Hash Metrics: Measuring the Performance Implications of PRNG on Various Hash Schemes

CS 6150
Christopher Earl, Gabe Rudy, Greg Szubzda

Introduction

The common problem of having the ability to maintain a set of elements from a large universal space that may change over time has bred a solid offering of hashing algorithms. However, some of the best of these algorithms need random number generators. Since purely random functions cannot be implemented, the best we have is pseudo-random number generators. Of course, not all algorithms are created equal, and so some will produce numbers that are "more random" than others will.

Randomness can be defined as the lack of a discernible pattern. For a sequence of numbers, or values in general, to be random, then what number (or value) is at a given position in the sequence is not determined or predictable based on its position, the previous numbers or values in the sequence, or any other discernible basis. Statistically speaking, the value each position in a sequence is independent of everything, for some vague definition of everything. So asking "how random" a sequence of values is equivalent to asking "how difficult" it is to predict a value in the sequence given the preceding values. So to test "how random" a sequence of values is, we can try to find patterns or biases in a given sequence.

There are many standard empirical tests that determine how random the products of pseudo-random number generators are. In general, they attempt pattern matching on or measuring for some bias in a given sequence. However, these empirical tests do not test the randomness of the pseudo-random number generators themselves, but their output. This distinction, while subtle, is important. For example, a theoretic pure random number generator can conceivable generate a sequence that contains only one's (i.e. 11111...). While this sequence was generated randomly, the sequence itself exhibits a pattern. Transient patterns (and biases) can also obviously appear in sequences generated by pseudo-random number generators. In fact, by definition pseudo-random number generators have persistent biases and patterns. For example, as will be discussed below, in the sequences produced by the PRand generator the numbers alternates between even and odd, which means the least significant bit has a constant pattern. These qualities can be measured by the empirical tests, so how frequent they appear can also be measured.

However, applications requiring randomness may be hindered, not be affected, or even benefit by such biases and patterns. For a simplistic example, the alternating even and odd pattern of PRand's sequences means that no two consecutive numbers in the sequence will be equal. It is theoretically possible that some application, such as a hashing scheme, would benefit from this small guarantee of non-repetition.

In this project we took a selection of pseudo-random number generators (PRNG), hashing functions and hashing schemes and investigate the variations of performance of their combinations in a hash table benchmark. Using known statistical methods of measuring PRNGs quality, we derive a ranking of the PRNGs based on how often empirical tests find obvious patters or biases. Because hash functions consume random numbers and should produce uniformly distributed outputs, we used the same measurement methods on the output of our hashing functions based on their use of various PRNGs. Finally we implemented different hashing schemes to test the performance of a hash table inserts, lookups and deletes based on their use of the different hash functions crossed with the different PRNGs. With this approach, we hope to show how randomness used in hash functions effects the performance of hash tables and what combinations show the greatest performance empirically.

Pseudo Random Number Generators Used

Although there has been a large body of research involved in constructing good uniform pseudo random number generators, a few published algorithms dominate the real-world implementations. We have chosen to focus on these algorithms for the purposes of testing. We used the R statistical language as a resource in finding the documentation and implementation of most popular PRNGs in their most widely used form. We also looked at the PHP source code (which uses MersenneTwister) and other sources to confirm that we had covered most PRNGs used in the implementation of rand() functions of various platforms and languages. Too counter the well researched and mature PRNGs, we also chose a decidedly simple number generator and finally the GNU C library rand() method.

For our first PRNG, we have the Witchmann Hill algorithm [1], which has a cycle length of *6.9536e12* (see *Applied Statistics* (1984) **33**, 123 which corrects the original article). The second is Marsaglia-Multicarry[2], which is a multiply with carry PRNG which some statisticians believe is superior. Third is an older PRNG from Marsaglia called Super-Duper from the 1970's. This is the original version which does *not* pass the MTUPLE test of the Diehard battery. It has a period of *about 4.6*10^18* for most initial seeds. The seed is two integers (all values allowed for the first seed: the second must be odd). We use the implementation by Reeds et al. (1982–84)[3]. Fourth from Matsumoto and Nishimura (1998) is the Mersenne-Twister[4]. A twisted GFSR with period *2^19937 - 1* and equidistribution in 623 consecutive dimensions (over the whole period). The ‘seed’ is a 624-dimensional set of 32-bit integers plus a current position in that set. Fifth Knuth-TAOCP[5] - A 32-bit integer GFSR using lagged Fibonacci sequences with subtraction. That is, the recurrence used is *X[j] = (X[j-100] - X[j-37]) mod 2^30* and the ‘seed’ is the set of the 100 last numbers (actually recorded as 101 numbers, the last being a cyclic shift of the buffer). The period is around *2^129*.

Then we have some other PRNGs to compare to these more rigorous ones. The GLIBC implementation of rand() returns 15 bits of randomness based on a PRNG that uses 128 bytes of state and a degree seven polynomial in a linear feedback shift register approach with a period around 7*(2^7-1). Because it only produced 15 bit random values (RAND_MAX is 2^15), to get a full 32-bit random value like the other PRNGs, we call the GLIBC rand() 3 times, filling bits 0-15, 15-30 and 30-32 respectively of an unsigned integer as our LibC PRNG for our tests. Finally to have some references of not so good PRNGs, we have an very simple XOR and SHIFT method called PRand and the regular 15-bits of GLIBC rand() mapped to the 32-bit range as a example or a sparse distribution which we label BadLibC as methods 7 and 8 respectively.

Hashing Functions

A hash function's purpose is to map a large domain to a much smaller co-domain in a well-defined and uniform fashion. This can be accomplished through universal hashing, a randomized algorithm. Universal hashing allows the random selection of a hash function which has a random but well-defined mapping. Alternatively, a traditional hash function may be used which does combination and mixing operations on the input data without using random numbers.

We use four hash functions: Zobrist [6], two generic universal functions [7,6], and a rotating function [6]. Zobrist and the generic functions are the only universal hash functions used in our evaluation and so only these use random numbers. Zobrist is a 3-universal function and the generic functions are 2-universal functions given the names "Generic" and "Universal". All hash functions are designed to hash 4 byte integer keys. The Generic hash function is takes four random integers as internal state. The random number are shifted to the right by one with the number one added. In this way the four random numbers are guaranteed to be odd. The hashing operation of Generic multiplies the data by each of the four random values and XOR's the results yielding the final hash. The Rotating hash function uses no internal random state. Its operation consists of, for each byte of the data, a rotation of 4 bits leftware and a XOR of the byte. The Universal hash function takes 32 random integers as internal state where each random value is associated with a single bit of the input data. For each bit in the input data, the corresponding random number is XOR'd to the hash (with initial value of zero). The final hash function, Zobrist, uses the most random state consisting of 1024 random integer values arranged in a 4 by 256 array. For each byte i of the input data whose byte value is j, the random number at the ith row and jth column is XOR'd the the running hash (with initial value of zero).

Based purely on implementation details, an intuitive ranking of hash function performance according to increasing uniformity is: Rotating, Generic, Universal, Zobrist. This initial ranking is supported by experimental evidence in figure 1 showing the ranking of the randomness of the output stream of each hash function. Ranks are made according to the number of successful tests indicated randomness. Random state is sampled from the PRNGs listed horizontally. The unhashed random samples are also charted for comparison.

Rotating ranks last which supports the fact that randomness cannot be synthesized from non-random data. A hash function like rotating is effectively dependent on the randomness of the input stream, as opposed to the internal state. This property of the input data cannot be relied about with certainty. Generic is clearly the least uniform of the universal functions owning to the fact that its internal store of random data is small. Using the simple metric which is the size of the internal random state, Zobrist naturally out-performs universal. Even so, the performance of the last two functions is a close tie. We expect Zobrist to induce the best performance out of the hashing schemes with Universal close behind.

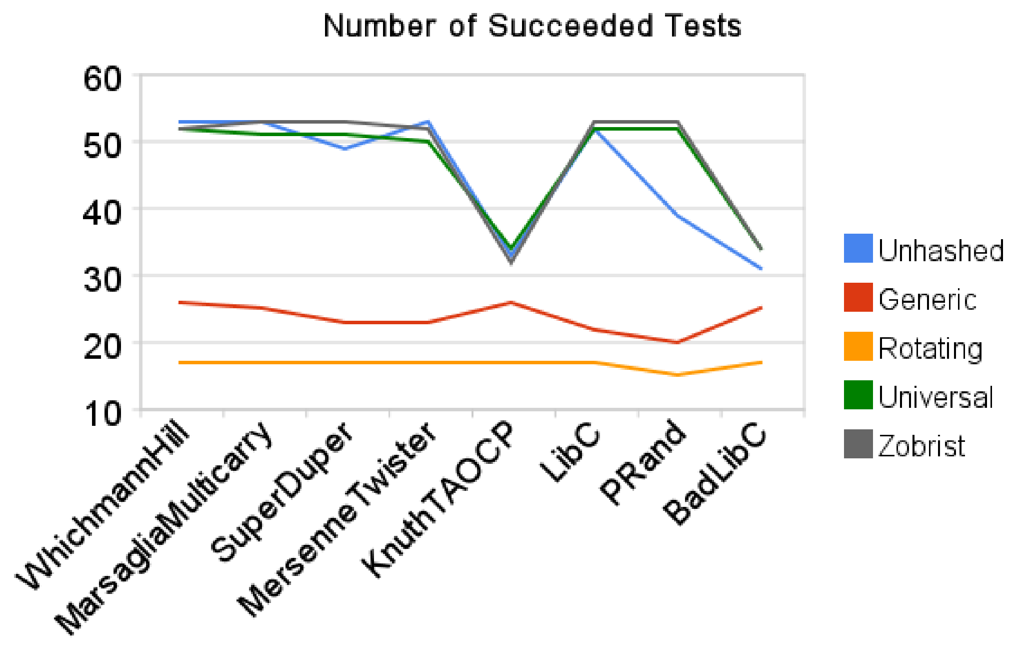


Figure 1

Visualizing PRNGs

Although to get an empirical and comparable metric for the quality of a PRNG, you will need some statistical tools (discussed in the following section), it is also quite useful have create a visual representation of some property of the PRNG for a quick subject comparison of methods. We used a method of generating a image that simply tests whether a number is even or odd, and fills in a pixel with a white or black as a result [8]. Because pixels are filled in a sequential manner, if there is a pattern in the PRNG, or if there is a disproportionate number of even or odd numbers, the image will reflect that.

We created images for every PRNG we used as well as the result of using each hash function on the PRGN numbers as input. As it would be expected the result images where very uniform for good PRNGs such as the MersenneTwister in Figure 2 (a) and for the robust hash functions regardless of the quality of its inputs. The "generic" hash function always produced even numbers and thus had all-white output images. Some interesting results from viewing the other PRNGs include that the KnuthTAOCP always produced odd outputs and there is a clear pattern in LibC's image as seen in Figure 1 (b). PRand as expected does not look that random with it's image of vertical black and white lines. The BadLibc is notably light meaning it produced more even than odd numbers.

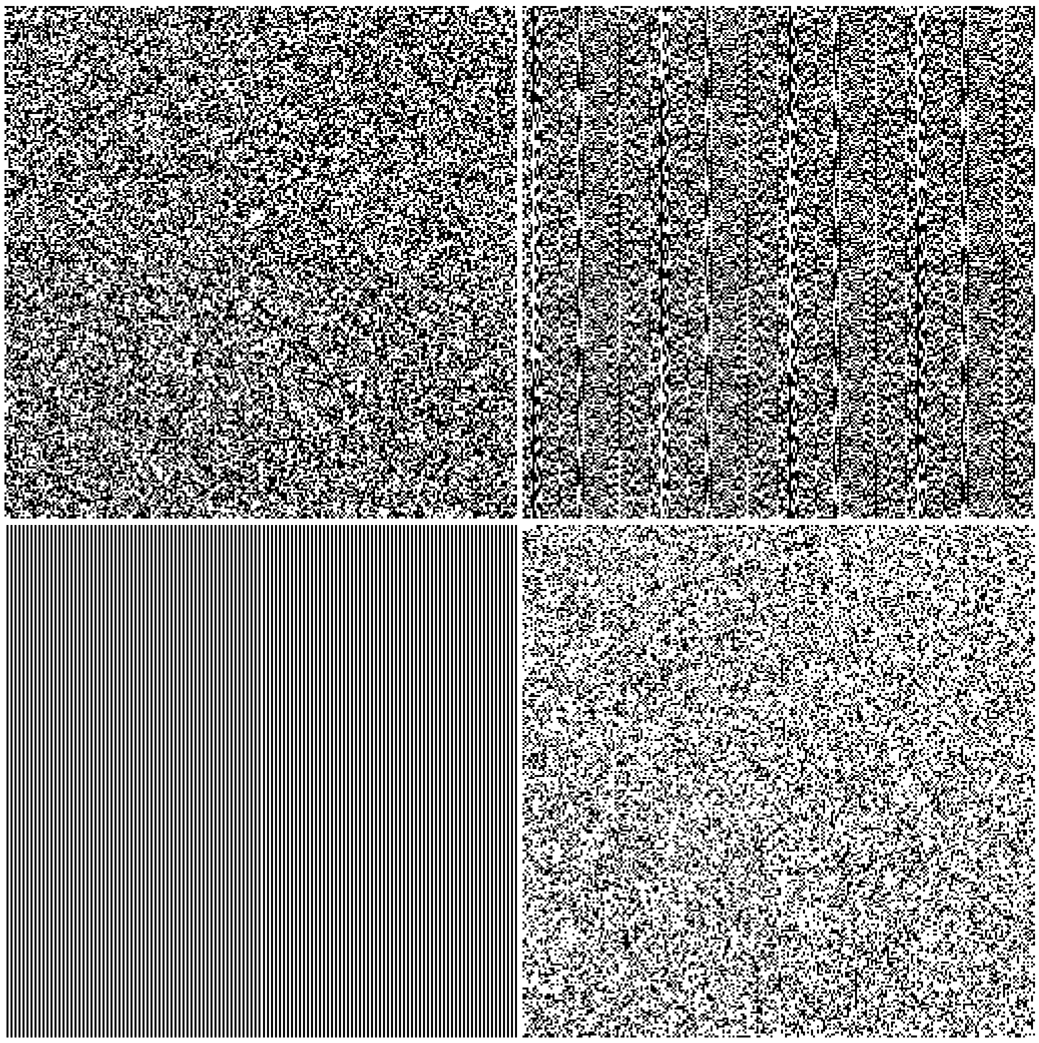


Figure 2. (a) The MersenneTwister (b) LibC (c) PRand and (d) BadLibC

Statistical Methods of Measuring PRNG Quality

For the statistical/empirical tests, the Diehard Battery of Tests of Randomness [9] was used. A sequence of 1 million integers was produced from each PRNG. This sequence was run through each hash function to come up with another sequence of random numbers. The results of the pairings can be seen in the graph titled "Number of Succeeded Tests" above. The numbers in the graph, as the title suggests are the number of passed tests. Each run had 145 tests. However, some of the tests always failed, as detailed below. 87 of the tests always failed, which means that the max any run passed was 58 tests.

KSTEST referenced below is a modified version of the Kolmogorov-Smirnov Test. Each test that was run is briefly explained below. Explanations are modified from the documentation of Diehard.

Birthday Spacing
Choose m birthdays in a year of n days. List the spacings between the birthdays. If j is the number of values that occur more than once in that list, then j is asymptotically Poisson distributed with mean m^3/(4n). Experience shows n must be quite large, say n>=2^18, for comparing the results to the Poisson distribution with that mean. This test

uses n=2^24 and m=2^9, so that the underlying distribution for j is taken to be Poisson with lambda=2^27/(2^26)=2. A sample of 500 j's is taken, and a chi-square goodness of fit test provides a p value. The first test uses bits 1-24 (counting from the left) from integers in the specified file. Then the file is closed and reopened. Next, bits 2-25 are used to provide birthdays, then 3-26 and so on to bits 9-32. Each set of bits provides a p-value, and the nine p-values provide a sample for a KSTEST. This test succeeded intermittently and usually not for the Generic and Universal Hash Functions.

Overlapping 5-Permutation Tests (2 tests)
This test looks at a sequence of one million 32-bit random integers. Each set of five consecutive integers can be in one of 120 states, for the 5! possible orderings of five numbers. Thus the 5th, 6th, 7th,...numbers each provide a state. As many thousands of state transitions are observed, cumulative counts are made of the number of occurences of each state. Then the quadratic form in the weak inverse of the 120x120 covariance matrix yields a test equivalent to the likelihood ratio test that the 120 cell counts came from the specified (asymptotically) normal distribution with the specified 120x120 covariance matrix (with rank 99). This version uses 1,000,000 integers, twice. This test consistently failed for all test cases. It is unclear why; however, it may have been effected by file size.

Rank test for binary matrices (31x31, 32x32, 6x8)
For the 31x31 and 32x32 tests: The leftmost n bits of n random integers from the test sequence are used to form a nxn binary matrix over the field {0,1}. The rank is determined. That rank can be from 0 to n, but ranks< 28 are rare, and their counts are pooled with those for rank 28. Ranks are found for 40,000 such random matrices and a chisquare test is performed on counts for ranks 32,31,30,29 and <=28. For the 6x8 tests: From each of six random 32-bit integers from the generator under test, a specified byte is chosen, and the resulting six bytes form a 6x8 binary matrix whose rank is determined. That rank can be from 0 to 6, but ranks 0,1,2,3 are rare; their counts are pooled with those for rank 4. Ranks are found for 100,000 random matrices, and a chi-square test is performed on counts for ranks 6,5 and <=4. These tests only succeeded for Unhashed and Zobrist and then inconsistently.

Bitstream Tests (20 tests)
The file under test is viewed as a stream of bits. Call them b1,b2,... . Consider an alphabet with two "letters", 0 and 1 and think of the stream of bits as a succession of 20-letter "words", overlapping. Thus the first word is b1b2...b20, the second is b2b3...b21, and so on. The bitstream test counts the number of missing 20-letter (20-bit) words in a string of 2^21 overlapping 20-letter words. There are 2^20 possible 20 letter words. For a truly random string of 2^21+19 bits, the number of missing words j should be (very close to) normally distributed with mean 141,909 and sigma 428. Thus (j-141909)/428 should be a standard normal variate (z score) that leads to a uniform [0,1) p value. The test is repeated twenty times. This test succeeded fairly frequently for Unhashed, Universal and Zobrist, and not at all for the others.

Overlapping-Pairs-Sparse-Occupancy Test (OPSO) (23 tests)
The OPSO test considers 2-letter words from an alphabet of 1024 letters. Each letter is determined by a specified ten bits from a 32-bit integer in the sequence to be tested. OPSO generates 2^21 (overlapping) 2-letter words (from 2^21+1 "keystrokes") and counts the number of missing words---that is 2-letter words which do not appear in the entire sequence. That count should be very close to normally distributed with mean 141,909, sigma 290. Thus (missingwrds-141909)/290 should be a standard normal variable. The OPSO test takes 32 bits at a time from the test file and uses a designated set of ten consecutive bits. It then restarts the file for the next designated 10 bits, and so on. So there are 23 tests from each possible starting point for 10 consecutive bits in a a 32-bit integer. The OPSO test failed for all test cases. It is unclear as to why.

Overlapping-Quadruples-Sparse-Occupancy Test (OQSO) (28 tests)

The test OQSO is similar to OPSO, except that it considers 4-letter words from an alphabet of 32 letters, each letter determined by a designated string of 5 consecutive bits from the test file, elements of which are assumed 32-bit random integers. The mean number of missing words in a sequence of 2^21 four-letter words, (2^21+3 "keystrokes"), is again 141909, with sigma = 295. The mean is based on theory; sigma comes from extensive simulation. The OQSO test failed for all test cases. It is unclear as to why, but given that the OQSO and OPSO tests were implemented together, it is likely the same problem for both.

The DNA test (31 tests)
The DNA test considers an alphabet of 4 letters:: C,G,A,T determined by two designated bits in the sequence of random integers being tested. It considers 10-letter words, so that as in OPSO and OQSO, there are 2^20 possible words, and the mean number of missing words from a string of 2^21 (overlapping) 10-letter words (2^21+9 "keystrokes") is 141909. The standard deviation sigma=339 was determined as for OQSO by simulation. (Sigma for OPSO, 290, is the true value (to three places), not determined by simulation. The DNA test failed for all test cases. It is unclear as to why. It was implemented with the OQSO and OPSO tests, so it probably suffered from the same problem.

Count 1's test for a stream of bytes/specific bytes (26 tests)
The Count 1's test worked both on a stream of bytes and on specific bytes. There were 25 tests for the specific bytes and one for the stream. For a stream of bytes test, the file under test is viewed as a stream of bytes (four per 32 bit integer). Each byte can contain from 0 to 8 1's, with probabilities 1,8,28,56,70,56,28,8,1 over 256. Now let the stream of bytes provide a string of overlapping 5-letter words, each "letter" taking values A,B,C,D,E. The letters are determined by the number of 1's in a byte: 0,1,or 2 yield A, 3 yields B, 4 yields C, 5 yields D and 6,7 or 8 yield E. Thus we have a monkey at a typewriter hitting five keys with various probabilities (37,56,70,56,37 over 256). There are 5^5 possible 5-letter words, and from a string of 256,000 (overlapping) 5-letter words, counts are made on the frequencies for each word. The quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chisquare test: Q5-Q4, the difference of the naive Pearson sums of (OBS-EXP)^2/EXP on counts for 5- and 4-letter cell counts. For specific bytes tests, the file under test is seen as a stream of 32-bit integers. From each integer, a specific byte is chosen , say the left-most:: bits 1 to 8. Each byte can contain from 0 to 8 1's, with probabilitie 1,8,28,56,70,56,28,8,1 over 256. Now let the specified bytes from successive integers provide a string of 5-letter words, each "letter" taking values A,B,C,D,E. The letters are determined as for the stream of bytes described above. Thus we again have the probabilities: 37,56,70,56,37 over 256 for each letter. The 5-letter words are counted for the frequency of each word. The quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chisquare test: Q5-Q4, the difference of the naive Pearson sums of (OBS-EXP)^2/EXP on counts for 5- and 4-letter cell counts. For the stream of bytes test, only Unhashed, Universal, and Zobrist succeeded at all and then not always. For specific bytes tests, all succeeded with varying likelihood. As was common throughout the tests, Unhashed, Universal, and Zobrist succeeded most often.

Parking Lot Test
In a square of side 100, use the input to determine the location to "park" a car---a circle of radius 1. Then try to park a 2nd, a 3rd, and so on, each time parking "by ear". That is, if an attempt to park a car causes a crash with one already parked, try again at a new random location. (To avoid path problems, consider parking helicopters rather than cars.) Each attempt leads to either a crash or a success, the latter followed by an increment to the list of cars already parked. If we plot n, the number of attempts, versus k, the number successfully parked, we get a curve that should be similar to those provided by a perfect random number generator. Theory for the behavior of such a random curve seems beyond reach, and as graphics displays are not available for this battery of tests, a simple characterization of the random experiment is used: k, the

number of cars successfully parked after n=12,000 attempts. Simulation shows that k should average 3523 with sigma 21.9 and is very close to normally distributed. Thus (k-3523)/21.9 should be a standard normal variable, which, converted to a uniform variable, provides input to a KSTEST based on a sample of 10. Except for Rotating, which failed every time, the Parking Lot Test was passed by every input.

Minimum Distance Test
The Minimum Distance test chooses n=8000 points in a square of side 10000 based on the input file. Find d, the minimum distance between the (n^2-n)/2 pairs of points. If the points are truly independent uniform, then d^2, the square of the minimum distance should be (very close to) exponentially distributed with mean .995 . Thus 1-exp(-d^2/.995) should be uniform on [0,1) and a KSTEST on the resulting 100 values from 100 tests serves as a test of uniformity for random points in the square. Every Minimum Distance test failed for unknown reasons.

3D Spheres test
This test uses the input file to choose 4000 random points in a cube of edge 1000. At each point, center a sphere large enough to reach the next closest point. Then the volume of the smallest such sphere is (very close to) exponentially distributed with mean 120pi/3. Thus the radius cubed is exponential with mean 30. (The mean is obtained by extensive simulation). The 3D Spheres test generates 4000 such spheres 20 times. Each min radius cubed leads to a uniform variable by means of 1-exp(-r^3/30.), then a KSTEST is done on the 20 p-values. Except for Rotating, which failed every time, the 3D Spheres Test was passed by every input.

Squeeze Test
Integers from the input file are floated to get uniforms on [0,1). Starting with k=2^31=2147483647, the test finds j, the number of iterations necessary to reduce k to 1, using the reduction k=ceiling(k*U), with U provided by floating integers from the file being tested. Such j's are found 100,000 times, then counts for the number of times j was <=6,7,...,47,>=48 are used to provide a chi-square test for cell frequencies. Every Squeeze test failed for an unknown reason.

Overlapping Sums Test
Integers from the input file are floated to get a sequence U(1),U(2),... of uniform [0,1) variables. Then overlapping sums, S(1)=U(1)+...+U(100), S2=U(2)+...+U(101),... are formed. The S's are virtually normal with a certain covariance matrix. A linear transformation of the S's converts them to a sequence of independent standard normals, which are converted to uniform variables for a KSTEST. The p-values from ten KSTESTs are given still another KSTEST. Except for Rotating, which failed every time, the Overlapping Sums was passed by every input.

Runs Test (4 tests)
The Runs test counts runs up, and runs down, in a sequence of uniform [0,1) variables, obtained by floating the 32-bit integers in the specified file. This example shows how runs are counted: .123,.357,.789,.425,.224,.416,.95 contains an up-run of length 3, a down-run of length 2 and an up-run of (at least) 2, depending on the next values. The covariance matrices for the runs-up and runs-down are well known, leading to chisquare tests for quadratic forms in the weak inverses of the covariance matrices. Runs are counted for sequences of length 10,000. This is done ten times. Then repeated. The Runs Test was passed for all input.

Craps Test (2 tests)
The Craps test plays 200,000 games of craps, and finds the number of wins and the number of throws necessary to end each game. The number of wins should be (very close to) a normal with mean 200000p and variance 200000p(1-p), with p=244/495. Throws necessary to complete the game can vary from 1 to infinity, but counts for all>21 are lumped with 21. A chi-square test is made on the no.-of-throws cell counts.

Each 32-bit integer from the test file provides the value for the throw of a die, by floating to [0,1), multiplying by 6 and taking 1 plus the integer part of the result. The second Craps test was always failed, but the first was usually passed by every input except for Rotating, which always failed.

Hash Table Schemes

We evaluate three hashing schemes: Cuckoo hashing [7], Chaining [10], and Two-Choice chaining [11]. Cuckoo hashing is an open addressing, or closed hashing, method of resolving hash collisions. Such a method relies on probing alternate locations after an initial collision. However, a successfully probed alternate location, meaning it's free, may collide with a future output of a the hash function on different input. In Cuckoo, by default a minimum of two hash functions are used, each associated with it's own table. Each key to be hashed has two possible locations determined by one of these hash functions. On insertion, the key is placed in the location specified by the first hash function. If it happens that the location is occupied by a different key, the insertion of the new key, nevertheless, proceeds to replace the old key. What follows is a greedy algorithm in that each evicted key is hashed with the alternate hash function and this key, in turn, evicts a potential old key in the alternate table. A situation may arise in which such eviction proceeds infinitely. To maintain the well-definedness of the hashing scheme, infinite cycles are detected with a bound on the number of evictions that can take place. When this condition arises, the open addressing nature of Cuckoo hashing warrants the need to double each table's size, select new hash functions, and rehash all data. Such action is necessary because open addressing stores keys directly in the table and not only would exhaust all table locations but would, with high probability, reenter an eviction cycle. Such rehashing is done in-place and is transparent to an outside entity doing the insertion. Lookup operations are done in worst-case constant time since only two locations need to be checked for any given key.

The two other hashing schemes we use are both closed addressing, or open hashing, schemes. We use the Chaining and the Two-Choice chaining scheme which are variations on the same idea. Unlike in the open addressing scheme where collisions are resolved by selecting new locations, in closed addressing collisions are resolved within the colliding location such that potentially more than one key maintains the same location. In the case of Chaining, colliding keys are maintained in a unordered array with the policy that new keys are appended and lookups are done sequentially. Worst case lookup time is proportional to the size of longest chain. It is therefore important to have short chains in practice. Short chains are made possible with good hash functions which uniformly distribute locations, thereby making chain sizes equal with greater probability, but also with a variation of Chaining called Two-Choice chaining.

Two-Choice chaining preserves the main aspects of regular chaining with the exception that two hash functions are now used with a single table. On insertion, both hash functions are evaluated on the key. That location is selected which currently has the shorter chain. On lookup, the chains at both locations are sequentially searched. By allowing two locations to be selected and the shortest chain chosen from among these locations, Two-Choice increases the probability of maintaining shorter chains than the general Chaining scheme. If the hash function employed lacks sufficient uniformity, this scheme may make up for this lacking by explicitly preferring the shorter chain. However, it is the case that two chains must be traversed during a lookup operation with two-choice whose combined size may not be advantageous over the general Chaining scheme. In light of this fact, Two-Choice does not have a clear intuitive advantage among the chaining schemes.

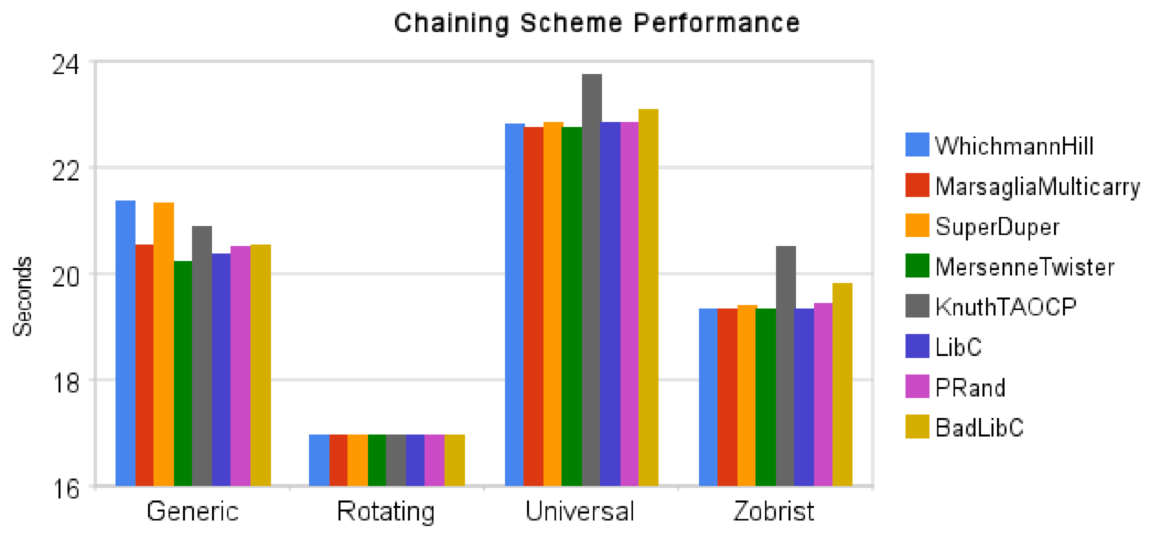
Given that Cuckoo is an open addressing scheme, it imperative that the scheme supports dynamic resizing of the table to prevent the exhaustion of locations. The Cuckoo implementation supports resizing by doubling the hash table, selecting new hash functions, and rehashing all keys. The closed addressing schemes used do not need resizing since chains may grow without exhausting locations.

Experimental Design

To exemplify real-world use of hash tables, our benchmarks measure the following set of operations: The insertion of all the keys, the lookup of all the keys, the removal of the first half of the keys, the re-insertion of those keys and finally the lookup of the full set of keys. For simplicity, no "data" was associated with keys. A benchmark program was compiled for each combination of hash function and hash scheme. Each PRNG generated 10 million numbers and wrote them to a file to be used later by each benchmark driver. The benchmark started a timer after all the of the random numbers were read into memory (so disk performance was not considered) and the hash functions were initialized. The benchmark stopped its timer after the full sequence of operations were finished.

The input data consistently used for all benchmarks is a stream of 4 byte keys totaling 64 MB (16M keys). The data is derived from a sample dump of the English Wikipedia [12]. However, the data sample enwik9 was not used unmodified in full. The first 16M characters of the sample were considered and the binary integer key corresponding to each character was synthesized as follows. Each character, whose integer value is byte encoded, was given an initial starting count equally spaced along the 4 byte integer value domain. Each character read from the original input sequence would output the character's corresponding count after which the count would be incremented by three. This attempts to, but does not completely, remove duplicate keys in the output. More than 95% of the keys in the resulting data is unique which is sufficient for our purposes.

Each hashing scheme was benchmarked with an initial table size, counted in number of locations, of 32M such that the table is twice as large as the number of keys to be hashed. Each benchmark number reported is the average of three individual runs.



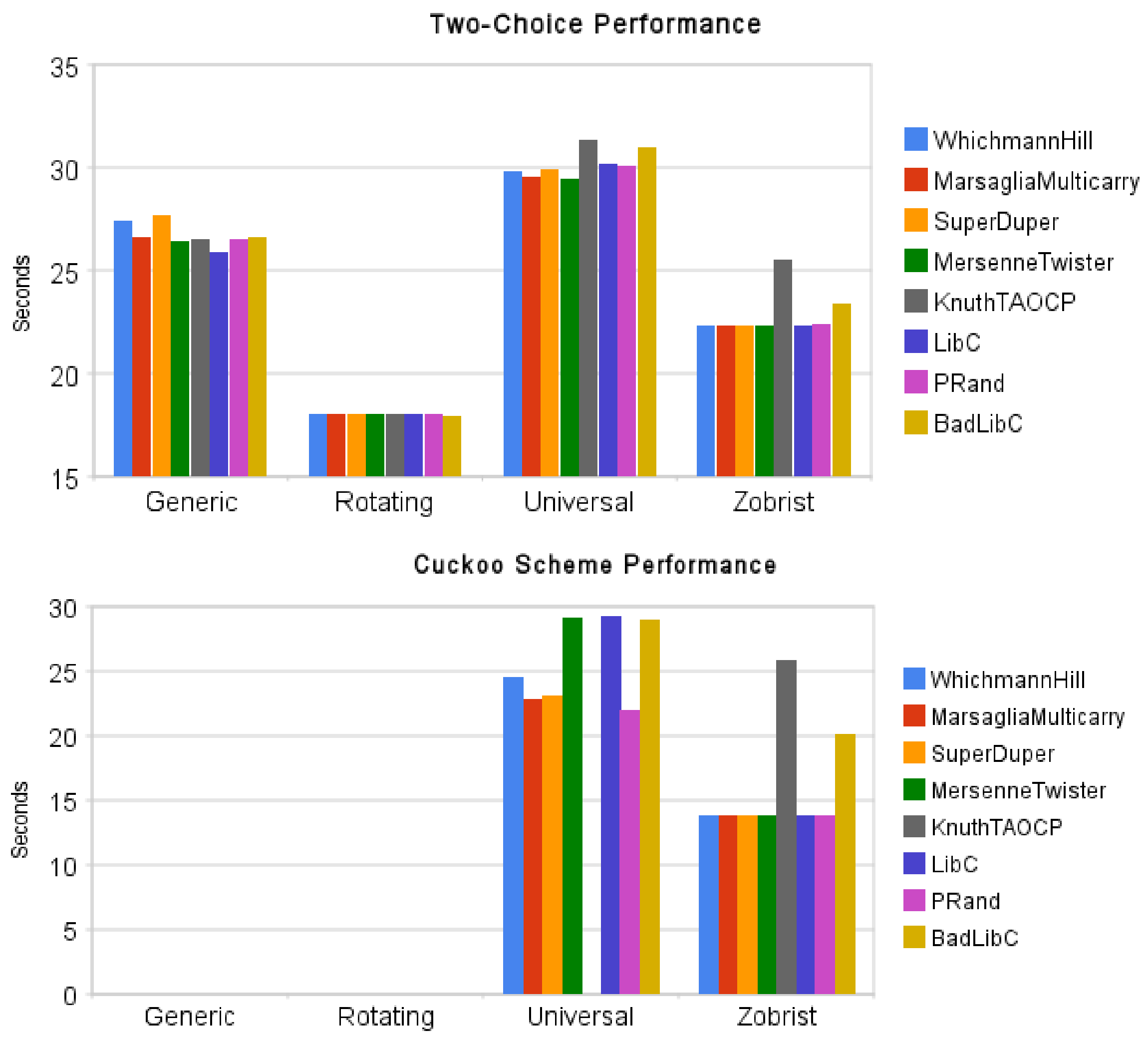


Figure 3. Each hashing scheme and its performance with the four hash functions and 8 PRNGs

Results

In Figure 3 we graph the results of running the set of experimental benchmarks. We were excited to see such a wide variation in performance from 13.8s to 31.4s. I is clear that the Rotating hash method has a very low computational overhead, and all else being equal comes out ahead. Of course, not all else is equal as we discuss below. Of particular interest is how our a measurably less desirable PRNG like KnuthTAOCP can as much as doubling of computation time compared to good PRNGs with certain hash schemes and functions like Cuckoo-Zobrist. On the flip side, you can see that Universal seems more robust to variant quality PRNG input compared to Generic and Zobrist. Remember that Rotating does not consume random numbers. Although we expected Zobrist to perform quite well, it was surprising that the Universal function was not faster. This can most likely be explained by the fact that it simply is more computationally expensive and in this test of speed that does not outweigh its good behavior as a hash function.

The Cuckoo hashing scheme is more complex in it's behavior and if used naively can be dangerous. The reasons there are no results for all of the Cuckoo-Generic and Cuckoo-Rotating schemes is that the hash functions provided too many keys that were identical and thus caused so many rehash calls that malloc failed as the operating system denied the process more memory. Both Generic and Rotating use integer overflow operations and with our real world input data resulted in "small" keys (higher order bits are all zero) that caused high collision rates and the degenerate behavior. The Cuckoo algorithm performs a rehash in these situations because when there are high coalition rates, growing the table should theoretically decrease collision rates. But because Cuckoo uses a masking mechanism to place hash keys in the table, it so happens that even after growing the table the hash functions producing keys with zeroed high order bits still have high collisions, causing another rehash operation. So although a simple hashing function like Rotating may seem like a win, one must understand fully the intended use to avoid catastrophically bad behavior.

In general, it is observed that clustering of hash keys caused by either weak hash functions or as empirically shown from weak PRNGs can have very real impact of the performance of a hash table system. And on the flip side, quality PRNGs will improve the uniformity of a hash functions keys and thus improve its performance. Although shortcuts can be made in the name of performance, fast and robust solutions take advantage of both good PRNGs and hash functions.

The entire set of source code for the PRNGs, hash functions, hash schemes and benchmark scripts is available on the mash-metric google code page we created for the purposes of this project [13].

References

[1] Wichmann, B. A. and Hill, I. D. (1982) Algorithm AS 183: An Efficient and Portable Pseudo-random Number Generator, Applied Statistics, 31, 188–190; Remarks: 34, 198 and 35, 89
[2] Marsaglia, G. (1997) A random number generator for C. Discussion paper, posting on Usenet newsgroup sci.stat.math on September 29, 1997.
[3] Reeds, J., Hubert, S. and Abrahams, M. (1982–4) C implementation of SuperDuper, University of California at Berkeley. (Personal communication from Jim Reeds to Ross Ihaka.)
[4] Matsumoto, M. and Nishimura, T. (1998) Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Transactions on Modeling and Computer Simulation, 8, 3–30.
Source code at http://www.math.keio.ac.jp/~matumoto/emt.html.
[5] Knuth, D. E. (2002) The Art of Computer Programming. Volume 2, third edition, ninth printing.
See http://Sunburn.Stanford.EDU/~knuth/news02.html.
[6] Bob Jenkins. Hash Functions. http://burtleburtle.net/bob/hash/doobs.html
[7] Rasmus Pagh, Flemming Friche Rodler. Cuckoo Hashing
[8] http://www.random.org/analysis/
[9] George Marsaglia. (1995). Diehard Battery of Tests of Randomness.
http://stat.fsu.edu/pub/diehard/
[10] Donald Knuth. (1998). The Art of Computer Programming. Volume 3: Sorting and Searching (2nd ed.)
[11] Michael Mitzenmacher, Andrea W. Richa, Ramesh Sitaraman. The Power of Two Random Choices: A Survey of Techniques and Results
[12] enwik9. http://cs.fit.edu/~mmahoney/compression/textdata.html
[13] http://code.google.com/p/hash-metrics/