dashed lines). This figure assumes that the input to stage 1a never stalls (i.e., 16 bases are available each clock cycle). The two double clocking curves vary in how they utilize the additional effective memory. One doubles k, while the other doubles m. Clearly doubling the clock rate of the block RAMs not only decreases the false positive rate for smaller query sizes, but also helps us support larger queries than would otherwise be possible.

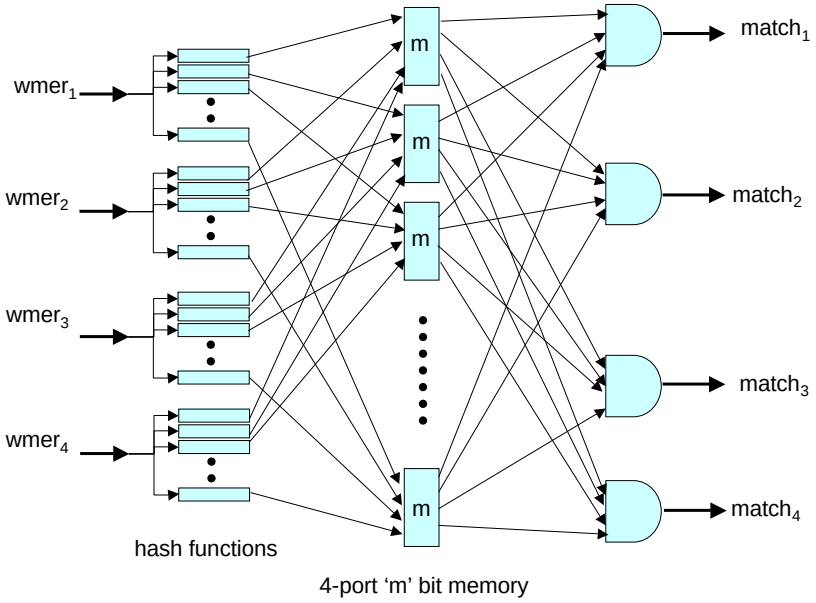

Figure 6. Four parallel Bloom filters

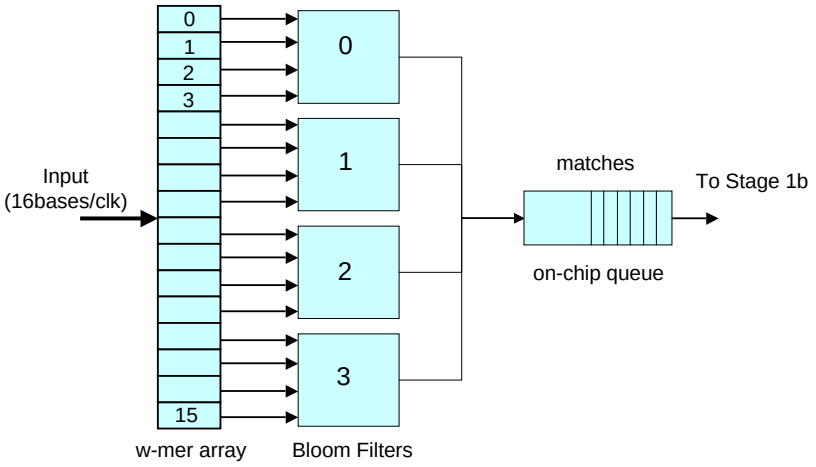

Figure 7. Sixteen parallel Bloom filters

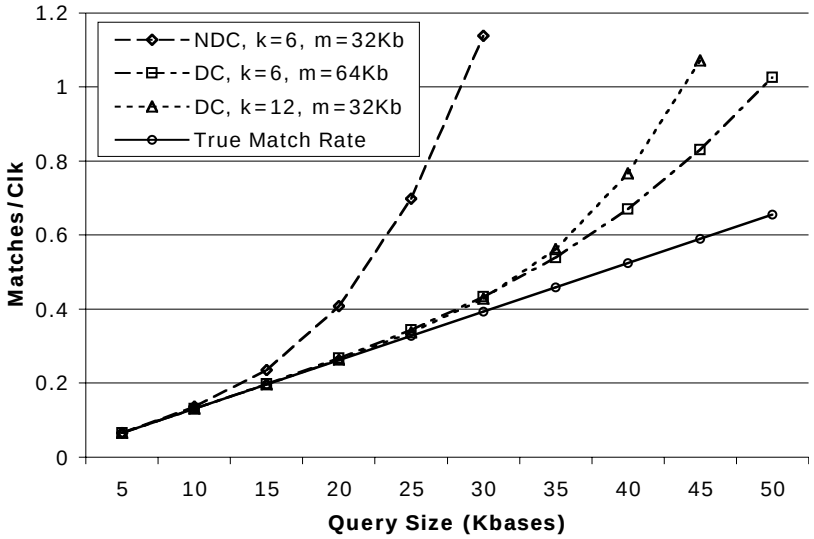

Figure 8. Bloom filter output match rate vs. query size, NDC: no double clocking of block RAMs, DC: double clocking of block RAMs

The maximum query size that can be supported on our prototype is partially determined by the rate of matches from stage 1a. Stage 1b (described below in Section 4.3) processes matches from this stage, and it is designed to support an input rate of approximately one match every clock cycle. Given an expected average input rate from the disk subsystem of 1.4 Gbases/s (700 MB/s $\times$ 2 bases/byte), and a maximum ingest rate into stage 1a of 2 Gbases/s, 50 Kbase queries are reasonably supported.

As similarities can exist between the query and database sequences, there is a good chance that matches from stage 1a will be bursty. We maintain an on-chip queue of size 1000 w-mers to accommodate such bursts. In the event that this queue fills up, for example in the case where long genes are conserved between sequences, we store the matches from stage 1a in off-chip DRAM.

4.3: Hash Lookup

The second substage of stage 1 uses a hash table to identify those w-mers from the database that actually occur in the query sequence. Each such w-mer must be mapped to its position or positions in the query. Note that, in contrast to NCBI BLASTN, we do not test only those w-mers falling on byte boundaries – every w-mer in the database is checked. The hash table is implemented in an external SRAM attached to the FPGA, since the latter's internal block RAMs are too limited in size to contain tables built from

large query sequences.

The need to access a single external SRAM is a potential source of pipeline bottlenecks. Suppose that the SRAM can sustain only one read per clock cycle, which is a reasonable assumption at high FPGA clock speeds. The data reduction achieved by our Bloom filters is sufficient to ensure an average input rate to this substage of at most one w-mer per clock. However, if processing a w-mer were usually to entail multiple, serial accesses to the SRAM (e.g. to resolve hash collisions), we could not sustain even this modest data rate. Our design for this stage therefore seeks to dispose of the vast majority of w-mers with only a single SRAM lookup.

4.3.1: Near-Perfect Hashing

Our design utilizes a "near-perfect" hashing strategy. This strategy approximates the desirable collision-free property of perfect hashing and so eliminates almost all hash lookups requiring more than one access to the SRAM. Even with this property, the design remains efficient both in time needed to find the hash function for a query and in on-chip memory needed to compute it.

A perfect hash [23] for a set $k_1 \ldots k_n$ of keys maps each key to a distinct table entry, with no collisions between keys in the set. Assuming we can find such a perfect hash function h for the query sequence, checking whether the query contains a given w-mer s reduces to retrieving a single entry $h(s)$ from SRAM.

A number of approaches have been described to compute perfect hashes [8]. However, these methods typically suffer two drawbacks. First, the family of possible hash functions h may be inappropriate for efficient hardware implementation. For example, computing h may require arithmetic modulo an arbitrary prime number; alternatively, it may require lookups in a table too large to be stored on chip. Second, finding a perfect hash function for a large set of keys may itself be an expensive operation, requiring seconds to minutes of computing time before the search itself can begin.

Suppose we want to map a set of w-mers, each consisting of $2w$ bits, into a table of size 2^n. To realize this mapping, we construct a hash function h_1 of the form

$$h_1(s) = A(s) \oplus \tau[B(s)],$$

where $A : \{0,1\}^{2w} \rightarrow \{0,1\}^n$ and $B : \{0,1\}^{2w} \rightarrow \{0,1\}^m$ are easily-computed functions of

the key, $\oplus$ is the bitwise XOR operation, and τ is a *displacement table* of 2^m small integers. We use the displacement table τ to resolve collisions: if w-mers s_1 and s_2 have $A(s_1) = A(s_2)$ but $B(s_1) \neq B(s_2)$, choose distinct values for $\tau[B(s_1)]$ and $\tau[B(s_2)]$. Efficient construction of perfect hash functions using displacement tables was studied by, e.g., Tarjan and Yao [24] and more recently by Hagerup et al. [13]; Fox et al. [11] discuss the use of such tables in practice.

We choose the functions A and B from the H_3 family described in [3, 20], which is efficiently computable in hardware. An H_3 function g from p to q bits consists of p q-bit vectors $v_1 \ldots v_p$. Let M be the $p \times q$ Boolean matrix whose rows are the v_i's. Then for a p-bit input vector s, $g(s)$ is the vector-matrix product $s \cdot M$, computed in the Boolean field whose addition and multiplication operators are respectively XOR and AND. It can be shown that, if the matrix M has full rank m over this field, then M maps exactly 2^{p-q} p-bit inputs to each of the 2^q possible outputs. The rank of M can be computed using Gaussian elimination, so one may generate a random H_3 function with full rank by generating random $p \times q$ Boolean matrices M and keeping the first full-rank matrix.

Although efficient algorithms are known [13] for selecting values for the displacement table τ to generate a perfect hash, these algorithms require that to hash N keys, τ must have at least N entries of $\log N$ bits apiece. Such a large table is inappropriate for hardware implementation, but a smaller table may be insufficient to achieve a perfect hash, or it may be computationally intractable to find values achieving it. We circumvent these limitations by relaxing the requirement that our hash be perfect, instead asking only that it achieve few collisions on the input keys. We call such a hash *near-perfect*.

The following efficient greedy heuristic chooses values for the displacement table τ that achieve few, and often no, collisions in our application. First, sort the 2^m possible values of B in decreasing order by the number of keys mapping to each. Let $\{b_1 \ldots b_{2^m}\}$ be the resulting sequence of B-values. For each i from 1 to 2^m, greedily assign a value to $\tau[b_i]$ so as to minimize the number of collisions with all keys that map to B-values $b_1 \ldots b_{i-1}$. Once the initial construction of τ is complete, we may heuristically further improve our hash by trying to modify each entry $\tau[b_i]$ in turn to minimize collisions given the remaining values in τ. Modification of τ may be performed repeatedly until the number of collisions does not change.

In our design, which supports 25 Kbase queries with $w = 11$, we set $n = 17$, $m = 10$, and permit each entry of τ to be up to 8 bits. We therefore map w-mers to a table of 2^{17} entries in SRAM, using a modest 2^{13} bits of on-chip block RAM for τ. On a sample of one hundred 25 Kbase DNA query sequences drawn at random from the human genome, our heuristic almost always generated perfect hash functions, only occasionally producing one or two collisions. On a 2.4 GHz AMD Opteron workstation, the average design time for h_1 on these queries was under 0.1 seconds.

4.3.2: Hash Table Structure

Each 32-bit entry in our hash table stores a 16-bit query position and a *collision bit*, which is set if two or more distinct w-mers from the query map to that entry. Collisions occur if our hash function design does not yield a perfect hash. Unoccupied entries are marked by a special value in the query position field.

If a w-mer s maps to a table entry $h_1(s)$ with the collision bit unset, lookup proceeds as for a perfect hash, using only a single SRAM access. In case of a collision, the implementation performs a second check in a smaller *secondary* hash table (2^{16} entries in our implementation) similar to the primary table, which contains only those query w-mers that collide in the first table. The hash function used for the secondary table, h_2, is chosen from the H_3 family, in similar fashion to A and B above. Because the primary hash function h_1 is near-perfect, the number of entries with collision bits set is small. Hence, accesses to the secondary table are rare, preserving our desired property of at most one SRAM lookup for most w-mers.

Query w-mers s_1 and s_2 that collide in both the primary and secondary tables are merged; that is, they are treated for hash lookup purposes as if they are two copies of the same w-mer. The duplicate facility described below ensures that probes of either s_1 or s_2 return all query positions at which either one occurs. Merging therefore creates false positive matches, which are removed by later pipeline stages, but does not create false negatives. Merging requires that two w-mers collide in both primary and secondary tables, which use independent hash functions, so it is extremely rare.

4.3.3: Resolving Duplicate Keys

When a w-mer occurs at two or more positions in the query, we wish to return not just one such position but all of them. We therefore extend each hash entry with a *duplicate flag*

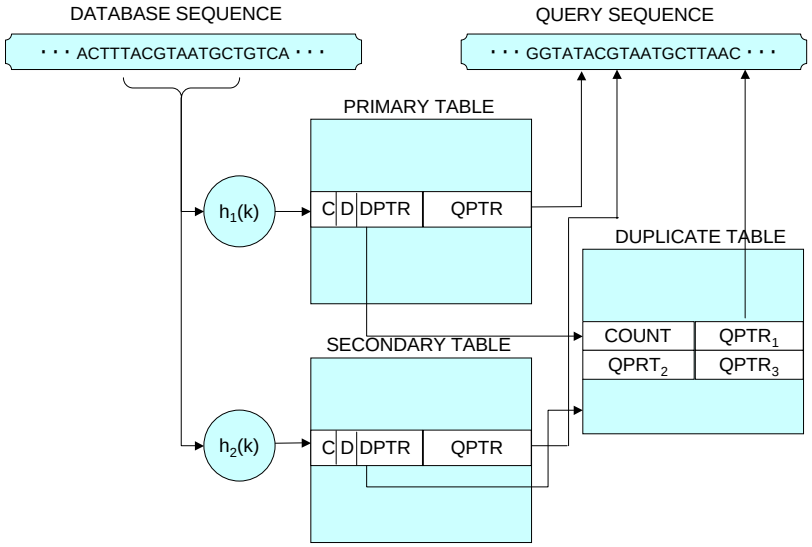

Figure 9. Hash table (stage 1b) datapath

and a pointer into a separate *duplicate table*, also in SRAM. Any w-mer s with more than one occurrence has its duplicate flag set. One occurrence of s is stored in the table entry itself, while the remainder are stored in a contiguous range of 16-bit cells in the duplicate table. The first cell in the range, which is pointed to by the hash table entry, specifies the number of additional occurrences, while each additional cell gives the position of one occurrence.

We align the beginning of each range of cells in the duplicate table on a 32-bit boundary. Hence, a 14-bit pointer in the hash table entry is sufficient to support a table of 2^{15} 16-bit cells – enough for even a fairly degenerate 25 Kbase query.

Accesses to the duplicate table compete with the hash table for SRAM bandwidth, so they increase the average number of probes per w-mer. However, a typical repeat-masked 25 Kbase DNA sequence from the human genome has only a few hundred duplicate 11-mers, so the additional accesses to the table add only slightly to the overall load on SRAM.

The datapath for the hash table (stage 1b) is illustrated in Figure 9, and the control logic that manages the datapath is shown in Figure 10.

There are two conditions under which a false positive result can come from stage 1b. The first, described earlier, is if there is a collision in both the primary and secondary hash tables. The second is when a false positive result from the Bloom filters collides in the primary hash table with a w-mer in the query. Our current implementation passes these

```
address = h1(w-mer);  /* compute address for w-mer in primary table */
if (C == 1) { /* collision flag set */
    address = h2(w-mer); /* change to accessing secondary table */
}
if (D == 1) { /* duplicate flag set */
    address = DOFFSET + DPTR;
    return QPTRs;  /* return all locations which match in query */
} else if (QPTR == NULL) { /* false positive identified */
    return NULL;
} else {  /* unique match in query */
    return QPTR;  /* return location in QPTR */
}
```

Figure 10. Hash table control logic

false positives to later stages (where they are discovered and discarded). An alternative approach would be to perform an explicit test, comparing the w-mer in the database with the w-mer in the query. This would completely eliminate any false positives from stage 1.

4.4: Redundancy Filter

To avoid expending the effort to inspect all the redundant w-mers, NCBI BLASTN uses a redundancy filter to discard the w-mers which fall within the range that already has been inspected by extension in stage 2. This range can be defined as two w-mers which have an overlapping diagonal. Each w-mer is represented by an ordered pair (q_j,d_k), where q_j and d_k are indices into the query and database, respectively. We define the diagonal, D_i, as $D_i = d_k - q_j$. To keep track of the redundant w-mers, a record for every diagonal is stored in a data structure which contains the previous non-redundant database index. If d_k of the current w-mer is greater than the stored value, the w-mer is queued for further inspection. In NCBI BLASTN, the redundancy filter is updated after extension in stage 2. This method can eliminate more than strictly overlapping w-mers without decreasing sensitivity.

We use a slightly different heuristic which does not require feedback from stage 2 [19]. Instead of storing the highest endpoint of the database from the stage 2 extension, Mercury BLASTN implements the redundancy filter by finding all the truly redundant w-mers as well as the ones close to them. The end of the first non-overlapping w-mer out of stage 1b is stored for each diagonal and is updated every time a new non-overlapping w-mer is seen on that diagonal. The redundancy filter is also parameterizable to discard any

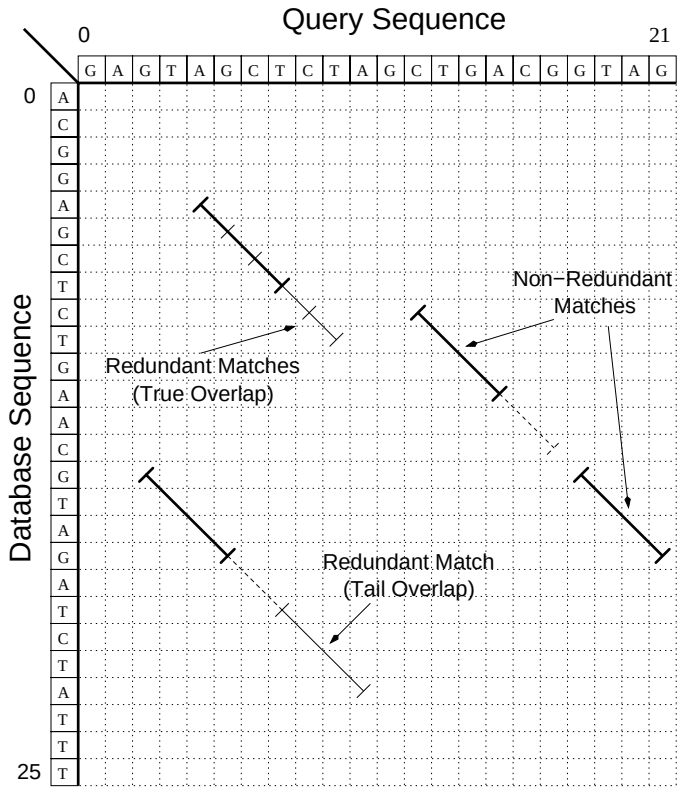

Figure 11. Redundancy filter example

w-mers on a diagonal that are within N bases of a previous diagonal even if they aren't truly overlapping. Setting this parameter allows for a trade off between filter strength and sensitivity.

Figure 11 illustrates an example query and database sequence plotted to form a matrix using a word length of 4. The non-redundant 4-mer matches are shown in bold and redundant 4-mers are shown normal weight. The filter keeps a record of the first 4-mer in each diagonal and passes them to the later stages. 4-mers that are truly overlapping a previous 4-mer on that diagonal are discarded. This is shown on the figure as "True Overlap." An example of a 4-mer that does not overlap the previous entry on that diagonal, but is within the tail region (with N=2) and discarded, is shown in Figure 11 labeled "Tail Overlap."

While the Mercury BLASTN redundancy filtering method is a less stringent filter than NCBI BLASTN's heuristic, it has a strong advantage when implemented in hardware. Since there is no feedback, the pipeline can advance independent of the outcome of stage 2. This increases throughput and consumes less resources than an exact implementation of NCBI BLASTN's redundancy filter.

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

We have built several firmware prototypes of stage 1 that can consume 16 bases/clk. The latest runs at a clock rate of 133 MHz. The prototype is limited to a query size of 50 Kbases. To process queries of size greater than 50 Kbases, we pass the database through the firmware stage multiple times (say r), each pass consuming a fraction of the query $\leq$ 50 Kbases. This results in an effective throughput of $\frac{1}{r}^{th}$ that of 50 Kbase queries for larger queries.

To assist in the performance analysis task, we developed a detailed simulator that provides a cycle-accurate model of the stage 1 design. This simulation model is used to validate the analytical models used to generate Figure 8, and our assumption that we can process a match from stage 1a every clock cycle in stage 1b. We chose 3 of the design points from Figure 8 and executed simulations using a minimum of 30 different queries for each configuration. For each of these simulations we assumed that the input rate into stage 1 is 2.128 Gbases/s, which is its maximum ingest rate. Table 4 compares the results we obtain from these simulations to the analytical predictions for a number of different parameters. We observe that the simulation performance predictions line up extremely well to the predictions using the analytic models.

While the mean time to process a match in stage 1b is slightly higher than the predicted value of 1, the input rate to this stage is such that the utilization remains $< 100\%$. Also, the nature of genomic sequences is such that the assumption of uniform distribution of bases in sequences turns out to be pessimistic. The observed mean throughput of 2.128 Gbases/s (with extremely small standard deviations) builds further confidence in our analytic model results.

The raw throughput supported by our stage 1 implementation is about 2 Gbases/sec.

Table 4. Validation of analytic predictions using simulation

Configuration	Parameter	Analytic	Simulation
25k, k=6, m=32Kb	f	0.369	0.34
	f_r	0.81	0.82
	Process time stage 1b	≈ 1 clk	1.03
	Tput	2.128 Gbases/s	$2.128 \pm 2.9 \times 10^{-2}$ Gbases/s
25k, k=6, m=64Kb	f	0.016	0.014
	f_r	0.81	0.82
	Process time stage 1b	≈ 1 clk	1.07
	Tput	2.128 Gbases/s	$2.128 \pm 7.6 \times 10^{-6}$ Gbases/s
40k, k=6, m=64Kb	f	0.15	0.12
	f_r	0.69	0.71
	Process time stage 1b	≈ 1 clk	1.07
	Tput	2.128 Gbases/s	$2.128 \pm 7.0 \times 10^{-5}$ Gbases/s

f: false positive rate from stage 1a
f_r: $\%f$ removed by stage 1b

Using the Mercury system infrastructure, which currently provides 700 MB/sec of input bandwidth, we can expect a data rate of 1.4 Gbases/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 5 compares the performance of stage 1 in firmware to the software BLASTN.

Table 5. Firmware vs. software stage 1 (throughput and speedup)

Query Size	10 Kbases	25 Kbases	50 Kbases	100 Kbases	1 Mbases	Units
NCBI BLASTN Stage 1 ($Tput_1$)	77.4	34.8	18.1	10.5	0.771	Mbases/sec
Mercury BLASTN Stage 1($Tput_1$)	1400	1400	1400	700	70	Mbases/sec
Speedup (S_1)	18.1	40.2	77.4	66.7	90.8	

As shown in Figure 7, a queue is present in the design to smooth the bursty nature of matches that are generated in stage 1a. The detailed simulation model described above was used to assess the usage of this queue, and also to assess the performance impact of bursty matches. Figures 12 through 14 plot the maximum queue length for 32 executions each of 3 system configurations. The 3 configurations shown are the same as those used for validation purposes above. Across the board, the maximum queue lengths are quite reasonable, with an overall maximum under 1600 matches.

In addition to reasonably bounded queue lengths, the bursty nature of the stage 1a match process has a negligible impact on the overall throughput performance as well. Observing the throughput predictions from simulation (see Table 4), the performance is quite stable

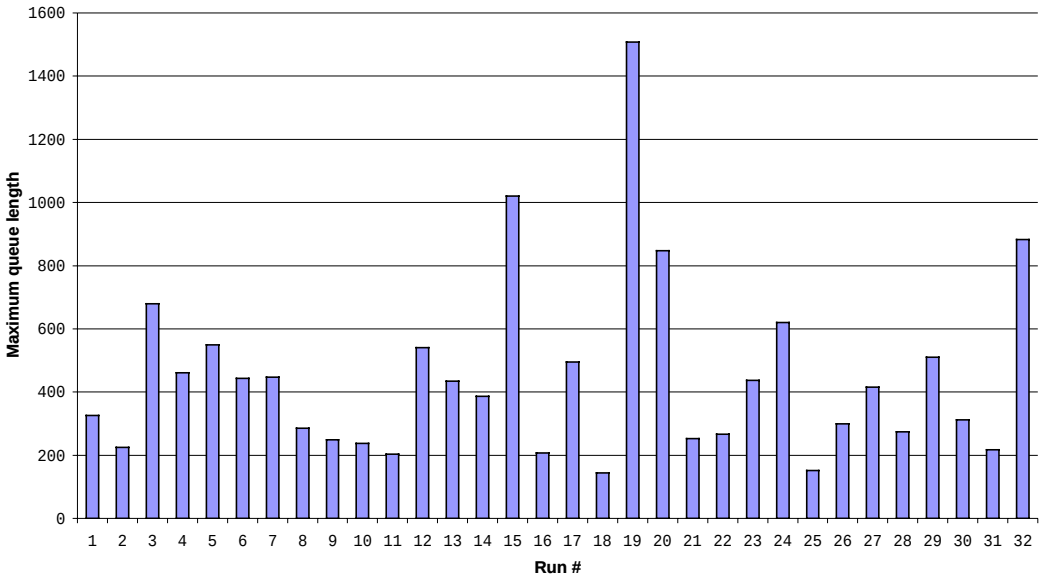

Figure 12. Maximum length of queue between stages 1a and 1b, query size = 25 Kbases, k=6, m=32Kb

(i.e., the standard deviations are quite low) and is not impacted significantly by the data variability.

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 5; 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 6 compares the overall performance of Mercury BLASTN with that of NCBI BLASTN. Though we have shown significant speedup for stage 1 in firmware (refer to Table 5), the overall speedup is limited to a factor of 5 to 8. Overall performance is now limited by the software-based stage 2. Hence, though we successfully deployed stage 1 in

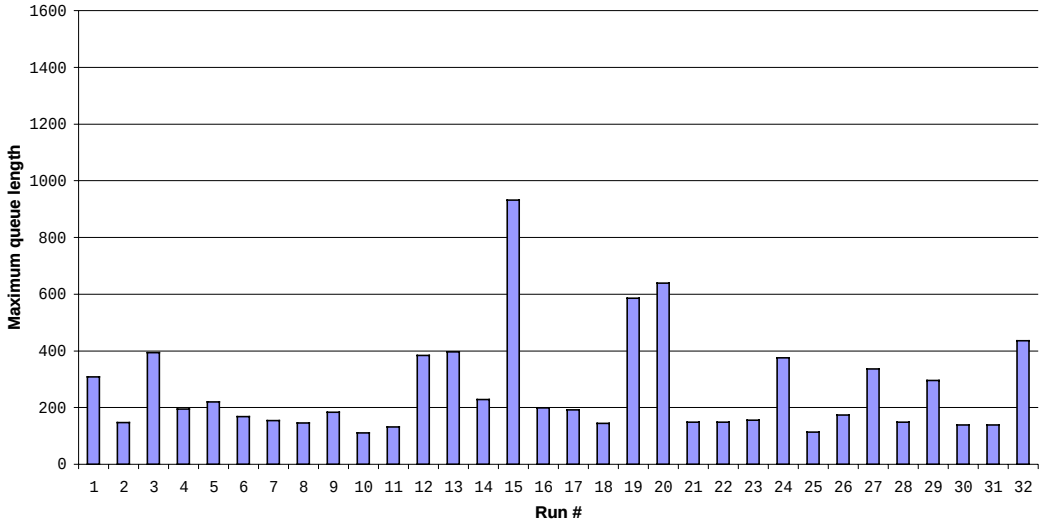

Figure 13. Maximum length of queue between stages 1a and 1b, query size = 25 Kbases, k=6, m=64Kb

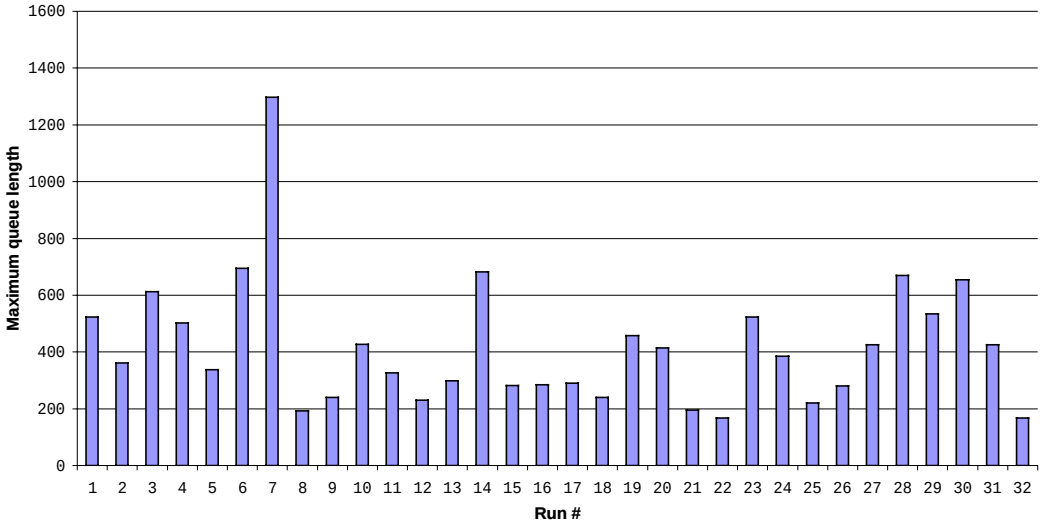

Figure 14. Maximum length of queue between stages 1a and 1b, query size = 40 Kbases, k=6, m=64Kb

firmware with high throughput, the overall application still suffers from limitations imposed by the remaining pipeline stages.

Table 6. Overall performance (throughput and speedup)

Query Size	10 Kbases	25 Kbases	50 Kbases	100 Kbases	1 Mbase	Units
NCBI BLASTN	67.0	29.2	14.9	8.76	0.657	Mbases/sec
Mercury BLASTN	497	181	86.0	52.6	4.44	Mbases/sec
Speedup	7.42	6.21	5.76	6.01	6.84	

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 $Tput_3 = 1/p_1p_2t_3$ (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 matches 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_1t_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 15 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 16 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.

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 [14] and

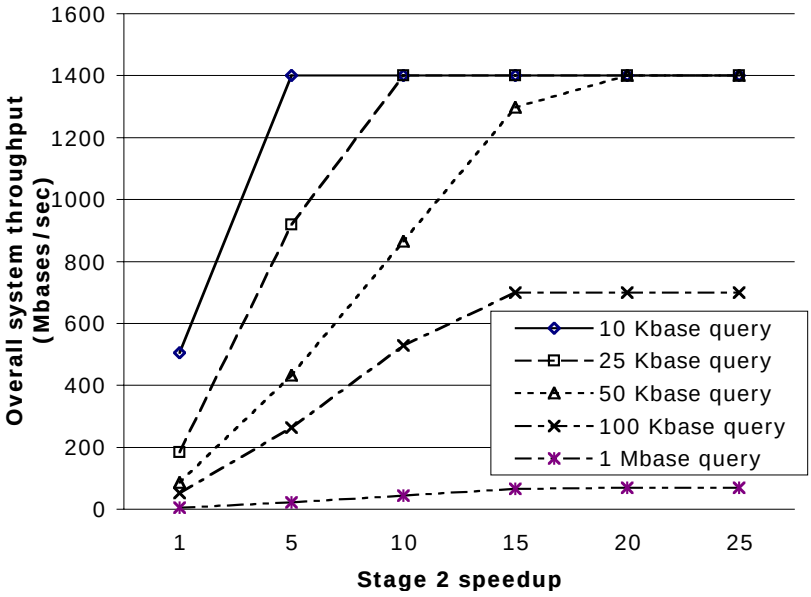

Figure 15. Throughput of Mercury BLASTN with improved stage 2

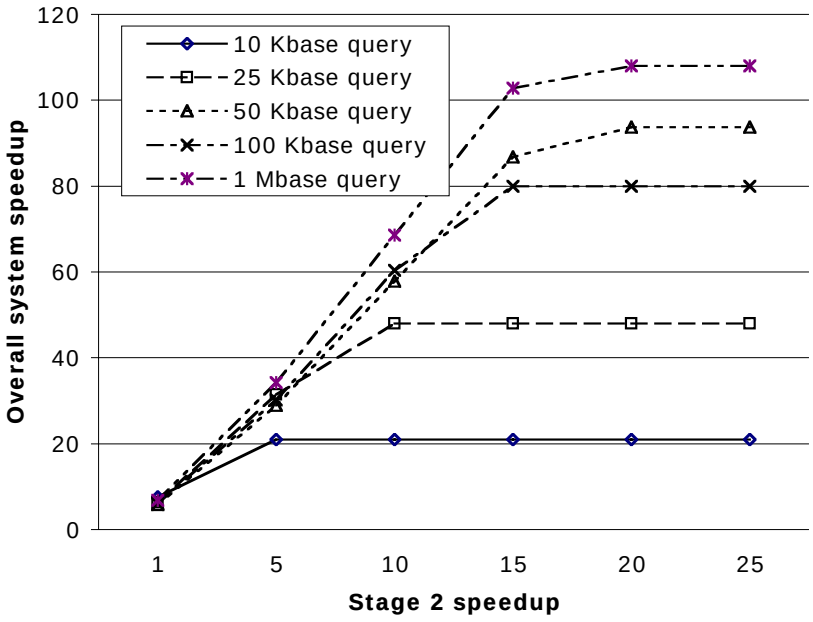

Figure 16. Speedup of Mercury BLASTN over NCBI BLASTN with improved stage 2

reconfigurable logic [15, 28]. 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 [18, 25]. The Paracel GeneMatcher^{TM} [18] 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 [16] 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 [18] and DeCypherBLAST^{TM} from TimeLogic [25]. 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 4.44 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, NSF grants ITR–0313203, ITR–0427794 and CCR–0217334, and NIH/NGHRI grant 1 R42 HG003225–01.

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. *Communications of the ACM*, 13(7):422–426, May 1970.

[3] L. Carter and M. Wegman. Universal classes of hashing functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[4] R. Chamberlain and R. Cytron. Novel techniques for processing unstructured data sets. In *Proc. of IEEE Aerospace Conf.*, March 2005. (In press).

[5] R. Chamberlain, R. Cytron, and N. McVicar. Quickly mining very large software repositories. In *Proc. of Workshop on Mining Software Repositories*, May 2005. (Submitted).

[6] R. Chamberlain, B. Shands, and J. White. Achieving real data throughput for an FPGA co-processor on commodity server platforms. In *Proc. of 1st Workshop on Building Block Engine Architectures for Computers and Networks*, October 2004.

[7] 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.

[8] Z. J. Czech, G. Havas, and B. S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182:1–143, 1997.

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

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

[11] E. A. Fox, L. S. Heath, Q.-F. Chen, and A. M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35:105–121, 1992.

[12] M. Franklin et al. An architecture for fast processing of large unstructured data sets. In *Proc. of 22nd Int'l Conf. on Computer Design*, pages 280–287, October 2004.

[13] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *Journal of Algorithms*, 41:69–85, 2001.

[14] 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.

[15] D. T. Hoang. Searching genetic databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, 1993.

[16] 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.

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

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

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

[20] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE Transactions on Computers*, 46:1378–1381, 1997.

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

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

[23] R. Sprugnoli. Perfect hashing functions: a single probe retrieving method for static sets. *Communications of the ACM*, 20(11):841–850, 1977.

[24] R. E. Tarjan and A. C. C. Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, 1979.

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

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

[27] 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.

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

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