Supplement Information

METHOD

Mix Function:

Wyhash and wyrand is based on a mix function call MUM that mix two 64-bit integer A and B to produce a 64-bit integer C: MUM (A, B) => C. @vnmakarov released the original version of MUM on Mother's Day [22].

```
uint64_t  mum(uint64_t A, uint64_t B){
  __uint128_t c=(__uint128_t)A*B;
  return  (c>>64)^c;
}
```

Despite the nominal 128-bit multiplication, the actual instructions on 64-bit machines are as simple as follow:

```
MUM(unsigned long, unsigned long):
mov rax, rdi
mul rsi
xor rax, rdx
ret
```

Our further improvements on MUM is the masked-MUM: MUM (A^secret, B^seed), where secret is a predefined 64-bit integer with 32 1bits and seed is current status with a uniform distributed number of 1bits. The masked-MUM can protect the MUM from being zero (Discussion), randomize the distribution of real data and produce an avalanche effect. We observed experimentally that just two rounds of masked-MUM suffice to pass all statistical tests.

wyhash Hash Function

wyhash hash function is based on masked-MUM and contains three parts: The batch part the minibatch part and finalization part. The batch part processes most of the data as 64-byte blocks while the minibatch part process the reminder of 64 bytes blocks as 16 bytes mini blocks before finalization. The finalization part processes the tail bytes (<=16). The code is shown below where the _wyr# functions reads # byte from the key using memcpy.

```
static inline uint64_t _wyfinish16(const uint8_t *p, uint64_t len, uint64_t seed, const uint64_t *secret, uint64_t i){
#if(WYHASH_CONDOM>0)
  uint64_t a, b;
  if(_likely_(i<=8)){
    if(_likely_(i>=4)){ a=_wyr4(p); b=_wyr4(p+i-4); }
    else if (_likely_(i)){ a=_wyr3(p,i); b=0; }
    else a=b=0;
  }
  else{ a=_wyr8(p); b=_wyr8(p+i-8); }
  return mum(secret[1]^len,mum(a^secret[1], b^seed));
#else
  #define oneshot_shift ((i<8)*((8-i)<<3))
  return mum(secret[1]^len,mum((_wyr8(p)<<oneshot_shift)^secret[1],(_wyr8(p+i-8)>>oneshot_shift)^seed));
#endif
}

static inline uint64_t _wyfinish(const uint8_t *p, uint64_t len, uint64_t seed, const uint64_t *secret, uint64_t i){
  if(_likely_(i<=16)) return _wyfinish16(p,len,seed,secret,i);
  return _wyfinish(p+16,len,mum(_wyr8(p)^secret[1],_wyr8(p+8)^seed),secret,i-16);
}

static inline uint64_t wyhash(const void *key, uint64_t len, uint64_t seed, const uint64_t *secret){
  const uint8_t *p=(const uint8_t *)key;
  uint64_t i=len; seed^=*secret;
  if(_unlikely_(i>64)){
    uint64_t see1=seed;
    do{
      seed=mum(_wyr8(p)^secret[1],_wyr8(p+8)^seed)^mum(_wyr8(p+16)^secret[2],_wyr8(p+24)^seed);
      see1=mum(_wyr8(p+32)^secret[3],_wyr8(p+40)^see1)^mum(_wyr8(p+48)^secret[4],_wyr8(p+56)^see1);
      p+=64; i-=64;
    }while(i>64);
    seed^=see1;
  }
  return _wyfinish(p,len,seed,secret,i);
}
```

wyrand PRNG

Our PRNG is named wyrand is even simpler. It keeps a 64-bit internal status and updates it by adding a 64-bit prime. The internal status is mixed with masked itself by MUM function to produce a pseudorandom number. It is obvious that its cycle length is 2^{64} as p0 is a large prime.

```
uint64_t wyrand(uint64_t *seed) {
  *seed+=p0;
  return mum(*seed^p1,*seed);
}
```

Benchmark

We validate and benchmark wyhash and wyrand on a server with 2X Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz, 64GB memory and 2*2TB SSD hard driver. SMHasher [9] is used to validate and benchmark hash functions. The original hash map speed test codes have an unnecessary overhead of string copying that slow down the benchmark. We replace the following lines

std::string line = *it;

with

std::string &line = *it;

in SpeedTest.cpp.

wyrand compiled code:

```
wyrand(unsigned long*):
        movabs  rax, -6884282663029611473
        add     rax, QWORD PTR [rdi]
        mov     rcx, rax
        mov     QWORD PTR [rdi], rax
        movabs  rax, -1800455987208640293
        xor     rax, rcx
        mul     rcx
        xor     rax, rdx
        ret
```

wyhash compiled code:

```
wyhash(void const*, unsigned long, unsigned long, unsigned lon
g const*):
        push    r14
        mov     r10, rsi
        push    r13
        push    r12
        push    rbp
        push    rbx
        xor     rdx, QWORD PTR [rcx]
        mov     r9, QWORD PTR [rcx+8]
        mov     r8, rdx
        cmp     rsi, 64
        ja      .L18
        cmp     r10, 16
        ja      .L4
.L9:
        cmp     r10, 8
        ja      .L5
.L19:
        cmp     r10, 3
        jbe     .L6
        mov     eax, DWORD PTR [rdi-4+r10]
        xor     r8, rax
        mov     eax, DWORD PTR [rdi]
        xor     rax, r9
.L7:
        mul     r8
```

```
        xor     rsi, r9
        pop     rbx
        pop     rbp
        pop     r12
        pop     r13
        pop     r14
        xor     rax, rdx
        mul     rsi
        xor     rax, rdx
        ret
.L18:
        lea     r14, [rsi-65]
        mov     r13, QWORD PTR [rcx+16]
        mov     r12, QWORD PTR [rcx+24]
        shr     r14, 6
        mov     rbp, QWORD PTR [rcx+32]
        mov     rcx, rdx
        lea     rbx, [r14+1]
        sal     rbx, 6
        add     rbx, rdi
.L3:
        mov     r10, QWORD PTR [rdi]
        mov     rax, QWORD PTR [rdi+8]
        add     rdi, 64
        xor     r10, r9
        xor     rax, r8
        mul     r10
        mov     r11, rdx
        mov     r10, rax
        mov     rdx, QWORD PTR [rdi-48]
        mov     rax, QWORD PTR [rdi-40]
        xor     rdx, r13
        xor     rax, r8
        mul     rdx
        xor     r10, rax
        mov     r8, rdx
        mov     rax, QWORD PTR [rdi-32]
        xor     r10, r11
        xor     r8, r10
        mov     r10, QWORD PTR [rdi-24]
        xor     rax, r12
        xor     r10, rcx
        xor     rcx, QWORD PTR [rdi-8]
        mul     r10
```

```
        mov     r10, rax
        mov     rax, QWORD PTR [rdi-16]
        mov     r11, rdx
        xor     rax, rbp
        mul     rcx
        xor     r10, rax
        mov     rcx, rdx
        xor     r10, r11
        xor     rcx, r10
        cmp     rdi, rbx
        jne     .L3
        neg     r14
        xor     r8, rcx
        sal     r14, 6
        lea     r10, [rsi-64+r14]
        cmp     r10, 16
        jbe     .L9
.L4:
        lea     rcx, [r10-17]
        shr     rcx, 4
        lea     r11, [rcx+1]
        sal     r11, 4
        add     r11, rdi
.L8:
        mov     rax, QWORD PTR [rdi]
        xor     r8, QWORD PTR [rdi+8]
        add     rdi, 16
        xor     rax, r9
        mul     r8
        mov     r8, rdx
        xor     r8, rax
        cmp     rdi, r11
        jne     .L8
        neg     rcx
        sal     rcx, 4
        lea     r10, [r10-16+rcx]
        cmp     r10, 8
        jbe     .L19
.L5:
        mov     rax, QWORD PTR [rdi]
        xor     r8, QWORD PTR [rdi-8+r10]
        xor     rax, r9
        jmp     .L7
.L6:
```

```
        test    r10, r10
        je      .L11
        lea     eax, [r10-1]
        movzx   edx, BYTE PTR [rdi]
        shr     r10d
        movzx   eax, BYTE PTR [rdi+rax]
        sal     rdx, 16
        or      rax, rdx
        movzx   edx, BYTE PTR [rdi+r10]
        sal     rdx, 8
        or      rax, rdx
        xor     rax, r9
        jmp     .L7
.L11:
        mov     rax, r9
        jmp     .L7
```

FigureS1: Compiled Code Size Hash Functions