

ARCHITECTURE-AWARE CLASSICAL TAYLOR SHIFT BY 1

Jeremy R. Johnson
Werner Krandick
and
Anatole D. Ruslanov

Technical Report DU-CS-05-06
Department of Computer Science
Drexel University
Philadelphia, PA 19104
May, 2005

ARCHITECTURE-AWARE CLASSICAL TAYLOR SHIFT BY 1

JEREMY R. JOHNSON, WERNER KRANDICK, ANATOLE D. RUSLANOV
DEPARTMENT OF COMPUTER SCIENCE
DREXEL UNIVERSITY, PHILADELPHIA, PA 19104
{JJOHNSON, KRANDICK, ANATOLE}@CS.DREXEL.EDU

ABSTRACT. We present algorithms that outperform straightforward implementations of classical Taylor shift by 1. For input polynomials of low degrees a method of the SACLIB library is faster than straightforward implementations by a factor of at least 2; for higher degrees we develop a method that is faster than straightforward implementations by a factor of up to 7. Our Taylor shift algorithm requires more word additions than straightforward implementations but it reduces the number of cycles per word addition by reducing memory traffic and the number of carry computations. The introduction of signed digits, suspended normalization, radix reduction, and delayed carry propagation enables our algorithm to take advantage of the technique of register tiling which is commonly used by optimizing compilers. While our algorithm is written in a high-level language, it depends on several parameters that can be tuned to the underlying architecture.

1. INTRODUCTION

Let A be a univariate polynomial with integer coefficients. Taylor shift by 1 is the operation that computes the coefficients of the polynomial $B(x) = A(x + 1)$ from the coefficients of the polynomial $A(x)$. Taylor shift by 1 is the most time-consuming subalgorithm of the monomial Descartes method [7] for polynomial real root isolation. Taylor shift by 1 can also be used to shift a polynomial by an arbitrary integer a . Indeed, if $B(x) = A(ax)$ and $C(x) = B(x + 1)$ and $D(x) = C(x/a)$, then $D(x) = A(x + a)$. According to Borowczyk [3], Budan proved this fact in 1811.

More recently, von zur Gathen and Gerhard [22, 9] compared six different methods to perform Taylor shifts. The authors distinguish between classical methods and asymptotically fast methods. When the shift amount is 1, the classical methods collapse into a single method which computes $n(n+1)/2$ integer sums where n is the degree of the input polynomial. In fact, von zur Gathen's and Gerhard's implementation of classical Taylor shift by 1 simply makes calls to an

integer addition routine. We will refer to such implementations as straightforward implementations.

We present algorithms that outperform straightforward implementations of classical Taylor shift by 1. For input polynomials of low degrees a method of the SACLIB library [8] is faster than straightforward implementations by a factor of at least 2 on our experimental platform (UltraSPARC III); for higher degrees we develop a method that is faster than straightforward implementations by a factor of up to 7 (Figure 8).

Our Taylor shift algorithm requires more word additions than straightforward implementations but it reduces the number of cycles per word addition (Figure 11) by reducing memory traffic (Figure 12) and the number of carry computations. The introduction of signed digits, suspended normalization, radix reduction, and delayed carry propagation enables our algorithm to take advantage of the technique of register tiling which is commonly used by optimizing compilers [2, 14]. While our algorithm is written in a high-level language, it depends on several parameters that can be tuned to the underlying architecture.

It is widely believed that computer algebra systems can obtain high performance by building on top of basic arithmetic routines that exploit features of the hardware. It is also believed that only assembly language programs can exploit features of the hardware. This paper suggests that both tenets are wrong.

In Section 2 we introduce some notation for the classical Taylor shift by 1. We also characterize two classes of test inputs for the algorithm. In Section 3 we define our notion of straightforward implementations; we single out the GNU-MP-based implementation as our point of reference. In Section 4 we describe the specialized algorithm used in SACLIB. In Section 5 we describe our new algorithm; Section 6 documents its performance.

2. NOTATION, CORRECTNESS, AND CLASSES OF TEST INPUTS

We will call a method that computes Taylor shift by 1 *classical* if the method uses only additions and computes the intermediate results given in Definition 1.

Definition 1. For any non-negative integer n let $I_n = \{(i, j) \mid i, j \geq 0 \wedge i+j \leq n\}$. If n is a non-negative integer and

$$A(x) = a_n x^n + \dots + a_1 x + a_0,$$

is an integer polynomial we let, for $k \in \{0, \dots, n\}$ and $(i, j) \in I_n$,

$$\begin{aligned} a_{-1,k} &= 0, \\ a_{k,-1} &= a_{n-k}, \\ a_{i,j} &= a_{i,j-1} + a_{i-1,j}, \end{aligned}$$

as shown in Figure 1.

$$\begin{array}{cccc}
 & & 0 & 0 & 0 & 0 \\
 & & \downarrow & \downarrow & \downarrow & \downarrow \\
 a_n & \rightarrow & a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\
 a_{n-1} & \rightarrow & a_{1,0} & a_{1,1} & a_{1,2} & \\
 a_{n-2} & \rightarrow & a_{2,0} & a_{2,1} & & \\
 a_{n-3} & \rightarrow & a_{3,0} & & &
 \end{array}$$

FIGURE 1. By Theorem 1, the pattern of integer additions in Pascal's triangle, $a_{i,j} = a_{i,j-1} + a_{i-1,j}$, can be used to perform Taylor shift by 1.

Theorem 1. *Let n be a non-negative integer, and let $A(x) = a_n x^n + \dots + a_1 x + a_0$ be an integer polynomial. Then, in the notation of Definition 1,*

$$A(x+1) = \sum_{h=0}^n a_{n-h,h} x^h.$$

Proof. The assertion clearly holds for $n = 0$; so we may assume $n > 0$. For every $k \in \{0, \dots, n\}$ let $A_k(x) = \sum_{h=0}^k a_{k-h,h} x^h$. Figure 2 shows that the coefficients of the polynomial A_k reside on the k -th diagonal of the matrix of Figure 1. Then, for all $k \in \{0, \dots, n-1\}$, we have $A_{k+1}(x) = (x+1)A_k(x) + a_{n-(k+1)}$. Now an easy induction on k shows that $A_k(x) = \sum_{h=0}^k a_{n-k+h}(x+1)^h$ for all $k \in \{0, \dots, n\}$. In particular, $A_n(x) = \sum_{h=0}^n a_h(x+1)^h = A(x+1)$. \square

Definition 2. Let a be an integer. The *binary-length* of a is defined as

$$L(a) = \begin{cases} 1 & \text{if } a = 0, \\ \lfloor \log_2 |a| \rfloor + 1 & \text{otherwise.} \end{cases}$$

Definition 3. The *max-norm* of an integer polynomial $A = a_n x^n + \dots + a_1 x + a_0$ is $|A|_\infty = \max(|a_n|, \dots, |a_0|)$.

The algorithm in Section 4 requires a bound on the binary lengths of the intermediate results $a_{i,j}$.

Theorem 2. *Let n be a non-negative integer, and let $A(x) = a_n x^n + \dots + a_1 x + a_0$ be an integer polynomial of max-norm d . Then, for all $(i, j) \in I_n$,*

$$(1) \ a_{i,j} = \binom{i+j}{j} a_n + \binom{i+j-1}{j} a_{n-1} + \dots + \binom{j}{j} a_{n-i}, \text{ and}$$

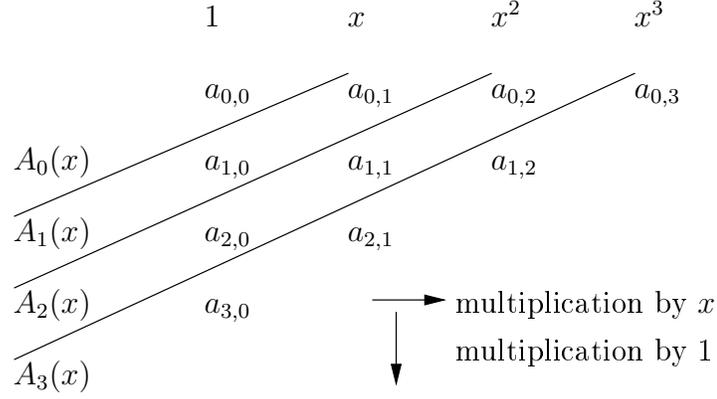


FIGURE 2. The coefficients of the polynomial $A_k(x)$ in the proof of Theorem 1 reside on the k -th diagonal. Multiplication of $A_k(x)$ by $(x + 1)$ can be interpreted as an addition that follows a shift to the right and a downward shift.

(2) $L(a_{i,j}) \leq L(d) + i + j$.

Proof. Assertion (1) follows from Definition 1 by induction on $i + j$. Due to assertion (1),

$$\begin{aligned}
 |a_{i,j}| &\leq \left(\binom{i+j}{j} + \binom{i+j-1}{j} + \dots + \binom{j}{j} \right) d \\
 &= \binom{i+j+1}{j+1} d \\
 &\leq 2^{i+j} d
 \end{aligned}$$

which proves assertion (2). □

Remark 1. Theorem 2 implies that, for degree n and max-norm d , the binary length of all intermediate results is at most $L(d) + n$. The algorithm in Section 4 can be slightly improved for small-degree polynomials by tightening that bound for $n \in \{8, \dots, 39\}$ to $L(d) + n - 1$, for $n \in \{40, \dots, 161\}$ to $L(d) + n - 2$, and for $n \in \{162, \dots, 649\}$ to $L(d) + n - 3$.

We will use Theorem 3 to prove lower bounds for the computing time of two classes of input polynomials.

Theorem 3. *Let n be a non-negative integer. Then at least $n/2$ of the binomial coefficients $\binom{n}{k}$, $0 \leq k \leq n$, have binary length $\geq n/2$.*

Proof. The assertion is clearly true for all $n \in \{0, \dots, 19\}$, so we may assume $n \geq 20$. We then have

$$n - \lfloor n/4 \rfloor + 1 \geq 4^2.$$

Also, for $0 < i < \lfloor n/4 \rfloor$,

$$\frac{n-i}{\lfloor n/4 \rfloor - i} > \frac{n}{\lfloor n/4 \rfloor} \geq \frac{n}{n/4} = 4,$$

so that

$$\begin{aligned} \binom{n}{\lfloor n/4 \rfloor} &= \frac{n}{\lfloor n/4 \rfloor} \cdot \frac{n-1}{\lfloor n/4 \rfloor - 1} \cdots \frac{n - \lfloor n/4 \rfloor + 1}{1} \\ &> 4^{\lfloor n/4 \rfloor + 1} = 2^{2\lfloor n/4 \rfloor + 2} > 2^{n/2}. \end{aligned}$$

Hence, the binary length of each binomial coefficient

$$\binom{n}{\lfloor n/4 \rfloor}, \binom{n}{\lfloor n/4 \rfloor + 1}, \dots, \binom{n}{n - \lfloor n/4 \rfloor}$$

is $> n/2$. But the number of those coefficients is $> n/2$. \square

Definition 4. For any positive integers n, d we define the polynomials

$$\begin{aligned} B_{n,d}(x) &= dx^n + dx^{n-1} + \cdots + dx + d, \\ C_{n,d}(x) &= x^n + d. \end{aligned}$$

Theorem 5 and the proof of Theorem 4 characterize the computing time functions of classical Taylor shift on the sets of polynomials $B_{n,d}$ and $C_{n,d}$. We use the concept of dominance defined by Collins [6] since it hides fewer constants than the more widely used big-Oh notation; Collins also defines the maximum computing time function.

Theorem 4. *Let $t^+(n, d)$ be the maximum computing time function for classical Taylor shift by 1 where $n \geq 1$ is the degree and d is the max-norm. Then $t^+(n, d)$ is co-dominant with $n^3 + n^2L(d)$.*

Proof. The recursion formula in Definition 1 is invoked $|I_n| = n(n+1)/2$ times. Hence the number of integer additions is dominated by n^2 . By Theorem 2, the binary length of any summand is at most $L(d) + n$. Thus the computing time is dominated by $n^2 \cdot (L(d) + n)$.

We now show that, for the input polynomials $B_{n,d}$, the computing time dominates $n^3 + n^2L(d)$. Since, for any fixed $n \geq 1$, the computing time clearly dominates $L(d)$ we may assume $n \geq 2$. By Theorem 2 (1),

$$a_{i,j} = \binom{i+j+1}{j+1} d$$

for all $(i, j) \in I_n$. For $k = i + j \geq 2$, Theorem 3 yields that at least $(k + 1)/2$ of the binomial coefficients

$$\binom{k+1}{1}, \binom{k+1}{2}, \dots, \binom{k+1}{k+1}$$

have binary length $\geq (k + 1)/2$. So, for all $k \in \{2, \dots, n\}$ there are least $(k + 1)/2$ integers $a_{i,j}$ with $i + j = k$ and

$$L(a_{i,j}) \geq L(d) - 1 + \frac{k+1}{2}$$

Now the assertion follows by summing all the lengths. \square

Our proof of Theorem 5 assumes that the time to add two non-zero integers a, b is co-dominant with $L(a) + L(b)$; Collins [6] makes the same assumption in his analyses.

Theorem 5. *The computing time function of classical Taylor shift by 1 on the set of polynomials $C_{n,d}$ of Definition 4 is co-dominant with $n^3 + L(d)$.*

Proof. By Theorem 2, $a_{n,0} = d + 1$ and, for $(i, j) \in I_n - \{(n, 0)\}$,

$$a_{i,j} = \binom{i+j}{j}.$$

Hence, by Theorem 3, for any $k \in \{0, \dots, n\}$, at least half of the integers $a_{k,0}, a_{k-1,1}, \dots, a_{0,k}$ have binary length $\geq k/2$. Since all of them—except possibly $a_{n,0}$ —have binary length $\leq k$, we have that

$$-L(a_{n,0}) + \sum_{k=0}^n \sum_{j=0}^k L(a_{k-j,j}) \sim n^3.$$

But the time to compute $a_{n,0}$ is co-dominant with $L(d)$, and so the total computing time is co-dominant with $n^3 + L(d)$. \square

3. STRAIGHTFORWARD IMPLEMENTATIONS

We call an implementation of classical Taylor shift by 1 *straightforward* if it uses a generic integer addition routine to compute one of the following sequences of the intermediate results $a_{i,j}$.

- (1) Horner's scheme—descending order of output coefficients

$$(a_{0,0}, a_{0,1}, a_{1,0}, a_{0,2}, a_{1,1}, a_{2,0}, \dots, a_{0,n}, \dots, a_{n,0}).$$

- (2) Horner's scheme—ascending order of output coefficients

$$(a_{0,0}, a_{1,0}, a_{0,1}, a_{2,0}, a_{1,1}, a_{0,2}, \dots, a_{n,0}, \dots, a_{0,n}).$$

(3) Synthetic division—ascending order of output coefficients

$$(a_{0,0}, a_{1,0}, \dots, a_{n,0}, a_{0,1}, \dots, a_{n-1,1}, \dots, a_{0,n}).$$

(4) Descending order of output coefficients

$$(a_{0,0}, a_{0,1}, \dots, a_{0,n}, a_{1,0}, \dots, a_{1,n-1}, \dots, a_{n,0}).$$

Von zur Gathen and Gerhard use method (1) [10]. The computer algebra system Maple [15, 17, 16], version 9.01, uses method (3) in its function `PolynomialTools[Translate]`. In methods (3) and (4) the output coefficients appear earlier in the sequence than in the other methods. The computing times of the four methods are very similar; they differ typically by less than 10%. In our experiments we will use method (3) to represent the straightforward methods; Figure 3 gives pseudocode.

```

for  $i = 0, \dots, n$ 
   $b_i \leftarrow a_i$ 
  assertion:  $b_i = a_{n-i,-1}$ 
for  $j = 0, \dots, n - 1$ 
  for  $i = n - 1, \dots, j$ 
     $b_i \leftarrow b_i + b_{i+1}$ 
    assertion:  $b_i = a_{n-i,j}$ 
    
```

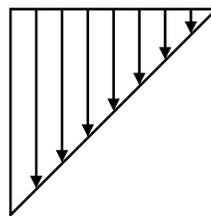


FIGURE 3. The straightforward method we consider uses integer additions to compute the elements of the matrix in Figure 1 from top to bottom and from left to right.

The efficiency of straightforward methods depends entirely on the efficiency of the underlying integer addition routine. Von zur Gathen and Gerhard use the integer addition routine of NTL [19, 18] in their experiments. But Tables 1 and 2 imply that the integer addition routine of GNU-MP [11] is faster. We use the GNU-MP routine in our implementation of a straightforward method.

The GNU-MP package represents integers in sign-length-magnitude representation. On the UltraSPARC III we have the package use the radix $\beta = 2^{64}$. Let n be a non-negative integer, and let $u = u_0 + u_1\beta + \dots + u_n\beta^n$, where $0 \leq u_i < \beta$ for all $i \in \{0, \dots, n\}$ and $u_n \neq 0$. The magnitude u is represented as an array \mathbf{u} of unsigned 64-bit integers such that $\mathbf{u}[i] = u_i$ for all $i \in \{0, \dots, n\}$. Let $v = v_0 + v_1\beta + \dots + v_n\beta^n$ be a magnitude of the same length. The routine `mpn_add_n` is designed to add u and v in $n + 1$ phases of 4 cycles each. Phase i computes the carry-in c_{i-1} and the result digit $r_i = (u_i + v_i + c_{i-1}) \bmod \beta$. Figure 4 gives a high-level description of the routine; all logical operators in the figure are bit-wise operators. In each set of four successive phases the operation `address`

computes new offset addresses for u_{i+1} , v_{i+1} , and r_{i+1} , respectively, during the first three phases; in the fourth phase, the operation `address` is replaced by a loop control operation. The routine consists of 178 lines of assembly code. In-place addition can be performed. Whenever the sum does not fit into the allocated result array, GNU-MP allocates a new array that is just large enough to hold the sum.

	cycle 1	cycle 2	cycle 3	cycle 4
IEU1	$a \leftarrow (u_{i-1} \vee v_{i-1}) \wedge \neg r_{i-1}$	$a \leftarrow a \vee b$	$c_{i-1} \leftarrow \lfloor a/2^{63} \rfloor$	$r_i \leftarrow b + c_{i-1} \bmod \beta$
IEU2	$b \leftarrow u_{i-1} \wedge v_{i-1}$	<code>address</code>	$b \leftarrow u_i + v_i \bmod \beta$	$u_i \vee v_i$
MEM	<code>load</code> u_{i+3}	<code>load</code> v_{i+3}	<code>store</code> r_{i-1}	—

FIGURE 4. The UltraSPARC III has two integer execution units (IEU1, IEU2) and one load/store unit (MEM). The GNU-MP addition routine adds each pair of 64-bit words in a phase that consists of 4 machine cycles. Digit additions are performed modulo $\beta = 2^{64}$; carries are reconstructed from the leading bits of the operands and the result.

The NTL library represents integers using a sign-length-magnitude representation similar to the one GNU-MP uses. But while GNU-MP allows the digits to have word-length, NTL-digits have 2 bits less than a word. As opposed to GNU-MP, NTL needs 1 bit of the word to absorb the carry when it adds two digits. This explains why NTL-digits are 1 bit shorter than GNU-MP-digits. Another bit is lost for the following reason. While GNU-MP represents an integer as a C-language `struct`, NTL represents it as an array and uses the first array element to represent the signed length of the integer. Since all array elements are of the same type, NTL-digits are signed as well—even though their sign is never used. Finally, due to its way of performing multiplications, NTL cannot take full advantage of a 64-bit word-length. In our experiments on the UltraSPARC III the NTL radix was 2^{30} . The NTL addition routine `ntl_zadd` consists of 113 lines of C++ code.

4. THE SACLIB METHOD

The SACLIB library of computer algebra programs [8] performs classical Taylor shift by 1 using the routine `IUPTR1`. The routine, consisting of 144 lines of C-code, is due to G. E. Collins and was originally written for the SAC-2 computer algebra system [5]. Figure 8 shows that the method is faster than straightforward methods for polynomials of small degrees.

SACLIB represents integers with respect to a radix β that is a positive power of 2; in our experiments, $\beta = 2^{62}$. Integers a such that $-\beta < a < \beta$ are called

β -digits and are represented as variables of type `int` or `long long`. Integers a such that $a \leq -\beta$ or $\beta \leq a$ are represented as lists (d_0, \dots, d_h) of β -digits with $a = \sum_{i=0}^h d_i \beta^i$ where $d_h \neq 0$ and, for $i \in \{0, \dots, h\}$, $d_i \leq 0$ if $a < 0$ and $d_i \geq 0$ if $a > 0$.

SACLIB adds integers of opposite signs by adding their digits. None of these digit additions produces a carry. The result is a list (d_0, \dots, d_k) of β -digits that may be 0 and that may have different signs. If not all digits are 0, the non-zero digit of highest order has the sign s of the result. The digits whose sign is different from s are adjusted in a step called *normalization*. The normalization step processes the digits in ascending order. Digits are adjusted by adding $s \cdot \beta$ and propagating the carry $-s$.

The routine IUPTR1 performs Taylor shift by 1 of a polynomial of degree n and max-norm d by performing the $n(n+1)/2$ coefficient additions without normalizing after each addition. A secondary idea is to eliminate the loop control for each coefficient addition. To do this the program first computes the bound $n+L(d)$ of Remark 1 for the binary length of the result coefficients. The program determines the number k of words required to store $n+L(d)$ bits. The program then copies the polynomial coefficients in ascending order, and in ascending order of digits, into an array that provides k words for each coefficient; the unneeded high-order words of each coefficient are filled with the value 0. This results in an array P of $k(n+1)$ entries such that, for $i \in \{0, \dots, k(n+1)-1\}$ and $i = qk+r$ with $0 \leq r < k$, $P[i+1] = a_{n-q}^{(r)}$ where $\sum_{j=0}^{k-1} a_{n-q}^{(j)} \beta^j$ is the coefficient of x^{n-q} in the input polynomial. After these preparations the Taylor shift can be executed using just the two nested loops of Figure 5. The principal disadvantage of the method is the cost of adding many zero words due to padding. This makes the method impractical for large inputs. Also, the carry computation generates branch mispredictions.

5. THE TILE METHOD

Our new method outperforms the existing Taylor shift methods by reducing the number of cycles per word addition. The GNU-MP addition routine of Figure 4 requires 4 cycles per word addition. To improve on this we reduce the number of carry computations by using a smaller radix and allowing carries to accumulate inside a computer word. Further, we reduce the number of read and write operations by performing more than one word addition once a set of digits has been loaded into registers. This requires changing the order of operations; only certain digits of the intermediate integer results $a_{i,j}$ in Definition 1 will be computed in one step. We perform only additions; signed digits will implicitly distinguish between addition and subtraction.

```

Step4: /* Apply synthetic division. */
m = k * (n + 1);
for (h = n; h >= 1; h--) {
  c = 0;
  m = m - k;
  for (i = 1; i <= m; i++) {
    s = P[i] + P[i + k] + c;
    c = 0;
    if (s >= BETA) {
      s = s - BETA;
      c = 1; }
    else
      if (s <= -BETA) {
        s = s + BETA;
        c = -1; }
    P[i + k] = s; } }

```

FIGURE 5. Taylor shift by 1 in SACLIB. The key ideas are the use of signed digits, suspended normalization, and the elimination of loop overhead for each integer addition.

The technique we use is an instance of register tiling—a computation method that groups the operands, loads them into machine registers, and operates on the operands without referencing the memory [2, 14]. We call our method *tile method*. The routine consists of 275 hand-written lines of C-code. In addition, we use a code generator to automatically unroll and schedule some parts of the code, which further improves performance but results in a total number of 848 lines of C-code.

5.1. Description of the algorithm. We partition the set of indices I_n of Definition 1 as shown in Figure 6 (a).

Definition 5. Let n, b be positive integers. For non-negative integers i, j let

$$T_{i,j} = \{(h, k) \in I_n \mid \lfloor h/b \rfloor = i \wedge \lfloor k/b \rfloor = j\},$$

and let T be the set of non-empty sets $T_{i,j}$.

Remark 2. The set T is a partition of the set of indices I_n ; some elements of T can be interpreted as squares of sidelength b , others as triangles and pentagons.

Definition 6. Let $T_{i,j} \in T$. The *sets of input indices* to $T_{i,j}$ are

$$N_{i,j} = \{(h, k) \in I_n \mid h = ib - 1 \wedge \lfloor k/b \rfloor = j\},$$

$$W_{i,j} = \{(h, k) \in I_n \mid \lfloor h/b \rfloor = i \wedge k = jb - 1\}.$$

The *sets of output indices* for $T_{i,j}$ are

$$S_{i,j} = \{(h, k) \in I_n \mid h = ib + b - 1 \wedge \lfloor k/b \rfloor = j\},$$

$$E_{i,j} = \{(h, k) \in I_n \mid \lfloor h/b \rfloor = i \wedge k = jb + b - 1\}.$$

Remark 3. Clearly, $N_{i,j} = S_{i-1,j}$, $W_{i,j} = E_{i,j-1}$ whenever these sets are defined.

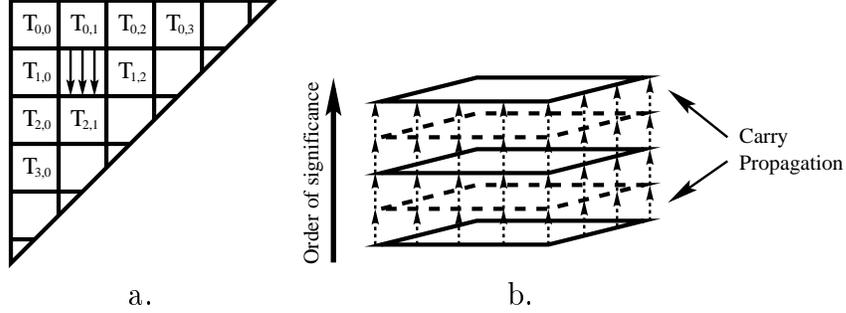


FIGURE 6. a. Tiled Pascal triangle. b. Register tile stack. A register tile is computed for each order of significance. Carries are propagated only along lower and right borders.

Definition 7. Let $a_{h,k}$ be one of the intermediate integer results in Definition 1, and let β be an integer > 1 . We write

$$a_{h,k} = \sum_r a_{h,k}^{(r)} \beta^r,$$

where $|a_{h,k}^{(r)}| < \beta$, and we define, for all i, j ,

$$N_{i,j}^{(r)} = \{a_{h,k}^{(r)} \mid (h, k) \in N_{i,j}\}$$

and, analogously, $W_{i,j}^{(r)}$, $S_{i,j}^{(r)}$, and $E_{i,j}^{(r)}$.

Let $I = \max\{i \mid T_{i,j} \in T\}$ and $J = \max\{j \mid T_{i,j} \in T\}$. The tile method computes, for $i = 0, \dots, I$ and $j = 0, \dots, J - i$, the intermediate integer results with indices in $S_{i,j} \cup E_{i,j}$ from the intermediate integer results with indices in $N_{i,j} \cup W_{i,j}$. The computation is performed as follows. A *register tile* computation at level r takes $N_{i,j}^{(r)}$ and $W_{i,j}^{(r)}$ as inputs and performs the additions described in Figure 7 (a); the additions are performed without carry but using a radix $B > \beta$. Once the register tile computations have been performed for all levels r , a carry propagation transforms the results into $S_{i,j}^{(r)}$ and $E_{i,j}^{(r)}$ for all levels r . Referring to Figure 6 (b) we call the collection of register tile computations for all levels r a *register tile stack*. The maximum value of r for each stack of index (i, j) depends on the precision of the stack which we define as follows.

Definition 8. The *precision* $L_{i,j}^*$ of the register tile stack with index (i, j) is defined recursively as follows.

$$\begin{aligned} L_{-1,j}^* &= \max(\{L(a_{h,k}) \mid (h, k) \in N_{0,j}\}), \\ L_{i,-1}^* &= \max(\{L(a_{h,k}) \mid (h, k) \in W_{i,0}\}), \\ L_{i,j}^* &= \max(\{L_{i-1,j}^*, L_{i,j-1}^*\} \cup \{L(a_{h,k}) \mid (h, k) \in S_{i,j} \cup E_{i,j}\}). \end{aligned}$$

To facilitate block prefetching we place the input digits to a register tile next to each other in memory. We thus have the following *interlaced polynomial representation* of the polynomial $A(x)$ in Definition 1 by the array \mathbf{P} . If i is a non-negative integer such that $i = g(n+1) + f$ and $0 \leq f < n+1$ then $\mathbf{P}[i]$ contains the value $a_{f,-1}^{(g)}$ defined in Definition 7.

Theorem 6. *The computation of a register tile requires at most $L(\beta-1) + 2b - 2$ bits for the magnitude of each intermediate result.*

Proof. Let $n = 2b - 1$, and let $B_{n,\beta-1}(x)$ be the polynomial defined in Definition 4. For all $(h, k) \in T_{0,0}$ we have $0 \leq h, k \leq b - 1$. Then, by Theorem 2, $L(a_{h,k}) \leq L(\beta - 1) + h + k \leq L(\beta - 1) + 2b - 2$. \square

Theorem 7. *If $L(B) \geq L(\beta - 1) + 2b - 2$ and 1 bit is available for the sign then the tile method is correct.*

Remark 4. The UltraSPARC III has a 64-bit word. We use 16 registers, and we let $b = 8$, $\beta = 2^{49}$, and $B = 2^{63}$.

5.2. Properties of the algorithm.

Theorem 8. *The tile method has the following properties.*

- (1) *Assuming the straightforward method must read all operands from memory and write all results to memory, the tile method will reduce memory reads by a factor of $b/2$, and memory writes by a factor of $b/4$.*
- (2) *Since the UltraSPARC processor architecture is capable of concurrent execution of 2 integer instructions and 1 memory reference instruction with a memory reference latency of at least 2 cycles, a $b \times b$ register tile computation takes at least $\frac{b^2}{2} + 7$ processor cycles.*

Proof. (1) Obvious. (2) In the register tile, the addition at the SE-corner must follow the other $b^2 - 1$ additions, and the addition at the NW-corner must precede all other additions. The first addition requires two summands in registers, which takes at least 3 cycles for the first summand and 1 more cycle for the second summand. The last sum needs to be written to two locations; the first write requires 3 cycles and the second 1 more cycle. Since we can perform the other

$b^2 - 2$ additions in $\frac{(b^2-2)}{2}$ cycles, the register tile will take at least $3 + 1 + (b^2 - 2)/2 + 1 + 3 = b^2/2 + 7$ cycles. \square

The 8×8 register tile computation should take at least $8^2/2 + 7 = 39$ processor cycles, see Figure 7 (b). By unrolling and manually scheduling the code for the register tile, the code compiled with the Sun Studio 9 C compiler and the optimization options `-fast -xchip=ultra3 -xarch=v9b` required 53 cycles. When the compiler was used to schedule the unrolled code the computation required 63 cycles.

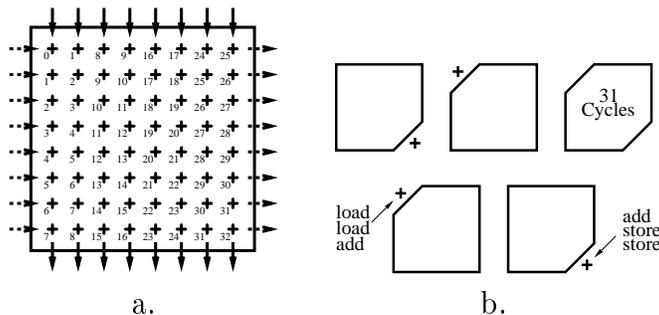


FIGURE 7. a. A scheduled 8×8 register tile. Arrows represent memory references, “+”-signs represent additions. Numbers represent processor cycles. The 2 integer execution units (IEU) perform 2 additions per cycle. b. Register tile: a sketch of the proof of Theorem 8.

5.3. Automatic tuning. The algorithm has several parameters such as tile size b , radix β , and the amount of unrolling and scheduling of various parts of the code. These parameters can be tuned to a particular architecture. On the UltraSPARC III, $b = 8$ turned out to be the largest register tile size for which the Sun Studio 9 C compiler stored all summands in registers.

6. PERFORMANCE

In the RAM-model of computation [1] the tile method is more expensive—with respect to the logarithmic cost function—than straightforward methods. Indeed, by reducing the radix the tile method increases the number of machine words needed to represent integers and hence requires more word additions than straightforward implementations. However, modern computer architectures [12, 4] are quite different from the RAM-model. We show that, on the UltraSPARC architecture, the tile method outperforms straightforward methods

by a significant factor—essentially by reducing the number of cycles per word addition. In Section 6.10 we compare our computing times with those published by von zur Gathen and Gerhard [22, 9].

6.1. Hardware and operating system. We use a Sun Blade 2000 workstation. The machine has two 900-MHz Sun UltraSPARC III processors, two gigabytes of random-access memory, and it operates under the Solaris 9 operating system. The UltraSPARC III has 32 user-available 64-bit integer registers [13, 21]. Its superscalar architecture provides six 14-stage pipelines, four of which can be independently engaged. Two of the pipelines perform integer operations, two floating point operations, one memory access, and one pipeline performs branch instructions. The processor is capable of speculative execution of branch instructions and memory loads. The data cache has 64 kilobytes, the instruction cache 32 kilobytes. Both caches are 4-way set-associative Level-1 (L1) on-die caches that use 32-byte blocks. The external Level-2 (L2) cache features a 1- to 8-megabyte unified 2-way set-associative design with a variable block size of 64 bytes to 512 bytes with 64-byte sub-blocks. The Sun Blade 2000 workstation has an 8-megabyte L2 cache.

6.2. Compilers. We wrote all our code in C and compiled it using the Sun Studio 9 [20] compiler with the optimization options `-xO3 -xarch=v9b`. The optimization options `-fast -xchip=ultra3 -xarch=v9b` yielded slightly slower run times. The Sun compiler generated executable code that was 10%-70% faster than code generated by versions 2.95.2 and 3.3.2 of the GNU C/C++ compiler with either of the optimization options `-O3` and `-O3 -mcpu=ultrasparc3 -m64`. We compiled version 4.1.2 of the GNU-MP library using the Sun Studio 7 compiler; we installed the library using the standard installation but substituting CFLAGS by `-fast`.

6.3. Performance counter measurements. We accessed the performance counters on our processor through the CPC-library that was provided with the operating system. We monitored processor cycles, instructions, and branch mispredictions as well as cache misses for the L1 instruction cache, the L1 data cache, and the L2 external cache. Execution times were calculated from the number of processor cycles; on our machine, 1 cycle corresponds precisely to $1/900 \mu\text{s}$.

Before each measurement, we flushed the L1 and L2 data caches by declaring a large integer array and writing and reading it once. We did not flush the L1 instruction cache; our measurements show that its impact on performance is insignificant. We obtained each data point as the average of at least 3 measurements. The fluctuation within these measurements was usually well under 1%. We did not remove any outliers.

6.4. Input polynomials. As inputs we use the polynomials $B_{n,d}$ and $C_{n,d}$ of degree n and max-norm d defined in Definition 4. For the polynomials $C_{n,d}$ we let $n \in \{22, 25\}$ and $d = 2^k - 1$, $k \in \{10, 20, \dots, 1000\}$. For the polynomials $B_{n,d}$ we let $n \in \{8, 10, 12, \dots, 200\}$ for “low” degrees and $n \in \{200, 300, \dots, 10000\}$ for “high” degrees; in both cases, $d = 2^{20} - 1$. The wide ranges of n and d serve to illustrate the influence of the cache size on performance; the cross-over points with asymptotically fast methods are not known.

6.5. Execution time. Figure 8 shows the speedup that the SACLIB and tile methods provide with respect to the straightforward method for the input polynomials $B_{n,d}$. The tile method is up to 7 times faster than the straightforward method for low degrees and 3 times faster for high degrees. The SACLIB method is up to 4 times faster than the straightforward method for low degrees but slower for high degrees. The speedups are not due to the fact that the faster methods avoid the cost of re-allocating memory as the intermediate results grow. Indeed, pre-allocating memory accelerates the straightforward method by a factor of only 1.25 for degree 50. As the degree increases that factor approaches 1.

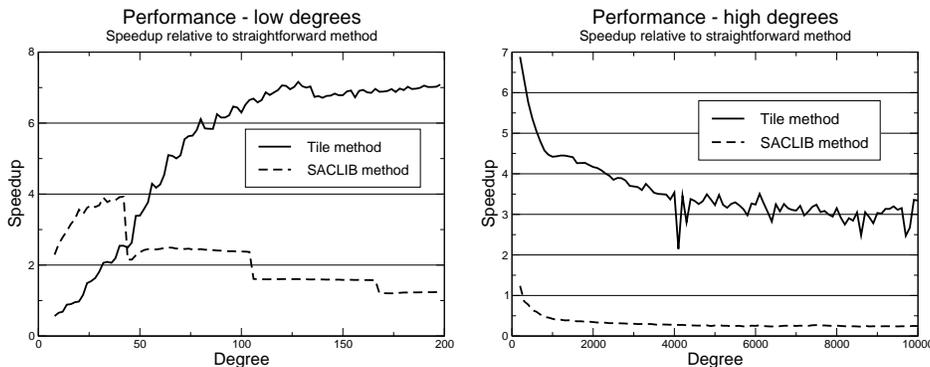


FIGURE 8. The tile method is up to 7 times faster than the straightforward method.

In Figure 9 the polynomials $C_{n,d}$ reveal a weakness of the tile method. The tile method does not keep track of the individual precisions of the intermediate results $a_{i,j}$ but instead uses the same precision for all the integers in a tile. The tile stack containing the constant term d of $C_{22,d}$ and $C_{25,d}$ consists of 28 and 3 integers $a_{i,j}$, respectively. Thus, when the degree stays fixed and d tends to infinity, the tile method becomes slower than the straightforward method by a constant factor. The figure shows that—even when the degree is small—the constant term d must become extremely large in order to degrade the performance.

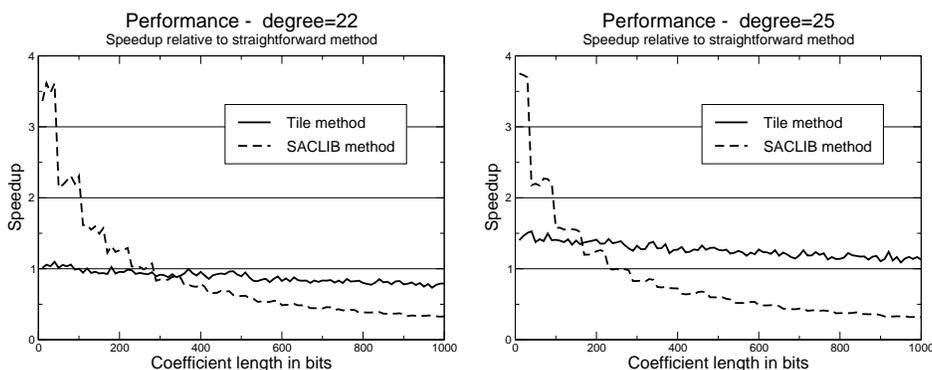


FIGURE 9. For the input polynomials $C_{n,d}$ the tile method computes a whole register tile stack at the precision required for just the constant term.

6.6. Cycles per word addition. Figure 10 shows the number of cycles per word addition for the GNU-MP addition routine described in Section 3. In the experiment all words of both summands were initialized to $2^{64} - 1$, and the summands were prefetched into L1 cache. The figure shows that the intended ratio of 4 cycles per word addition is nearly reached when the summands are very long and fit into L1 cache; for short integers GNU-MP addition is much less efficient.

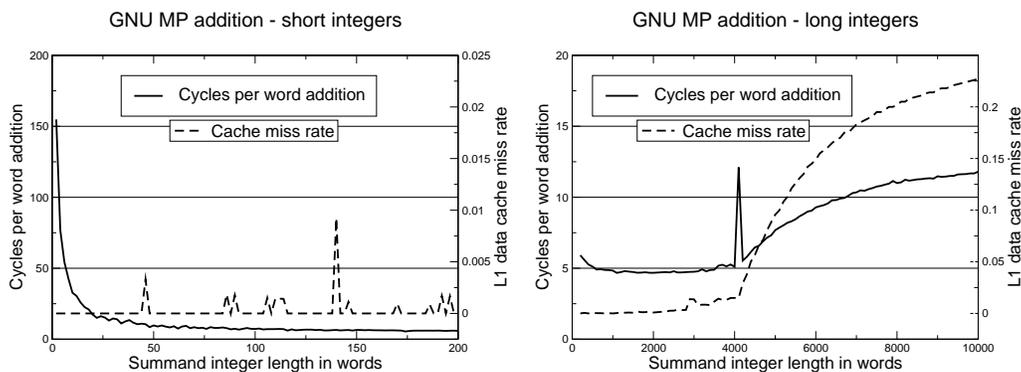


FIGURE 10. In GNU-MP addition the ratio of cycles per word addition (left scale) increases with the cache miss rate (right scale).

Figure 11 shows the number of cycles per word addition for the GNU-MP-based straightforward Taylor shift described in Section 3 and for the tile method described in Section 5; the polynomials $B_{n,d}$ were used as inputs. For large degrees

the methods require about 5.7 and 1.4 cycles per word addition, respectively. Since the tile method uses the radix 2^{49} and the straightforward method uses the radix 2^{64} the tile method executes about $64/49 \approx 1.3$ times more word additions than the straightforward method. As a result the tile method should be faster than the straightforward method by a factor of $5.7/(1.4 \cdot 1.3) \approx 3.1$. The measurements shown in Figure 8 agree well with this expectation.

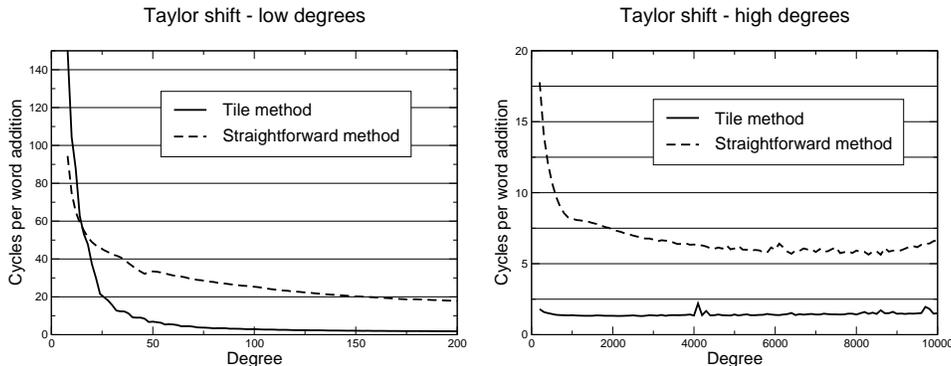


FIGURE 11. In classical Taylor shift by 1 the tile method requires fewer cycles per word addition than the straightforward method.

6.7. Memory traffic reduction. Figure 12 shows that the tile method reduces the number of memory reads with respect to the straightforward method by a factor of up to 7. The polynomials $B_{n,d}$ were used as inputs. The number of memory reads in the GNU-MP-based straightforward method is independent of the compiler since the implementation relies to a large extent on an assembly language routine. However, the number of memory reads in the tile method depends on how well the compiler is able to take advantage of our C-code for the computation of register tiles. The figure shows that the Sun Studio 9 C-compiler with the option `-xO3 -xarch=v9b` works best for the tile method.

6.8. Cache miss rates. Figure 13 shows the L1 data cache miss rates for the straightforward method and the tile method; the polynomials $B_{n,d}$ were used as inputs. As the degree increases the cache miss rate of the straightforward method rises sharply as soon as the polynomials no longer fit into the cache. The cache miss rate levels off at about 13%. Indeed, by Section 6.1 the block size is 8 words; so, one expects 7 cache hits for each cache miss.

6.9. Branch mispredictions. Figure 14 shows the number of branch mispredictions per cycle for the straightforward method and the tile method; the polynomials $B_{n,d}$ were used as inputs. Since either method produces at most one

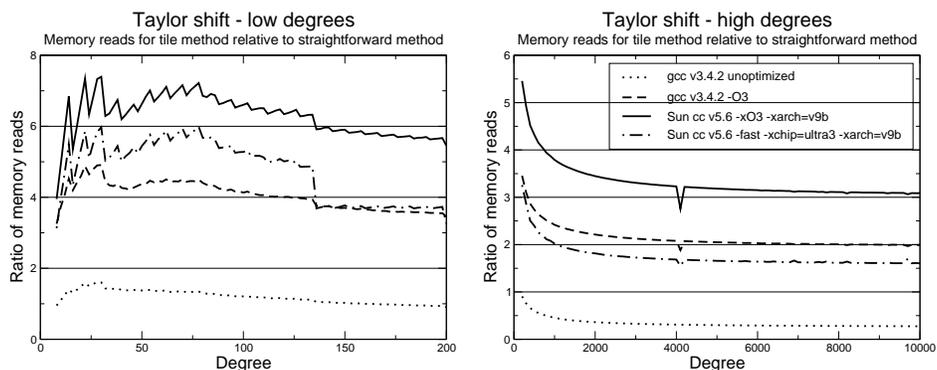


FIGURE 12. The tile method substantially reduces the number of memory reads required for the Taylor shift; the extent of the reduction depends on the compiler.

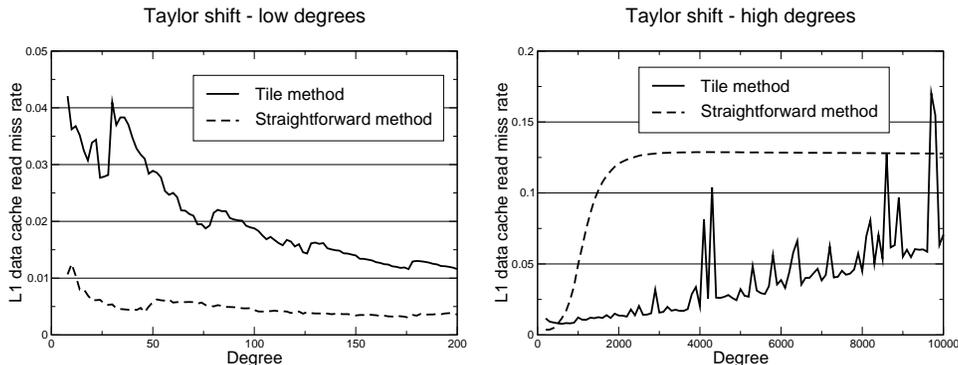


FIGURE 13. For large degrees the tile method has a lower cache miss rate than the straightforward method. Moreover, the number of cache misses generated by the tile method is small because the tile method performs few read operations.

branch misprediction every 200 cycles, branch mispredictions do not significantly affect the two methods. However, the branch misprediction rate of the SACLIB method is 60 times greater than that of the straightforward method when the degree is high.

6.10. Computing times in the literature. Von zur Gathen and Gerhard [22, 9] published computing times for the NTL-based implementation of the straightforward method described in Section 3. Tables 1 and 2 quote those computing times and compare the NTL-based straightforward method with the GNU-MP-based straightforward method and the tile method.

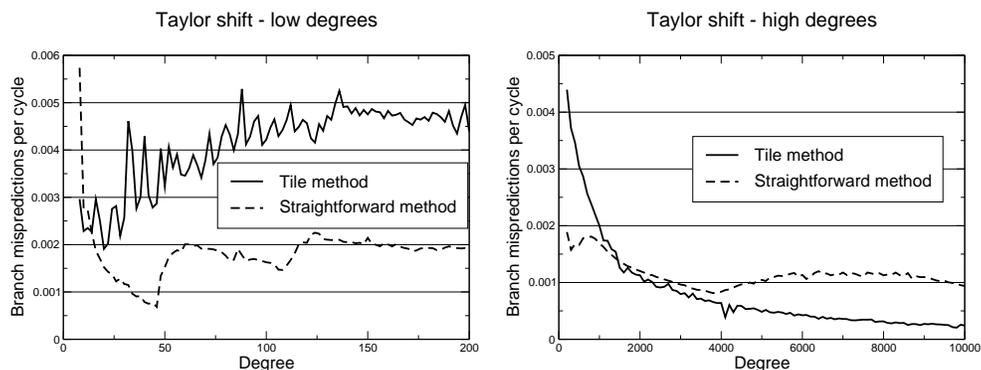


FIGURE 14. The number of branch mispredictions per cycle is negligible for the tile method and the straightforward method.

TABLE 1. Computing times (s.) for Taylor shift by 1 —“small” coefficients.

degree	straightforward				tile
	NTL-addition		GMP-add		
	UltraSPARC [22]	Pentium III [9]	UltraSPARC III		
127	0.004	0.001	0.001	0.00076	0.00010
255	0.019	0.005	0.004	0.00327	0.00046
511	0.102	0.030	0.016	0.01475	0.00286
1023	0.637	0.190	0.101	0.08261	0.03183
2047	4.700	2.447	0.710	0.56577	0.27114
4095	39.243	22.126	4.958	3.73049	1.97799
8191	—	176.840	44.200	29.91298	18.48445

TABLE 2. Computing times (s.) for Taylor shift by 1 —“large” coefficients.

degree	straightforward				tile
	NTL-addition		GMP-add		
	UltraSPARC [22]	Pentium III [9]	UltraSPARC III		
127	0.006	0.002	0.001	0.00096	0.00016
255	0.036	0.010	0.005	0.00434	0.00099
511	0.244	0.068	0.029	0.02154	0.00838
1023	1.788	0.608	0.231	0.17607	0.09183
2047	13.897	8.068	1.773	1.27955	0.83963
4095	111.503	65.758	13.878	9.97772	6.27948
8191	—	576.539	140.630	151.27732	61.04515

The computing times we quote were obtained on an UltraSPARC workstation rated at 167 MHz [22] and on a Pentium III 800 MHz Linux PC; the latter experiments were performed using the default installation of version 5.0c of NTL [9]. We installed NTL in the same way on our experimental platform but while the default installation uses the gcc compiler with the `-O2` option we used the Sun compiler with the options `-fast -xchip=ultra3`. This change of compilers sped-up the NTL-based straightforward method by factors ranging from 1.06 to 1.63.

Von zur Gathen and Gerhard ran their program letting $k = 7, \dots, 13$ and $n = 2^k - 1$ for input polynomials of degree n and max-norm $\leq n$ for Table 1, and max-norm $< 2^{n+1}$ for Table 2; the integer coefficients were pseudo-randomly generated. We used the same input polynomials in our experiments.

The NTL-based straightforward method runs faster on the UltraSPARC III than on the Pentium III, but the speedup ratios vary. This is likely due to differences between the processors in cache size and pipeline organization. The computing time ratios between the NTL- and GNU-MP-based straightforward methods on the UltraSPARC III are more uniform and range between 0.9 and 1.7. If these computing time ratios can be explained by the difference in radix size— 2^{30} for NTL and 2^{64} for GNU-MP—then there is no justification for the use of assembly language in the GNU-MP-addition routine. Again, the tile method outperforms the straightforward methods.

7. ACKNOWLEDGEMENTS

Thomas Decker invented the interlaced polynomial representation to improve the SACLIB method. Jürgen Gerhard kindly made his Taylor shift code [22, 9] available. Our research was supported in part by NSF grants ECS-0424475 (W.K.) and ITR/NGS-0325687 (J.R.J.).

REFERENCES

- [1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.
- [2] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architecture: A Dependence-Based Approach*. Morgan Kaufmann Publishers, New York, 2002.
- [3] Jacques Borowczyk. Sur la vie et l'œuvre de François Budan (1761–1840). *Historia Mathematica*, 18:129–157, 1991.
- [4] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.
- [5] G. E. Collins and R. G. K. Loos. Specifications and index of SAC-2 algorithms. Technical Report WSI-90-4, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 1990.
- [6] George E. Collins. The computing time of the Euclidean algorithm. *SIAM Journal on Computing*, 3(1):1–10, 1974.

- [7] George E. Collins and Alkiviadis G. Akritas. Polynomial real root isolation using Descartes' rule of signs. In R. D. Jenks, editor, *Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation*, pages 272–275. ACM Press, 1976.
- [8] George E. Collins et al. SACLIB User's Guide. Technical Report 93-19, Research Institute for Symbolic Computation, RISC-Linz, Johannes Kepler University, A-4040 Linz, Austria, 1993.
- [9] Jürgen Gerhard. *Modular Algorithms in Symbolic Summation and Symbolic Integration*, volume 3218 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
- [10] Jürgen Gerhard. Personal communication, 2005.
- [11] Torbjörn Granlund. *GNU MP: The GNU Multiple Precision Arithmetic Library*. Swox AB, September 2004. Edition 4.1.4.
- [12] John L. Hennessy, David A. Patterson, and David Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2002.
- [13] Tim Horel and Gary Lauterbach. UltraSPARC-III: Designing third-generation 64-bit performance. *IEEE MICRO*, 19(3):73–85, 1999.
- [14] Marta Jiménez, José M. Llabería, and Agustín Fernández. Register tiling in nonrectangular iteration spaces. *ACM Transactions on Programming Languages and Systems*, 24(4):409–453, 2002.
- [15] Maplesoft. *Maple 9: Learning Guide*, 2003.
- [16] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 9: Advanced Programming Guide*. Maplesoft, 2003.
- [17] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 9: Introductory Programming Guide*. Maplesoft, 2003.
- [18] Victor Shoup. *NTL: A Library for Doing Number Theory*. <http://www.shoup.net/ntl>.
- [19] Victor Shoup. A new polynomial factorization algorithm and its implementation. *Journal of Symbolic Computation*, 20(4):363–397, 1995.
- [20] Sun Microsystems. Sun Studio Collection. <http://www.sun.com/software/products/studio/>.
- [21] Sun Microsystems. *UltraSPARC III Cu: User's Manual*, 2004.
- [22] Joachim von zur Gathen and Jürgen Gerhard. Fast algorithms for Taylor shifts and certain difference equations. In W. W. Küchlin, editor, *International Symposium on Symbolic and Algebraic Computation*, pages 40–47. ACM Press, 1997.