Multiplication by a Constant is Sublinear

Vassil Dimitrov
ATIPS Labs, CISaC
University of Calgary
2500 University drive NW
T2N 1N4, Calgary, AB
Canada

Laurent Imbert
LIRMM, Univ. Montpellier 2, CNRS
Montpellier, France
& ATIPS Labs, CISaC
University of Calgary
Canada

Andrew Zakaluzny
ATIPS Labs
University of Calgary
2500 University drive NW
T2N 1N4, Calgary, AB
Canada

***Abstract*— This paper explores the use of the double-base number system (DBNS) for constant integer multiplication. The DBNS recoding scheme represents integers – in this case constants – in a multiple-radix way in the hope of minimizing the number of additions to be performed during constant multiplication. On the theoretical side, we propose a formal proof which shows that our recoding technique diminishes the number of additions in a sublinear way. Therefore, we prove Lefèvre's conjecture that the multiplication by an integer constant is achievable in sublinear time. In a second part, we investigate various strategies and we provide numerical data showcasing the potential interest of our approach.**

I. Introduction

Multiplication by an integer constant has many applications; for example in digital signal processing, image processing, multiple precision arithmetic, cryptography and in the design of compilers. In certain applications, like the discrete cosine transform (DCT), the implementation of integer constant multiplications is the bottleneck as it largely determines the speed of the entire process. Therefore, it is imperative that multiplications by integer constants in these high throughput applications are optimized as much as possible.

The problem of optimizing multiplication by constants is that not all constants behave the same. In other words, a technique for optimizing the multiplication by the specific constant c may not optimize the multiplication by another constant c'. Therefore, finding solutions that optimize multiplication for most constants over a specified range is an important problem which has been sought after by many authors.

Given an integer constant c, the goal is to find a program which computes $c \times x$ with as few operations as possible. A basic complexity model is to count the number of additions only, assuming that multiplications by powers of 2 are free (left shifts). The number of additions is highly dependent upon the number of non-zero digits in the representation of the constant c. For example, if c is a n-bit constant, then one needs on average $n/2$ additions with the double-and-add method, sometimes referred to as the binary method; and $n/3$ additions if c is expressed in the Canonic Signed Digit (CSD) representation: a variant of Booth's recoding technique [1]. Other classes of algorithms based on cost functions or the search for patterns in the binary expansion of c are described in the papers from Bernstein [2], Lefèvre [3], and Boullis and Tisserand [4]. These algorithms give very good results in practice at the cost of quite expensive computations. Moreover, the asymptotic complexities of the generated programs are difficult to analyze.

In this paper, we propose several variants of an algorithm based on integer recoding, where the constant c is represented as a sum of mixed powers of two coprime bases; e.g. 2 and 3 or 2 and 5. This very sparse representation scheme, called Double-Base Number System (DBNS), has been used in digital signal processing [5] and cryptographic [6] applications with great success. By restricting the largest exponent of the second base to some well chosen bound, we obtain a sublinear constant multiplication algorithm; the resulting program requires $O(\log c/\log\log c)$ additions and/or subtractions. Our numerical experiments confirm the asymptotic sublinear behavior, even for relatively small numbers (32 to 64 bits).

This paper is organized as follows: We define the problem and present some previous works in Section II and III. In Section IV, we introduces the Double-Base Number System. We present our new algorithm and the proof of sublinearity in Section V. In Section VI, we propose different heuristics and several numerical experiments.

II. Problem Definition

In many applications, multiplication of integers is simply done with an all purpose multiplier. (On recent general purpose processors, it is sometimes even faster to use the floating-point multiplier to perform an integer multiplication.) As mentioned before, many applications require a high throughput of constant integer multiplications, and would benefit from a customized integer multiplier suited to the specific constants used. In essence, multiplication is a series of shifts and additions, but in some cases, it might be a good idea to allow subtractions. The central point of the constant multiplication problem is to minimize the total number of additions/subtractions required for each integer multiplication.

In the following, we shall use $x \ll k$ to denote the value obtained when the variable x is shifted to the left by k places (bits); i.e., the value $x \times 2^k$. For simplicity, we consider a complexity model where the cost of shifts is ignored. Note that this widely assumed assumption might not correspond to practical reality, especially in the context of multiple precision arithmetic. For hardware implementations, however, this assumption seems reasonable. Therefore, in order to simplify

the presentation, we only take into account the number of additions and/or subtractions. We also assume that addition and subtraction have the same speed and cost. Hence, we will sometimes refer to the number of additions, or even to the number of operations, by which terminology we include subtractions.

Let us start with a very small example. We want compute the product of the unknown integer x by the integer constant $c = 151 = 10010111_2$. Using a naive approach, we can shift each non-zero bits of c to the left to its corresponding position and sum them all together. If $c = \sum_{i=0}^{n-1} c_i 2^i$, this is equivalent to

$$c \times x = \sum_{i=0}^{n-1} c_i 2^i \times x = \sum_{i=0}^{n-1} c_i x 2^i. \quad (1)$$

For example, using the $\ll$ notation, we have

$$151x = (x \ll 7) + (x \ll 4) + (x \ll 2) + (x \ll 1) + x.$$

Such a constant multiplier by $c = 151$ would require 4 additions. In the general case, the number of additions is equal to the Hamming weight (i.e., the number of non-zero digits) of c minus 1. In the next section, we present some more sophisticated methods to perform a constant multiplication.

III. Previous Works

A widely used approach to reduce the number of non-zero digits, and therefore the number of additions, is to consider variants of Booth's recoding [1] technique, where long strings of ones are replaced with equivalent strings with many zeros. An improvement to the multiplication example presented above can be achieved if we represent our constant using signed digits. In the so-called Signed Digit (SD) binary representation, the constant c is expressed in radix 2, with digits in the set $\{\overline{1} = -1, 0, 1\}$. This recoding scheme is clearly redundant. A number is said to be in the Canonical Signed Digit (CSD) format if there are no consecutive non-zero digits in its SD representation. In this case, it can be proved that the number of non-zero digits is minimal among all SD representations [7]. For a n-bit[1] constant, it is bounded by $(n+1)/2$ in the worst case, and is roughly equal to $n/3$ on average (the exact value is $n/3 + 1/9$; see [8]). For example, since $151 = (10010111)_2 = (1010\overline{1}00\overline{1})_2$, the product $c \times x$ reduces to 3 additions:

$$151x = (x \ll 7) + (x \ll 5) - (x \ll 3) - x.$$

Representing the constant in a different format is known as a direct recoding method. The double-base encoding scheme we present in Section IV also falls into this type. Several other constant multiplication methods have been proposed in the literature. Solutions based on genetic algorithms such as evolutionary graph generation seem to provide very poor results. A drawback of typical recoding methods is the impossibility to reuse intermediate values. The first proposed method which takes advantage of intermediate computations is due to Bernstein [2], which is implemented in the GNU Compiler Collection (GCC) [9]. Methods based on pattern search in the binary or SD representation of the constant have also been widely studied. For example, in 2001, Lefèvre proposed an algorithm [3] to efficiently multiply a variable integer x by a given set of integer constants. This algorithm can also be used to multiply x by a single constant. Using similar techniques, Boullis and Tisserand [4] recently proposed improvements in the case of multiplication by constant matrices; a detailed presentation of all the methods mentioned above can be found in their respective papers with the corresponding references. Methods based on cost functions or pattern search generates optimized results at the expense of large computational time. In addition, lower bounds on the maximum left shifts must be considered carefully to minimize overflow – this to the detriment of the optimization. Another interesting method was proposed by MacLeod and Dempster [10] in 1999. It relies on graph generation, and again requires immense computational time as well as large memory requirements with the benefit of greatly optimized results.

[1] In the standard binary representation.

IV. The Double-Base Number System

In this section, we present the main properties of the double-base number system, along with some numerical results to provide the reader with some intuitive ideas about this representation scheme. We have intentionally omitted the proofs of previously published results. The reader is encouraged to check the references for further details.

We will need the following definitions.

Definition 1 (S-integer): Given a set of primes S, an S-integer is a positive integer whose prime factors all belong to S.

Definition 2 (Double-Base Number System): Given p, q, two distinct prime integers, the double-base number system (DBNS) is a representation scheme into which every positive integer n is represented as the sum or difference of distinct $\{p, q\}$-integers, i.e., numbers of the form $p^a q^b$.

$$n = \sum_{i=1}^{\ell} s_i \, p^{a_i} q^{b_i}, \quad (2)$$

with $s_i \in \{-1, 1\}$, $a_i, b_i \geq 0$ and $(a_i, b_i) \neq (a_j, b_j)$ for $i \neq j$.

The size, or length, of a DBNS expansion is equal to the number ℓ of terms in (2). In the following, we will only consider bases $p = 2$ and $q \in \{3, 5, 7\}$.

Whether one considers signed ($s_i = \pm 1$) or unsigned ($s_i = 1$) expansions, this representation scheme is highly redundant. Indeed, if one considers unsigned double-base representations (DBNR) only, with bases 2 and 3, then one can prove that 10 has exactly 5 different DBNR; 100 has exactly 402 different DBNR; and 1000 has exactly 1295579 different DBNR. The following theorem holds.

Theorem 1: Let n be a positive integer and let q be a prime > 2. The number of unsigned DBNR of n with bases 2 and q is given by $f(1) = 1$, and for $n \geq 1$

$$f(n) = \begin{cases} f(n-1) + f(n/q) & \text{if} \quad n \equiv 0 \pmod{q}, \\ f(n-1) & \text{otherwise.} \end{cases} \tag{3}$$

Remark: The proof consists of counting the number of solutions of the diophantine equation $n = h_0 + qh_1 + q^2h_2 + \cdots + q^th_t$, where $t = \lfloor \log_q(n) \rfloor$ and $h_i \geq 0$.

Not only this system is highly redundant, but it is also very sparse. Probably, the most important theoretical result about the double-base number system is the following theorem from [11], which gives an asymptotic estimate for the number of terms one can expect to represent a positive integer.

Theorem 2: Every positive integer n can be represented as the sum (or difference) of at most $O\left(\log n / \log\log n\right)$ $\{p, q\}$-integers.

The proof is based on Baker's theory of linear forms of logarithms and more specifically on the following result by R. Tijdeman [12].

Theorem 3: There exists an absolute constant C such that there is always a number of the form p^aq^b in the interval $\left[n - n/(\log n)^C, n\right]$.

Theorem 1 tells us that there exists very many ways to represent a given integer in DBNS. Some of these representations are of special interest, most notably the ones that require the minimal number of $\{p, q\}$-integers; that is, an integer can be represented as the sum of m terms, but cannot be represented with $(m-1)$ or fewer terms. These so-called canonic representations are extremely sparse. For example, with bases 2 and 3, Theorem 1 tells us that 127 has 783 different unsigned representations, among which 6 are canonic requiring only three $\{2, 3\}$-integers. An easy way to visualize DBNS numbers is to use a two-dimensional array (the columns represent the powers of 2 and the rows represent the powers of 3) into which each non-zero cell contains the sign of the corresponding term. For example, the six canonic representations of 127 are given in Table I.

Finding one of the canonic DBNS representations in a reasonable amount of time, especially for large integers, seems to be a very difficult task. Fortunately, one can use a greedy approach to find a fairly sparse representation very quickly. Given $n > 0$, Algorithm 1 returns a signed DBNR for n.

Although Algorithm 1 sometimes fails in finding a canonic representation (the smallest example is 41; the canonic representation is $32 + 9$, whereas the algorithm returns $41 = 36 + 4 + 1$) it is very easy to implement and it guarantees a representation of length $O\left(\log n / \log\log n\right)$.

The complexity of the greedy algorithm mainly depends on the complexity of step 3: finding the best approximation of n of the form p^aq^b. An algorithm based on Ostrowski's number system was proposed in [13]. It is possible to prove that its complexity is $O(\log\log n)$. (The algorithm proposed in [13] focuses on base 2 and 3 but the results extend to bases p, q.) Since Algorithm 1 finishes in $O(\log n / \log\log n)$ iterations, its overall complexity is thus optimal in $O(\log n)$. Another solution for Step 3 was recently proposed by Doche and Imbert in [14]; it uses lookup tables containing the binary representations of some powers of q and can be implemented very quickly, even for large numbers.

Algorithm 1 Greedy algorithm

INPUT: A positive integer n

OUTPUT: The sequences $(s_i, a_i, b_i)_{i \geq 0}$ s.t. $n = \sum_i s_i p^{a_i} q^{b_i}$ with $s_i \in \{-1, 1\}$, $a_i, b_i \geq 0$ and $(a_i, b_i) \neq (a_j, b_j)$ for $i \neq j$

1: $s \leftarrow 1$ {To keep track of the sign}
2: while $n \neq 0$ do
3: Find the best approximation of n of the form $z = p^aq^b$
4: print (s, a, b)
5: if $n < z$ then
6: $s \leftarrow -s$
7: $n \leftarrow |n - z|$

V. SUBLINEAR CONSTANT MULTIPLICATION

In this core section, we propose a generic algorithm for constant multiplication that takes advantage of the sparseness of the double-base encoding scheme. Our algorithm computes a special DBNS representation of the constants, where the largest exponent of the second base q is restricted to an arbitrary (small) value B. It uses a divide and conquer strategy to operate on separate blocks of small sizes. For each block, it is possible to generate those specific DBNS representations using a modified version of the greedy algorithm, or to precompute and store them in a lookup table in a canonical form; i.e., a DBNS expansion with a minimal number of terms. We show that both approaches lead to sublinear constant multiplication algorithms.

Let us illustrate the algorithm on a small example. We express $c = 10599 = (10100101100111)_2$ in radix 2^7; that is, we split c in two blocks of 7 bits each. We obtain $c = 82 \times 2^7 + 103$ and we represent the "digits" 82 and 103 in DBNS with bases 2 and 3 using as few terms as possible, where the exponents of the second base $q = 3$ are at most equal to 2. We find that 82 can be written using two terms as $64 + 18$ and 103 using only three terms as $96 + 8 - 1$. (We have results which prove that these values are optimal). By sticking the two parts together, we obtain the representation given in Table II.

Using this representation, the product $c \times x$ is decomposed as follows:

$$\begin{aligned} x_0 &= (x \ll 8) \\ x_1 &= 3x_0 + (x \ll 5) \\ x_2 &= 3x_1 + (x \ll 13) + (x \ll 3) - x \end{aligned}$$

Since multiplications by 3 can be performed by a shift

TABLE I

THE SIX CANONIC UNSIGNED DBNR OF 127

$2^2 3^3 + 2^1 3^2 + 2^0 3^0 = 108 + 18 + 1$

	1	2	4
1	1		
3			
9		1	
27			1

$2^2 3^3 + 2^4 3^0 + 2^0 3^1 = 108 + 16 + 3$

	1	2	4	8	16
1					1
3	1				
9					
27			1		

$2^5 3^1 + 2^0 3^3 + 2^2 3^0 = 96 + 27 + 4$

	1	2	4	8	16	32
1			1			
3						1
9						
27	1					

$2^3 3^2 + 2^1 3^3 + 2^0 3^0 = 72 + 54 + 1$

	1	2	4	8
1	1			
3				
9				1
27		1		

$2^6 3^0 + 2^1 3^3 + 2^0 3^2 = 64 + 54 + 9$

	1	2	4	8	16	32	64
1							1
3							
9	1						
27		1					

$2^6 3^0 + 2^2 3^2 + 2^0 3^3 = 64 + 36 + 27$

	1	2	4	8	16	32	64
1							1
3							
9			1				
27	1						

TABLE II

A DBNS REPRESENTATION OF $c = 10599$ OBTAINED USING TWO BLOCKS OF 7 BITS EACH

	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}
3^0	-1			1										1
3^1						1								
3^2									1					
	103							82						

followed by an addition, the resulting sequence of shift-and-add becomes:

$$x_0 = (x \ll 8)$$
$$x_1 = ((x_0 \ll 1) + x_0) + (x \ll 5)$$
$$x_2 = ((x_1 \ll 1) + x_1) + (x \ll 13) + (x \ll 3) - x$$

Let us give a formal description of the algorithm outlined in the previous example. We express c in DBNS as

$$c = \sum_{j=0}^{b_{max}} \sum_{i=0}^{a_{max}} c_{i,j} 2^i q^j, \tag{4}$$

with digits $c_{i,j} = \{\overline{1}, 0, 1\}$. Algorithm 2 below can be used to compute $c \times x$. We remark that each step of the algorithm requires a multiplication by q. It is therefore important to select the second base q such that the multiplication by q only requires a single addition; i.e., with $q = 3$, we have $3x = (x \ll 1) + x$. At the end, the result is given by $x_{b_{max}} = c \times x$. If ℓ is the length of the double-base expansion; i.e, the number of non-zero digits $c_{i,j}$ in (4), and if b_{max} is the largest exponent of q, then the overall number of additions is equal to

$$\ell + b_{max} - 1. \tag{5}$$

The goal is to set B, the predefined upper bound for the

Algorithm 2 Double-base constant multiplication

INPUT: A constant $c = \sum_{i,j} c_{i,j} 2^i 3^j$, with $c_{i,j} = \{\bar{1}, 0, 1\}$; and an integer x

OUTPUT: $c \times x$

1: $x_{-1} \leftarrow 0$
2: **for** $j = 0$ to b_{max} do
3: $\quad x_j \leftarrow q \times x_{j-1}$
4: $\quad x_j \leftarrow x_j + \sum_i c_{i,b_{max}-j}\ (x \ll i)$
5: Return $x_{b_{max}}$

exponents of q, such that the overall number of addition is minimal. (Note that b_{max} might be different from B, but $b_{max} \leq B$ holds.)

The following theorem shows that the number of additions required to evaluate the product $c \times x$ using our algorithm is sublinear in the size of the constant c.

Theorem 4: Let c be a positive integer constant of size n (in bits). Then, the multiplication by c can be computed in $O(n/\log n)$ additions.

Proof: We split c in blocks of size $n/\log n$ bits each. Clearly, one needs $\log n$ such blocks. Each block corresponds to an integer of size $n/\log n$ bits and can thus be represented in DBNS with exponents all less than $n/\log n$. In particular, we have $b_{max} \in O(n/\log n)$. From Theorem 2, we know that the number of non-zero digits in the DBNS representations of each block belongs to $O\left(n/(\log n)^2\right)$. Note that this is true whether one uses the greedy algorithm or considers a canonic double-base representation for each block. Therefore, since we have $\log n$ blocks, the number of non-zero digits in the DBNS representation of c belongs to $O\left(n/\log n\right)$. From (5), since $b_{max} \in O(n/\log n)$, we obtain that the overall complexity of the algorithm is in $O(n/\log n)$. □

VI. HEURISTICS

The algorithm presented in the previous section must be seen as a generic method; it must be adapted for each specific application. In particular, there are several parameters that need to be defined carefully: the second base q, the upper bound B on the exponents of q, and the size of the blocks.

As mentioned previously, when the block size is not too large, it is possible to store the canonic representations of each possible number in the range in a lookup table. The values given in Table III have been computed using exhaustive search. For integers of size up to 21 bits, we report the average number of non-zero digits in canonic double base representation with bases 2 and $q = 3$, maximum binary exponent $a_{max} = 19$ and ternary exponent $b_{max} \in \{3, 4, 5\}$. We also give the maximum number of terms ever needed in the corresponding range and the first integer x for which it occurs[2].

Let us analyze some options offered by our algorithm for a multiplication by a 64-bit constant c. For this example, we only consider bases 2 and 3 with maximum ternary exponent $b_{max} = 3$. If we split our 64-bit constant in 7 blocks of 10 bits, we know from Table III that, in the worst case, the DBNS decomposition of c will require $7 \times 3 = 21$ non-zero digits. Therefore, we know that the number of additions will be ≤ 23. If, instead, we consider four blocks of 16 bits each, we obtain 22 additions in the worst case. We remark that our worst case is similar to the average complexity if one uses the CSD representation ($64/3 \simeq 21.3333$). The average number of operations in our case is roughly equal $3.64 \times 4 + 2 \simeq 16.56$, which represents a speedup of about 22% compared to the CSD approach. This is the kind of analysis a programmer should do in order to define an appropriate set of parameters for his specific problem.

This approach is encouraging but is is possible to do better. In the algorithm presented so far, the blocks are all of the same size. This is to the detriment of efficiency since there might exist better way to split the constant than these regular splitting. In the next two sections, we present to families of heuristics that operates from right-to-left or from left-to-right.

A. Right-to-left splitting

The regular splitting does not exploit the nature of the binary representation of the constant c. The idea here is to try to avoid blocks with long strings of zeros and rather use these strings to split the constant c. For a predefined integer $m > 1$, we define a separating string as a string of m consecutive zeros. The heuristic works as follows: starting from the least significant bit, we look for the first separating string. If such a string is found at position j, the first block corresponds to the bits of c of weight less than j. We then look for the next separating string starting from the first non-zero digit of weight $> j$. Therefore, every block is an odd number and there is no need to store the canonic representations of even numbers in our lookup table. The separating size m must be carefully chosen in function of the size of c. If it is too small, there will be too many blocks and the overall number of additions will increase accordingly. Reversely, if it is too large, there might not be any such strings and we might end up with the entire constant c, for which we do not know a canonic DBNS representation. In the second case, the solution is to fix the largest block size (according to the amount of memory available for the lookup table) and to split the constant c either when we find a separating string or when we reach this largest block size. In Figures 1 and 2, we have plotted the average number of additions as a function of the largest block size for $m = 2, 3, 4, 5$, for $a_{max} = 19$ and $b_{max} = 3$, for 100000 random 32-bit and 64-bit constants. We also have similar plots for $b_{max} = 2, 4, 5$ but $b_{max} = 3$ seems to give the best results.

B. Left-to-right splitting

Another possible strategy is to start from the most significant bit of c and to look for the largest odd number of size less than the predefined largest block size. As previously, we impose that each block starts and finishes with a non-zero digit in order to store odd numbers only. This strategy might

[2]We also have similar data for $q = 5$ and $q = 7$.

TABLE III

NUMERICAL RESULTS FOR PARAMETERS $q = 3$, $a_{max} = 19$, $b_{max} = 3, 4, 5$

size	$b_{max} = 3$			$b_{max} = 4$			$b_{max} = 5$		
(in bits)	Avg	Max	at $x =?$	Avg	Max	at $x =?$	Avg	Max	at $x =?$
1	0.5	1	1	0.5	1	1	0.5	1	1
2	0.75	1	1	0.75	1	1	0.75	1	1
3	1.125	2	5	1.125	2	5	1.125	2	5
4	1.375	2	5	1.375	2	5	1.375	2	5
5	1.5625	2	5	1.5625	2	5	1.5625	2	5
6	1.71875	2	5	1.71875	2	5	1.71875	2	5
7	1.89844	3	77	1.84375	3	103	1.83594	3	103
8	2.10547	3	77	2.02734	3	103	1.96875	3	103
9	2.31836	3	77	2.23047	3	103	2.15234	3	103
10	2.51074	3	77	2.42773	3	103	2.34961	3	103
11	2.68408	4	1229	2.59863	4	1843	2.52881	3	103
12	2.86743	4	1229	2.75391	4	1843	2.67871	4	2407
13	3.06897	4	1229	2.92224	4	1843	2.81982	4	2407
14	3.27203	4	1229	3.10913	4	1843	2.97766	4	2407
15	3.46136	5	19661	3.29990	5	29491	3.15594	4	2407
16	3.64391	5	19661	3.47905	5	29491	3.33882	5	52889
17	3.83374	5	19661	3.64627	5	29491	3.50820	5	52889
18	4.03194	5	19661	3.81557	5	29491	3.66204	5	52889
19	4.22856	6	314573	3.99545	6	471859	3.81476	5	52889
20	4.44634	6	314573	4.20838	6	471859	3.99837	5	52889
21	4.67745	6	314573	4.41817	6	471859	4.19770	6	1103161

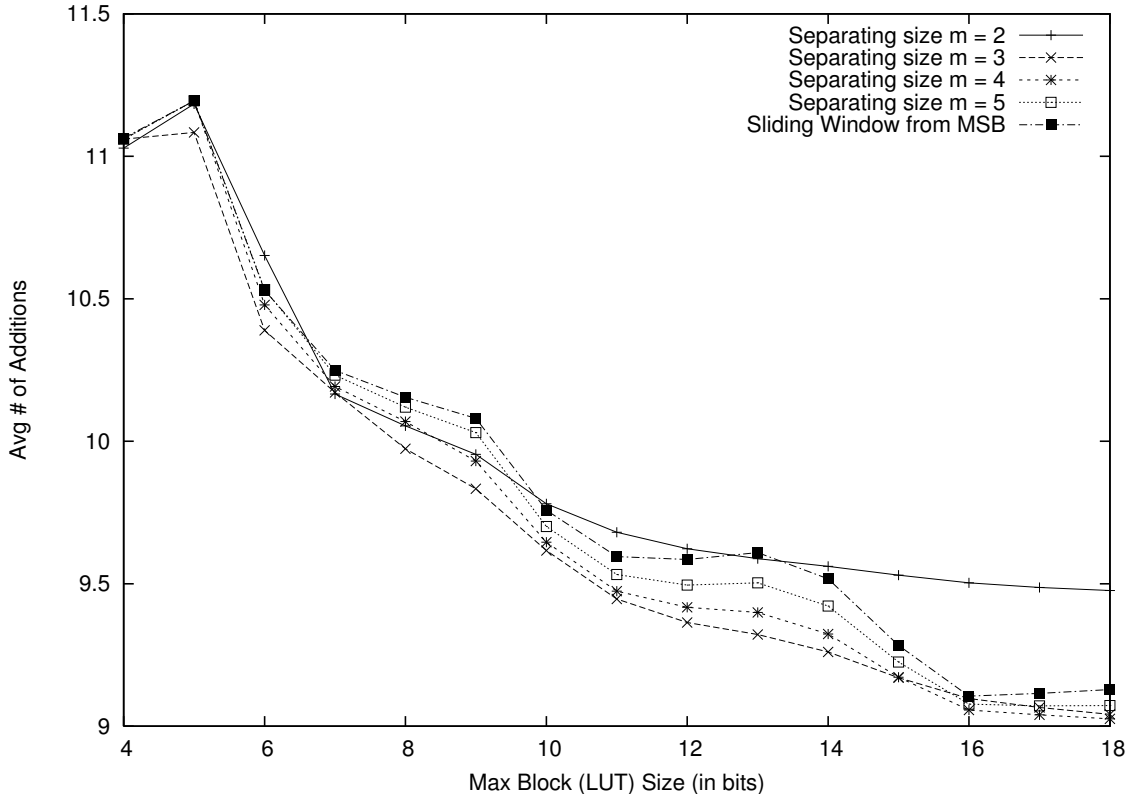

Fig. 1. Average number of additions for 100000 randomly chosen 32-bit constants, using bases 2 and 3, with $a_{max} = 19$ and $b_{max} = 3$

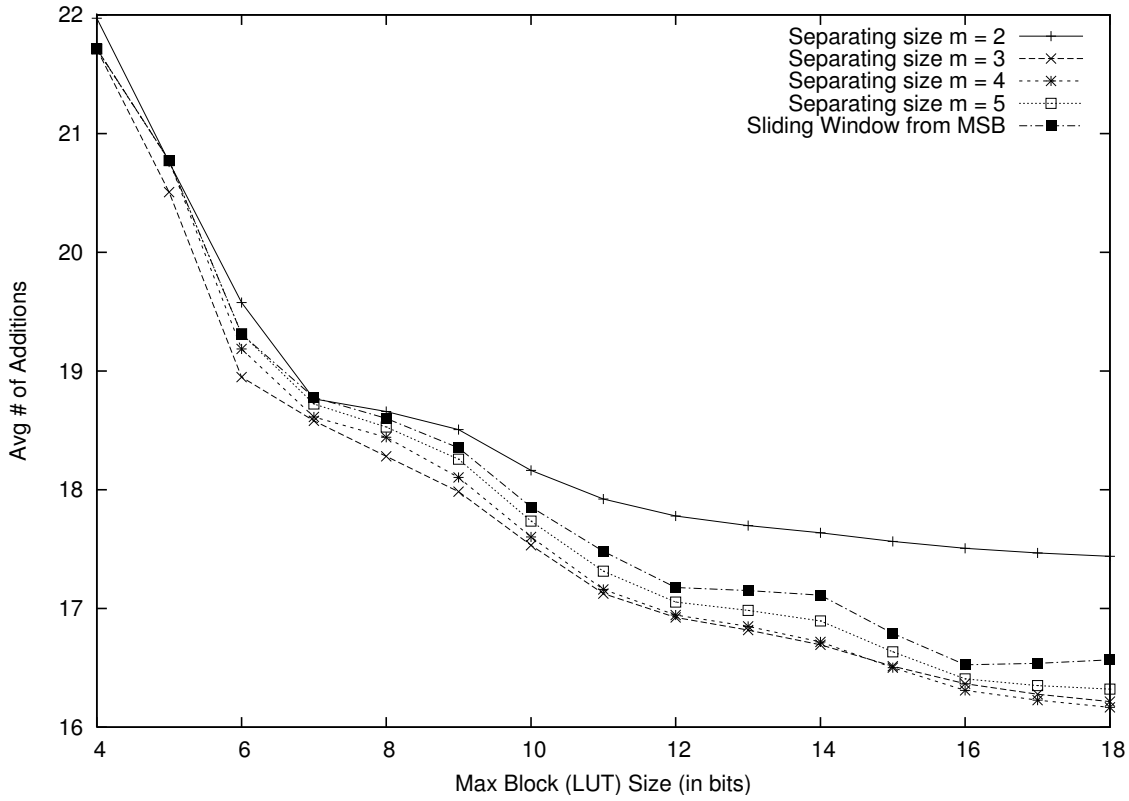

Fig. 2. Average number of additions for 100000 randomly chosen 64-bit constants, using bases 2 and 3, with $a_{max} = 19$ and $b_{max} = 3$

look optimal as it best exploit the precomputed values, but Figures 1 and 2 show that this is not the case.

C. Remarks

1) In Figures 1 and 2, we remark that for 32-bit and 64-bit constants, with lookup tables of reasonable size (10 to 12 input bits), the best results seems to be given for separating string of size $m = 3$.
2) In Table IV, we give the average number of additions and the worst case for 100000 randomly chosen 64-bit constants (with separating size $m = 3$). We remark that Lookup tables of 10 to 12 input bits lead to roughly 17 additions on average and 22 in the worst case. Using much larger lookup tables only provides small improvements. For 64-bit constants, lookup tables of 10 to 12 input bits seems to be a good choice. For 32-bit numbers, tables of size 8 to 10 input bits lead to < 10 additions on average and 13 in the worst case.
3) We have performed the same kind of experiments with second base $q = 5$ and $q = 7$. Bases 2 and 3 seem to provide the best results.
4) In terms of comparisons, our recoding algorithm requires more additions, both on average and in the worst case, than Boullis and Tisserand's algorithm [4] (using the graph heuristic strategy); which is the best known algorithm so far for multiplication by constant matrices. Using their approach for a single constant, one get about 13.5 additions on average and 19 in the worst case. This is not surprising since their approach based on pattern search generates very optimized results. However, the computational cost of our DBNS recoding algorithm, both in time and memory, is smaller, which might allow its use in compilers.

TABLE IV
AVERAGE AND WORST CASE NUMBER OF ADDITIONS FOR 64-BIT CONSTANTS

Max block size	Avg # add	Worst case
4	21.7133	31
5	20.5069	28
6	18.9489	24
7	18.5809	26
8	18.2813	25
9	17.9844	24
10	17.5323	22
11	17.1257	22
12	16.9249	23
13	16.818	22
14	16.694	21
15	16.5134	21
16	16.366	22
17	16.277	21
18	16.2151	21

5) Note that it is possible to reduce the average and worst case number of additions. Indeed, the canonic representations stored in the lookup tables we used for our experiments are not the "best" possible ones; i.e., among all the canonic representations for a given number, we do not necessarily store the representation with the smallest second exponent. By doing so, we can probably save some additions.

VII. Conclusions

In this paper, we proposed a new recoding algorithm for the constant multiplication problem. Our approach uses a divide and conquer strategy combined with the double-base number system. We proved that our approach leads to a sublinear algorithm. To our knowledge, this is the first sublinear algorithm for the constant multiplication problem. We illustrate the potential interest of our approach with several numerical experiments. The sequence of shifts-and-adds obtained with our algorithm is not as "good" as the sequences obtained with the best known methods based on pattern search or cost funcions. However, our DBNS-based generation algorithm requires much less computational effort than these optimal methods and it gives better results than the other direct recoding methods. A natural extension to the problem is the multiplication by constant vectors and matrices, where the high redundancy of the DBNS can certainly be exploited.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments. This work was partly funded by an NSERC strategic grant on the Canadian side and by an ACI grant on the French side.

References

[1] A. D. Booth, "A signed binary multiplication technique," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951, reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[2] R. Bernstein, "Multiplication by integer constants," *Software – Practice and Experience*, vol. 16, no. 7, pp. 641–652, jul 1986.

[3] V. Lefèvre, "Multiplication by an integer constant," INRIA, Research Report 4192, May 2001.

[4] N. Boullis and A. Tisserand, "Some optimizations of hardware multiplication by constant matrices," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1271–1282, Oct. 2005.

[5] V. S. Dimitrov, G. A. Jullien, and W. C. Miller, "Theory and applications of the double-base number system," *IEEE Transactions on Computers*, vol. 48, no. 10, pp. 1098–1106, Oct. 1999.

[6] V. Dimitrov, L. Imbert, and P. K. Mishra, "Efficient and secure elliptic curve point multiplication using double-base chains," in *Advances in Cryptology, ASIACRYPT'05*, ser. Lecture Notes in Computer Science, vol. 3788. Springer, 2005, pp. 59–78.

[7] G. W. Reitwiesner, "Binary arithmetic," *Advances in Computers*, vol. 1, pp. 231–308, 1960.

[8] R. I. Hartley, "Subexpression sharing in filters using canonic signed digit multipliers," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 43, no. 10, pp. 677–688, Oct. 1996.

[9] "GCC, the GNU compiler collection," http://www.gnu.org.

[10] A. G. Dempster and M. D. Macleod, "Constant integer multiplication using minimum adders," *IEE Proc. Circuits Devices Syst.*, vol. 141, no. 5, pp. 407–413, 1994.

[11] V. S. Dimitrov, G. A. Jullien, and W. C. Miller, "An algorithm for modular exponentiation," *Information Processing Letters*, vol. 66, no. 3, pp. 155–159, May 1998.

[12] R. Tijdeman, "On the maximal distance between integers composed of small primes," *Compositio Mathematica*, vol. 28, pp. 159–162, 1974.

[13] V. Berthé and L. Imbert, "On converting numbers to the double-base number system," in *Advanced Signal Processing Algorithms, Architecture and Implementations XIV*, ser. Proceedings of SPIE, vol. 5559. SPIE, 2004, pp. 70–78.

[14] C. Doche and L. Imbert, "Extended double-base number system with applications to elliptic curve cryptography," in *Progress in Cryptology, INDOCRYPT'06*, ser. Lecture Notes in Computer Science, vol. 4329. Springer, 2006, pp. 335–348.