Report Number: WUCSE-2007-13

2007

Mercury BLAST dictionaries: analysis and performance measurement

Jeremy Buhler

This report describes a hashing scheme for a dictionary of short bit strings. The scheme, which we call near-perfect hashing, was designed as part of the construction of Mercury BLAST, an FPGA-based accelerator for the BLAST family of biosequence comparison algorithms. Near-perfect hashing is a heuristic variant of the well-known displacement hashing approach to building perfect hash functions. It uses a family of hash functions composed from linear transformations on bit vectors and lookups in small precomputed tables, both of which are especially appropriate for implementation in ardware logic. We show empirically that for inputs derived from genomic DNA... **Read complete abstract on page 2.**

Mercury BLAST dictionaries: analysis and performance measurement

Jeremy Buhler

Complete Abstract:

This report describes a hashing scheme for a dictionary of short bit strings. The scheme, which we call near-perfect hashing, was designed as part of the construction of Mercury BLAST, an FPGA-based accelerator for the BLAST family of biosequence comparison algorithms. Near-perfect hashing is a heuristic variant of the well-known displacement hashing approach to building perfect hash functions. It uses a family of hash functions composed from linear transformations on bit vectors and lookups in small precomputed tables, both of which are especially appropriate for implementation in ardware logic. We show empirically that for inputs derived from genomic DNA sequences, our scheme obtains a good tradeoff between the size of the hash table and the time required to ompute it from a set of input strings, while generating few or no collisions between keys in the table. One of the building blocks of our scheme is the H_3 family of hash functions, which are linear transformations on bit vectors. We show that the uniformity of hashing performed with randomly chosen linear transformations depends critically on their rank, and that randomly chosen transformations have a high probability of having the maximum possible uniformity. A simple test is sufficient to ensure that a randomly chosen H3 hash function will not cause an unexpectedly large number of collisions. Moreover, if two such functions are chosen independently at random, the second function is unlikely to hash together two keys that were hashed together by the first. Hashing schemes based on H3 hash functions therefore tend to distribute their inputs more uniformly than would be expected under a simple uniform hashing model, and schemes using pairs of these functions are more uniform than would be assumed for a pair of independent hash functions.

2007-13

Mercury BLAST dictionaries: analysis and performance measurement

Authors: Jeremy Buhler

Corresponding Author: jbuhler@cse.wustl.edu

Abstract: This report describes a hashing scheme for a dictionary of short bit strings. The scheme, which we call near-perfect hashing, was designed as part of the construction of Mercury BLAST, an FPGA-based accelerator for the BLAST family of biosequence comparison algorithms.

Near-perfect hashing is a heuristic variant of the well-known displacement hashing approach to building perfect hash functions. It uses a family of hash functions composed from linear transformations on bit vectors and lookups in small precomputed tables, both of which are especially appropriate for implementation in ardware logic. We show empirically that for inputs derived from genomic DNA sequences, our scheme obtains a good tradeoff between the size of the hash table and the time required to ompute it from a set of input strings, while generating few or no collisions between keys in the table.

One of the building blocks of our scheme is the H_3 family of hash functions, which are linear transformations on bit vectors. We show that the uniformity of hashing performed with randomly chosen linear transformations depends critically on their rank, and that randomly chosen transformations have a high probability of having the

Type of Report: Other

Mercury BLAST Dictionaries: Analysis and Performance Measurement

Jeremy Buhler
jbuhler@cse.wustl.edu

1 Introduction

This report describes a hashing scheme for a dictionary of short bit strings. The scheme, which we call *near-perfect hashing*, was designed as part of the construction of Mercury BLAST [7], an FPGA-based accelerator for the BLAST family of biosequence comparison algorithms [3, 2, 4].

Near-perfect hashing is a heuristic variant of the well-known displacement hashing approach [9] to building perfect hash functions. It uses a family of hash functions composed from linear transformations on bit vectors and lookups in small precomputed tables, both of which are especially appropriate for implementation in hardware logic. We show empirically that for inputs derived from genomic DNA sequences, our scheme obtains a good tradeoff between the size of the hash table and the time required to compute it from a set of input strings, while generating few or no collisions between keys in the table.

One of the building blocks of our scheme is the H_3 family of hash functions, which are linear transformations on bit vectors. We show that the uniformity of hashing performed with randomly chosen linear transformations depends critically on their rank, and that randomly chosen transformations have a high probability of having the maximum possible uniformity. A simple test is sufficient to ensure that a randomly chosen H_3 hash function will not cause an unexpectedly large number of collisions. Moreover, if two such functions are chosen independently at random, the second function is unlikely to hash together two keys that were hashed together by the first. Hashing schemes based on H_3 hash functions therefore tend to distribute their inputs *more* uniformly than would be expected under a simple uniform hashing model, and schemes using pairs of these functions are more uniform than would be assumed for a pair of independent hash functions.

The remainder of this report is organized as follows. In the first part, we prove the claimed properties for hash functions in the H_3 family. In the second part, we describe the near-perfect hashing scheme and provide empirical results on how closely it approximates the ideal of perfect hashing on genomic DNA.

2 H_3 Hash Functions of Full Rank

In their seminal paper on universal hashing, Carter and Wegman [5] defined the class H_3 of hash functions. Elements of H_3 are linear transformations over $\mathcal{Z}_2$, the field of bits equipped with AND and XOR operations. H_3 hash functions with a particular input and output width are *universal*; that is, a random hash function is likely to hash any two keys in its input space to different values. The practical utility of H_3 functions was shown empirically by [6], while Alon et al. [1]

showed analytically that these functions are quite unlikely to produce large bin sizes when used in a hash table. Because H_3 functions can be implemented using bit operations, they are particularly attractive for implementation in hardware.

Although the H_3 class has nice properties on average, many functions in the class do *not* distribute their inputs as evenly as possible, and a minority have pathologically bad behavior. In this section, we show that, to guarantee the best-performing hash functions in practice, one may straightforwardly sample from a subspace of the H_3 functions, namely those that have *full rank* when considered as linear transformations. We show that these functions have strong uniformity properties, and in particular that a pair of random full-rank H_3 functions will likely produce a much more uniform result that would be expected for two functions satisfying the tenets of simple uniform hashing.

2.1 Definitions

Let $\mathcal{Z}_2 = (\{0,1\}, +, \times)$ be the finite field of integers modulo 2 equipped with the XOR operation $+$ and the AND operation $\times$. Let $\mathcal{Z}_2^n$ be the n-dimensional vector space over $\mathcal{Z}_2$.

An H_3 *hash function* $H : \mathcal{Z}_2^n \to \mathcal{Z}_2^k$ is a linear transformation from $\mathcal{Z}_2^n$ to $\mathcal{Z}_2^k$, that is, a mapping from n-bit to k-bit numbers, described by a $k \times n$ matrix of bits. The rows of H may be viewed as vectors $e_1 \dots e_k \in \mathcal{Z}_2^n$, and the operation of H on $v \in \mathcal{Z}_2^n$ to produce a hash value $h \in \mathcal{Z}_2^k$ is simply $h = Hv$.

We denote by $\mathcal{H}_k^n$ the space of all *non-singular* H_3 hash functions from $\mathcal{Z}_2^n$ to $\mathcal{Z}_2^k$, that is, all such functions H for which the k vectors $e_1 \dots e_k$ are linearly independent over $\mathcal{Z}_2^n$.

2.2 Goodness of $\mathcal{H}_k^n$

We now show that the family $\mathcal{H}_k^n$ of hash functions has particularly desirable properties. Our first observation is that every function in $\mathcal{H}_k^n$ is highly uniform, in the following sense:

Lemma 2.1. *Any $H \in \mathcal{H}_k^n$ maps exactly 2^{n-k} inputs to every vector in $\mathcal{Z}_2^k$.*

Proof. Because H has rank k, it has a nullspace of dimension $n-k$. Let $V = \{v_1 \dots v_{n-k}\}$ be any basis for this nullspace. Every v for which $Hv = 0$ can be obtained as a linear combination of vectors from V; moreover, no two distinct combinations yield the same vector; otherwise, V would not have rank $n-k$. Conclude that there are exactly 2^{n-k} distinct vectors v for which $Hv = 0$.

Let $x \in \mathcal{Z}_2^n$ be given, and let $w = Hx$. Then for any v in the nullspace of H, $H(x+v) = Hx + Hv = w + 0 = w$. Moreover, *every* $y \in \mathcal{Z}_2^n$ with $Hy = v$ can be decomposed as $x + v$, where $v = y + x$ is an element of H's nullspace. We conclude that there are exactly 2^{n-k} distinct vectors x for which $Hx = w$.

Since there are 2^n inputs, and each w to which some input maps receives only 2^{n-k} inputs, there must be 2^k different values of w realizable by some vector. Conclude that H covers its entire range and maps an equal number of vectors to each element of that range. □

We note that H_3 hash functions of less than full rank are guaranteed *not* to be as uniform as $\mathcal{H}_k^n$. Indeed, if such a function F has rank $r < k$, then it maps all keys onto a subspace of $\mathcal{Z}_2^k$ of size 2^r, leaving the remaining hash values unused!

Next, we prove a nice property of pairs of full-rank H_3 hash functions. Consider a pair of such functions F and G, mapping n-bit strings into a-bit and b-bit hash values, respectively. If each of

F and G are chosen independently at random, then one might expect, reasoning from the usual model of simple uniform hashing, that for vectors x and y, $\Pr(Fx = Fy \wedge Gx = Gy)$ would be the product $\Pr(Fx = Fy)\Pr(Gx = Gy)$. For example, If $n = 22$, $a = 17$, and $b = 10$, then one might expect x and y to map to the same pair of hash values with probability $1/2^{17+10}$. By a Birthday Paradox argument, we therefore expect to see simultaneous collisions under both hash functions when there are more than about 11600 keys.

In fact, viewing our hash functions as linear transformations yields a stronger result: provided that a and b together are at least n, then for any fixed F, a randomly chosen G is likely to separate *all* pairs of keys that collide under F.

Lemma 2.2. *Let $a + b \geq n$, let $F \in \mathcal{H}_a^n$ be any fixed hash function, and let and $G \in \mathcal{H}_b^n$ be a randomly chosen hash function. Then with probability at least $1/4$, for every $x \neq y \in \mathcal{Z}_2^n$, the pairs (Fx, Gx) and (Fy, Gy) are distinct.*

Proof. It suffices to show that with the claimed probability, the rows of F and G together form an $(a + b) \times n$ matrix S of rank n. Since S has full rank, it maps x and y to the same value iff $H(x + y) = 0$, which is possible only if $x + y = 0$, i.e. if $x = y$.

Suppose first that $a + b = n$, and let $e_1, e_2, \ldots e_b$ be the vectors of G. What is the probability that S has rank n? We require that each e_i must be different from any combination of the $e_i's$ *and the vectors of* F. Hence, there are only $2^n - 2^{a+i-1}$ feasible choices for e_i given F and $e_1 \ldots e_{i-1}$, compared to $2^n - 2^{i-1}$ choices for a random G of rank b. Hence, the probability q_S that S has rank k is given by

$$q_S = \prod_{i=1}^{b} \left(1 - \frac{2^n - 2^{i-1+a}}{2^n - 2^{i-1}}\right).$$

Recognizing that $a = n - b$, we make two observations:

- For any fixed n, the probability q_S attains its minimum value for $b = n/2$.
- When $b = n/2$, q_S attains its minimum value in the limit as $n \to \infty$, and this value is

$$\lim_{n\to\infty} \prod_{j=1}^{n/2} \left(1 - \frac{1}{2^j}\right) \geq 0.2887.$$

Conclude that $q_S \geq 0.2887 > 1/4$.

Finally, consider the case when $a + b > n$. The matrix S now has full rank provided that *any subset* of $n - a < b$ linearly independent vectors from G form a rank-n collection together with F. Hence, the probability that S has full rank is at least as great as in the case when $a + b = n$. □

We note that, if it is the user's intent *a priori* to produce two hash functions of sizes a and b, $a + b \geq n$, that generate unique pairs of values, one could also produce a random full-rank list of n vectors, then adjoin any $a + b - n$ rows to this list and divide it to produce the desired pair of functions. However, random generation of the two functions independently will likely work, particularly when $a + b > n$.

2.3 Efficient Sampling from $\mathcal{H}_k^n$

To achieve the guarantees of the previous section, one must draw from the space of hash functions $H \in \mathcal{H}_k^n$. We now show how to sample efficiently from this space. We analyze the following sampling procedure:

Sample(n,k)
    do
        choose k random non-zero n-bit vectors $e_1 \dots e_k$
    until ($e_1 \dots e_k$ are linearly independent)
    return the matrix $H_{k \times n}$ whose ith row is e_i

This procedure samples uniformly from the space of $k \times n$-bit matrices with no zero rows, rejecting any matrix that is not a member of $\mathcal{H}_k^n$. The elimination of zero rows is natural, since the presence of such a row would trivially render the matrix singular. Testing for linear independence can be performed efficiently using, e.g., Gaussian elimination.

Lemma 2.3. *For any $k \leq n$, a single iteration of the do-loop in procedure **Sample**(n,k) produces a member of H_k^n with probability greater than $1/4$.*

Proof. Let $p_{n,k}$ be the probability that a set of k non-zero n-bit vectors, chosen at random with replacement, has rank k. To estimate this probability, we consider the choices of vectors sequentially and define E_i to be the event that e_i is linearly independent of $e_1 \dots e_{i-1}$, given that these $i-1$ vectors are themselves linearly independent. The event E_i occurs precisely when e_i is not equal to the XOR of any nonempty subset $S \subseteq \{e_1 \dots e_{i-1}\}$. One can show that XOR'ing every distinct subset S from this set yields a *distinct* vector; hence, e_i must be different from *exactly* $2^{i-1}-1$ other vectors in $\mathcal{Z}_2^n$.

Because there are $2^n - 1$ non-zero vectors in $\mathcal{Z}_2^n$, we have

$$\Pr(E_i) = \left(1 - \frac{2^{i-1}-1}{2^n - 1}\right).$$

It follows that

$$\begin{aligned} p_{n,k} &= \prod_{i=2}^{k} \Pr(E_i) \\ &= \prod_{i=2}^{k} \left(1 - \frac{2^{i-1}-1}{2^n - 1}\right) \\ &\geq \prod_{j=1}^{n-1} \left(1 - \frac{2^j - 1}{2^n - 1}\right). \end{aligned}$$

In the limit as $n \to \infty$, the last expression achieves its minimum value, which is

$$\lim_{n\to\infty} \prod_{j=1}^{n-1} \left(1 - \frac{1}{2^j}\right) \geq 0.2887.$$

Hence, $p_{n,k} \geq 0.2887 > 1/4$. □

From the lemma, we may conclude that the procedure **Sample(n, k)** succeeds on average after at most 4 trials. Moreover, since all $k \times n$-bit matrices without zero rows are chosen with equal probability, this procedure produces a random member of $\mathcal{H}_k^n$.

The bound given above is reasonably tight for $n = k$; for example, when $n = k = 8$, $p_{n,k}$ is already < 0.3. However, it is quite conservative for hash functions that reduce the size of their input, since it is easier to choose a small set of linearly independent vectors than a large set. For example, $p_{16,16} \approx 0.289$, but $p_{16,15} > 0.577$, and $p_{16,8} > 0.996$. In general, H_3 functions from n to k bits with full rank form a minority only when $n = k$, but one cannot be very sure of choosing such a function without checking its rank unless the gap between n and k is on the order of 10 or more.

2.4 Universality of $\mathcal{H}_k^n$

For completeness, we verify that the family $\mathcal{H}_k^n$ is still *universal*; in other words, a random hash function from this family is expected to cause no more collisions than a truly uniform random mapping from $\mathcal{Z}_2^n$ to $\mathcal{Z}_2^k$. [5] showed that this result holds for the family of all H_3 functions from $\mathcal{Z}_2^n$ to $\mathcal{Z}_2^k$, so it is not surprising that it continues to hold for the "smoothest" functions in this family.

Lemma 2.4. *Let $x \neq y$ be any two distinct elements of $\mathcal{Z}_2^n$, and let H be a random hash function from H_k^n. Then*

$$\Pr(Hx = Hy) \leq \frac{1}{2^k}.$$

Proof. Let $v = x + y$. We have that $Hx = Hy$ iff $Hv = 0$. Let $e_1 \ldots e_k$ be the rows of H; $Hv = 0$ iff $e_i \cdot v = 0$ for every i.

We will use several facts to prove the lemma:

- Firstly, if we consider only the first row of a random $H \in \mathcal{H}_k^n$, this row is equally likely to be any nonzero vector in $\mathcal{Z}_2^n$. Indeed, the analysis of Lemma 2.3 implies that, if we set any non-zero n-bit vector as e_1, there are any equal number of ways to "complete" the $k \times n$ matrix with non-zero vectors so as to obtain a matrix of rank k.

- Secondly, a random $z \in \mathcal{Z}_2^n$ has
$$\Pr(z \cdot v = 0) = 1/2.$$
If we additionally require that z be different from one or more fixed vectors z_i for which $z_i \cdot v = 0$, then this probability becomes *less* than $1/2$.

- Thirdly, if $e_i \cdot v = e_j \cdot v = 0$, then $(e_i + e_j) \cdot v = 0$.

We now ask, what is the probability that $e_i \cdot v = 0$ for every i? Again, we consider the vectors in increasing order by index. By our first fact, the first row of e_1 is a random nonzero n-bit vector, and so by our second fact, $\Pr(e_1 \cdot v = 0) < 1/2$. Now consider the probability that $e_i \cdot v = 0$ for $i > 1$. The vector e_i must be different from the XOR of any subset of $\{e_1 \ldots e_{i-1}\}$; by our third fact, all of these subsets yield vectors s for which $s \cdot v = 0$. It follows that there are strictly fewer ways to choose e_i to ensure $e_i \cdot v = 0$ than there were ways to choose e_1, and so

$$\Pr(e_i \cdot v = 0 \mid e_1 \cdot v = \ldots e_{i-1} \cdot v = 0) < 1/2.$$

We conclude that

$$\Pr(e_i \cdot v = 0, 1 \leq i \leq n) < \frac{1}{2^k},$$

which proves the lemma. □

3 The Near-Perfect Hashing Scheme

This section describes *near-perfect hashing*, a heuristic method for rapidly creating a static dictionary from a collection $C \subseteq \mathcal{Z}_2^n$ of keys. Before describing our construction, we first review briefly the design considerations of Mercury BLAST that prompted it. We then describe the design and compare it to related work. Finally, we study the performance of our design.

3.1 Motivation for Design

The target application for our dictionary is Mercury BLAST [7], an FPGA-based search engine for DNA sequences. Mercury BLAST compares a series of query strings against a large database. A query is converted to a dictionary composed of all its k-base substrings; because DNA sequences contain only four possible letters, these substrings are bit strings in $\mathcal{Z}_2^{2k}$. The dictionary is loaded into SRAM memory attached to the FPGA hardware. The database is then streamed through the hardware at high speed, and each of its k-base substrings is checked to see if it is in the dictionary. A "hit" between query and dictionary is used as a heuristic signal to search more carefully for similarity between the query and the part of the database adjacent to the hit.

To be of use in Mercury BLAST, a dictionary must balance three competing design pressures. Firstly, *the bandwidth of the memory containing the dictionary is limited.* To avoid a performance bottleneck, it is important that lookups in the dictionary usually require only one probe. Secondly, *the available storage for the dictionary and for its hash function are limited.* The available SRAM can hold records for between 2^{17} and 2^{18} keys, which is within a factor of four of the actual number that the dictionary must hold in practice. The hash function itself must be stored on-chip; because it competes for space with other parts of the design, its available storage is limited to 8-16 kilobits. Finally, *the dictionary must be generated quickly.* The search engine deals with multiple queries by reading the entire database once for each query. For a database the size of the human genome, this read takes less than a second. To keep the search engine busy, we must construct the dictionaries for later queries while earlier queries are being processed. If it takes longer to build the dictionary than to perform the search, the engine will be starved for work.

3.2 Dictionary Design

Our dictionary maps a collection $C \subseteq \mathcal{Z}_2^n$ of keys into a sparse table M of 2^a entries. Keys are mapped into the table using a *displacement hash function* [9] consisting of three components: two H_3 hash functions $A \in \mathcal{H}_a^n$ and $B \in \mathcal{H}_b^n$, and a (dense) displacement table T consisting 2^b numbers, in the range $\{0 \ldots 2^m - 1\}$, $m \leq a$. To evaluate the hash function for a key x, we compute $h(x) = Ax \oplus T[Bx]$.

To construct a dictionary from C, we proceed in two stages. First, we randomly select A and B and map each $x \in C$ to the pair (Ax, Bx). If this mapping is not 1:1 on C, we discard A and B and try again. Our result in Lemma 2.2 ensures that a few (and usually just one) selections of A and B suffice for this step to succeed. Second, we choose the entries in table T as follows. We divide the

inputs into up to 2^b *B-groups* $\beta_i = \{x \in C \mid Bx = i\}$ and sort these B-groups in descending order by size. Then, for each β_i in this order, we choose a value for $T[i]$. We try all 2^m possible values for $T[i]$ and keep the one that results in the fewest collisions between β_i and $\beta_1 \ldots \beta_{i-1}$. Finally, we mark each entry of M as either containing no key, containing a single key, or containing a collision of two or more keys.

Looking up a key x in the dictionary entails probing $M[h(x)]$. If this probe finds no key, we are done; if it finds a single key, we may simply check whether this key equals x. If the probe finds a collision, we must resolve it using additional memory accesses; details of the resolution mechanism are given in [7].

3.3 Related Work

Displacement hashing as a mechanism for generating perfect hash functions has a long history, starting with the work of Tarjan and Yao [9]. More recently, Pagh gave an efficient dictionary construction [8] similar to our own that uses displacement hashing to produce a perfect hash. He showed that, provided $m = a$ and $2^b \geq 2|C|$, a displacement table T that yields perfect hash exists with positive probability if the underlying hash functions A and B are universal. Moreover, roughly doubling the size of b is sufficient to ensure that T can be found quickly.

Pagh's construction requires a displacement table of at least $2|C| \log |C|$ bits to produce a perfect hash for all keys in C. For $|C| = 2^{15}$, this requires almost a megabit of storage, well beyond the amount of on-chip storage we can dedicate to the dictionary. Ensuring quick generation of this perfect hash requires even more space. We were therefore motivated to investigate how few collisions could be achieved in practice by an efficient construction that is restricted to a much smaller displacement table.

3.4 Performance

In this section, we empirically investigate the number of colliding keys produced by our dictionary construction using practically feasible parameters and real DNA queries.

We chose the parameters of construction as follows. Our SRAM currently supports a table size M of 2^{17} entries, and we expect this to increase to 2^{18} in the near term. Hence, we tested $a \in \{17, 18\}$. We allowed the total size of the displacement table T to be either 8 or 16 kilobits. For these sizes, the fewest collisions were obtained by setting $m = 8$ and b to 10 and 11, respectively.

Tables with the specified a and b values were tested on collections of 30 non-repetitive DNA sequences chosen at random from the human genome. We tested two sizes of sequence, 12500 bases and 25000 bases, that reflect the range of target query sizes for Mercury BLAST. Each test sequence, together with its reverse complement, was converted to a collection of 11-base keys (i.e. keys in $\mathcal{Z}_2^{22}$); each sequence of length ℓ yields roughly 2ℓ such keys. For each test sequence, we performed the table construction five times using different random hash functions A and B.

Table 1 illustrates the performance of our dictionary construction. For each combination of parameters and query size, we give the mean number of table slots containing a collision, along with a 95% confidence interval based on our 150 trials. To illustrate the benefits of the displacement table, we also give collision counts for a simple, non-displacement strategy in which $h(x) = Ax$. In general, setting $a = 17$ and $b = 10$ sufficed to generate perfect hashes in almost every trial for 12.5 kb sequences, while setting $a = 18$ and $b = 11$ sufficed to generate perfect hashes for 25 kb sequences.

b (bits)	a (bits)	# collisions (12.5 kb)	# collisions (25 kb)
(none)	17	3881 ± 60	14724 ± 97
	18	1957 ± 33	7718 ± 66
10	17	0.067 ± 0.058	4718 ± 53
	18	0 ± 0	600 ± 15
11	17	0 ± 0	1591 ± 33
	18	0 ± 0	0.040 ± 0.045

Table 1: Numbers of colliding keys in small-displacement hash tables. We give averages and 95% confidence intervals for the number of collisions based on 30 non-repetitive sequences from the human genome, each hashed five times using randomly chosen hash functions.

We implemented our dictionary generator in the C++ language. To assess its efficiency, we measured the (wall-clock) time required to generate dictionaries, including reading of input and writing of the dictionary to a file, on a 2 GHz AMD Opteron processor. For $a = 17$, $b = 10$, and $m = 8$, a 12.5kb query required 0.44 seconds to convert to a dictionary; for $a = 18$, $b = 11$, and $m = 8$, a 25kb query required 0.65 seconds. Both times showed very little variation across the 150 random trials.

In conclusion, our near-perfect hashing scheme appears to cause many fewer collisions for given table and input size than a similar scheme without displacement hashing, while requiring well under a second to generate the displacement table component of the hash function. These results make the scheme appropriate for use in the demanding Mercury BLAST application.

References

[1] N. Alon, M. Dietzfelbinger, P. B. Miltersen, E. Petrank, and G. Tardos. Linear hash functions. *Journal of the ACM*, 46:667–83, 1999.

[2] S. F. Altschul and W. Gish. Local alignment statistics. *Methods: a Companion to Methods in Enzymology*, 266:460–80, 1996.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, et al. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–10, 1990.

[4] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–402, 1997.

[5] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–54, 1979.

[6] 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.

[7] P. Krishnamurthy et al. Biosequence similarity search on the *Mercury* system. *Journal of VLSI Signal Processing*, 2007. In press.

[8] R. Pagh. Hash and displace: efficient evaluation of minimal perfect hash functions. In *Proceedings of WADS 1999*, pages 49–54, 1999. LNCS vol. 1663.

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