Converting to and from Dilated Integers

Rajeev Raman and David S. Wise *Member, IEEE-CS*

***Abstract*— Dilated integers form an ordered group of the cartesian indices into a d-dimensional array represented in Morton order. Efficient implementations of its operations can be found elsewhere; this paper offers efficient casting (type)conversions to and from ordinary integer representation. As Morton-order representation for two- and three-dimensional arrays attracts more users because of its excellent block locality, the efficiency of these conversions becomes important. They are essential for programmers who would use cartesian indexing there.**

Two algorithms for each casting conversion are presented here: including to-and-from dilated integers, for both $d = 2$ and $d = 3$. They fall into two families. One family uses newly compact table-lookup so cache capacity is better preserved. The other generalizes better to all d, using processor-local arithmetic that is newly presented as abstract d-ary and $(d-1)$-ary recurrences. Test results for 2 and 3 dimensions generally favor the former.

CCS Categories and subject descriptors: E.1 [Data Structures]:Arrays; D.1.0 [Programming Techniques]: General; B.3.2 [Memory Structures]: Design Styles; F.2.1 [Analysis of Algorithms and Problem Complexity]: Numerical algorithms and problems–computations on matrices.
General Term: Algorithms, Design, Performance
Additional Key Words: memory hierarchy, caches, paging, compilers, classes, quadtrees, octrees, dilation, contraction.

I. INTRODUCTION.

Dilated integers form an ordered group for the cartesian indices into a d-dimensional array represented in Morton order [1]. Figure 1 illustrates the Morton order and the use of dilated integers as its cartesian indices. With an extra high-order bit, they also suffice for casts from cartesian to/from Ahnentafel indices [2]. The amount of dilation, called d-dilation, varies with the dimension d of the represented array. Morton order is used to index arrays in applications from computer vision and graphics, cartography and geographical information systems, volumetric analyses such as axial tomography (CAT scans), and the conventional matrix computations of scientific computing. An array in Morton order can be conveniently decomposed as a 2^d-ary tree whose subtrees have contiguous addresses; such locality minimizes page and cache misses. Practically, $d = 2$ for matrices and quadtrees, and $d = 3$ for three-dimensional aggregates and octrees.

A comfortable programming paradigm to take advantage of such locality uses recursion on 2^d blocks of each array/operand to descend those trees [3], [4]. When the base case is reached (well above 1×1, typically a 32×32 block) all of it resides in cache. More importantly, its larger and larger enveloping blocks fit into L2, L3, RAM, swapping disk, and so forth up the memory hierarchy—to include communication of blocks among both shared-memory and distributed co-processors. The programmer need not worry about which size lands where. This implicit mapping of memory has been called "cache-oblivious" because she can ignore details of block size, cache lines, and memory capacity [5]; all these are local minutiae anyway. Whichever blocks fit will reside in L2 cache for a good while, with subblocks rolling into L1 repeatedly; meanwhile, translation-lookaside buffer (TLB) misses seem to disappear [6]. This style enhances portability of code that no longer needs to be retuned to specific memory architecture.

As these array indexings become better used for their high performance—particularly in the context of cache-oblivious matrix processing—their conversions to and from ordinary integer representations become important. They are essential manifestations, for instance, of the common abstractions from generic programming [7]. Like the I/O conversions of IEEE floating-point numbers, however, their use really should be infrequent, and be displaced by efficient computation within that type. Nevertheless, unlike the decimal/float conversions [8], [9], knowledge of the details of these algorithms now seems a prerequisite for the acceptance of the type into common use, particularly in the face of lore identifying that conversion as an obstacle [5, p. 288], [10, §4.1], [11, §4.1].

In order to hasten its promulgation we offer eight algorithms—two conversions in each direction for $d = 2$ and $d = 3$—to suit specific architectures. In all cases the conversion is fast, faster than a random access into an array which is their common run-time context. On current processors, table lookup is shown to be the faster for repeated castings. For sequential access, however, efficient implementations of the additive operators are available directly on both dilated integers and their cousins, masked integers [12], [13]. So, C++ iterators can be even faster than inline conversions for cartesian indexing into Morton-ordered matrices.

Section III, below, uses table lookup to solve these problems; Section IV uses register-local operations. Section IV-D uses an arbitrary d as d-ary and $(d-1)$-ary recurrences, and Section V describes our timings and offers conclusions. New contributions are the halving of tables necessary for contractions in Section III, new multiplication-based algorithms in Section IV-D, the generalized d-ary and $(d-1)$-ary recurrences in Section IV-D, and Section V's demonstration that table lookup is faster on current RISC processors.

Manuscript received: 2006 December 29. Revised: 2007 May 3, July 5.
R. Raman is with the University of Leicester.
D. S. Wise is with Indiana University.
Supported, in part, by the NSF under grants numbered CCR–0073491, EIA–0202048, and CCF–0541364. POWER5 tests used IU computing systems.

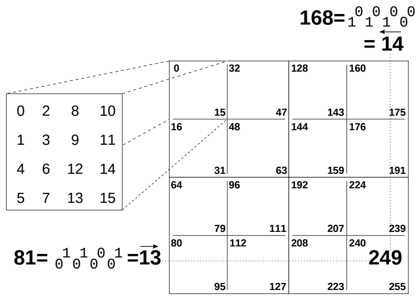

Fig. 1. Morton-order indexing of a 16×16 matrix. As this block-recursive И order extends across the plane, the dilation of integer 13 indexes the entire fourteenth row, and the doubled dilation of integer 14 indexes the whole fifteenth column. The transposed Morton-Z order is also available.

II. Definitions.

Informally, 2-dilating an integer interleaves one 0 bit between all the meaningful bits in the binary representation of an integer. As a simple example, consider $i = 2^8 - 1 = 11111111_2 = \mathrm{FF}_{16}$. Its 2-dilation, denoted $\overrightarrow{\imath}$ and read "i dilated," is $0101010101010101_2 = 5555_{16}$, written `0x5555` in the C language. Similarly, if j is `0xF0`, then $\overrightarrow{\jmath}$ is `0x5500` and, again notationally, $\overleftarrow{\jmath} = 2\overrightarrow{\jmath} =$ `0xAA00`; see Figure 1 for an example of their use as row and column indices.

Notation 2.1: Let *w*
be the number of bits in a (short) word.

Notation 2.2: Let q_ℓ be a quaternary digit.

Notation 2.3: Let o_ℓ be an octal digit.
Each q_ℓ can alternatively be expressed as $q_\ell = 2i_\ell + j_\ell$ where i_ℓ and j_ℓ are bits. Each o_ℓ can alternatively be expressed as $o_\ell = 4i_\ell + 2j_\ell + k_\ell$ where i_ℓ, j_ℓ, and k_ℓ are bits.

Cartesian indices are here limited to w or fewer bits; Morton indices and 2-dilated integers have $2w$ bits. The restriction, $w = 16$, can be relaxed for 64-bit architectures although, frankly, dense matrices don't often have orders exceeding the 65,535 that it provides. If necessary, these algorithms are easily generalized to larger w.

Definition 2.1: [2] The root of a d-dimensional array has Morton-order index 0. A subarray (block) at Morton-order index i is either an element (scalar), or it is composed of 2^d subarrays, with indices $2^d i + 0, 2^d i + 1, \ldots, 2^d i + (2^d - 1)$ at the next level.

For $d = 2$ the indexing of a 16×16 matrix is illustrated in Figure 1. At each level of the recurrence the four quadrants are labeled *northwest*, *southwest*, *northeast*, and *southeast*, respectively. This yields the И order, to meet the conventions of scientific computation where matrices are tall rather than wide. Computer graphics, where images are wider than they are tall, uses the transposed Z order, available by exchanging *southwest* and *northeast* just above, and using $\overleftarrow{\imath}$ and $\overrightarrow{\jmath}$ to index rows and columns.

Theorem 2.1: [1] The Morton index into a matrix is $\sum_{\ell=0}^{w-1} q_\ell 4^\ell = 2\sum_{\ell=0}^{w-1} i_\ell 4^\ell + \sum_{\ell=0}^{w-1} j_\ell 4^\ell$ and corresponds to the cartesian index for row $i = \sum_{\ell=0}^{w-1} i_\ell 2^\ell$ and column $j = \sum_{\ell=0}^{w-1} j_k 2^\ell$.
Concatenated quaternary digits, above, index the nested quadrants of a matrix; octal digits, below, index nested octants of a 3D array.

Theorem 2.2: [2] The Morton index into a 3-dimensional array is $\sum_{\ell=0}^{w-1} o_\ell 8^\ell = 4\sum_{\ell=0}^{w-1} i_\ell 8^\ell + 2\sum_{\ell=0}^{w-1} j_\ell 8^\ell + \sum_{\ell=0}^{w-1} k_\ell 8^\ell$ and corresponds to the cartesian index for row $i = \sum_{\ell=0}^{w-1} i_\ell 2^\ell$, column $j = \sum_{\ell=0}^{w-1} j_\ell 2^\ell$, and rod $k = \sum_{\ell=0}^{w-1} k_\ell 2^\ell$.

These theorems reflect the so-called "bit interleaving" [15], [16], [1], also described in the book [17, pp. 105–110]. The purpose of the algorithms below is to convert between the binary representations of $i = \sum_{\ell=0}^{w-1} i_\ell 2^\ell$ and of $\overrightarrow{\imath} = \sum_{\ell=0}^{w-1} i_\ell 2^{d\ell}$, *i.e.* the dilation of integers.

The following two sections define four functions each, for casting ordinary integers to and from 2-dilated and 3-dilated integers using table lookup and $w$ord-width arithmetic. The former functions run in about $w/8$ steps (on tables of 2^8 entries), but the latter use roughly $\lg w$ steps. Even with w growing to 32, addressing a square matrix of order $4 \cdot 10^9$, these proportions are about the same. Section IV-D offers two more functions for even wider dilations.

III. Converting via table lookup.

The following algorithms are conversions and inverse conversions (*dilations* and *contractions*) from integers to 2-dilations: $\sum_{\ell=0}^{w-1} i_\ell 2^\ell \leftrightarrows \sum_{\ell=0}^{w-1} i_\ell 4^\ell$ with inline instructions. The first casting conversions use table lookup. They require a vector of 256 entries, either short integers or bytes; its aligned footprint in an L1 cache is very small, and the inline code is short.

For the first pair of algorithms, assume that $d = 2$, thus spreading bits apart by one, that the dilated integers are 32 bits wide, that ordinary integer indices are 16 bits wide, and that all integers are unsigned. Their generalization is straightforward for 64-bit indexing into large arrays. Generalizations to $d = 3$ follow.

The table for 2-dilation of Algorithm 1 appears as Figure 2 [18], [19]; it is composed of 256 short integers, occupying twice the necessary bits but avoiding final operations to mask out alternating bits from them. Those extra operations are illustrated in Algorithm 3 for 3-dilation.

Algorithm 1:
```
inline unsigned int dilate_2(unsigned short x){
  return dilate_tab2[    0xFF & x      ]
      | (dilate_tab2[ (0xFFFF & x) >>8 ] <<16);
}
```
Algorithm 2:
```
inline unsigned short undilate_2(unsigned int x){
  return undilate_tab2[0xFF & ((x>>7) |x)      ]
      | (undilate_tab2[0xFF &(((x>>7) |x) >>16)]
            << 8);
}
```
To invert a dilation, the dilated integer is split and folded into a single pattern, byte by byte, which then indexes a table (Figure 3) to recover its contraction. This folding halves the fetches from others' tables [18], [19]. Algorithms 1–4 use bitwise inclusive-or `|` to sum bit patterns. Alternatives are simple addition `+` and bitwise exclusive-or `^` because the patterns are bitwise disjoint.

Algorithms 3 and 4 use the precomputed vector of 256 bytes in Figure 4, instead of two: one of shorts and another of bytes. The algorithms are written for 3-dilations of 16-bit short integers to 32-bit integers, and vice versa.

```
const unsigned short int dilate_tab2[256] = {
        0x0000, 0x0001,  0x0004, 0x0005,    0x0010, 0x0011,  0x0014, 0x0015,
        0x0040, 0x0041,  0x0044, 0x0045,    0x0050, 0x0051,  0x0054, 0x0055,
        0x0100, 0x0101,  0x0104, 0x0105,    0x0110, 0x0111,  0x0114, 0x0115,
        0x0140, 0x0141,  0x0144, 0x0145,    0x0150, 0x0151,  0x0154, 0x0155,
        0x0400, 0x0401,  0x0404, 0x0405,    0x0410, 0x0411,  0x0414, 0x0415,
        0x0440, 0x0441,  0x0444, 0x0445,    0x0450, 0x0451,  0x0454, 0x0455,
        0x0500, 0x0501,  0x0504, 0x0505,    0x0510, 0x0511,  0x0514, 0x0515,
        0x0540, 0x0541,  0x0544, 0x0545,    0x0550, 0x0551,  0x0554, 0x0555,
        0x1000, 0x1001,  0x1004, 0x1005,    0x1010, 0x1011,  0x1014, 0x1015,
        0x1040, 0x1041,  0x1044, 0x1045,    0x1050, 0x1051,  0x1054, 0x1055,
        0x1100, 0x1101,  0x1104, 0x1105,    0x1110, 0x1111,  0x1114, 0x1115,
        0x1140, 0x1141,  0x1144, 0x1145,    0x1150, 0x1151,  0x1154, 0x1155,
        0x1400, 0x1401,  0x1404, 0x1405,    0x1410, 0x1411,  0x1414, 0x1415,
        0x1440, 0x1441,  0x1444, 0x1445,    0x1450, 0x1451,  0x1454, 0x1455,
        0x1500, 0x1501,  0x1504, 0x1505,    0x1510, 0x1511,  0x1514, 0x1515,
        0x1540, 0x1541,  0x1544, 0x1545,    0x1550, 0x1551,  0x1554, 0x1555,
        0x4000, 0x4001,  0x4004, 0x4005,    0x4010, 0x4011,  0x4014, 0x4015,
        0x4040, 0x4041,  0x4044, 0x4045,    0x4050, 0x4051,  0x4054, 0x4055,
        0x4100, 0x4101,  0x4104, 0x4105,    0x4110, 0x4111,  0x4114, 0x4115,
        0x4140, 0x4141,  0x4144, 0x4145,    0x4150, 0x4151,  0x4154, 0x4155,
        0x4400, 0x4401,  0x4404, 0x4405,    0x4410, 0x4411,  0x4414, 0x4415,
        0x4440, 0x4441,  0x4444, 0x4445,    0x4450, 0x4451,  0x4454, 0x4455,
        0x4500, 0x4501,  0x4504, 0x4505,    0x4510, 0x4511,  0x4514, 0x4515,
        0x4540, 0x4541,  0x4544, 0x4545,    0x4550, 0x4551,  0x4554, 0x4555,
        0x5000, 0x5001,  0x5004, 0x5005,    0x5010, 0x5011,  0x5014, 0x5015,
        0x5040, 0x5041,  0x5044, 0x5045,    0x5050, 0x5051,  0x5054, 0x5055,
        0x5100, 0x5101,  0x5104, 0x5105,    0x5110, 0x5111,  0x5114, 0x5115,
        0x5140, 0x5141,  0x5144, 0x5145,    0x5150, 0x5151,  0x5154, 0x5155,
        0x5400, 0x5401,  0x5404, 0x5405,    0x5410, 0x5411,  0x5414, 0x5415,
        0x5440, 0x5441,  0x5444, 0x5445,    0x5450, 0x5451,  0x5454, 0x5455,
        0x5500, 0x5501,  0x5504, 0x5505,    0x5510, 0x5511,  0x5514, 0x5515,
        0x5540, 0x5541,  0x5544, 0x5545,    0x5550, 0x5551,  0x5554, 0x5555
};
```

Fig. 2. Vector of 256 dilated short integers, each of which 2-dilates the byte that indexes it for Algorithm 1.

```
const unsigned char undilate_tab2[256] = {
    0x00,0x01,0x10,0x11, 0x02,0x03,0x12,0x13,  0x20,0x21,0x30,0x31, 0x22,0x23,0x32,0x33,
    0x04,0x05,0x14,0x15, 0x06,0x07,0x16,0x17,  0x24,0x25,0x34,0x35, 0x26,0x27,0x36,0x37,
    0x40,0x41,0x50,0x51, 0x42,0x43,0x52,0x53,  0x60,0x61,0x70,0x71, 0x62,0x63,0x72,0x73,
    0x44,0x45,0x54,0x55, 0x46,0x47,0x56,0x57,  0x64,0x65,0x74,0x75, 0x66,0x67,0x76,0x77,
    0x08,0x09,0x18,0x19, 0x0A,0x0B,0x1A,0x1B,  0x28,0x29,0x38,0x39, 0x2A,0x2B,0x3A,0x3B,
    0x0C,0x0D,0x1C,0x1D, 0x0E,0x0F,0x1E,0x1F,  0x2C,0x2D,0x3C,0x3D, 0x2E,0x2F,0x3E,0x3F,
    0x48,0x49,0x58,0x59, 0x4A,0x4B,0x5A,0x5B,  0x68,0x69,0x78,0x79, 0x6A,0x6B,0x7A,0x7B,
    0x4C,0x4D,0x5C,0x5D, 0x4E,0x4F,0x5E,0x5F,  0x6C,0x6D,0x7C,0x7D, 0x6E,0x6F,0x7E,0x7F,
    0x80,0x81,0x90,0x91, 0x82,0x83,0x92,0x93,  0xA0,0xA1,0xB0,0xB1, 0xA2,0xA3,0xB2,0xB3,
    0x84,0x85,0x94,0x95, 0x86,0x87,0x96,0x97,  0xA4,0xA5,0xB4,0xB5, 0xA6,0xA7,0xB6,0xB7,
    0xC0,0xC1,0xD0,0xD1, 0xC2,0xC3,0xD2,0xD3,  0xE0,0xE1,0xF0,0xF1, 0xE2,0xE3,0xF2,0xF3,
    0xC4,0xC5,0xD4,0xD5, 0xC6,0xC7,0xD6,0xD7,  0xE4,0xE5,0xF4,0xF5, 0xE6,0xE7,0xF6,0xF7,
    0x88,0x89,0x98,0x99, 0x8A,0x8B,0x9A,0x9B,  0xA8,0xA9,0xB8,0xB9, 0xAA,0xAB,0xBA,0xBB,
    0x8C,0x8D,0x9C,0x9D, 0x8E,0x8F,0x9E,0x9F,  0xAC,0xAD,0xBC,0xBD, 0xAE,0xAF,0xBE,0xBF,
    0xC8,0xC9,0xD8,0xD9, 0xCA,0xCB,0xDA,0xDB,  0xE8,0xE9,0xF8,0xF9, 0xEA,0xEB,0xFA,0xFB,
    0xCC,0xCD,0xDC,0xDD, 0xCE,0xCF,0xDE,0xDF,  0xEC,0xED,0xFC,0xFD, 0xEE,0xEF,0xFE,0xFF
};
```

Fig. 3. Vector of all possible bytes, each indexed by the folding of its respective 2-dilated integer for Algorithm 2.

```
const unsigned char dilate_tab3[256] = {
    0x00,0x01,0x08,0x09, 0x40,0x41,0x48,0x49,  0x02,0x03,0x0A,0x0B, 0x42,0x43,0x4A,0x4B,
    0x10,0x11,0x18,0x19, 0x50,0x51,0x58,0x59,  0x12,0x13,0x1A,0x1B, 0x52,0x53,0x5A,0x5B,
    0x80,0x81,0x88,0x89, 0xC0,0xC1,0xC8,0xC9,  0x82,0x83,0x8A,0x8B, 0xC2,0xC3,0xCA,0xCB,
    0x90,0x91,0x98,0x99, 0xD0,0xD1,0xD8,0xD9,  0x92,0x93,0x9A,0x9B, 0xD2,0xD3,0xDA,0xDB,
    0x04,0x05,0x0C,0x0D, 0x44,0x45,0x4C,0x4D,  0x06,0x07,0x0E,0x0F, 0x46,0x47,0x4E,0x4F,
    0x14,0x15,0x1C,0x1D, 0x54,0x55,0x5C,0x5D,  0x16,0x17,0x1E,0x1F, 0x56,0x57,0x5E,0x5F,
    0x84,0x85,0x8C,0x8D, 0xC4,0xC5,0xCC,0xCD,  0x86,0x87,0x8E,0x8F, 0xC6,0xC7,0xCE,0xCF,
    0x94,0x95,0x9C,0x9D, 0xD4,0xD5,0xDC,0xDD,  0x96,0x97,0x9E,0x9F, 0xD6,0xD7,0xDE,0xDF,
    0x20,0x21,0x28,0x29, 0x60,0x61,0x68,0x69,  0x22,0x23,0x2A,0x2B, 0x62,0x63,0x6A,0x6B,
    0x30,0x31,0x38,0x39, 0x70,0x71,0x78,0x79,  0x32,0x33,0x3A,0x3B, 0x72,0x73,0x7A,0x7B,
    0xA0,0xA1,0xA8,0xA9, 0xE0,0xE1,0xE8,0xE9,  0xA2,0xA3,0xAA,0xAB, 0xE2,0xE3,0xEA,0xEB,
    0xB0,0xB1,0xB8,0xB9, 0xF0,0xF1,0xF8,0xF9,  0xB2,0xB3,0xBA,0xBB, 0xF2,0xF3,0xFA,0xFB,
    0x24,0x25,0x2C,0x2D, 0x64,0x65,0x6C,0x6D,  0x26,0x27,0x2E,0x2F, 0x66,0x67,0x6E,0x6F,
    0x34,0x35,0x3C,0x3D, 0x74,0x75,0x7C,0x7D,  0x36,0x37,0x3E,0x3F, 0x76,0x77,0x7E,0x7F,
    0xA4,0xA5,0xAC,0xAD, 0xE4,0xE5,0xEC,0xED,  0xA6,0xA7,0xAE,0xAF, 0xE6,0xE7,0xEE,0xEF,
    0xB4,0xB5,0xBC,0xBD, 0xF4,0xF5,0xFC,0xFD,  0xB6,0xB7,0xBE,0xBF, 0xF6,0xF7,0xFE,0xFF
};
```

Fig. 4. Vector of all 256 bytes, each of which 3-dilates the byte that indexes it, *and* each indexed by the folding of its respective 3-dilated integer for *both* Algorithms 3 and 4.

Algorithm 3:
```
inline unsigned int dilate_3(unsigned short x){
  return ((
      ((dilate_tab3[(0xFFFF & x) >>8]  )<<24)
     |  dilate_tab3[   0xFF & x     ]
    ) * 0x010101) & 0x49249249;
}
```
Algorithm 4:
```
inline unsigned short undilate_3(unsigned int x){
  return dilate_tab3[0xFF & ((((x>>8) | x) >>8)
                              | x)      ]
      | (dilate_tab3[0xFF &(((((x>>8) | x) >>8)
                              | x) >>24)] <<8);
}
```
Just as Algorithm 2 halves the space of Algorithm 1 (Figure 3 *vs.* 2), the next theorem nearly halves their total space once again.

Theorem 3.1: The tables of bytes required for Algorithms 3 and 4 coincide.

Proof: [Suggested by a referee] A table entry effects a dilating/folding permutation, P, of the bit at an individual position, $0 \leq i < 8$, to position $(3{\cdot}i) \mod 8$, and it so permutes all eight bits in a single lookup. This permutation is its own inverse because $P(P(i)) = (3{\cdot}3{\cdot}i) \mod 8 = i \mod 8 = i$. ■

A similar proof applies for any d-dilation and permuting table of 2^p different p-bit integers, such that $d^2 \mod p = 1$. For example, one table of 256 bytes also suffices for 5-dilation and contraction, though computations like the next sections' are faster.

IV. Conversion without table Lookup.

Conversion can also be done entirely within the processor without the cache impact of table lookup. A review of existing approaches follows, based on shifts and masking, which leads to new ones based on integer multiplication. This section concludes by considering the abstract case for all d. Discussed first are d-undilation algorithms for $d = 2, 3$.

As before, consider the case where the dilated integer is contained in a w-bit (32-bit) word. The w limit constrains the number of significant digits in the undilated integer to $s = \lfloor w/d \rfloor$. As shown in several cases below, abstract multiplication generates high-order overflow bits that are ignored.

Notation 4.1: Let s denote the number of significant dilatable bits in an undilated integer.

A. Shift-Or Algorithms

Merkey presents straight-line codes based on an algorithm by Stocco and Schrack [20], [19], that use repeated shift–or operations and masks for conversion between normal and dilated integers. They also can be found without citation among sample codes in machine manuals [21, pp. 136–139, 184–185]. Instances of the shift-based 2-undilation and 2-dilation algorithms for $s = 16$ are shown in Figure 5. Each algorithm works in four rounds, where a round comprises three operations: a shift and two bitwise operations. Merkey presents a variant that performs one shift and three bitwise operations per round, but works equally well with superscalar instructions. Another round is necessary for $w = 64, s = 32$.

Figure 5 illustrates the whole family of algorithms. In `undilate_2` there, for instance, each round coalesces pairs of groups from the previous round; that is, that algorithm performs "binary coalescing." Unnumbered here because it is replaced below, it uses a mask to remove the redundant "middle muddle" that results from reflexive bit merges. It collapses the dilation in only $\lceil \lg s \rceil$ rounds.

B. Undilation via Multiplication

With fast integer multiplication the shift-ors in a round of the undilation algorithm might be reduced to a single multiplication that implements them. When a constant factor has only a few bits set, its multiplication might be better replaced by inline shift-adds at compile time, or (even if a variable) decoded by the processor to shift-adds at run time. For example, multiplications by integers like 3 or 17 that have only two bits set in their binary representation can be implemented by a single shift-add. In writing such a multiplication we intend it to allow such an implementation if faster. Indeed, we have seen compilers that enforce it (sometimes *slowing* the code). Faster multipliers or relatively slower barrel shifters might favor the multiply as coded.

The multiplication is safe here because the intervening/dilating zero bits absorb any carries from the implied addition within the middle muddle, that is about to be masked off anyway. Since any dilated integer has a sparse set of significant bit positions, it is safe to implement the shift-ors of similar undilations using multiplication. For instance, 2- and 3-undilation can be implemented as follows:

Algorithm 6:
```
    inline u_short undilate_2(u_int t){
        t = (t *   3) & 0x66666666;
        t = (t *   5) & 0x78787878;
        t = (t *  17) & 0x7F807F80;
        t = (t * 257) & 0x7FFF8000;
        return ((u_short) (t >> 15));
    }
```
Algorithm 7:
```
    inline u_short undilate_3(u_int t){
        t = (t * 0x00015) & 0x0E070381;
        t = (t * 0x01041) & 0x0FF80001;
        t = (t * 0x40001) & 0x0FFC0000;
        return ((u_short) (t >> 18));
    }
```
The moderately experienced programmer will recognize the constant factors in Algorithm 6 as successors of powers-of-powers-of-2 (Fermat primes even), and so we express them as decimal numbers. In fact and as derived in Section IV-D, in Round i the factor is composed of 2 one bits, separated by $\frac{1}{2}2^i - 1$ zero bits. In Algorithm 7 they are presented as hexadecimal numbers, revealing their similarly regular bit patterns: one bits are separated by $\frac{2}{3}3^i - 1$ zero bits in Round i.

Algorithm 6 mimics Figure 5's `undilate_2`, with the difference that the partial results shift to the left as rounds proceed because the products are emulating right shifts. For example, its first statement is mimicked almost directly by the the assignment `t = (t * 3) & 0x66666666` in Algorithm 6, except for that shift. Because of it, a cumulative correcting shift is necessary in the return value.

```
inline u_short undilate_2(u_int t) {
   t = (t | (t >> 1)) & 0x33333333;
   t = (t | (t >> 2)) & 0x0F0F0F0F;
   t = (t | (t >> 4)) & 0x00FF00FF;
   t = (t | (t >> 8)) & 0x0000FFFF;
   return((u_short) t);
}
```

Algorithm 5:

```
inline u_int dilate_2(u_short t) {
  u_int r = t;
  r = (r | (r << 8)) & 0x00FF00FF;
  r = (r | (r << 4)) & 0x0F0F0F0F;
  r = (r | (r << 2)) & 0x33333333;
  r = (r | (r << 1)) & 0x55555555;
  return(r);
}
```

Fig. 5. Inline algorithms described by Merkey for 2-dilation and 2-undilation. `u_int` and `u_short` are assumed to be `typedef`'ed to `unsigned int` and `unsigned short`, respectively [20], [19]. The left one remains unnumbered because it is replaced by Algorithm 6, above.

In summary, Algorithm 6 requires 9 operations, and Algorithm 7 requires 7. For 64-bit dilated indices, an extra round is needed by both algorithms, raising the instruction counts to 11 and 9 respectively. See Table I for comparisons.

C. Dilation

Algorithm 5 stands as our processor-local algorithm for 2-dilation. Multiplication, alone, cannot replace its shift-or operations because it introduces carry bits whose effects survive the subsequent masking. For example, one is tempted to substitute `t = t*257 & mask` for its first statement `t = (t | (t << 8)) & mask`, but that fails, *e.g.* at $t = 257$, because of the carry that shifts into the 2^9 bit of the product. Indeed, we are not aware of any multiplication-based approach to 2-dilation that uses fewer instructions than the shift-based approach.

The multiplication-based approach, however, *can* be used to 3-dilate an integer via "binary splitting" with fewer instructions than a shift-based binary-splitting algorithm.

Algorithm 8:

```
inline u_int dilate_3(u_short t) {
    u_int r = t;
    r =  (r * 0x10001) & 0xFF0000FF;
    r =  (r * 0x00101) & 0x0F00F00F;
    r =  (r * 0x00011) & 0xC30C30C3;
    r =  (r * 0x00005) & 0x49249249;
    return(r);
}
```

As before, the constant factors in Algorithm 8 follow a pattern that can be inferred from their hexadecimal presentation: one bits are separated by $2^{5-i} - 1$ zero bits at Round i. This algorithm delivers a binary splitting pattern of partial results: at the end of each round, there is a maximum group size, which changes as 8, 4, 2 and then 1.

Whereas Algorithm 8 needs as many rounds as a shift-based approach, each round requires just two instructions, multiply and mask, that might become the shift-based algorithms' three. Thus, dilating for $d = 3$ and $s = 10$ takes 8 instructions rather than 12, and for 64-bit dilated integers it would require only 10.

Although a "ternary split" could perform 3-dilation in fewer rounds, such an approach is inferior for both the multiplication- and shift-based approaches. Multiplication fails because of the presence of undesirable carries. As before, shift-based approaches would also have an excessive number of operations per round, more than nullifying any reduction in the number of rounds. However, as the next section shows, multiplication implements ternary split well for 4-dilation and, in general, $(d-1)$-ary split does well for wider d-dilations.

D. Even Wider Dilations

The case of arbitrary $d > 1$ and $s > 1$ is now addressed. Specifically, how can one derive the "magic" constants above and how is one convinced of the correctness of these algorithms? (Those constants are denoted b, c, y, z below, subscripted for dilation d and the round i.) In the arbitrary case, s is subject only to the condition that $1 < s \leq \lfloor w/d \rfloor$, where w denotes the natural word width of the processor. The general algorithms follow:

Algorithm 9:

```
    DILATE$(t, s, d)$ (where $d > 2$)
1) $r \leftarrow t$;
2) for $i = 1, 2, \ldots, \lceil \log_{d-1} s \rceil$ do
       $r \leftarrow (r \times b_{d,i})$ AND $y_{d,i}$.
3) return $r$.
```

Algorithm 10:

```
    UNDILATE$(t, s, d)$
1) $r \leftarrow t$;
2) for $i = 1, 2, \ldots, \lceil \log_d s \rceil$ do
       $r \leftarrow (r \times c_{d,i})$ AND $z_{d,i}$.
3) $a \leftarrow d(s-1)+1$;
4) return $\lfloor r/2^{a-s} \rfloor$.
```

When $d > 2$, the multiplication-based algorithms use $2\lceil \log_{d-1} s \rceil$ instructions for d-dilation, and for any $d \geq 2$, $2\lceil \log_d s \rceil + 1$ instructions for d-undilation. The bases of these logarithms lead to the terms $(d-1)$-*recursion* and d-*recursion.* Compared with the shift-based algorithms' bound of $3\lceil \log_2 s \rceil$ instructions, the multiplication-based algorithms use far fewer instructions, particularly when d becomes large, due to the linearity of d in the base of the logarithm. (Table I presents exact counts for practical values of d and s.)

Consider now the correctness of the algorithms and the values of b, c, y, and z.

Notation 4.2: Let $x_{p,q}$ denote $\sum_{\ell=0}^{p-1} 2^{\ell q}$ for integers $p, q \geq 1$.

The binary expansion of $x_{p,q}$ has p 1s, with each pair of successive 1s separated by exactly $q-1$ zeros, and with the rightmost bit equal to 1.

The correctness of the undilation Algorithm 10 is implied by an induction on two invariants. They determine what the constants $c_{d,i}$ and $z_{d,i}$ should be. The initial/base case is used below as if it occurred at "the end of" a null Round 0:

1) For all rounds, the leftmost bit of r stays in the same bit position, i.e. position $d(s-1)+1$.
2) At the end of Round i for $i = 0, 1, \ldots$ there are groups of consecutive significant digits of size d^i, with the rightmost group possibly smaller if s is not a power

Strategy	Alg.	Dilation			Undilation		
Table Lookup		fetch	shift	mask	fetch	shift	mask
$d = 2:$	1,2	2	2	3	2	2	4
$d = 3:$	3,4	2	2^1	3^1	2	4	5
Processor-local		multiply	shift	mask	multiply	shift	mask
$d = 2:$	5,6	0	4	8	4	1	4
$d = 3:$	8,7	4	0	4	3	1	3

TABLE I

INSTRUCTION COUNTS FOR UNDILATION AND DILATION, FOR PRACTICAL ALGORITHMS USING TABLE LOOKUP, SHIFT-BASED, AND MULTIPLICATION-BASED ALGORITHMS.

of d. Each group is separated from the one before it by $d^i(d-1)$ zero bits.

It is easy to verify that the multiplication constant for the $i{+}1^{\text{st}}$ round, $c_{d,i+1}$, should be $x_{d,(d-1)d^i}$. This constant can be more than w bits long but, by the properties of modular arithmetic, only the low-order w bits of any constant are necessary for the result to be correct. For instance, when $d = 3$, $s = 10$ and $w = 32$, $c_{3,3} = x_{3,18} = 2^{36} + 2^{18} + 1 =$ `0x1000040001`, a 37-bit value. However, the constant `0x40001` in Algorithm 7 is just the low-order 32 bits of $c_{3,3}$.

Algorithm 9, d-dilation for $d > 2$ in $t = \lceil \log_{d-1} s \rceil$ rounds, has invariants:

3) For all rounds, the rightmost bit of r stays in position 0.
4) At the end of Round i, for $i = 0, 1, \ldots, t$, there are groups of consecutive significant digits of size $(d{-}1)^{t-i}$, plus possibly one (the leftmost) group of smaller size if s is not a power of $(d-1)$. Each group is separated from the one before it by $(d-1)^{t-i+1}$ blank bits.

Therefore, one round of d-dilation reduces the maximum size of a group of consecutive significant digits by a factor of $(d-1)$, which may be viewed as a $(d-1)$-way split. Figure 6 illustrates why this works. As it shows, one can use a multiplication to create $(d-1)$ adjacent copies of a block of $(d-1)^j$ contiguous significant bits. From these adjacent copies, one can extract the appropriate contiguous blocks of size $(d-1)^{j-1}$, which are spaced exactly $(d-1)^j$ bits apart, as required by Invariant 4. The multiplication constants are given by $b_{d,i} = x_{d,(d-1)^{t-i+1}}$.

V. TIMINGS AND CONCLUSION

Algorithms 1–8 have been tested on PowerPCs G4 and G5, Pentiums 2 and 4, Xeons, an Itanium, Athlons, Opterons, and an IBM POWER5. Timings were taken for repeated runs with the small tables of Algorithms 1–4 preloaded in cache, a reasonable assumption when they are being used often, as in a loop. (The alternative of condemning cache after every test, to simulate rare random conversions, is too stringent.) All are very fast. Algorithms 1–2 are consistently faster than Algorithms 5–6. Algorithms 3–4 compare differently to 8–7 on different machines. For wider dilations, $d > 3$, unpacking the table entries becomes tedious and the direct, multiplication-based algorithms shine.

[1]Multiplication by `0x010101` in Algorithm 3 is not counted here. Depending on your processor, it may be faster as the constant multiplication, or as 2 shifts and 2 additions.

		$d = 2$ for Matrices			
		Table Lookup		Shift	Multiply
Processor	Compiler Version	Dilate Alg.1	Undilate Alg. 2	Dilate Alg. 5	Undilate Alg. 6
Itanium	`gcc` 3.4	2.64	4.65	9.66	4.65
Itanium	`icc` 8.1	3.11	5.11	10.12	8.12
Opteron	`gcc` 4.1.1	1.05	2.57	6.08	4.54
Pentium4	`gcc` 3.4.6	4.04	6.55	15.17	10.85
Pentium-M	`gcc` 3.3.6	3.01	4.14	8.37	7.96
POWER5	`gcc` 3.3.3	1.08	2.95	8.39	4.85
Xeon	`gcc` 3.4.5	4.06	6.48	15.18	10.45
		$d = 3$			
Measured in average processor cycles.		Table Lookup		Multiply	
		Dilate Alg. 3	Undilate Alg. 4	Dilate Alg. 8	Undilate Alg. 7
Itanium	`gcc` 3.4	7.68	6.65	7.65	6.65
Itanium	`icc` 8.1	10.12	8.12	8.12	8.12
Opteron	`gcc` 4.1.1	3.10	5.05	5.19	2.99
Pentium4	`gcc` 3.4.6	10.81	8.56	14.16	13.72
Pentium-M	`gcc` 3.3.6	5.02	4.83	8.61	5.56
POWER5	`gcc` 3.3.3	4.88	4.97	8.64	2.98
Xeon	`gcc` 3.4.5	10.71	8.49	13.94	13.68

TABLE II

AVERAGE TIMES FOR THE ALGORITHMS MEASURED IN PROCESSOR CYCLES ON DIFFERENT MACHINES.

Table I compares the operation counts for the eight principal algorithms. The trade-off between fetch and multiplication is apparent. Table II compares the running times measured in clock cycles for these algorithms, run on several machines. The indicated C compilers and versions were used only with option `-O3`.

This paper presents several results advancing the use of dilated integers for indexing Morton-ordered matrices. First, it is an introduction to those structures and how simply they deliver block-locality with conventional cartesian indexing. Second, it reintroduces efficient processor-based algorithms that should be more widely known, and establishes new time- and space-efficient table-based alternatives. Of course, either could be displaced by hardwired bit shufflers. Third, it generalizes the former algorithms with d-ary and $(d-1)$-ary recurrences for the general case.

Expansion of the memory hierarchy and the advent of chip multiprocessors create a need for arrays with highly local access within any cache-resident block. These algorithms allow dilated integers, as well as methods for direct arithmetic on them [12], help deliver that locality with ordinary cartesian indexing on ordinary hardware.

Acknowledgments: We thank the sponsors of Schloss Dagstuhl in the Saarland, for creating the magnificent environment and international interactions that inspired this note. Thanks also to the students in the B679 seminar at Indiana University, particularly K. Patrick Lorton, who timed these codes on their favorite machines, and to anonymous diligent referees.

REFERENCES

[1] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," IBM Ltd., Ottawa, Ontario, Tech. Rep., Mar. 1966.

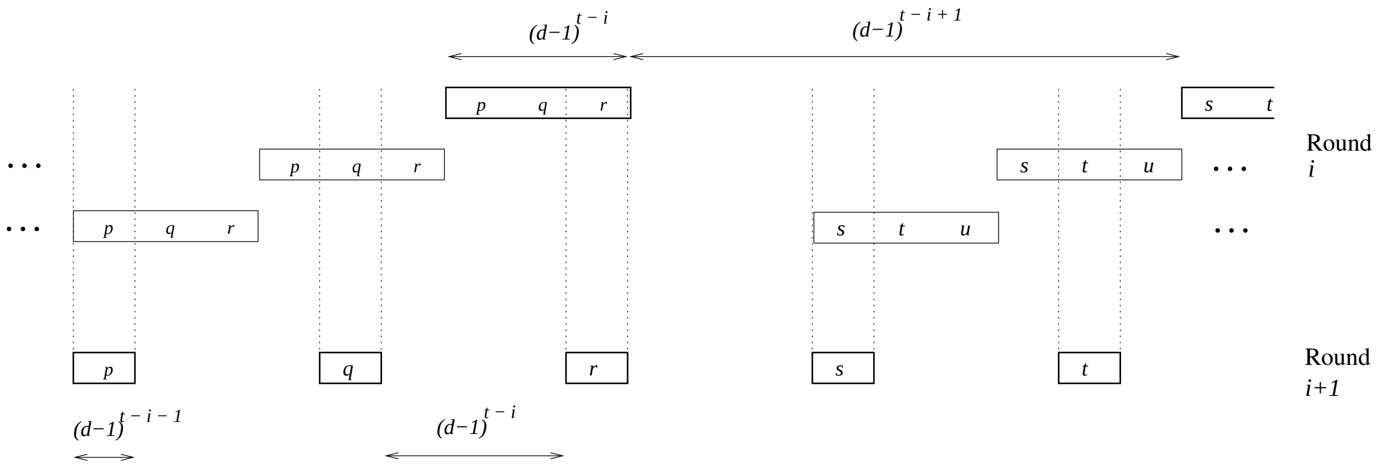

Fig. 6. Schematic view of Round $i + 1$ in the general dilation algorithm. Each rectangle represents a block of consecutive significant digits. The top row is the state at the end of Round i, the bottom row is the state at the end of Round $i + 1$ and the middle rows show the effect of the multiplication in Round $i + 1$. The dotted lines indicate the parts of the product that are retained by the mask operations.

[2] D. S. Wise, "Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free," in *Euro-Par 2000 – Parallel Processing*, ser. Lecture Notes in Comput. Sci., A. Bode, T. Ludwig, W. Karl, and R. Wismüller, Eds. Heidelberg: Springer, 2000, vol. 1900, pp. 774–883. [Online]. Available: http://www.springerlink.com/content/0pc0e9gfk4x9j5fa

[3] J. D. Frens and D. S. Wise, "QR factorization with Morton-ordered quadtree matrices for memory re-use and parallelism," *Proc. 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program, SIGPLAN Not.*, vol. 38, no. 10, pp. 144–154, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/781498.781525

[4] J. Sang Park, M. Penner, and V. K. Prasanna, "Optimizing graph algorithms for improved cache performance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 9, pp. 769–782, Sept. 2004. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2004.44

[5] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache–oblivious algorithms," in *Proc. 40th Ann. Symp. Foundations of Computer Science*. Washington, DC: IEEE Computer Soc. Press, Oct. 1999, pp. 285–298. [Online]. Available: http://dx.doi.org/10.1109/SFFCS.1999.814600

[6] M. D. Adams and D. S. Wise, "Seven at one stroke: Results from a cache-oblivious paradigm for scalable matrix algorithms," in *MSPC '06: Proc. 2006 Wkshp. Memory System Performance and Correctness*. New York: ACM Press, Oct. 2006, pp. 41–50. [Online]. Available: http://doi.acm.org/10.1145/1178597.1178604

[7] J. G. Siek and A. Lumsdaine, "The matrix template library: generic components for high-performace scientific computing," *Computing in Science and Eng.*, vol. 1, no. 6, pp. 70–78, Nov. 1999. [Online]. Available: http://dx.doi.org/10.1109/5992.805137

[8] W. D. Clinger, "How to read floating-point numbers accurately," *SIGPLAN Not.*, vol. 39, no. 4, pp. 360–371, Apr. 2004, best of PLDI: originally published *25*, 6 (June 1990), 92–101. [Online]. Available: http://doi.acm.org/10.1145/989393.989430

[9] G. L. Steele, Jr. and J. L. White, "How to print floating-point numbers accurately," *SIGPLAN Not.*, vol. 39, no. 4, pp. 372–389, Apr. 2004, best of PLDI: originally published *25*, 6 (June 1990), 112–126. [Online]. Available: http://doi.acm.org/10.1145/989393.989431

[10] M. Thottethodi, S. Chatterjee, and A. R. Lebeck, "Tuning Strassen's matrix multiplication for memory efficiency," in *Proc. Supercomputing '98*. Washington, DC: IEEE Computer Soc. Press, Nov. 1998, ch. 36. [Online]. Available: http://dx.doi.org/10.1109/SC.1998.10045

[11] V. Valsalam and A. Skjellum, "A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels," *Concur. Comp. Prac. Exper.*, vol. 14, no. 10, pp. 805–839, 2002. [Online]. Available: http://dx.doi.org/10.1002/cpe.630

[12] G. Schrack, "Finding neighbors of equal size in linear quadtrees and octrees in constant time," *CVGIP: Image Underst.*, vol. 55, no. 3, pp. 221–230, May 1992.

[13] M. D. Adams and D. S. Wise, "Fast additions on masked integers," *SIGPLAN Not.*, vol. 41, no. 5, pp. 39–45, May 2006. [Online]. Available: http://doi.acm.org/10.1145/1149982.1149987

[14] D. E. Knuth, *Fundamental Algorithms*, 3rd ed., ser. The Art of Computer Programming. Reading, MA: Addison-Wesley, 1997, vol. 1.

[15] G. Peano, "Sur une courbe, qui remplit toute une aire plaine," *Math. Ann.*, vol. 36, pp. 157–160, 1890.

[16] K. D. Tocher, "The application of automatic computers to sampling experiments," *J. Roy. Statist. Soc. Ser. B*, vol. 16, no. 1, pp. 39–61, 1954, see pp. 53–55.

[17] H. Samet, *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley, 1990, section 2.7.

[18] F. C. Holroyd and D. C. Mason, "Efficient linear quadtree construction algorithm," *Image Vision Comput.*, vol. 8, no. 3, pp. 218–224, Aug. 1990.

[19] L. Stocco and G. Schrack, "Integer dilation and contraction for quadtrees and octrees," in *Proc. IEEE Pacific Rim Conf. Communications, Computers, and Signal Processing*. New York: IEEE, May 1995, pp. 426–428. [Online]. Available: http://dx.doi.org/10.1109/PACRIM.1995.519560

[20] P. Merkey, "Z-ordering and UPC," Michigan Technological Univ., Tech. Rep., June 2003. [Online]. Available: http://www.upc.mtu.edu/papers/zorder.pdf

[21] AMD, *AMD Athlon™ Processor x86 Code Optimization Guide*, K$^{\text{th}}$ ed., Advanced Micro Devices, Inc., Sunnyvale, CA, Feb. 2002, publication 22007. [Online]. Available: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf

Professor **Rajeev Raman** obtained his B.Tech. degree from IIT Delhi in 1986, and his M.S. and Ph.D. degrees from the University of Rochester, NY USA, in 1988 and 1993. He worked as a post-doc at the Max-Planck-Institut für Informatik and UMIACS, University of Maryland, and held a faculty position at King's College London before joining the University of Leicester in 2001.

Raman's address: Dept. of Computer Science, University Road, Leicester LEQ 7RH, UK. `http://www.cs.le.ac.uk/people/rraman/`

David S. Wise is Professor of Computer Science at Indiana University and a Fellow of the Association for Computing Machinery. He earned his bachelor's degree at the Carnegie Institute of Technology and his graduate degrees at The University of Wisconsin. This collaboration stems from his recent sabbatical at the University of Washington, Seattle.

Wise's address: Computer Science Dept., 215 Lindley Hall, Bloomington, IN 47405–7104, USA. `http://www.cs.indiana.edu/~dswise/`