Circuits Pattern

Name
Circuits

Problem
In many problems the output is a logical function or bitwise calculation of the input. However, most of the instruction sets are restricted to operate on a word size of data. How can we efficiently compute the output?

Context
Consider computations which operate at the smallest possible granularity: the bit level. Such computations arise in many places. The well-known examples are:

- Network data-link layer protocol operations: error correcting codes, parity checks, and checksums [1].
- RAID arrays: Parity checks [2].
- Security and cryptography-related operations: encryption, decryption, and code cracking [3][4][5][6].
- Database: creating hashes for a large amount of objects [7].

Bit-level problem is difficult to represent using traditional programming languages like C, because the languages do not operate on such a small level of granularity. Even if the programmer declares a Boolean variable, the language still represents it as the smallest word, not a bit. This problem arises from the fact that processor ISAs (Instruction Set Architectures) are usually restricted to word size chunks of data, which are far too coarse for expressing bit level data access. To compensate this factor, some languages provide bitwise operations on a word. For example, the bitwise AND operation on two words is to apply the AND operation on each bit of the words, and save the results into another word. Whether the bitwise operations can be executed efficiently depends on how the instruction sets are optimized for these bitwise operations.

Forces

- Words vs. bits: Machines typically have a word width of 32 or 64 bits, which approaches 2 orders of magnitude wider than is most natural for bit-level parallelism. This pulls algorithms toward word-based implementations, even though that can be unnatural.
Solution

There are 4 steps for applying the bitwise calculation on a specific problem:

(1) Divide the input data into blocks that match the word width of the platform. For example, if the word width of the platform is 32 bits per word, we can divide the input data into 32-bit blocks. If the number of bits in the input data is not an integer multiple of the platform word width, we can pad 0 to the input data to make it an integer multiple of the platform word width.

(2) Use the following 6 basic bitwise operations to represent the calculation of each block:
   a. Bitwise not: ~x
   b. Bitwise and: x & y
   c. Bitwise or: x | y
   d. Bitwise exclusive or: x ^ y
   e. Right shift: x >> y
   f. Left shift: x << y

(3) Optimize the calculation within each block.

(4) If necessary, gather the calculation results of each block for further computations.

Parallelism Issues:

The way that we explore parallelism for the bitwise calculation can be divided into two categories:

(1) Coarse-grained parallelism

   • If the blocks are independent to each other, we can manipulate each block in parallel.
   • If the manipulation on later blocks depends on the previous blocks, we can apply the pipeline pattern (please see the pipeline pattern for more information) on the operations, and explore the parallelism within the pipeline pattern.
   • If we need to gather the results from each block for further computations, we can apply the map-reduce pattern (please see the map-reduce pattern for more information) to explore the possible parallelism.

(2) Fine-grained parallelism

   • Bit-level parallelism is generally very data parallel (please see the data parallel pattern for more information), since breaking the data into bits greatly multiplies the amount of data items. The SWAR (SIMD within a register) technology is trying to multiply the parallelism within a register by applying the same instructions on different fields of a register [8]. If the SWAR technology comes to the bit level parallelism, we can achieve the maximum parallelism for bitwise operations. Then we can take advantage of the bit level parallelism by using the bitwise operations within each block.
**Invariant**

Precondition: The word width of the platform is well-defined.

**Examples**

- **Gray Code [9]:**

  Gray code is a cyclic code which two successive codes differ only 1 bit. For example, the 3-bit Gray code is: 000 001 011 010 110 111 101 100

  We will follow the 4 steps to find nth Gray code.

  1. Divide the input data into blocks:
     The only input data is n, which already fits into a word. We do not need to divide it into blocks.
  2. Use Boolean operations to represent the calculation:
     \[ G = n \oplus (n >> 1) \]
  3. Optimize the calculation:
     No optimization is required.
  4. Gather computation results if necessary:
     No gathering is required.

  Pseudo code for finding the nth Gray code:

  ```plaintext
  Procedure GrayCode(n)
  return n ^ (n >> 1)
  ```

- **Bits counting [9]:**

  Given a sequence of bits, we need to calculate the number of 1-bits in the sequence.

  1. Divide the input into blocks:
     Suppose that the word width of the platform is 32 bits, we will divide the input sequence into blocks with width equal to 32 bits.
  2. Use Boolean operations to represent the calculation:
     We can count the number of 1-bits in a word by applying the map-reduce pattern. We calculate the number of 1-bits within every block with 2 bits, and then we sum up the
number of 1-bits within every block with 4 bits, and so on. Following is an example, showing how we can use Boolean operations to achieve this goal.

Pseudo code:

```plaintext
Procedure BitCount(int x)
    x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
    x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
    x = (x & 0x0000FFFF) + ((x >>16) & 0x0000FFFF);
    return x;
```

(3) Optimize the calculation:

By observing that

\[(x >> 1) & 0x55555555 + (x >> 1) & 0x55555555 = x & 0xAAAAAAAA,\]

we can optimize the first line of the code into

\[x = x - ((x >> 1) & 0x55555555);\]

By observing that there will be no overflow for \[x + (x >> 4)\] in line 3, \[x + (x >> 8)\] in line 4, and \[x + (x >> 16)\] in line 5, we can simplify line 3-5 into

\[x = (x + (x >> 4)) & 0x0F0F0F0F;\]
\[x = (x + (x >> 8)) & 0x00FF00FF;\]
\[x = (x + (x >> 16)) & 0x000FFFFF;\]

The maximum number of 1-bits for a 32 bit word is 32, and 32 can be represented by 6 bits as 100000. By observing that the 8 bits wide and the 16 bits wide operations are both larger than the width of 6 bits, we can delay the AND operation in line 4 and 5 to the end of the procedure:

\[x = (x + (x >> 8));\]
\[x = (x + (x >> 16));\]

return \[x & 0x0000003F;\]

The optimized code is:

```plaintext
Procedure BitCount(int x)
    x = x - ((x >> 1) & 0x55555555);
```
(4) Gather computation results if necessary:
We need to summarize the number of 1-bits for the whole data sequence. Therefore, the reduction operation is required. We can adopt the map-reduce pattern to sum up all the number of 1-bits within each block to finalize the computation.

• Parity Check [9]:
The parity of a data sequence refers to whether it contains odd or even number of 1-bits.

To solve this problem, the first 2 steps are very similar to that of the bits counting problem.

(1) Suppose that the word width of the platform is 32 bits, we will divide the input sequence into blocks with width equal to 32 bits.
(2) Use Boolean operations to represent the calculation:
We calculate the parity of 1-bits within every block with 2 bits, and then we check the parity within every block with 4 bits, and so on. The pseudo code is:

```c
Procedure ParityCheck(int x)
    x = x ^ (x >> 1);
    x = x ^ (x >> 2);
    x = x ^ (x >> 4);
    x = x ^ (x >> 8);
    x = x ^ (x >> 16);
    return x;
```

(3) Optimize the calculation:
After we have got the parity of each hex unit, we can adopt the multiplication operation to add all 8 parity bits by one instruction. To be more specific, consider the multiplication between the input string x and 0x11111111:
We can see that $y$ is the summation over the 8 parity bits. By using this trick, a better implementation is:

```c
Procedure ParityCheck(int x)
    x = x ^ (x >> 1);
    x = x ^ (x >> 2) & 0x11111111;
    x = x * 0x11111111;
    return (x>>28) & 0x00000001;
```

(4) Gather computation results if necessary:

We need to calculate the parity of the whole input sequence. Therefore, after the parity of each block is calculated, we need to summarize the overall parity for the input sequence. We can apply the map-reduce pattern to accomplish this goal.

- Generalized Extract [9]:

The generalized extract is an operation that extracts bits from a source string according to a mask string and compacts all the extracted bits toward right. For example, if the source string is $x$, and the mask is $m$, the resulting extracted string $x'$ will be the following:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>=</td>
<td>abcd</td>
<td>efgh</td>
<td>ijk1</td>
<td>mnop</td>
<td>qrst</td>
<td>uvwx</td>
<td>yzAB</td>
<td>CDEF</td>
</tr>
<tr>
<td>m</td>
<td>=</td>
<td>1000</td>
<td>1000</td>
<td>1110</td>
<td>0000</td>
<td>0000</td>
<td>1111</td>
<td>0101</td>
<td>0101</td>
</tr>
<tr>
<td>x'</td>
<td>=</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>000a</td>
<td>eijk</td>
<td>uvwx</td>
<td>zBDF</td>
</tr>
</tbody>
</table>

(1) Suppose that the word width of the platform is 32 bits, we will divide the input sequence into blocks with width equal to 32 bits.
(2) Use Boolean operations to represent the calculation:

We can scan the string $x$ from the rightmost bit to the leftmost bit. If we meet a bit with mask equal to one, we will put that bit into the extracted string $x'$. The pseudo code for this naive implementation is:
Procedure Extraction(unsigned int x, unsigned int m)
    unsigned r, s, b;    // Result, shift, mask bit.
    r = 0;
    s = 0;
    do {
        b = m & 1;
        r = r | ((x & b) << s);
        s = s + b;
        x = x >> 1;
        m = m >> 1;
    } while (m != 0);
    return r;

(3) Optimize the calculation:
Instead of moving bits one by one toward right, we can shift several bits toward right for a fixed distance at the same time. To find appropriate group of bits for shifting toward right, we will decompose the distance between each bit and the rightmost bit into binary representations. The distance between a bit and the rightmost bit is defined as the number of 0s between it and the rightmost bit. We will shift the group of bits toward right with shifting distance equal to 1, 2, 4, 8, and 16, respectively. When the shifting distance is equal to \(2^k\), the bits with 1 in the \(k\)th bit of their binary decomposed distance will be put into the shifting group. For example, if the distance between a bit and the rightmost bit is 10011, it will be put into the shifting group when the shifting distance is 1, 2, and 16. Following is a more detailed example of how this optimization works:

1. Clear irrelevant bits by an and operation: \(x = x \& m\)
   \[x = \text{a000 e000 ijk0 0000 0000 uvwx 0z0B 00DF}\]
   distance(a) = 10011
   distance(e) = 10000
   distance(i) = distance(j) = distance(k) = 01101
   distance(u) = distance(v) = distance(w) = distance(x) = 00100
   distance(z) = 00011
   distance(B) = 00010
   distance(D) = 00001
2. Shift a, i, j, k, z, and D toward right for 1 bit.
   \[x = \text{a000 e000 ijk0 0000 0000 uvwx 0z0B 00DF} \quad ^{\wedge} \quad ^{\wedge\wedge}\]
   \[x' = \text{0a00 e000 0ijk 0000 0000 uvwx 00zB 00DF} \quad ^{\wedge} \quad ^{\wedge}\]
3. Shift a, z, and B toward right for 2 bits.
   \[x = \text{0a00 e000 0ijk 0000 0000 uvwx 00zB 00DF} \quad ^{\wedge} \quad ^{\wedge}\]
   \[x' = \text{000a e000 0ijk 0000 0000 uvwx 0000 zBDF}\]
4. Shift i, j, k, u, v, w, and x toward right for 4 bits.
   \[x = \text{000a e000 0ijk 0000 0000 uvwx 0000 zBDF}\]
5. Shift i, j, and k toward right for 8 bits.
\[ x = 000a\ e000\ 0000\ 0ijk\ 0000\ 0000\ uvwx\ zBDF \]
\[ x' = 000a\ e000\ 0000\ 0000\ 0000\ 0ijk\ uvwx\ zBDF \]

6. Shift a and e toward right for 16 bits.
\[ x = 000a\ e000\ 0000\ 0000\ 0000\ 0000\ 0ijk\ uvwx\ zBDF \]
\[ x' = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0ijk\ uvwx\ zBDF \]

The pseudo code for this optimized method is:

```c
Procedure Extraction(unsigned int x, unsigned int m)
    unsigned mk, mp, mv, t;
    int i;

    x = x & m;           // Clear irrelevant bits.
    mk = ~m << 1;        // We will count 0's to right.

    for (i = 0; i < 5; i++) {
        mp = mk ^ (mk << 1);           // Parallel prefix.
        mp = mp ^ (mp << 2);
        mp = mp ^ (mp << 4);
        mp = mp ^ (mp << 8);
        mp = mp ^ (mp << 16);
        mv = mp & m;                   // Bits to move.
        m = m ^ mv | (mv >> (1 << i)); // Extract m.
        t = x & mv;
        x = x ^ t | (t >> (1 << i));   // Extract x.
        mk = mk & ~mp;
    }
    return x;
```

(4) Gather computation results if necessary:
After the extraction for all the blocks, we still need to shift the extracted bits toward right. The map-reduce pattern can be used to achieve this goal.

- MD5 Hash Function [10]:

MD5 (Message Digest algorithm 5) is a widely used cryptographic hash function. Given an input string, it will first pad the input string until it is divisible by 512. Then the MD5 will compute a 128 bits hash value for the padded input string.
(1) The input string will be padded until its length is divisible by 512. To pad an input string, a 1 will be padded at the end of the string. It will follow by as many 0s as are required for bringing the length of the whole string up to 64 bits fewer than a multiple of 512. The remaining 64 bits are filled with a 64 bits integer representing the length of the original input string. After the padding procedure, the 512-divisible input string will be divided into blocks with width equal to 512 bits.

(2) The 512 bits input string will be broken into 16 words with the length of each word equal to 32. The 16 words (represented by 16 unsigned integers) will be manipulated with the 128 bits (represented by 4 unsigned integers) hash value generated from the previous block. The pseudo code of the standard MD5 algorithm is:

```c
Procedure MD5Block(unsigned int w[16], unsigned int h[4])
    int r[64] = {
        7, 12, 17, 22,  7, 12, 17, 22,
        7, 12, 17, 22,  7, 12, 17, 22,
        5, 9, 14, 20,  5, 9, 14, 20,
        5, 9, 14, 20,  5, 9, 14, 20,
        4, 11, 16, 23,  4, 11, 16, 23,
        4, 11, 16, 23,  4, 11, 16, 23,
        6, 10, 15, 21,  6, 10, 15, 21};
    for (i = 0; i < 64; i++)
        k[i] = floor(abs(sin(i + 1)) \times (2^32));

    int a = h[0];
    int b = h[1];
    int c = h[2];
    int d = h[3];
    int f, g;

    for (i = 0; i < 64; i++){
        if (i < 16){
            f = (b & c) | ((~b) & d);
            g = i;
        }
        else if (i >= 16 && i < 32){
            f = (d & b) | ((~d) & c);
            g = (5\times i + 1) \mod 16;
        }
        else if (i >=32 && i < 48){
            f = b ^ c ^ d;
            g = (3\times i + 5) \mod 16;
        }
        else if (i < 64){
            f = c ^ (b \or (~d));
            g = (7\times i) \mod 16;
        }
    }
```
int temp = d;
d = c;
c = b;
b = b + leftrotate((a + f + k[i] + w[g]) , r[i]);
a = temp;
}
h[0]+=a;
h[1]+=b;
h[2]+=c;
h[3]+=d;

Subroutine leftrotate (unsigned x, unsigned c)

    return (x << c) | (x >> (32-c));

(3) The MD5 algorithm is a standard. No optimization is required.
(4) Gather computation results if necessary:
    The output 128 bits hash value of a block will be the input for the next block. The overall
    MD5 calculation for a string with n blocks is:

Procedure MD5(unsigned int x[n], unsigned int n)
    int h[4] = {0x67452301, 0xEFCDAB89, 0x98BADCFE,0x1 0325476};
    for (i = 0; i < n; i++){
        break x[i] into 16 words w[16];
        MD5Block(w, h);
    }
    return h;

The output of a block will be the input of the successive block. Therefore, we can apply
the pipeline pattern (please see the pipeline pattern for more information) on the blocks to
calculate the final hash value of the input string.

Known uses

- Cyclic redundancy check [12]
- Linear Feedback Shift Register [13]
- Error correcting codes [14]
- Popcount [15]
- DES [5][16]
- Rijndael [4][17]
- AES [17]
- SHA [18]
- Regular expressions [19]
- String manipulation [9]
- String hashing [10]

**Related Patterns**

- **Map-Reduce**
  If the computation within each block is independent to each other, and we need to gather the results from each block, we can adopt the map-reduce pattern for modeling our computation.

- **Data Parallelism**
  By breaking the data into bits, we greatly multiplies the amount of data items. Therefore, we can increase the data parallelism of the data by exploring the bit level parallelism.

- **Pipeline**
  If the output of a block will be the input of next block (for example, the MD5 [10] function has such property), we can model the computation as a pipeline pattern.

- **Digital Circuits**
  Digital circuits deals with bitwise operations as well as the arithmetic operations (ADD, SUBTRACT, ALU) on bits or bit-vectors. Therefore, it is a natural fit for the circuits pattern from the view of algorithmic strategies.

- **SIMD**
  When applying bitwise operations (AND, OR, NOT) on a word, we can adopt the SIMD pattern to model the parallel implementation of the operation.

**References**

3. Seidel, E., "Preparing tomorrows cryptography: Parallel computation via multiple processors, vector processing, and multi-cored chips." (Senior Honors Project, Lawrence University)

Author

Revised 03/15/2009: Author: Bor-Yiing Su   Shepherd: Sayak Ray
Revised 01/19/2009: Mark Hoemmen
Revised 07/22/2008: Kurt Keutzer, Tim Mattson, Youngmin Yi
Original Author: Bryan Catanzaro