ESE532 Project P1 Report

Ritika Gupta, Taylor Nelms, and Nishanth Shyamkumar

October 30, 2019

1. Our group makeup is Ritika Gupta, Taylor Nelms, and Nishanth Shyamkumar.
2. (a) We end up with $64ns$ to process each $64b$ word of input, which comes out to 76.8 (so, 76) cycles for a 1.2GHz processor.
   (b) By similar logic as the last question, with a 200MHz clock, we end up with 12.8 (so, 12) cycles to process all of the input.
3. (a) (i) **Content-Defined Chunking**:

```
skip input to minChunkSize - windowSize
buffer = input[minChunkSize - windowSize : minChunkSize]
curHash = 0
for byte in buffer:
    curHash += hash(byte)
if curHash == 0:
    markChunkBreak()
else:
    while (curHash != 0 and (notAtMaxChunkSize())):
        curHash -= hash(buffer[0])
        moveBufferWindow()
        readNextByte()
        curHash += hash(buffer[windowSize - 1])
    markChunkBreak()
```

   (ii) **SHA-256**:

```
# RITIKA'S VERSION
hash[0:7] = initializeHashValues()
k[0:63] = initializeMessageSchedule()

for each chunk
        for each 512_bit_subchunk m[]    # 64 byte sunchunk m[]
                for ( ; i < 64; ++i)     # For eevry byte in subchunk
                        m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15])
                                                + m[i - 16];

                # call update function
                sha_update(data, len)

        # Compute hash for last incomplete chunk, if any
        if (last_subchunk < 448 bits)
                append 1
                append 0s to make 448 bits
        else
                append 1
                append 0s to make 512 bits
                sha_update(last_subchunk, 64)
                last_chunk[] = {0}

        append_0s_till_448_bit_subchunk()
```

```
        append_msg_len_in_last_64_bits()
        sha_update(last_subchunk)
        convert_hash_values_big_endian()

sha_update(data, len)
        # initialize working variables with previous hash values
        a = hash[0]
        b = hash[1]
        c = hash[2]
        d = hash[3]
        e = hash[4]
        f = hash[5]
        g = hash[6]
        h = hash[7]

        # update_working_variables()
        for ( ; i < 64; ++i)     # For every byte in subchunk
                t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];
                t2 = EP0(a) + MAJ(a,b,c);
                h = g;
                g = f;
                f = e;
                e = d + t1;
                d = c;
                c = b;
                b = a;
                a = t1 + t2;

        # increment hash values by corresponding working variable
        hash[0] += a
        hash[1] += b
        hash[2] += c
        hash[3] += d
        hash[4] += e
        hash[5] += f
        hash[6] += g
        hash[7] += h

# TAYLOR'S VERSION
h[0:7] = initializeHashValues()
k[0:63] = initializeRoundConstants()
padInitialMessage()#pads to a 512-bit boundary
for chunk512bitSection in chunk:
    w[0:15] = chunk512bitSection

    #Extend the first 16 words into the remaining 48 words w[16..63] of the message s
    for i from 16 to 63
        s0 := (w[i-15] rightrotate  7) xor (w[i-15] rightrotate 18) xor (w[i-15] righ
3)
        s1 := (w[i- 2] rightrotate 17) xor (w[i- 2] rightrotate 19) xor (w[i- 2] righ
        w[i] := w[i-16] + s0 + w[i-7] + s1

    a:h = h[0:7]
    #Compression function main loop:
    for i from 0 to 63
        S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
        ch := (e and f) xor ((not e) and g)
        temp1 := h + S1 + ch + k[i] + w[i]
        S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
        maj := (a and b) xor (a and c) xor (b and c)
```

```
            temp2 := S0 + maj

            h := g
            g := f
            f := e
            e := d + temp1
            d := c
            c := b
            b := a
            a := temp1 + temp2

        h[0:7] += [a:h]

    digest = h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7
```

Credit: Wikipedia

(iii) **Chunk Matching**:

```
if shaResult in chunkDictionary:
    send(shaResult)
else:
    send(LZW(rawChunk))
```

(iv) **LZW Encoding**:

```
table = {}
for i in range(256):
    table[i] = i
curPos = 256
STRING = Input.read()
while(True):
    CHAR = Input.read()
    if STRING + CHAR in table.values():
        STRING += CHAR
    else:
        Output.write(table[STRING])
        table[STRING + CHAR] = curPos
        curPos += 1
        STRING = CHAR
    if Input.isDone():
        break
```

Credit: https://www.dspguide.com/ch27/5.htm

(b) **Memory Requirements**

(i) **Content-Defined Chunking**:
We'll need a rolling hash window's worth of working memory, spanning 16ish bytes.

(ii) **SHA-256**:
Eight 32-bit span of memory to hold hash values, sixty four 32-bit span of memory for storing message schedule, ten 32-bit span of memory for working vraiables, and 64-byte SHA-block span of memory

(iii) **Chunk Matching**:
We'll want a table to store hash values for index purposes, which would require at least 8 bytes times the maximum number of chunks to be processed.

(iv) **LZW Encoding**:
This is a somewhat tricky question given the associative memory involved, but it will be on the scale of roughly MAX_CHUNK_SIZE entries times 12 bits.

(c) **Computational Requirements**

(i) **Content-Defined Chunking**:

(ii) **SHA-256**:

(iii) **Chunk Matching**:

(iv) **LZW Encoding**:

(d) **Memory Access Requirements**

(i) **Content-Defined Chunking**:

(ii) **SHA-256**:

(iii) **Chunk Matching**:

(iv) **LZW Encoding**:

(e)

4. (a) The **LZW** and **SHA-256** operations can feasibly be done in parallel, as neither depends on the other.

(b) **Task-Level Parallelism**

(i) **Content-Defined Chunking**:

(ii) **SHA-256**:

(iii) **Chunk Matching**:
There aren't many tasks here, so task-level parallelism seems a rather useless thing to pursue.

(iv) **LZW Encoding**:

(c) **Data-Level Parallelism**

(i) **Content-Defined Chunking**:
Any particular window of data could, feasibly, be rabin-fingerprinted at the same time; however, given the computational efficiency of doing a rolling hash sequentially, this seems ill-advised.

(ii) **SHA-256**:

(iii) **Chunk Matching**:
With each table lookup result depending (potentially) on the last table entry, engaging in any parallelism here seems foolish.

(iv) **LZW Encoding**:

(d) **Pipeline Parallelism**

(i) **Content-Defined Chunking**:
Any particular window of data could, feasibly, be rabin-fingerprinted at the same time; however, given the computational efficiency of doing a rolling hash sequentially, this seems ill-advised.

(ii) **SHA-256**:
The entire SHA main loop could be feasibly pipelined.

(iii) **Chunk Matching**:
There aren't really enough tasks here to pipeline sensibly.

(iv) **LZW Encoding**:
If the end goal is to fill some kind of input/output buffers while dealing with streaming I/O data, any processes like that could be pipelined pretty well. For any of the internals, though, the logic should be atomic enough that the pipelining approach may not work optimally.

(e)

5. (a)

(b)

(c)

(d)

(e)

(f)