More On Squaring and Multiplying Large Integers

Dan Zuras

Abstract—**Methods of squaring and multiplying large integers are discussed. The obvious $O(n^2)$ methods turn out to be best for small numbers. Existing $O(n^{\log 3/\log 2}) \approx O(n^{1.585})$ methods become better as the numbers get bigger. New methods that are $O(n^{\log 5/\log 3}) \approx O(n^{1.465})$, $O(n^{\log 7/\log 4}) \approx O(n^{1.404})$, and $O(n^{\log 9/\log 5}) \approx O(n^{1.365})$ are presented. In actual experiments, all of these methods turn out to be faster than FFT multiplies for numbers that can be quite large (> 37,000,000 bits). Squaring seems to be fundamentally faster than multiplying but it is shown that $T_{\text{multiply}} \leq 2T_{\text{square}} + O(n)$.**

Index Terms— **Integer multiply, Karatsuba and Ofman, Toom–Cook, Schönhage and Strassen, FFT multiply.**

I. Introduction

THIS is an expansion of the paper On Squaring and Multiplying Large Integers [8] presented at the IEEE Symposium on Computer Arithmetic (ARITH-11) in July 1993. This paper presents more of the ideas behind these methods; some new results; and some speculations for areas of future work.

Much of the work presented here is based on methods that appear in Don Knuth's classic work *The Art of Computer Programming* [4] as well as the work of A. L. Toom [7].

In actual experiments, some of the simpler methods of squaring and multiplying are shown to be best in a surprisingly large number of cases. Some new methods are also discussed which are useful out to quite large numbers. In light of these results, it seems that methods such as the Schönhage and Strassen FFT multiply [5], while of theoretical interest, may never be best for any reasonably sized numbers.

II. The Problem

The problem to be discussed here is how to find the best (fastest) ways to square and multiply large numbers in software. The methods presented here were implemented in C and assembly language on an HP-9000/720 but the general observations and conclusions should be true in wider areas of application.

The approach used was to write a collection of routines for squaring and multiplying w-word numbers producing $2w$-word results for various specific values of w. These were general purpose routines in the sense that w was a parameter.

In the course of writing these routines, it became clear that some common operations on large integers stored as arrays of unsigned 32-bit words would be needed.[1] These operations were written in assembly language and performed the functions detailed in Table I.

TABLE I

Given operands a[0,w-1], b[0,w-1], producing result r[0,w-1]	
vzconst(r,w,c)	r[0,w-1] = ccc...c (w c's)
vzcopy(r,w,a)	r[0,w-1] = a[0,w-1]
vzneg(r,w,a)	r[0,w-1] = -a[0,w-1]
vzshlM(r,w,a,c)	r[0,w-1] = a[0,w-1]|c<<M for M in [1,2,4,8,16,31]
vzshr(r,w,a,M)	r[0,w-1] = a[0,w-1]>>M for M in [0,31]
vzadd(r,w,a,b)	r[0,w-1] = a[0,w-1] + b[0,w-1]
vzsub(r,w,a,b)	r[0,w-1] = a[0,w-1] - b[0,w-1]
Cout = vzaddwco(r,w,a,b)	r[0,w-1] + Cout = a[0,w-1] + b[0,w-1]
vzshMadd(r,w,a,b,M)	r[0,w-1] = a[0,w-1]<<M + b[0,w-1]
vzshMsub(r,w,a,b,M)	r[0,w-1] = a[0,w-1]<<M - b[0,w-1]
t = vzge(b,w,a)	t = (b[0,w-1] ≥ a [0,w-1])

A collection of routines for squaring and multiplying "small" integers were also written in assembly. These provided a basis upon which the larger routines could be built.

All other routines were written in C and compiler optimized.

Running time was measured for w in the range 1 to 1,175,553 words (37,617,696 bits) by counting instruction cycles on a 50 MHz HP-9000/720.

III. Finding the Best

We will define $T_{Si}(w)$ as the time required for method i to square a number of length w and $T_{Mi}(w)$ as the time required for method i to multiply a number of length w. (We will use the expression $T_i(w)$ when discussing either method or method i in general.)

Method i is considered *best* for some length w if

$$T_i(w) \leq T_j(w'), \forall j \forall w' \geq w.$$

That is, a method is best for a given length if there is no faster way of squaring or multiplying numbers at least as large as this one.

Manuscript received October 13, 1993; revised March 2, 1994.
The author is with Hewlett-Packard, Co., Mailstop 47LA, 19447 Pruneridge Avenue, Cupertino, CA 95014 USA; email: Zuras@cup.hp.com.
IEEE Log Number 9403087.

[1] These arrays were oriented "big-endian." That is, the most significant word of $a[0, w-1]$ was stored in $a[0]$ and the least significant word was stored in $a[w-1]$. While the author preferred "little-endian," both C and the HP-9000 PA-RISC architecture disagreed. They won, in the end.

IV. Method 1: The w^2 Basis

To start the ball rolling, basis routines for operating on small integers were written using the obvious method.

Roughly speaking, if the algorithm for a multiply is

```
1. FOR $i = w - 1$ to 0:
    a. FOR $j = w - 1$ to 0:
        • ($r[i + j, i + j + 1]$ and CarryOut)
        • $+= a[i] \times b[j]$,
```

then squaring can be made strictly faster than multiply by accumulating the upper half off-diagonal elements separately and combining them with the diagonals according to the formula

$$\text{Result} = 2\ \text{Offdiagonals} + \text{Diagonals}.$$

Obviously, the running time, $T_1(w)$, increases quadratically. $T_1(w)$, in cycles, is well approximated on the HP-9000/720 by the least squares formulas

$$T_{S1}(w) = 4.53w^2 + 10w + 30.71,$$

$$T_{M1}(w) = 9.82w^2 - 9.06w + 90.26.$$

These routines turned out to be best for all w up to 33 for square (23 for multiply).

V. The 2-Way Method

It seems that the first discussion of a method of multiplication (squaring, actually) that was better than w^2 in the width of the operand was a short paper written by Karatsuba and Ofman [3] that appeared in the Soviet journal Doklady in 1963. The method involves splitting the operand into 2 parts: the high order bits and the low order bits, based on the polynomial[2]

$$(A_1x + A_0)^2 = A_1^2(x^2 - x) + (A_1 + A_0)^2x + A_0^2(1 - x),$$

where $x = 2^n$.

Knuth presented a minor improvement on this based on the slightly different polynomial

$$(A_1x + A_0)^2 = A_1^2(x^2 + x) - (A_1 - A_0)^2x + A_0^2(x + 1).$$

Both of these may be regarded as special cases of solving for the C_i in the polynomial

$$(A_1x + A_0)^2 = C_2x^2 + C_1x + C_0.$$

Karatsuba and Ofman solve this equation by letting x take on the values $\{\infty, 1, 0\}$.[3]

Rather than express this system in the usual polynomial form, I will express it as a linear transformation of the operand in the following way

$$\begin{bmatrix} V_2 \\ V_1 \\ V_0 \end{bmatrix} = \begin{bmatrix} A_1 \\ A_1 + A_0 \\ A_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A_1 \\ A_0 \end{bmatrix}.$$

[2]I will not, however, inflict the "big-endian" notation on the reader. Here, I will use the somewhat more conventional notation of mathematics. That is, "little-endian". So there.

[3]In this context, $x = \infty$ corresponds to $\lim_{x\to\infty}(A_1x + A_0)^2/x^2 = \lim_{x\to\infty}(C_2x^2 + C_1x + C_0)/x^2$.

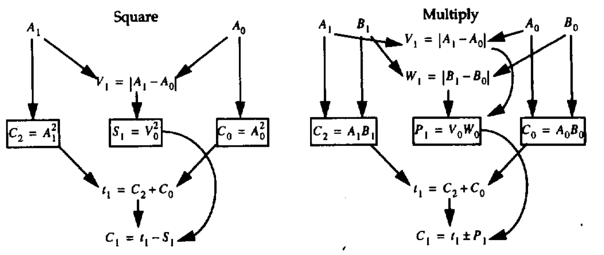

Fig. 1. The two-way method, $O(w^{1.585})$.

After squaring each element of the vector, the solution to the resulting system

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_2 \\ S_1 \\ S_0 \end{bmatrix} = \begin{bmatrix} A_1^2 \\ (A_1 + A_0)^2 \\ A_0^2 \end{bmatrix}$$

may be expressed as

$$\begin{bmatrix} C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_2 \\ S_1 \\ S_0 \end{bmatrix}.$$

This method has a small problem in that the term $A_1 + A_0$ requires one more bit than the other two. Karatsuba and Ofman address this problem by showing how you can square an $n+1$-bit number using an n-bit algorithm together with a couple of shifts and adds.

Knuth solves the equation by letting x take on the values $\{\infty, -1, 0\}$. A similar transformation

$$\begin{bmatrix} V_2 \\ V_1 \\ V_0 \end{bmatrix} = \begin{bmatrix} A_1 \\ A_1 - A_0 \\ A_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A_1 \\ A_0 \end{bmatrix}$$

results, after squaring, in a similar system

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_2 \\ S_1 \\ S_0 \end{bmatrix} = \begin{bmatrix} A_1^2 \\ (A_1 - A_0)^2 \\ A_0^2 \end{bmatrix},$$

which has the solution

$$\begin{bmatrix} C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_2 \\ S_1 \\ S_0 \end{bmatrix}.$$

Knuth presented this method as Karatsuba and Ofman's but Knuth's variation leads to a real improvement. By rearranging the algebra a bit, he was able to replace the $A_1 + A_0$ term with $A_1 - A_0$. Since the only use of the term is to square it, its sign is unimportant. Therefore, one can always subtract the lesser from the greater and save that extra bit.

The test and branch involved results in much less overhead than Karatsuba and Ofman's method.

This method can be turned into a multiply method by replacing the S_i with the P_i in the formula

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} P_2 \\ P_1 \\ P_0 \end{bmatrix} = \begin{bmatrix} A_1B_1 \\ (A_1 - A_0)(B_1 - B_0) \\ A_0B_0 \end{bmatrix}$$

and doing the implied extra work.

These methods are diagrammed in Fig. 1.

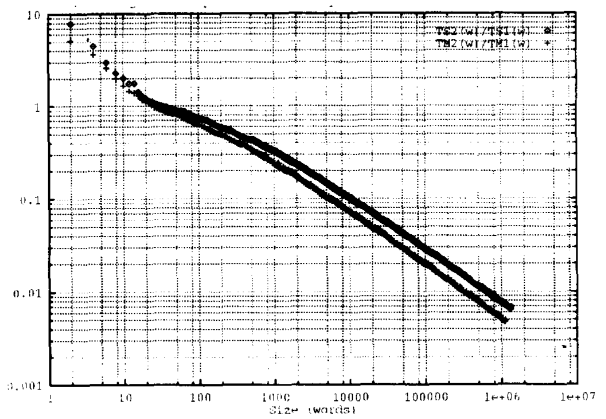

Fig. 2. Improvement of two-way methods over w^2 basis.

Assuming that the time for an add (or subtract) of length w is $O(w)$, the time to square a number (or multiply two numbers) of length $2w$ would be

$$T_2(2w) = 3T_{\text{best}}(w) + O(w).$$

The overhead involved, one add before the squaring step (two adds before the multiplying step) and two after, is strictly larger than the overhead for a w^2 method. Thus, although a w^2 squaring (or multiply) of length $2w$ would take

$$T_1(2w) \approx 4T_{\text{best}}(w) + O(w),$$

there is a definite crossover point as w increases where $T_1(w)$ is roughly the same as $T_2(w)$. This point occurs where the time involved in the extra overhead of Knuth's method (which I will call the two-way method from this point on) is roughly equivalent to the time for the fourth multiply in the w^2 method.

Below this point the w^2 method wins because of its simplicity and above this point the two-way method wins because of its better asymptotic behavior.

Figure 2 illustrates this by plotting the ratio $T_2(w)/T_1(w)$ for some two-way routines. ($T_1(w)$ is actually the least squares approximation.)

In the limit, the running time of the two-way method triples each time the size of the problem doubles. Therefore, its time increases as $w^{\log 3/\log 2} \approx w^{1.585}$.

The running time of the two-way routines (in cycles) is approximated by the least squares formulas

$$T_{S2}(w) \approx 26.64w^{\log 3/\log 2} - 54.63w + 227$$

$$T_{M2}(w) \approx 46.2w^{\log 3/\log 2} - 84.11w + 979.$$

Building on a basis set of w^2 routines up to 33 words for square (23 words for multiply), the two-way method takes over at 34 words (24 words) and generally dominates in these experiments up to 768 words (384 words).

M. Shand [6] has pointed out that this crossover point is a strong function of the ratio of the multiply time to add time on a given machine. The HP-9000/720 performs an unsigned $32 \times 32 \rightarrow 64$ multiply in two cycles (5 to 7 cycles if you include load/store overhead). On a machine with a relatively slower multiply the crossover point can be much lower.

VI. The General Idea

We now have enough background to show the general idea behind these squaring (and multiplying) methods.

In each squaring case, the operand is split into k parts A_j. These parts are transformed via a Q_{ij} (derived from the original polynomial) into m parts $V_i = Q_{ij}A_j$. Then, each of the V_i is squared with the best method available, producing $S_i = V_i^2$.

In each multiply case, the operands are split into k parts A_j and B_j. These parts are transformed via Q_{ij} into m parts $V_i = Q_{ij}A_j$ and $W_i = Q_{ij}B_j$. Then, each of the V_i is multiplied by W_i with the best method available, producing $P_i = V_iW_i$.

Now, the solution to the system $M_{ij}C_j = S_i$ ($M_{ij}C_j = P_i$) (derived from the square of the original polynomial) can be expressed as $C_i = \sum_{ik}^{-1} R_{kj}S_j$ ($C_i = \sum_{ik}^{-1} R_{kj}P_j$) with all the coefficients in the integers. (Up to now, $\sum_{ik}^{-1}$, which expresses the integer divides, has not been needed as it was the identity matrix.) Finally, the C_i are shifted and added up to produce the result.

VII. The 3-Way Method

Toom [7] showed that circuits could be constructed for squaring integers where the size of the circuit was bounded by $O(nc^{\sqrt{\log n}})$ and the delay was bounded by $O(c^{\sqrt{\log n}})$.

Cook [1] showed that an algorithm for squaring integers could be realized which had a running time of $O(n2^{5\sqrt{\log n}})$.

This algorithm, which is actually a collection of algorithms, has come to be known as the Toom–Cook method.

The first method in this collection is equivalent to Karatsuba's method.

The second method divides a number into three parts and involves solving for the C_i in the polynomial

$$(A_2x^2 + A_1x + A_0)^2 = C_4x^4 + C_3x^3 + C_2x^2 + C_1x + C_0.$$

Cook solves this equation by letting x take on the values $\{4, 3, 2, 1, 0\}$ in the polynomial, leading to the transformation

$$\begin{bmatrix} V_4 \\ V_3 \\ V_2 \\ V_1 \\ V_0 \end{bmatrix} = \begin{bmatrix} 16 & 4 & 1 \\ 9 & 3 & 1 \\ 4 & 2 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} A_2 \\ A_1 \\ A_0 \end{bmatrix},$$

which, after squaring, yields the system

$$\begin{bmatrix} 256 & 64 & 16 & 4 & 1 \\ 81 & 27 & 9 & 3 & 1 \\ 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} = \begin{bmatrix} (16A_2 + 4A_1 + A_0)^2 \\ (9A_2 + 3A_1 + A_0)^2 \\ (4A_2 + 2A_1 + A_0)^2 \\ (A_2 + A_1 + A_0)^2 \\ A_0^2 \end{bmatrix},$$

the solution of which I will express as

$$\begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} 24 & 0 & 0 & 0 & 0 \\ 0 & 12 & 0 & 0 & 0 \\ 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} 1 & -4 & 6 & -4 & 1 \\ -3 & 14 & -24 & 18 & -5 \\ 11 & -56 & 114 & -104 & 35 \\ -3 & 16 & -36 & 48 & -25 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix}.$$

The diagonal terms in the first matrix are the denominators in divides of intermediate results. In practice, there turn out to be four divides by 3 since the rest amounts to shifts of 2 or 3 bits.

Knuth solves this with x taken from $\{2, 1, 0, -1, -2\}$ leading to the transformation

$$\begin{bmatrix} V_4 \\ V_3 \\ V_2 \\ V_1 \\ V_0 \end{bmatrix} = \begin{bmatrix} 4 & 2 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & -1 & 1 \\ 4 & -2 & 1 \end{bmatrix} \begin{bmatrix} A_2 \\ A_1 \\ A_0 \end{bmatrix}.$$

and the system

$$\begin{bmatrix} 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 16 & -8 & 4 & -2 & 1 \end{bmatrix} \begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} = \begin{bmatrix} (4A_2 + 2A_1 + A_0)^2 \\ (A_2 + A_1 + A_0)^2 \\ A_0^2 \\ (A_2 - A_1 + A_0)^2 \\ (4A_2 - 2A_1 + A_0)^2 \end{bmatrix},$$

which has the somewhat more symmetrical solution

$$\begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} 24 & 0 & 0 & 0 & 0 \\ 0 & 12 & 0 & 0 & 0 \\ 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} 1 & -4 & 6 & -4 & 1 \\ 1 & -2 & 0 & 2 & -1 \\ -1 & 16 & -30 & 16 & -1 \\ -1 & 8 & 0 & -8 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix}.$$

Although both of these solutions look similar, Knuth's is somewhat better than the previous one. The sign symmetry simplifies the system and greatly reduces the overhead (in the form of add, subtract, shift, and the like).

If, however, one uses reciprocal symmetry as well as sign symmetry in the choices of x (as in the set $\{\infty, 2, 1, 1/2, 0\}$,[4] then the greater symmetry in the symmetry results in even greater simplification and greater reduction in overhead.

This system

$$\begin{bmatrix} V_4 \\ V_3 \\ V_2 \\ V_1 \\ V_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 2 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} A_2 \\ A_1 \\ A_0 \end{bmatrix}$$

and

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} = \begin{bmatrix} A_2^2 \\ (4A_2 + 2A_1 + A_0)^2 \\ (A_2 + A_1 + A_0)^2 \\ (A_2 + 2A_1 + 4A_0)^2 \\ A_0^2 \end{bmatrix}$$

has the solution

$$\begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -21 & 2 & -12 & 1 & -6 \\ 7 & -1 & 10 & -1 & 7 \\ -6 & 1 & -12 & 2 & -21 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix}.$$

(Oddly enough, it appears that reciprocal symmetry is more important than sign symmetry in the sense that, if one must break one or the other, the system seems simpler if sign symmetry is broken rather than reciprocal symmetry.)

This solution has only 2 divides by 3 and 3 shifts as well as much less overhead in the computation of the intermediate results.

The computation of S_2 requires 2 extra bits and S_1 and S_3 each require 3.

The squaring method implied by this solution is shown in Fig. 3.

Notice that it comes out looking like a complicated version of the two-way method in that there is an initial transformation stage; a squaring stage; and a final transformation stage.

All of the operations shown can be made $O(w)$. Shifts can be done separately or as part of the combined adds and subtracts.

The time to square (multiply) a number of length $3w$ becomes

$$T_3(3w) \approx 5T_{\text{best}}(w) + O(w).$$

This suggests that the asymptotic running time of the three-way method increases as $w^{\log 5/\log 3} \approx w^{1.465}$.

[4]In general, $x = p/q$ corresponds to $q^{2n}(A_n(p/q)^n + \cdots + A_0)^2 = q^{2n}(C_{2n}(p/q)^{2n} + \cdots + C_0)$.

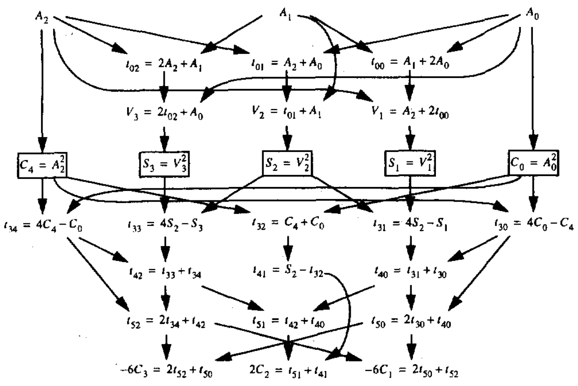

Fig. 3. The three-way method, $O(w^{1.465})$.

The improvement is not nearly so dramatic as the improvement in the two-way method over the w^2 method. The advantages of more complex routines will be even less dramatic for much more work.

The running time of the 3-way method is approximated by the formulas

$$T_{S3}(w) \approx 64.59w^{\log 5/\log 3} - 243.51w + 33913$$

$$T_{M3}(w) \approx 90.15w^{\log 5/\log 3} + 75.91w - 44016.$$

As before, to turn this method into a multiply, one duplicates those operations before the squaring stage and multiplies the corresponding terms instead of squaring them.

VIII. Running Time of Toom–Cook Style Methods

The running time of a Toom–Cook, Knuth, and k-way methods will all have three components: $m = 2k-1$ squarings of elements of size w/k; some $O(w)$ overhead; and some fixed overhead.

Assuming that one starts with a basis routine of length w_0 that takes t_0 time, we have

$$T_k(w_0) = t_0$$

$$T_k(k^{r+1}w_0) = mT_k(k^r w_0) + c_1 k^r w_0 + c_0$$

$$T_k(kw_0) = mt_0 + c_1 w_0 + c_0$$

$$T_k(k^2 w_0) = m^2 t_0 + c_1 w_0(m+k) + c_0(m+1)$$

$$T_k(k^3 w_0) = m^3 t_0 + c_1 w_0(m^2 + mk + k^2) + c_0(m^2 + m + 1)$$

$$\cdots$$

$$T_k(k^r w_0) = m^r t_0 + c_1 w_0 \frac{m^r - k^r}{m-k} + c_0 \frac{m^r - 1}{m-1}.$$

Rearranging terms in this last equation gives

$$T_k(k^r w_0) = \left(t_0 + \frac{c_1 w_0}{m-k} + \frac{c_0}{m-1}\right) m^r - \frac{c_1 w_0 k^r}{m-k} - \frac{c_0}{m-1}$$

or, in terms of $w = k^r w_0$ becomes

$$T_k(w) = \left(t_0 + \frac{c_1 w_0}{m-k} + \frac{c_0}{m-1}\right)\left(\frac{w}{w_0}\right)^{\frac{\log m}{\log k}} - \frac{c_1 w}{m-k} - \frac{c_0}{m-1}.$$

This formula has the expected form: a dominant $w^{\log m/\log k}$ term, an $O(w)$ term, and a constant term, both of which may be neglected for large w.

But, and this is important, the dominant term contains terms *proportional* to the overhead. Thus, while the overhead terms may be neglected for large w, the overhead *itself* may not as it *directly* contributes to the magnitude of the dominant term.

IX. The Four-Way Method and Beyond

Extension to a k-way method involves solving for the C_i in the polynomial

$$(A_k x^k + \cdots + A_0)^2 = C_{2k}x^{2k} + \cdots + C_0.$$

Cook would solve this polynomial by letting x take on the values $\{0, \cdots, 2k\}$.

Knuth would solve this polynomial by letting x take on the values $\{-k, \cdots, k\}$.

But if one chooses x from the reciprocally symmetric set $\{1, \infty, 0, 2, \frac{1}{2}, -2, -\frac{1}{2}, 3, \frac{1}{3}, -3, -\frac{1}{3}, \frac{3}{2}, \frac{2}{3}, \cdots\}$, that is, the set of small rational numbers, $\pm\frac{p}{q}$, such that the $\gcd(p, q) = 1$, the resulting system of equations is very symmetrical and the algorithm is simpler.

I will illustrate the various methods for $k = 4$.

Cook:

$$Q = \begin{bmatrix} 216 & 36 & 6 & 1 \\ 125 & 25 & 5 & 1 \\ 64 & 16 & 4 & 1 \\ 27 & 9 & 3 & 1 \\ 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} 46656 & 7776 & 1296 & 216 & 36 & 6 & 1 \\ 15625 & 3125 & 625 & 125 & 25 & 5 & 1 \\ 4096 & 1024 & 256 & 64 & 16 & 4 & 1 \\ 729 & 243 & 81 & 27 & 9 & 3 & 1 \\ 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 720 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 240 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 144 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 48 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 360 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 60 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R = \begin{bmatrix} 1 & -6 & 15 & -20 & 15 & -6 & 1 \\ -5 & 32 & -85 & 120 & -95 & 40 & -7 \\ 17 & -114 & 321 & -484 & 411 & -186 & 35 \\ -15 & 104 & -307 & 496 & -461 & 232 & -49 \\ 137 & -972 & 2970 & -5080 & 5265 & -3132 & 812 \\ -10 & 72 & -225 & 400 & -450 & 360 & -147 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Knuth:

$$Q = \begin{bmatrix} 27 & 9 & 3 & 1 \\ 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ -1 & 1 & -1 & 1 \\ -8 & 4 & -2 & 1 \\ -27 & 9 & -3 & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} 729 & 243 & 81 & 27 & 9 & 3 & 1 \\ 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 64 & -32 & 16 & -8 & 4 & -2 & 1 \\ 729 & -243 & 81 & -27 & 9 & -3 & 1 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 720 & 0 & 0 & 0 & 0 & 0 & \\ 0 & 240 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 144 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 48 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 360 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 60 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R = \begin{bmatrix} 1 & -6 & 15 & -20 & 15 & -6 & 1 \\ 1 & -4 & 5 & 0 & -5 & 4 & -1 \\ -1 & 12 & -39 & 56 & -39 & 12 & -1 \\ -1 & 8 & -13 & 0 & 13 & -8 & 1 \\ 2 & -27 & 270 & -490 & 270 & -27 & 2 \\ 1 & -9 & 45 & 0 & -45 & 9 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

Four-way:

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 8 & 4 & 2 & 1 \\ -8 & 4 & -2 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & -2 & 4 & -8 \\ 1 & 2 & 4 & 8 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

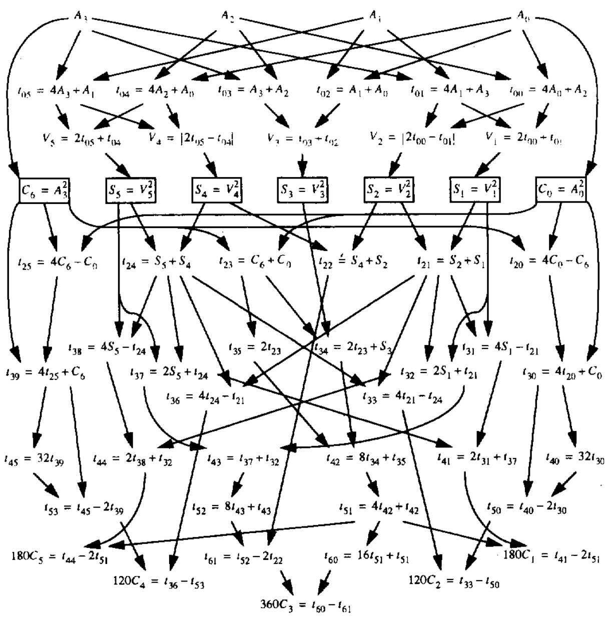

Fig. 4. The four-way method, $O(w^{1.404})$.

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 64 & -32 & 16 & -8 & 4 & -2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -2 & 4 & -8 & 16 & -32 & 64 \\ 1 & 2 & 4 & 8 & 16 & 32 & 64 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 180 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 120 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 360 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 120 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 180 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -180 & 6 & 2 & -80 & 1 & 3 & -180 \\ -510 & 4 & 4 & 0 & -1 & -1 & 120 \\ 1530 & -27 & -7 & 680 & -7 & -27 & 1530 \\ 120 & -1 & -1 & 0 & 4 & 4 & -510 \\ -180 & 3 & 1 & -80 & 2 & 6 & -180 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

This last solution was implemented as shown in the rather busy Fig. 4.

Except for its complexity, it is much the same as the other methods.

The computation of the S_3 term requires two extra bits and S_1, S_2, S_4, and S_5 each require 4.

The four-way method is asymptotically $w^{\log 7/\log 4} \approx w^{1.404}$ and is approximated on the HP-9000/720 by the formula

$$T_{S4}(w) = 120.79w^{\log 7/\log 4} - 858.7w + 976623.$$

$$x \in \{1, \infty, 0, 2, \frac{1}{2}, -2, -\frac{1}{2}, 3, \frac{1}{3}\}$$

$$\begin{bmatrix} V_8 \\ V_7 \\ V_6 \\ V_5 \\ V_4 \\ V_3 \\ V_2 \\ V_1 \\ V_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 16 & 8 & 4 & 2 & 1 \\ 16 & -8 & 4 & -2 & 1 \\ 81 & 27 & 9 & 3 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 9 & 27 & 81 \\ 1 & -2 & 4 & -8 & 16 \\ 1 & 2 & 4 & 8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} A_4 \\ A_3 \\ A_2 \\ A_1 \\ A_0 \end{bmatrix} \quad \& \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 256 & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 256 & -128 & 64 & -32 & 16 & -8 & 4 & -2 & 1 \\ 6561 & 2187 & 729 & 243 & 81 & 27 & 9 & 3 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 9 & 27 & 81 & 243 & 729 & 2187 & 6561 \\ 1 & -2 & 4 & -8 & 16 & -32 & 64 & -128 & 256 \\ 1 & 2 & 4 & 8 & 16 & 32 & 64 & 128 & 256 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_8 \\ C_7 \\ C_6 \\ C_5 \\ C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_8 \\ S_7 \\ S_6 \\ S_5 \\ S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} = \begin{bmatrix} V_8^2 \\ V_7^2 \\ V_6^2 \\ V_5^2 \\ V_4^2 \\ V_3^2 \\ V_2^2 \\ V_1^2 \\ V_0^2 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2100 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 12600 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8400 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 12600 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8400 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 12600 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2100 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} C_8 \\ C_7 \\ C_6 \\ C_5 \\ C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \Sigma^{-1} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2100 & -42 & -2 & 3 & 700 & 1 & 1 & -21 & -9100 \\ -54600 & 588 & 76 & -24 & -14000 & -24 & -29 & 483 & 1050 \\ -700 & 770 & -102 & -47 & -9100 & -5 & 10 & 154 & 146300 \\ -219450 & -2079 & 97 & 102 & 59500 & 102 & 97 & -2079 & -2194500 \\ 146300 & 154 & 10 & -5 & -9100 & -47 & -102 & 770 & -700 \\ 1050 & 483 & -29 & -24 & -14000 & -24 & 76 & 588 & 54600 \\ -9100 & -21 & 1 & 1 & 700 & 3 & -2 & -42 & -2100 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_8 \\ S_7 \\ S_6 \\ S_5 \\ S_4 \\ S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix}$$

Fig. 5. The five-way equations, $O(w^{1.365})$.

Figure 5 shows the equations necessary to implement a five-way method. This method would be asymptotically $w^{\log 9/\log 4} \approx w^{1.365}$ but has not yet been implemented. Out of a sense of mercy to the reader, I won't even *attempt* a graph of the solution.

X. The FFT Multiply

The best known method of squaring or multiplying integers is the FFT multiply discovered by Schönhage and Strassen [5]. That is, this is the method with the best known *asymptotic* behavior of $O(w \log w \log\log w)$.

I shall not present this method here in any great detail but I recommend the excellent description in Aho, Hopcroft, and Ullman [1, ch. 7].

In brief, the FFT multiply of order k splits a number into 2^{k-1} parts for increasingly large values of k, performs an order k FFT on it, squares (or multiplies) those 2^k elements, and performs an inverse FFT on the result. All arithmetic is performed in a ring with a solution to the equation $\omega^{2^{k-1}} = -1$.

If the ring chosen is the integers mod some base, b, the equation becomes $\omega^{2^{k-1}} = -1 \bmod (b)$ and b must be larger than the result coefficients.

It is common to choose $b = 2^{m2^{k-1}} + 1$ with $m2^{k-1} > 2w \log w / \log 2$ so that $\omega = 2^m$, although other choices are possible. (This transform is also known as a Number Theoretic Transform or NTT.)

The FFT multiply has the advantages that: it is easily extensible to any value of k; both the initial and final transformations can be made to take place in stages that have identical

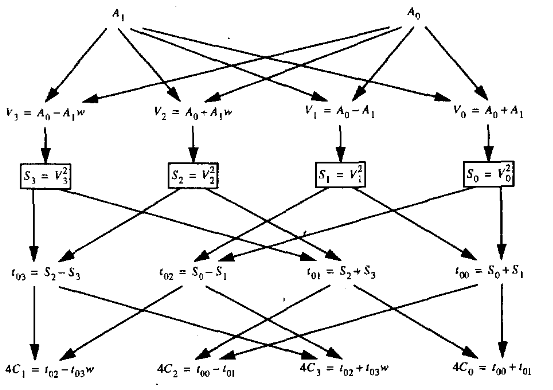

Fig. 6. The FFT-2 method.

connectivity; and, since all the divides are by powers of 2, they can be done as shifts.

But, it has a major disadvantage in that, since the ring must be large enough to represent the result coefficients, all of the basic operations are *twice* as large as in a corresponding Toom–Cook style method.

Therefore, the overhead involved can be large compared to the k-way methods.

For example, with $\omega = 2^m$ and $b = 2^{2m} + 1$, we have $\omega^2 = -1 \bmod (b)$ and the transformation for an FFT-2 would look like

$$\begin{bmatrix} V_3 \\ V_2 \\ V_1 \\ V_0 \end{bmatrix} = \begin{bmatrix} A_0 + A_1 \\ A_0 - A_1 \\ A_0 + A_1\omega \\ A_0 - A_1\omega \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & \omega \\ 1 & -\omega \end{bmatrix} \begin{bmatrix} A_1 \\ A_0 \end{bmatrix}$$

and

$$\begin{bmatrix} \omega & -1 & -\omega & 1 \\ -\omega & -1 & \omega & 1 \\ -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix} = \begin{bmatrix} V_3^2 \\ V_2^2 \\ V_1^2 \\ V_0^2 \end{bmatrix}$$

with the solution

$$\begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 0 & 4 & 0 \\ 0 & 4 & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & -1+\omega & -\omega \\ 1 & 1 & -1 & -1 \\ 1 & 0 & -1-\omega & \omega \end{bmatrix} \begin{bmatrix} S_3 \\ S_2 \\ S_1 \\ S_0 \end{bmatrix}.$$

This method is shown in Fig. 6.

XI. Running Time of FFT-k Methods

Since b must be large enough to contain the result coefficients, each of the four squares in an FFT-2 method must be done with a method which is slightly *larger* than the entire operand! Therefore, this method is not useful for constructing larger squares.

The first method that is useful in that sense is the FFT-3. The operand is divided into four parts and eight squares are performed in a ring that is slightly larger than half the operand. Therefore, this method is asymptotically $O(w^{\log 8/\log 2}) = O(w^3)$!

In general, an FFT-k will have 2^k squares in a ring slightly larger than 2^{k-2} times smaller than the operand. The asymptotic behavior is therefore $O(w^{k/(k-2)}) = O(w^{1+2/(k-2)})$.

A timing formula for an FFT-k may be derived in the same way as the previous methods as follows:

$$T_k(w_0) = t_0$$

$$T_k(2^{(r+1)(k-2)}w_0) = 2^k T_k(2^{r(k-2)}w_0) + c_1 k 2^{rk} w_0 + c_0$$

$$T_k(2^{k-2}w_0) = 2^k t_0 + c_1 k w_0 + c_0$$

$$T_k(2^{2(k-2)}w_0) = 2^{2k} t_0 + 2c_1 k w_0 2^k + (2^k + 1)c_0$$

$$T_k(2^{3(k-2)}w_0) = 2^{3k} t_0 + 3c_1 k w_0 2^{2k} + (2^{2k} + 2^k + 1)c_0$$

$$\cdots$$

$$T_k(2^{r(k-2)}w_0) = 2^{rk} t_0 + c_1 k w_0 r 2^{rk} + \left(\frac{2^{rk}-1}{2^k-1}\right)c_0.$$

Rearranging terms gives

$$T_k(2^{r(k-2)}w_0) = \left(t_0 + 2^{-k}c_1 k w_0 r + \frac{c_0}{2^k-1}\right)2^{rk} - \frac{c_0}{2^k-1}$$

or, with $w = 2^{r(k-2)}w_0$, becomes

$$T_k(w) = \left(t_0 + \frac{c_1 k w_0 \log\frac{w}{w_0}}{2^k(k-1)\log 2} + \frac{c_0}{2^k-1}\right) \cdot \left(\frac{w}{w_0}\right)^{\frac{k}{(k-2)}} - \frac{c_0}{2^k-1}.$$

While this timing formula is much like that for the Toom–Cook style methods, there is a sense in which the exponent of the dominant term shrinks more slowly with increasing k relative to the growth in overhead.

XII. Some Actual Results

Figure 7 compares the squaring methods discussed here and Fig. 8 compares the multiples. (The FFT's only appear in the squaring results since, as it will soon become clear, they do not show very much promise.) Time is in cycles at 50 MHz on the HP-9000/720 and size is in 32-bit words. For scale, this shows that one can square a 360,000-bit number in one second on this machine.

As a log-log plot, Fig. 7 diminishes the roughly 3-to-1 differences between the various methods. We can illustrate those differences better by rescaling the data by the least squares approximation to $T_2(w)$.

Figure 9 shows a log-linear plot of $T_i(w)/T_2(w)$. We can now see that the k-way methods are faster as a class than the FFT-k methods for $w < 1.1 \times 10^6$ words (37×10^6 bits) but we still don't have a clear idea of the asymptotic behavior.

For that, the log of the ratio of the time to the basis time over $\log k$, which is

$$\frac{\log\frac{T_k(w)}{T_k(w/k)}}{\log k}$$

for the k-way methods and

$$\frac{\log\frac{T_k(w)}{T_k(w/2^{k-2})}}{(k-2)\log 2}$$

for the FFT-k methods, gives a good approximation to the

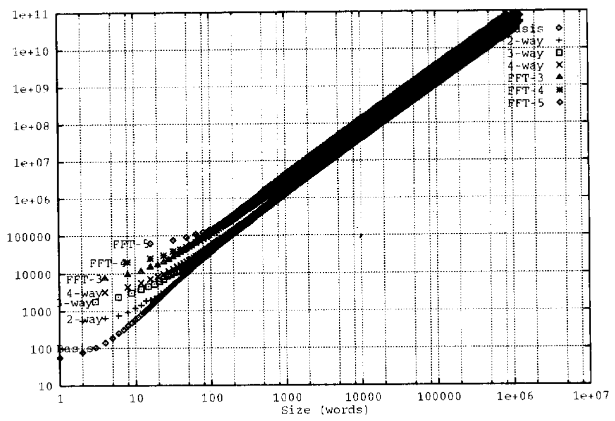

Fig. 7. Squaring methods: Time (cycles) vs. size (words).

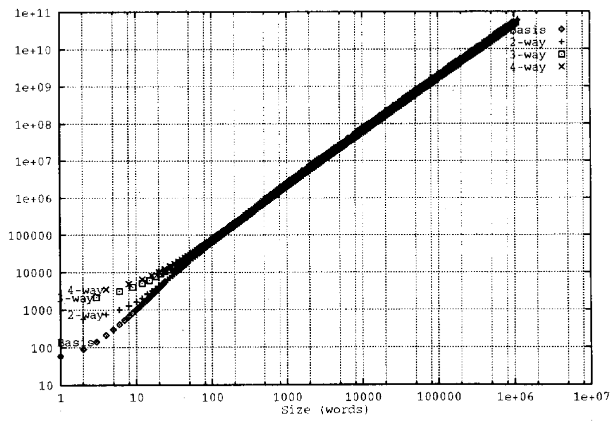

Fig. 8. Multiply methods: Time (cycles) vs. size (words).

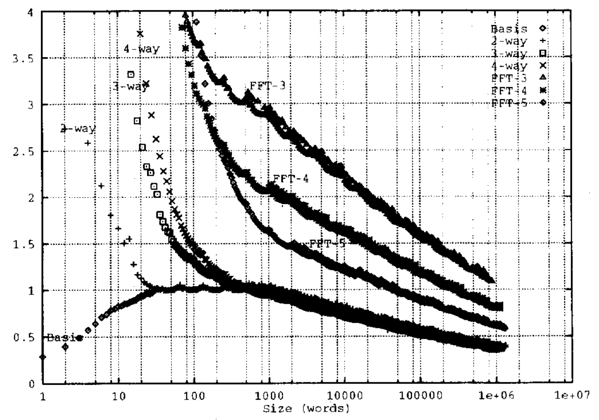

Fig. 9. Squaring methods: $T_{Si}(w)/T_{S2}(w)$ vs. size (words).

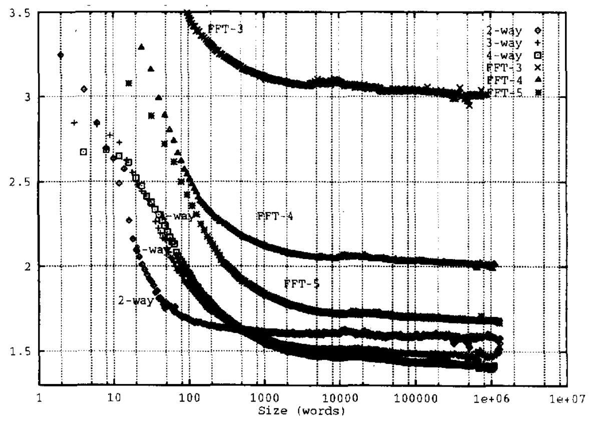

Fig. 10. Squaring methods: $\log(T/T_0)/\log k$ vs. size (words).

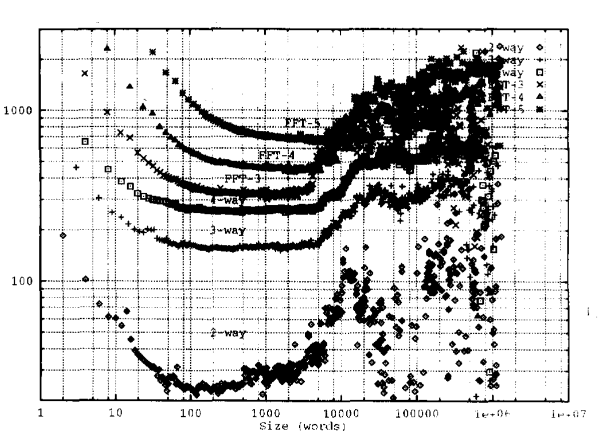

Fig. 11. Squaring methods: $(T - mT_0)/kw$ vs. size (words).

exponent in the dominant term and will illustrate how quickly each approaches its asymptote. Figure 10 shows this.

Another graph of interest is Fig. 11. This shows a term which is asymptotically equal to the first overhead term.

Besides showing that the overhead for the k-way methods is much less than for the FFT-k's, Fig. 11 seems to show the beginnings of cache effects and suggests that the k-way methods also have slightly better locality of reference than the FFT-k's.

None of the FFT-k methods tried have any chance of being best. An FFT-6, asymptotically $O(w^{\log 64/\log 16}) = O(w^{1.5})$, will also be too slow. An FFT-7, $O(w^{\log 128/\log 32}) = O(w^{1.4})$, is asymptotically faster than the four-way method but the five-way method, $O(w^{\log 9/\log 5}) = O(w^{1.365})$, very likely has much less overhead and will win first.

In spite of this behavior for small orders, it is possible that the FFT-k methods will eventually win for some $w \gg 1{,}175{,}000$ words.

The overhead in an FFT-k is known to grow as something like $O(k2^k)$. A Toom–Cook style k_{TC}-way method is asymptotically comparable to an FFT-k_{SS} method when $k_{TC} \sim 2^{\frac{k_{SS}-2}{2}}$. If the overhead of the k-way methods, dominated by a

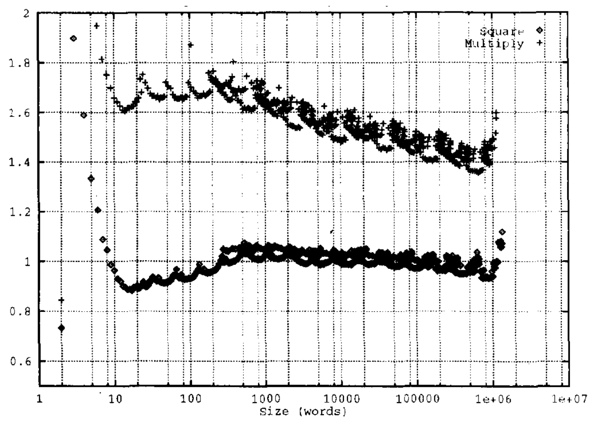

Fig. 12. T/T_{Best} vs. size (words).

matrix inversion, grows as fast as $O(k_{TC}^3) \sim O\left(2^{\frac{3}{2}(k_{SS}-2)}\right)$, the FFT-$k$'s may eventually win.

XIII. Is Squaring Faster than Multiplying

Squaring is a special case of multiplying. Therefore, we have trivially that

$$T_{\text{square}} \leq T_{\text{multiply}}.$$

But, in all the methods presented here, squaring involves *strictly* less work than multiply. Further, most of this savings is in the overhead and, in the limit of large numbers, virtually all of the time spent in a multiply is in the overhead.

Therefore, we are led to ask the question: Is it possible that there are squaring methods that are *of an order* faster than any multiply methods?

The answer is unfortunately: no.

While it is possible that we may discover some method of squaring that is strictly faster than any *existing* multiply method, any squaring method can be used to construct a multiply method that is no more than a constant slower.

A simple proof is

$$XY = ((X+Y)^2 - (X^2+Y^2))/2,$$

which, assuming that add and shift are no worse than $O(n)$, shows that

$$T_{\text{multiply}} \leq 3T_{\text{square}} + O(n).$$

Karatsuba presented a better method

$$XY = ((X+Y)^2 - (X-Y)^2)/4,$$

which gives a multiply in two squares and $O(n)$. Therefore,

$$T_{\text{multiply}} \leq 2T_{\text{square}} + O(n).$$

Figure 12 compares the best squaring times and the best multiply times.

The data in Fig. 12 are scaled by the formula

$$T_{\text{best}} \approx 200.377w^{1.40059-(6.51172-3.74336/\ln(w))/\ln(w)},$$

which roughly approximates the best squaring times.

XIV. Conclusions, Speculations, Etc.

It would appear that many of the simpler methods of multiplying are best all the way out to quite large numbers. Certainly, into the tens of millions of bits. Possibly, much farther.

In spite of the fact that squaring is fundamentally faster than multiply, it can be no better than a constant faster in the limit of large numbers.

It is still possible that the Schönhage and Strassen method will win in the end in spite of a slight asymptotic disadvantage. This would be a useful area for further work.

A closely related area is that of what is the minimum amount of overhead possible in Toom–Cook style methods. Different assumptions about the cost of overhead might lead to different trade-offs.

For example, in the approach taken in this paper, what are the best choices of $x = \pm\frac{p}{q}$ so as to minimize add, shift, and divide overhead?

References

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Reading, MA: Addison-Wesley, 1974, ch. 7.

[2] S. A. Cook, "On the minimum computation time of functions," thesis, Harvard Univ., May 1966, pp. 51–77.

[3] A. Karatsuba and Yu Ofman, "Multiplication of multidigit numbers on automata," *Soviet Phys. Doklady*, vol. 7, no. 7, pp. 595–596, Jan. 1963.

[4] D. E. Knuth, *The Art of Computer Programming*, vol. 2, second ed. Reading, MA: Addison-Wesley, 1981, ch. 4, section 3.3, pp. 278–301.

[5] A. Schönhage and V. Strassen, *Computing*, vol. 7, pp. 281–292, 1971 (in German).

[6] M. Shand, Digital Equipment Corp., Paris Res. Lab., personal communication, July 1993.

[7] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," *Soviet Math.*, vol. 3, pp. 714–716, 1963.

[8] D. Zuras, "On squaring and multiplying large integers," in *11th IEEE Symp. Comput. Arithmetic*, pp. 260–271.

Dan Zuras received the B.S. degree in mathematics from Stanford University in 1975.

He is now a Research and Development Engineer at Hewlett-Packard in Cupertino, CA, in the Open Systems Software Division. He has been with HP for 17 years working primarily on floating-point applications in both hardware and software. Outside of HP, Dan is a founding member of Group-70, a nonprofit organization that is building a 70″ telescope.