An Implementation of the AKS Primality Test

Robert G. Salembier and Paul Southerington, *Member, IEEE*

***Abstract*— We implement the Agrawal-Kayal-Saxena primality testing algorithm. We discuss optimizations to the implementation that resulted in improved performance over the initial implementation. We further discuss methods of obtaining faster runtimes for candidate primes of increasing size.**

***Index Terms*—AKS, Prime Numbers, Primality Testing**

I. Introduction

One of the most useful applications of mathematics is distinguishing between prime and composite numbers. Primality tests are algorithms that are used to directly determine whether a specified number is prime or composite without resorting to factoring. Such primality tests may be classified into two discrete types – probabilistic and deterministic.

Probabilistic primality tests are often quite efficient, but do not provide an absolute guarantee of accuracy. Such tests may provide a bound on the degree of inaccuracy, however. With the Miller-Rabin test, for example, the probability of error may be reduced to an arbitrarily small number by increasing the number of iterations performed.

In contrast to this approach, a deterministic primality test provides a guaranteed result. Examples of deterministic primality tests include the cyclotomy, Lucas-Lehmer, Adleman-Pomerance, and Elliptic curve methods[1]. Such tests are often impractical to implement. The most common problem with such tests is a runtime that is exponential or worse with regard to the number being tested. The amount of time required to determine whether a number is prime in such cases will be unacceptably long. Another problem of some deterministic tests is a dependence on mathematical statements that have not been mathematically proven.

On 6 August 2002 a paper, "Primes is in P" [2] was published that presented a deterministic primality test with polynomial time execution. Polynomial time simply means that the runtime can be represented by a polynomial of degree less than n, where n is the candidate prime. Manindra Agrawal, Neeraj Kayal and Nitin Saxena developed this algorithm which is commonly called AKS. The calculated execution time of the algorithm according to the original paper is $\tilde{O}((\log n)^{12})$. The "$\tilde{O}$" indicates the complexity of the algorithm. In this case, the dominant term affecting execution time will be of the order $(\log n)^{12}$.

In the field of cryptography prime numbers are used in RSA, and DSA. These are two of the most prominent public key algorithms in use today. Public and private key generation is the slowest part of both of these algorithms due to the lengthy time it takes to find adequate large prime numbers. Currently a fast primality test such as Miller Rabin is in use in the RSA algorithm. The problem is that this test has the potential for marking a composite number as prime, although the probability is very small for the number of iterations that are used. It would be very advantageous to be able to run a deterministic primality test in and execution time that is in relative proximity to the fast probabilistic algorithms in use today.

II. The AKS Algorithm

The AKS algorithm depends highly on Fermat's Little Theorem. The theorem is a special case of Euler's theorem and is necessary, but insufficient to determine primality. The test fails for pseudoprime numbers and Carmichael numbers. Fermat's Little Theorem is:

Let $a \in \mathbb{Z}$, $n \in \mathbb{N}$, $n \geq 2$ and $\gcd(a, n) = 1$. Then n is prime iff the polynomial equation holds:

$$(x+a)^n \equiv x^n + a \pmod{n} \qquad \textbf{(Eqn. 1)}$$

Unfortunately, this test fails for a specific class of numbers, known as pseudoprimes, which include the Carmichael numbers. These pseudoprimes leave a residue of zero when tested, causing the test to fail.

An additional problem with this algorithm is that it has $\tilde{O}(n)$ complexity. This is unacceptably high for a primality test which is not correct for all numbers. In order to reduce this complexity the following modification was formulated by the authors of the AKS algorithm. For an appropriately chosen small value of r, then for all values of a and r:

$$(x+a)^n \equiv x^n + a \pmod{x^r - 1, n} \qquad \textbf{(Eqn. 2)}$$

The number of values for a that must be tested and the value of r are limited by a polynomial in $\log n$, thus resulting in a primality test that is both deterministic and can execute in polynomial time [3]. The AKS algorithm as shown below is relatively easy to follow and fairly simple to implement:

Manuscript received May 12, 2005.

R. G. Salembier is a graduate student in Computer Engineering at George Mason University, Fairfax, VA 22030 USA (e-mail: rsalembi@gmu.edu).

P. Southerington is a graduate student in Computer Engineering at George Mason University, Fairfax, VA 22030 USA (e-mail: psouther@gmu.edu).

AKS Algorithm:
($n \in \mathbb{N}$, $n > 1$)
1. If ($n = a^b$ for $a \in \mathbb{N}$ and $b > 1$), output COMPOSITE.
2. Find the smallest r such that $O_r(n) > 4 \log^2 n$
3. If $1 < GCD(a, n) < n$ for all $a \leq r$, output COMPOSITE.
4. If $n \leq r$, output PRIME.
5. For a = 1 to $\lfloor 2\sqrt{\varphi(r)} \log n \rfloor$ do:
   if $((x + a)^n \neq x^n + a \pmod{x^r - 1, n})$,
   output COMPOSITE.
6. Output PRIME.

This algorithm executes in $\tilde{O}(\log^{10.5} n)$ time. Thus as *n* grows larger the increase in execution time will be proportional to $(\log n)^{10.5}$. This is a polynomial time function, which although not as fast as the probabilistic tests used today, has the advantage of being fully deterministic.

Stepping through the algorithm is the best way to understand what is being tested at each phase. The initial step is testing whether n is a perfect power, which would immediately allow it to be determined composite. The second step is where Fermat's Little theorem comes into play. The AKS team chooses this point to find *r* because the next steps are bounded by the value of *r*. The value of *r* is chosen in such a manner that it is the smallest value greater than $4 \log^2 n$ that still holds for **Equation 2**. Step three determines the greatest common divisor between n and all values less than or equal to *r*. These are done to make sure that all values r and below are relatively prime to *n*. The next step simply states that if the value found for *r* is greater than *n*, then *n* is prime. Step five is the all-encompassing step, and as we have found is the most time consuming step. All operations in this step are conduced in the polynomial ring $(\mathbb{Z}/n)[x]/(x^r-1)$. Polynomial exponentiation is being conducted on the left side in the ring and the right side is simply reduced to remain in the ring. These two values are compared to determine if they are equivalent. This relation has to be tested for all values of *a* from 1 to the bound set forth in the algorithm [3]. If the relation holds true for all values of *a*, than the number is prime. The computations of the algorithm are dominated by this step; this step determines the order of the algorithm.

III. Revised Algorithm

Initial review of the AKS algorithm by mathematicians was predominately positive. The most significant observed flaw was that the order of the algorithm was unacceptably high. As a result, the execution time was far too great in comparison to times of the current deterministic primality tests. There have been many people that have made improvements on the potential run-time of the algorithm, such as H. Lenstra, C. Pomerance, and D. Bernstein. Others have attempted, in great detail to explain some of the changes made in the improvements made by the aforementioned people. The algorithm that we will detail below was taken from the improvements made by Lenstra and Pomerance [4]. They have reduced the algorithm to what is now four steps:

AKS Algorithm (Lenstra & Pomerance Improvements):
($n \in \mathbb{N}$, $n > 1$)
1. If ($n = a^b$ for $a \in \mathbb{N}$ and $b > 1$), output COMPOSITE.
2. Find the smallest r such that $O_r(n) > \log^2 n$
3. If $\gcd(a, n) \neq 1$ for all $a \leq r$, output COMPOSITE.
4. For a = 1 to $\lfloor \sqrt{r} \log n \rfloor$ do
   if $((x + a)^n \equiv x^n + a \pmod{f(x), n})$, output PRIME.

This revised version includes several changes from the original algorithm. Initially we must address the function $f(x)$. The function is actually the same as before, x^r-1, but as a result of the changes in *r* its definition must be restated. The function $f(x)$ is a monic polynomial of degree *r* with integer coefficients such that the ring $\mathbb{Z}[x]/(n, f(x))$ becomes a pseudofield. A simple method for calculating the value of *r* is given as follows:

- $q > \text{floor}(\log^2 n)$
- Compute $n^j \bmod q$ for $j = 1, 2, \ldots, [(\log^2 n)]$
- If residue equals 1 mod q than q++
- Else r = q

This portion of the algorithm has a runtime order of $\tilde{O}(r(\log^2 n))$ [5].

The improvements gained with this modified algorithm are considerable. The value of *r* is reduced by approximately a factor of four. This is significant because it greatly reduces the number of iterations required in the final loop of the algorithm.

The difference between the square root of r and the square root of $\varphi(r)$ is not a large amount, primarily because *r* was required to be prime in the original algorithm. As a result, $\varphi(r)$ was simply r-1. However, in addition to the decreased value of *r*, the upper bound of the loop has been further reduced by a factor of 2. Proofs supporting this simplification may be found in [4]. This modified algorithm operates with $\tilde{O}(\log^{7.5} n)$ complexity.

Additionally, much of the literature available regarding the AKS algorithm discusses a conjecture that can result in dramatic improvements in the speed of the algorithm. [3],[6] This conjecture states:

If *r* is a prime number that does not divide *n* and if $n \neq 0 \bmod r$, then either *n* is prime or $n^2 = 1 \bmod r$. This conjecture allows us to significantly reduce the time spent in the final loop of the algorithm and instead test only a single value of *a*, provided *r* is carefully chosen. This conjecture reduces the overall time complexity of the algorithm to $\tilde{O}(\log^3 n)$.

IV. Implementation

A. Approach

Our implementation of the AKS algorithm was performed using the LiDIA [7] library. This decision was made based on previous work on the algorithm and faculty recommendation. LiDIA is a C++ library that was developed specifically for

computational number theory. The library contains a large number of highly optimized implementations of different functions.

Testing was conducted on the linux platform using kernel version 2.6.8 on a Pentium 4-M processor running at 1.8 GHz. The C++ compiler was GNU g++ version 3.3.5. The kernel, system libraries, and AKS implementation were all compiled with i686 optimizations enabled. Version 2.1.3 of LiDIA was used. NTL version 5.2 was used. Both LiDIA and NTL were compiled with support for GMP large integers.

We based our implementation on the Lenstra & Pomerance variation of the algorithm. In our initial implementation, the LiDIA-provided functions performed acceptably. Of the first three steps, only the GCD calculation required by step 3 of the algorithm incurred a noticeable performance penalty. This will be discussed below.

Steps one and two of the algorithm were remarkably simple to implement. LiDIA provides an `is_power()` function to determine whether a given integer value is a perfect power. Step two was easily implemented using a for loop, shown in **Fig. 1**.

B. Profiling

Throughout the development of the AKS implementation, we used execution profiling to identify areas of poor performance. In most cases, the profiling data merely confirmed our expectations, but the profile data did expose a significant performance penalty in Greatest Common Denominator calculation. The use of profiling also exposed a performance issue with the LiDIA `div_rem()` function used in our first attempt at polynomial reduction.

Profiling was conducted using the native profiling support of the g++ compiler. Executables compiled with profiling enabled produced a trace file, `gmon.out`, each time they were executed. One limitation of this approach is that the trace file was not created if the program exited unexpectedly due to a crash or when interrupted from the keyboard by pressing Ctrl-C. At first, this limitation prevented us from obtaining profiling data for long test runs processing multiple candidate primes in sequence. Shortly after observing the problem, we added a signal handler to allow graceful interruption and shutdown of the test program. In our implementation, pressing Ctrl-C once signaled the program to exit after the next candidate prime completed testing. Sending a second interrupt signal caused the program to terminate immediately. In either case, however, a graceful shutdown was performed and both the profiling data and a log of the program's output were created.

```
// Step 2
logn = log((bigfloat)n);
logn2 = power(logn, 2);
ceil(q, logn2);

// Choose value of r
while(1) {
   found_r= true;
   for (j = 1; j <= logn2; j++) {
      power_mod(res, n, j, q, 0);
      if (res == 1) {
         found_r = false;
         break;
      }
   }
   if (found_r) {
      r = q;
      break;
   }
   q++;
}
```

Fig. 1: Selection of r

C. GCD Calculation

Our initial implementation had favored functionality over optimization. This first implementation used the `dgcd`() function call provided by the LiDIA library. LiDIA provides six different functions for GCD calculation. Of these, three provide the basic result, while the other three provide extended information. The basic `gcd()` function uses the traditional method for determining the GCD. The `dgcd()` function uses the method based on Euclidian division . Finally, the `bgcd()` function uses a binary method [8]. We originally chose the Euclidian variant provided by `dgcd()` based on our familiarity with that algorithm. Data from the profiler indicated that GCD calculation using accounted for approximately 30% of the program's runtime using a test vector containing ten 32-bit numbers, four of which were prime and six of which were composite. This was unexpected and clearly unacceptable. Replacement of the call to `dgcd()` with a call to `bgcd()` resulted in considerable improvement. The cost incurred by GCD calculation dropped from 30% to consume only a negligible amount of the total processing time. This is not surprising, as further investigation showed that the binary method for GCD calculation was based on a sequence of shifts and adds, rather than the more CPU-intensive division operation [9].

D. Polynomial Exponentiation

The single most time-consuming operation of the AKS algorithm is the exponentiation of polynomials. This operation is needed in order to calculate the left side of the central equation in the final step of the algorithm.

Our first implementations used LiDIA's native functions for polynomial exponentiation and resulted in dismal performance. This is easily explained if the native function performs all exponentiation before reducing coefficients and/or exponents. This would require multiplication of polynomials of increasing degree up to degree *n* with each number tested.

We implemented our own `ReducedPower()` function to

```
INLINE void ReduceByDivision(poly &p,
const bigint &r) {
  // Reduce one polynomial by
  // dividing by another fixed
  // polynomial ( x^r - 1 )

  poly in = p;
  poly akspoly;
  long rl;

  long x;
  for (x = p.degree(); x >= rl; x--) {
    p[x - rl] = p[x - rl] + p[x];
    p[x] = 0;
  }
  p.remove_leading_zeros();
}
```

Fig. 2: Algorithm for Optimized Polynomial Reduction

solve this problem. Our implementation performs left-to-right expansion in traditional fashion. At each bit, the coefficients of the resulting polynomial were reduced mod n. Additionally, the result was reduced modulo another polynomial (x^r - 1).

The implementation of `ReducedPower()` did provide an improvement in speed, but less of one than originally hoped. Profile data indicated that the polynomial division function `div_rem()` caused significant overhead. The profiler indicated an astounding 73.62% of processing time spent in the div_rem function when operating on a test vector of prime numbers of increasing bitlengths up to 27 bits.

Based on a recommendation by Dr. Patrick Baier [10], we were able to improve this considerably. Instead of implementing the polynomial reduction as a division and taking the remainder, we used the following method.

When a polynomial is squared during exponentiation and the degree exceeds that of *r* it must be reduced. The reduction is performed by finding the residue after dividing the polynomial by x^r-1. A method exists to reduce the squared polynomial by x^r-1 with no division and only additions [10].

A polynomial in *x* of degree *r-1* may be represented as shown:

$$c_{2r-2}\, x^{2r-2} + \ldots + c_r\, x^r + c_{r-1}\, x^{r-1} + c_{r-2}\, x^{r-2} + \ldots + c_1\, x + c_0$$

This polynomial may then be grouped into blocks of terms with coefficients above and below the degree of the original polynomial. Assuming we wish to reduce mod x^r-1, we then obtain:

$$(c_{2r-2}\, x^{2r-2} + \ldots + c_r\, x^r) + c_{r-1}\, x^{r-1} + (c_{r-2}\, x^{r-2} + \ldots + c_1\, x + c_0)$$

This leaves us with a block above degree *r-1* and a block with degree below *r-1*. If the exponents are to be reduced mod *r*, we may simply shift each exponent down to its current degree minus *r* to obtain:

$$c_{r-1}\, x^{r-1} + (c_{r-2}\, x^{r-2} + c_{r-3}\, x^{r-3} + \ldots + c_1\, x + c_0) + (c_{2r-2}\, x^{2r-2} + c_{2r-1}\, x^{2r-1} + \ldots + c_r\, x^r)$$

This yields as a final result:

$$c_{r-1}\, x^{r-1} + (c_{r-3} + c_{2r-1})\, x^{r-3} + \ldots + (c_1 + c_r)) + c_0$$

The reduction method described above dramatically reduced the overall time of the algorithm. Additional profiling feedback indicated that over 90% of the runtime was now spent on polynomial multiplication, which corresponds to our initial performance expectations.

E. Polynomial Multiplication and Squaring

With exponentiation now implemented as a series of repeated squarings and multiplications, the next target of optimization was basic multiplication of polynomials. At this stage, our implementation was based on the LiDIA native functions for polynomial multiplication and squaring. Based on the profiler output, the squaring appeared to be implemented as a pass-through call to multiplication. Our first target for optimization was the squaring operation.

When multiplying polynomials, each pair of coefficients must be multiplied together. In the special case of squaring, these coefficient pairs will be repeated. Thus an optimized version of squaring should be able to reduce the number of integer multiplications by half. We implemented the squaring algorithm as shown in **Fig 3**. Unfortunately, we observed this implementation to actually decrease performance. When running with a test vector consisting of prime numbers of increasing bitlengths from 2 to 21 bits, the profiler showed that over 96% of the total execution time was spent in the Square() function.

Our initial theory to explain this behaviour was that it was caused by overhead from the LiDIA library. One possible source of such overhead was array access. While the code fragment shown in **Fig. 3** appears to be accessing the members of an array, the brackets shown are in fact overloaded operators calling LiDIA functions. We also considered the possibility that the frequent changes in the

```
  // Set the first and last terms
  result[0] = p[0] * p[0];
  if (k > 0) result[k] = p[n] * p[n];

  // Set intermediate terms
  for (i = k - 1; i > 0; i--) {
    ir = i / 2.0;
    a = 0;
    for (j = 0; j <= ir; j++) {
      if (i-j > n) continue;
      product = p[j] * p[i-j];

      a = a + product;
      if (j != i - j) a = a + product;
    }
    result[i] = a;
  }
```

Fig. 3: Algorithm for Polynomial Squaring

magnitude of coefficients for each exponent could be resulting in memory allocation, which could contribute significantly to overall time. Unfortunately, profiling data was of only minimal benefit in this case.

Our first attempt to prove or disprove these theories was to try operating against an array of `bigint` values rather than against a polynomial object. The array was declared statically in an attempt to help avoid memory reallocation each time the squaring function was called. Unfortunately, this implementation did not provide the improvement in speed that we desired. A partially optimized array implementation performed with comparable speed to the polynomial version. It is important to note, however, that memory reallocation could still be a concern with the array representation. While the coefficients themselves were allocated in advance, each coefficient was represented using the LiDIA `bigint` data type, which is itself capable of memory reallocation.

We then created a separate implementation of portions of the AKS algorithm using the GMP large integer library directly rather than using LiDIA. This was done in order to verify that LiDIA itself was not the bottleneck. Although the performance of the initial GMP trials was disappointing, it did provide the needed verification. We observed similar profiling results and similar performance problems to the LiDIA implementation. This indicated that the delay is attributable to our squaring algorithm itself.

F. NTL Implementation

After further research, we discovered that the NTL library already contained much more efficient polynomial multiplication and squaring libraries than those found in LiDIA. NTL automatically selects between classical multiplication, the Karatsuba method, and two different variations of Fast Fourier Transform (FFT). [11]. Our implementation represents integer coefficients mod *n* using the `ZZ_p` data type, and polynomials using the `ZZ_pEX` data type. This approach reduces coefficients automatically, but requires manual reduction of exponents.

LiDIA does contain several functions that are not present in NTL, however. In particular, NTL does not contain a function to determine whether or not an integer is a perfect power. Fortunately, GMP does contain such a function, and our version of NTL was compiled with GMP as the underlying multiprecision integer library. Our solution was therefore to convert our value of *n* from NTL's `ZZ` data type to the `mpz_t` type used by GMP, at which point we were able to use the native GMP function `mpz_perfect_power_p()`. The NTL representation was converted to a character (text) string, which was then converted back to GMP's representation. A better implementation might be to port the algorithm used by GMP to perform this function to native NTL. Since GMP and NTL are both released under the GNU Public License, there are no barriers to prevent such an implementation.

The NTL implementation provided a drastic improvement in performance, as will be shown in **Fig. 7**. Furthermore, NTL did appear to be making use of multiple techniques for the multiplication and squaring functions. Operating on a list of candidate primes of bitlengths up to 61 bits, the profiler reported 52% of the execution time being spent in the `FFT()` function. Smaller amounts of time (typically less than 1%) were spent in functions corresponding to traditional and Karatsuba multiplication and squaring. Additional performance for short bitlengths might be gained by adjusting the cutoff values at which NTL switches between algorithms.

V. Summary of Results

Initially our results progressed much as expected. Our first implementation, while slow, was able to correctly determine primality for each number tested. Test vectors included a list of the first 10,000 prime numbers, the first 10,000 integers, and several Carmichael numbers.

As we continued to develop the program, we were able to identify and remove several bottlenecks to performance, including gcd calculation and the removal of polynomial division during reduction. This performance in improvement is shown in **Fig. 4**. **Fig 5**. shows additional tests performed with the same optimized implementation with longer bitlengths.

Polynomial multiplication and squaring in particular proved to be a continuing problem, however. Our current implementation continues to show slow performance in spite

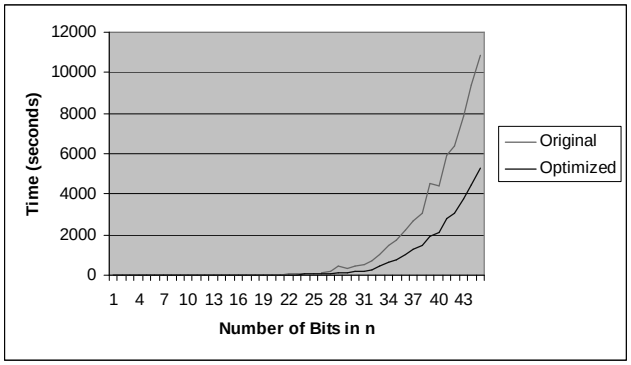

Fig 4: Original vs. Optimized LiDIA Implementations

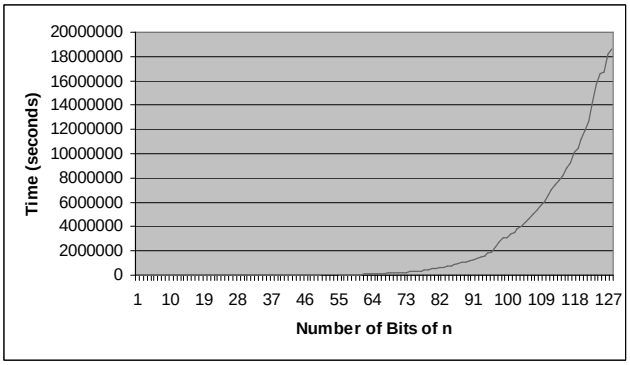

Fig 5: Optimized LiDIA Test Results Through 128 Bits

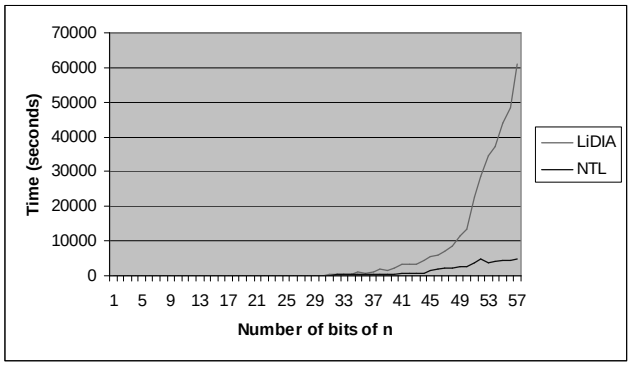

Fig. 7: NTL vs. LiDIA Implementations

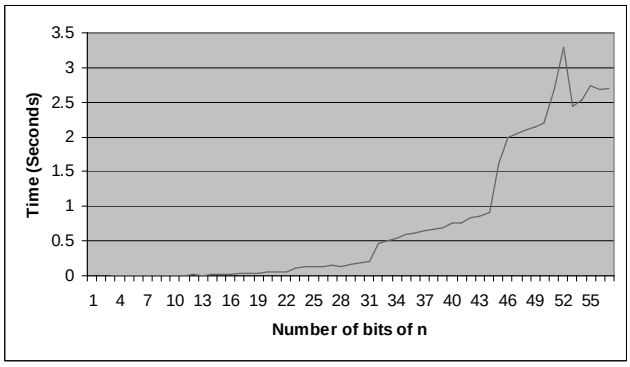

Fig 8: Extrapolated NTL Times With Conjecture

of ongoing optimizations. Profiling data continues to show poor performance in polynomial squaring, and it is this area which is most in need of continued work.

After this phase of testing, we were able to achieve additional performance improvements by moving from LiDIA to NTL as our choice of library. We achieved our best results to date by re-implementing the algorithm using NTL; these results are shown in **Fig. 7**.

Based on the conjecture discussed in Section III, we extrapolated the average time required for testing each value of *a* using the NTL implementation. This provides an approximate indication of the time required to prove primality if only one value is tested. This extrapolation is shown in **Fig. 8**. Since this conjecture has not been mathematically proven, the algorithm ceases to be truly deterministic in this configuration. However, the significant decrease in testing may be helpful when comparing the relative performance of different versions of the implementation.

It is worth noting all tests that were conducted using the full implementation of AKS were shown to be prime if they passed testing with any value of a. While this is far from formal proof, it does support the conjecture, and provides additional reassurance when using this abbreviated version of the algorithm for performance comparisons.

VI. Performance Comparisons

Approximate performance comparisons may be drawn between our implementation and those of others. In particular, we observed general comparison between our own implementations and four other alternatives.

First, we can compare our results to previous GMU student implementations. In the fall semester of 2004, two student groups completed AKS implementations. The first of these achieved bitlengths of up to only 13-20 bits in practical trials [12]. Our implementation was able to determine primality for bitlengths on the order of 64 bits in a few hours. The second of these implementations was performed in Java rather than C++, which renders direct comparisons somewhat meaningless; however, this Java implementation showed running times of over two minutes for a 10-bit number [13]. Our final NTL implementation processed a 44 to 45 bit number in this amount of time; bitlengths of up to approximately 12 bits could be completed in under a second.

A recent implementation constructed at Carnegie Mellon University provided results that appear to be comparable to ours.[14]. It is difficult to obtain a detailed comparison because of differences in the testing methodology. The CMU implemented used both long integer and GMP large integer representations for testing. The long integer version appears to outperform our implementation. This is unsurprising given the overhead required by multiprecision integer representation. The corresponding GMP version appears slower than our implementations Additionally, the CMU tests were performed on somewhat faster hardware than ours.

The fastest implementation that we have found was performed by Crandall and Papadopoulos, working for Apple Computer and the University of Maryland at College Park [15]. **Fig. 9** shows the comparison between their implementation and ours. The results shown for the Apple implementation were obtained on a 1.42GHz Mac Mini running Mac OS version 10.3.9. Although the running times are very similar, the clock speed of the Macintosh was slower than that of the PC used for NTL testing. This indicates that the Apple/UMD implementation is in fact faster than ours. Since this implementation is optimized for the Apple/PowerPC architecture, testing on our Intel reference machine would also have led to some inaccuracy.

VII. Future Work

Several possibilities for future work on the present NTL implementation are available. Some minor gains might be

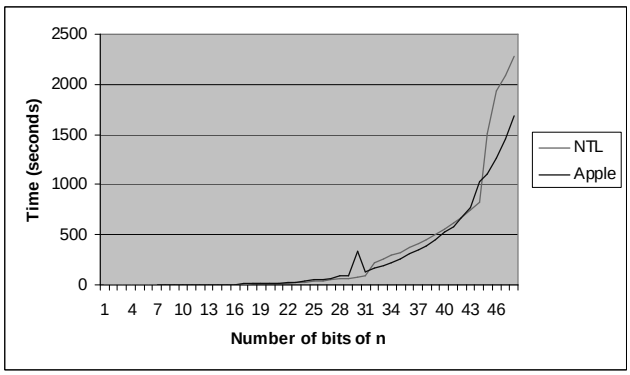

Fig 9: GMU NTL vs. Apple/UMD Implementations

realized through different choices in GCD calculation. Another area for improvement is in the current implementation of Step 1 of the algorithm. This step currently requires conversion of the representation of n from NTL to GMP data types. Porting the GMP `mpz_perfect_power_p()` function to native NTL might offer slight performance gains. In both of these cases, however, the functions are called only once per test cycle.

Additional performance improvements in the NTL version might be obtained using different data types to represent polynomials. NTL allows a choice of whether to perform modular reduction of coefficients and exponents manually or automatically through the use of data types. NTL offers several different representations both for large integers and for polynomials, and a different representation may results in increased performance.

There are also many improvements to the AKS algorithm available in paper form, Lenstra and Pomerance [4] and Bernstein [16] are the two that seem to provide the largest amount of improvement in performance.

When referencing Bernstein's improvements the one we found most intriguing and provided the largest potential for time improvement was his exponentiation idea. Bernstein proposed the idea of translating polynomials into integers and then performing exponentiation functions in the new integer ring. All operations are currently in the ring $(\mathbb{Z}/n)[x]/(x^r-1)$. Bernstein's idea is to compute a value k = ceiling($\log rn^2$) and use this to transition to $\mathbb{Z}[x]/x^r-1$, obtaining a polynomial with coefficients (0, n-1). The polynomial is then mapped to $\mathbb{Z}/(2^{kr}-1)$ by the relation $x \rightarrow 2^k$. All squaring operations take place in the integer ring. The polynomial form may then be recovered by applying the inverse relation. The performance improvement by this method could be substantial, because it would allow the use of existing functions for integer squaring and multiplication. Such functions are more commonly used than their polynomial counterparts, and are thus likely to be more highly optimized. Bernstein has also proposed a number of other improvements, but this may be the most promising.

There is another proposed improvement to the algorithm that appears to offer a very large decrease in execution time. Qi Cheng built on the work by P. Berrizbeitia [17] to develop a method that would require one round of Elliptical Curve Primality Proving (ECPP) and a single iteration of the AKS algorithm [18]. This is a large variant from AKS, but can significantly improve execution time. The construction of one round of ECPP test will be the largest challenge, because this algorithm is much more difficult to understand than AKS.

One last optimization is Bernstein's generalization [19] of Berrizbeitia's improvements to AKS. The idea behind this approach is based on essentially a totally restructuring of the AKS algorithm. The resulting algorithm produces an answer much more quickly; however, it is not guaranteed to always provide an answer. When it does provide such an answer, it is always correct, as is the current form of AKS. Unfortunately, the number of iterations required to determine primality using this method will not be known beforehand.

VIII. Conclusion

The AKS algorithm in its current form is not suitable for general use due to the length of time required to verify primality. Although our implementations were able to achieve results comparable to those of others, this is not sufficient for general use. Currently available probabilistic algorithms provide much faster response times within margins of error that are generally acceptable. Primality tests for numbers that might take hours or days using AKS could be performed in seconds or even less using probabilistic methods such as Miller-Rabin. If the conjecture described in Section III can be proven, however, the practical application of AKS begins to become more realistic. Similarly, the Qi Cheng method may offer hope for faster practical implementations. Continuing improvements to AKS are thus best focused in improvements to the algorithm itself.

In our implementation we were able to overcome several performance bottlenecks, particularly those related to choice of specific functions such as GCD calculation and in multiplication and reduction. Much of this initial effort was caused by the initial focus on use of the LiDIA library, however. After moving to an NTL-based implementation, performance improved dramatically.

Additional implementation improvements may be directed in two primary areas. First, continuing implementations using NTL or similar libraries that offer efficient polynomial operations may allow improved speed. In particular, adjustment of the boundaries between different multiplication methods chosen by NTL may result in improved performance.

An alternate approach to optimization minimizes the use of such libraries and focuses on customized methods. These implementations may use special-purpose libraries as in the Apple/UMD implementation. Additionally, representation of the polynomial in alternate forms may provide significant speed improvements. This includes mapping the existing polynomial ring onto an integer ring and other such methods.

References

[1] E. Weisstein, "Primality Test", 1999, CRC Press LLC, Available from the World Wide Web: <http://mathworld.wolfram.com/PrimalityTest.html>

[2] M. Agrawal, N. Kayal, and N. Saxena, "Primes in P," Department of Computer Science & Engineering, Indian Institute of Technology Kanpur. Available from the World Wide Web: <http://www.cse.iitk.ac.in/news/primality.pdf>

[3] M. Agrawal, N. Kayal, and N. Saxena, "Primes in P," (Revised) Department of Computer Science & Engineering, Indian Institute of Technology Kanpur. Available from the World Wide Web: <http://www.cse.iitk.ac.in/news/primality_v3.pdf>

[4] H. Lenstra, C. Pomerance, "Primality Testing with Gaussian Periods." Manuscript 2003.

[5] A. Granville, “It Is Easy To Determine Whether a Given Integer Is Prime,” Bulletin (New Series) of the American Mathematical Society Vol. 42 No 1, pp 3-38.
[6] R. Bhattacharjee and P. Pandey. “Primality Testing”. Technical Report, IIT Kanpur, 2001. Available from the World Wide Web: http://www.cse.iitk.ac.in/research/btp2001/primality.html
[7] LiDIA - A Library for Computational Number Theory. 1994 - 2004, The LiDIA Group. Available from the World Wide Web: http://www.informatik.tu-darmstadt.de/TI/LiDIA/
[8] S. Hamdy, “LiDIA Reference Manual”. 2004. The LiDIA Group. Available from the World Wide Web: ftp://ftp.informatik.tu-darmstadt.de/pub/TI/systems/LiDIA/current/LiDIA.pdf
[9] A. Menezes, P. OOshot, S. Vanstone. *Handbook of Applied Cryptography*. CRC Press 1996. pp. 606-608.
[10] P. Baier, National Organization for Research at The University of Chicago
[11] V.Shoup: “A Tour of NTL:: NTL Implementation and Portability”. Available from the World Wide Web: <http://www.shoup.net/ntl/doc/tour-impl.html>
[12] C. Lanka, J. Nightingale, D. Patel, H. Vasudevan, “Generating large prime numbers for cryptographic applications”, Fall 2004, George Mason University, Fairfax, VA.
[13] K. Shah, “Generating large primes using AKS”, Fall 2004, George Mason University, Fairfax, VA.
[14] C. Rotella, “An Efficient Implementation of the AKS Polynomial-Time Primality Proving Algorithm”, Carnegie Mellon University, May 2005, Pittsburgh, PA.
[15] R. Crandall and J. Papadopoulos, “On the implementation of AKS-class primality tests”, March 18, 2003
[16] D. Bernstein, “Proving Primality After Agrawal-Kayal-Saxena.” Department of Mathematics, Statistics, and Computer Science, University of Illinois. Available from the World Wide Web: <http://cr.yp.to/papers/aks.pdf>
[17] P. Berrizbeitia, “Sharpening Primes Is In P For a Large Family of Numbers.” 2003. Available from the World Wide Web: <http://lanl.arxiv.org/PS_cache/math/pdf/0211/0211334.pdf>
[18] Q. Cheng, “Primality Proving Via One Round In ECPP and One Iteration In AKS.” Available from the World Wide Web: <http://www.cs.ou.edu/%7Eqcheng/paper/aksimp.pdf>
[19] D. Bernstein, “Proving Primality in Essentially Quartic Random Time.” Department of Mathematics, Statistics, and Computer Science, University of Illinois. Available from the World Wide Web: <http://cr.yp.to/primetests/quartic-20041203.pdf>