Complexity Measurement and Fault Detection Techniques for H.264 Optimized Functions

Hao Zhang
School of Information Science and Engineering Central South University, Changsha, Hunan ,China
Email: hao@csu.edu.cn

Yuetang Deng[2], Zhenye Liu[3], Yuan Zhao[1] , Haiyan Zhan[1]
[1]School of Information Science and Engineering Central South University, Changsha, Hunan, China
[2]Tencent, Inc. ShenZhen, China
[3]AT&T Labs, CA, USA
Email:hyzhanm@gmail.com

***Abstract*—In this paper, we systematically studied the complexity measurement, SIMD (Single Input Multiple Data) fault types and testing methodologies for H.264 codec. To the best of our knowledge, it is the first attempt to address these problems. Firstly, two complexity metrics are calculated for various optimized functions in H.264 reference software. These measures have been found to be strongly correlated to the number of faults in software testing. Secondly, we introduced a new category of SIMD faults. Conformance testing, random testing and manual testing are proposed to deal with these SIMD faults as well as conventional faults. Results have shown that, conformance testing, often used as a mechanism to verify the conformity of a decoder under test (DUT), can also be used to discover faults in the studied optimized functions. Random testing is able to detect simple faults at both encoder and decoder functions. Manual testing is especially effective for difficult faults. In practice, one or more of the three techniques can be chosen as needed to increase fault detection rate and speed for H.264 video codec testing.**

***Index Terms*—video coding, software testing, single input multiple data, optimized functions**

I. INTRODUCTION

Video coding standards have been developing rapidly in recent years. Compression ratios have been greatly improved with greater coder complexity [1], [2]. The upcoming High Efficiency Video Coding (HEVC) could be more computationally expensive than H.264 with further improvement on compression efficiency [3]. Although advancement of hardware is speeding up the encoding/decoding process, computational efficiency is still considered as an important performance metric for video codec.

It has been found that some functions in H.264 video decoder such as deblock filter and interpolation are of considerable computational complexity [4], [5]. A common way to reduce their execution time is hand optimizing them with SIMD technologies that are available in various hardware platforms, e.g., CPU, GPU, DSP, etc. [5], [6]. Because each platform supports a unique SIMD instruction set, various optimized versions are required to be developed, tested and maintained for each function. It is well known that functions coded with assembly languages are generally hard to read and maintain and thus very likely to contain faults that might generate mismatches. Deploying such a system leads to unpleasant artifacts and finally costs system makers a lot of time and money on patch distributions. Fig.1 gives such an example when decoding the conformance bitstream 'BA1_FT_C' [7] with a faulty decoder. Subjective viewing could be employed to find easy faults, while it is insufficient to detect difficult faults in the required time frame. This is due to many reasons, e.g., the condition to intrigue the fault is not met, or the fault related artifacts are only visible after a long period of time. Therefore, it is essential to conduct thorough software testing before deployment.

Recommendation ITU-T H.264.1 specifies bitstreams to check the conformance of decoders under test (DUT) [7]. A decoder is considered conformant to the specification if the decoded frames are identical as those decoded by reference software. Those bitstreams are provided to test various H.264 features for different profiles. However, whether the conformance test could be used to detect programming faults in the source code of a DUT is unknown. This issue is going to be investigated in this paper.

Conformance testing is insufficient for video codec testing because it can only test decoders. Moreover, it is time consuming compared to some other testing techniques. Lastly, it is not able to locate faulty functions. Therefore, other software testing techniques are needed. Among them, unit testing is one of the most important procedures for quick error revelation. To unit test different versions of an optimized function, a correct version should be obtained before hand. This version is usually coded with a high level language (such as C) and considered error free (by thorough code review, comparisons with the specifications or reference software, extensive experiments, etc.). Unit testing is then performed to find out whether the outputs of the optimized versions match that of the correct version with some selected test cases.

doi:10.4304/jsw.7.8.1845-1854

Figure 1. A decoded image demonstrating some artifacts with a fault

Actually, thoroughly testing all the optimized function is sometimes a formidable task due to the limitation of engineer resource. Two widely adopted software complexity metrics--- LOC (line of code) and Cyclomatic Complexity measures could be used to predict bug densities. Furthermore, two code coverage metrics are calculated when random testing is used. These metrics reveal the software complexity of various optimized functions that is helpful to codec engineers in task prioritization.

Another important procedure in software testing is fault injection. We use typical faults in the software testing literature to mimic faults in real applications. Additionally, we consider the conventional fault types insufficient for optimized functions and we introduce a new SIMD fault category. With various faults injected into the optimized functions, test cases are designed by random and manual testing and their effectiveness is compared. Random testing generates random input of all the parameters within their admissible ranges; for the manual test case design, each test case is designed manually based on past experiences. This technique aims at difficult faults that could not be efficiently detected by conformance testing and random testing techniques.

The paper is organized as follows. Sec. Ⅱ briefly describes related work on video codec testing. A brief introduction of SIMD technologies is then given in Sec. Ⅲ. Two complexity metrics are calculated in Sec.Ⅳ. Sec. Ⅴ describes the proposed new fault category. Sec. Ⅵ gives two examples of the manual test case design technique. In Sec.Ⅶ, experiments are conduct to compare the effectiveness of the three testing techniques. Finally, conclusions and future work are described in Sec.Ⅷ.

II. Related Work

Software testing has long been investigated. It helps find all sorts of faults to make the final product work as expected. There exist various testing methods such as random testing, mutation testing, data flow testing, etc. [8-10]. Although these testing techniques have been used widely in many fields, their applications to video coding are still very limited. The only known research activities focus on the conformance testing of standard compliant decoders [11-13]. These conformance bitstreams may be useful to test whether the features in some specific profile and level are supported. Their effectiveness is not verified for general software faults, especially in optimized functions written with SIMD instructions. In this paper, we consider the video codec testing from a different angle --- what are the frequently encountered SIMD faults and how to detect them effectively.

III A Brief Introduction To SIMD Technologies

SIMD was a technology exploited by super computers, but nowadays it is deployed in all sorts of hardware platforms and widely exploited to accelerate various applications, e.g., image processing, video coding, etc. Big corporations are pushing this technology to a new level. For example, Altivec was used in Powerpc to support various computationally expensive operations. Moreover, starting from 64-bit MMX registers, Intel and Amd have been frequently introducing new instructions into their powerful SIMD instruction sets. For example, the SIMD instruction set in Intel CPUs has been evolving from MMX to SSE(Streaming SIMD Extensions), SSE2, SSE3, SSSE3 and SSE4. During this process, eight 128-bit SIMD registers are added for better performance. AMD recently developed SSE5 and introduced more instructions. Longer registers will be available in Advanced Vector Extensions (AVX). Some DSPs also support SIMD instruction sets and this could be very useful in embedded applications. For convenience, we would like to use Intel CPUs to generate our test cases, though many of them could be used by other processors.

Basically, SIMD instructions realize parallelization and improve the computational efficiency by processing multiple operations simultaneously. Intel&Amd gradually add extensions to their SIMD instruction set. For backward compatibility, programmers need to write various versions of optimized functions for all the supported CPUs. A CPUID instruction could be used to find the supported instruction sets [14]. However, the coexistence of different code versions for the same functionality creates a potential source of fault generation. To make things worse, SIMD programmers may choose various assemblers under different operating systems. For instance, some compilers (such as GAS) support AT&T syntax, while others (such as NASM) support Intel syntax. These two assembly syntaxes differ in many places such as function names, register locations, constant representations, etc. Those different assembly syntaxes further increase the number of optimized versions. One way to avoid developing the same function in two syntaxes is to use (or write your own) automatic syntax converters, however, any fault in the converter may generate new faults and finally result in a correct version for one OS and a wrong version for the other OS. Mismatch problem will arise when two clients using such code versions in different OSs talk to each other. Another option is to use intrinsic instead of pure assembly [14]. For convenience, we will use intrinsic to create all the optimized versions in our experiments.

IV Complexity Measurement

There exist many metrics for program complexity evaluation, where LOC and Cyclomatic Complexity are widely used. Both LOC and Cyclomatic Complexity have been shown to be correlated to fault density [15], and hence, it is necessary to calculate these metrics for H.264 optimized functions. LOC is the number of lines in the source code, and it could be easily calculated from the source code of each function. Cyclomatic Complexity is a software metric based on the control flow graph. It is given by the following formula [15], [16]:

$$v(G) = e - n + p \quad (1)$$

Where $v(G)$ denotes the Cyclomatic Complexity of the control flow graph G, which contains e edges, n vertices and p connected components. It is actually equal to the number of linearly independent path in graph G and it is known that larger Cyclomatic Complexity values correspond to more test cases for path coverage [15], [16]. An example is given in Fig.2, based on the specification and source code of JM16.0. In the graph, S denotes the entry point and E represents the exit point. For this simple example, $e = 9, n = 7, p = 1$, so Cyclomatic Complexity is calculated as $e - n + p = 3$.

```
Function Start

Void forward4x4(int **block,
int **tblock, int pos_y, int
pos_x)
{
1……
1  int *pTmp = tmp, *pblock;
1……

   //Horizontal
2  for (i=pos_y;
        i<pos_y+BLOCK_SIZE;
        i++)
  {

3  ……

   }

   //Vertical
4 for
(i=0;i<BLOCK_SIZE;i++)
  {

5  ……

  }
              }
        Function End
```

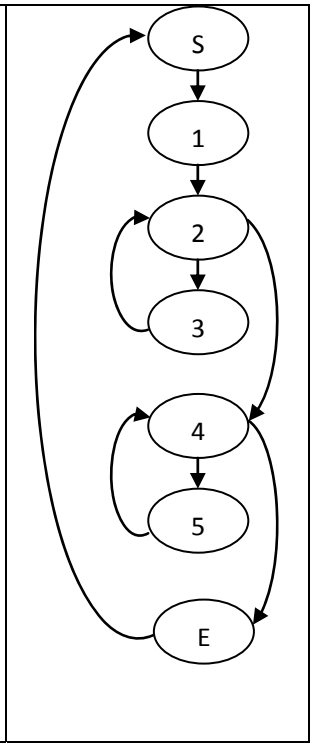

Figure 2. A decoded image demonstrating some artifacts with a fault in the optimized code of luma interpolation.

LOC and Cyclomatic Complexity metrics are calculated for the optimized functions as shown in Tab. Ⅰ. A simplified calculation method for Cyclomatic Complexity is used: it is equal to the number of predicates plus one in a structured program with only one entrance point and one exit point [16]. Among the tested modules in Tab. Ⅰ, Luma Deblock and Luma Interpolation have larger complexity values than other functions. It is recommended that Cyclomatic Complexity should be no more than 10 [16]. We can see that the complexity metrics of these two modules are much larger than this recommended value. Therefore, codec engineers should pay attention to these functions and conduct thorough testing for them.

V LIST OF FAULTS

A. Regular Faults

TABLEⅠ
COMPLEXITY METRICS FOR VARIOUS ENCODER MODULES IN JM16.0

Module Names	Loc	Cyclomatic Complexity
Forward Transform4x4	40	3
Inverse transform4x4	40	3
Quant+Dequant4x4	56	7
Luma Interpolation	527	88
Luma Deblock	108	18

We are attempting to attack two categories of faults. The first category in Tab.Ⅱ includes regular faults used in [17]. These faults could occur in modules written with any programming language and so they will be used in our tests for H.264 optimized functions.

B. SIMD Faults

Other than the regular faults listed in Sec.Ⅴ-A, we propose a new category of faults, which are caused by incorrect implementations of various SIMD instructions. It consists of five types as shown in Tab.Ⅲ: overflow, incorrect signed/unsigned right shift, incorrect signed/unsigned extension, incorrect signed/unsigned saturation, incorrect rounding. Please note that Tab.Ⅲ only includes those faults we encountered in the past. We It consists of five types as shown in Tab.Ⅲ: overflow,

TABLE II
FAULT TYPES IN CATEGORY I

Fault types	
Missing path faults	
Incorrect predicate faults	Relational operator replacement
	Logical operator replacement
Incorrect Computation Statement	Incorrect initialization
	Incorrect constant
	Incorrect precedence
	Incorrect array element reference
	Incorrect pointer operation
	Same type variable replacement
	Arithmetic operator replacement
	Miscellaneous
Missing computation statement	Delete a complete statement
	Delete a part of a statement
Incorrect number of loop iterations	
Missing clause in predicates	

incorrect signed/unsigned right shift, incorrect signed/unsigned extension, incorrect signed/unsigned saturation, incorrect rounding. Please note that Tab.Ⅲ only includes those faults we encountered in the past. We hope to see more discussions and results on this topic in the future.

1) Overflow:

A 128-bit XMM register can only hold eight 16-bit or four 32-bit integers. Without careful analysis of variable dynamic ranges, SIMD programmers may allocate 16 bits for a variable when 32 bits are actually needed. This is usually intrigued by ambitions to simultaneously process as many pixels as possible (so higher coding speeding may be achieved). The quantization equation shown in (2) provides a good example for this fault type.

TABLEIII
FAULT TYPES IN CATEGORY II

Fault Types
Overflow
Incorrect signed/unsigned right shift
Incorrect signed/unsigned extension
Incorrect signed/unsigned saturation
Incorrect rounding

$$\begin{aligned} |Z_{ij}| &= \left(|W_{ij}| \cdot MF + f\right) >> qbits \\ sign(Z_{ij}) &= sign(W_{ij}) \end{aligned} \tag{2}$$

Variables in (2) are explained in [18]: f is $2^{qbits}/3$ for Intra blocks or $2^{qbits}/6$ for Inter blocks, qbits = 15+floor(QP/6), MF is the multiplication factor defined in Tab. Ⅳ, '>>' denotes a binary right shift, $W = C_f X C_f^T$ is the horizontal and vertical forward transform of residual matrix X with C_f^T being the transpose of C_f.

C_f is given as below[18]:

TABLE IV.
MULTIPLICATION FACTOR MF [18]

QP	Position (0,0),(2,0),(2,2),(0,2)	Positions (1,1),(1,3),(3,1),(3,3)	Other positions
0	13107	5243	8066
1	11916	4660	7490
2	10082	4194	6554
3	9362	3647	5825
4	8192	3355	5243

$$C_f = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & 1 \end{pmatrix}$$

With C programming, only one element in W can be processed at a time with (2). Using 16 bits for each element in W enables a XMM register to load 8 variables. However, using 16 bits in this case will cause the overflow fault because the multiplication $|W_{ij}|MF$ could easily go out of the 16-bit range (W_{ij} has a dynamic range of -9180 to 9180 and MF could be as large as 13107 [18]). The field testing may not immediately result in artifacts, because sometimes the residual signal aptitudes are too small to lead to the overflow. In the experimental section, it can be seen that this fault is detectable with random testing.

2) Incorrect Signed/Unsigned Right Shift:

Right shifts are widely used in video coding. It's known that there are two types of right shifts: the so called 'logical right shift' and 'arithmetic right shift'. Arithmetic right shift differs from logical right shift by preserving a number's sign bit [14]. C programmers can solely use '>>' and the compiler will automatically convert it into the right instruction. On the contrary, SIMD programmers have to select the correct instruction by themselves (e.g., psrlw and psraw represent logical and arithmetic right shift of each 16-bit element in a XMM register respectively). This should be done carefully; otherwise, faults could be made out of incorrect right shift implementations. An example of this fault occurred in the Luma Deblock function is given in (3) [19]:

$$\begin{aligned} p_1^{'} = p_1 + clip\ 3(-t_{C0}, t_{C0}, (p_2 + \\ ((p_0 + q_0 + 1) >> 1) - (p_1 << 1) \\ \text{where } a_p = |p_2 - p_0|. \end{aligned} \tag{3}$$

The last right shift is an arithmetic one. A fault is created when it is replaced by a logical right shift. This fault is not difficult to detect with both random testing and conformance testing.

3) Incorrect Signed/Unsigned Extension:

In SIMD implementations, it is up to the programmer to decide the number of bits for each variable. During calculation, some intermediate results may have wider dynamic ranges than the original input and so signed or unsigned extensions are needed. However, before SSE4.1, there are no signed SIMD extension instructions for Intel CPUs, i.e., only unsigned zero extension instructions such as 'punpcklbw' and 'punpcklwd' are supported. Faults could be generated if a SIMD programmer uses zero extensions when signed extensions are required. An example for this can be found in Half-pel Luma Interpolation formula shown in (4) [19], where *A, C, G, M, R, T* denote the pixel values at integer positions with a dynamic range of 0 to 255, inclusive; *cc, dd, ee, m_1, ff* are half-pel pixel values derived in the same manner as the derivation of h_1, which have a dynamic range of -2550 to 10710, inclusive [19],[20]. Suppose h_1 is represented by 16 bits within a XMM register, signed extensions should be used because the dynamic range of j_1 exceeds 16 bits. If zero extensions are used here instead, a fault is

generated that is detectable with random testing and conformance testing.

$$h_1 = (A - 5*C + 20*G + 20*M - 5*R + T)$$
$$j_1 = (cc - 5*dd + 20*h_1 + 20*m_1 - 5*ee + ff \tag{4}$$

4) Incorrect Signed/Unsigned Saturation:

One of signed/unsigned saturation instructions in MMX/SSE2 is PACKSSWB/PACKUSWB, which packs signed 16-bit integers into 8-bit signed/unsigned integers and saturate [14]. Signed and unsigned saturations map variables into different ranges, e.g., '-1' would be mapped into 0 with unsigned saturation and '-1' with signed saturation. Hence, faults are generated when incorrect instructions are used.

5) Incorrect Rounding:

Code lines like $(x + y + 2^{n-1}) >> n$ appear frequently in many functions. It is not a rare case for a programmer to forget adding the rounding offset 2^{n-1} in this expression. Additionally, a programmer may use a wrong rounding offset value 2^{m-1} with $m \neq n$. Furthermore, in the case when various rounding offsets are stored in a constant array, it is not uncommon for programmers to modify the constant array without changing the values of corresponding address registers.

An example is given in Tab. Ⅴ , where a simple program is shown with two versions: a pure C code version on the top of the table and a SIMD version written with inline assembly (Intel syntax). The function includes eight additions and right shift operations, which is replaced by three SSE2 instructions. The rounding offset is added by 'paddw xmm1, [constArray+16]'. A fault is generated if a programmer modifies constarray by

TABLE VI.
AN EXAMPLE OF INCORRECT ROUNDING FAULTS

```
short x[8] = {1,2,3,4,5,6,7,8};
short x[8] = {1,1,1,1,1,1,1,1};
_declspec(align(16)) short constArray[] = {
     2,2,2,2,2,2,2,2,
     4,4,4,4,4,4,4,4,
};
for (int i= 0; i< 8; i++)
    z[i] = (x[i]+y[i]+4)>>3;
```

```
_asm {
          movdqu xmm1, [x]
          movdqu xmm2, [y]
          paddw   xmm1, xmm2
          paddw   xmm1, [constArray+16]
          psraw   xmm1, 3
          movdqu [z], xmm1
      }
```

adding into it another constant (Tab.Ⅵ) while forgetting to change the address offset in the assembly code.

Another example is incorrect implementation of instruction 'pavgb xmm1, xmm2', which calculates the average of xmm1 and xmm2 with rounding. A programmer creates a fault by mistakenly implementing it as an average instruction without rounding.

TABLE V.
A MODIFIED ROUNDING OFFSET TABLE

```
_declspec(align(16)) short constArray [] = {
      1,1,1,1,1,1,1,1,
      2,2,2,2,2,2,2,2,
      4,4,4,4,4,4,4,4,
};
```

VI TEST CASE DESIGN VIA MANUAL TESTING

Since the number of optimized functions in H.264 video codec is not very large, and most of those functions (at least in the decoder) are not revisable after the standards are finalized, it is meaningful and useful to design a test suite for those faults that are difficult if not possible to detect by random testing and conformance testing. Test cases for some faults are easy to design. For instance, for the incorrect signed/unsigned right shift fault described in Sec.Ⅴ-B2, $bS = 1, \alpha = 255, \beta = 18,$ $t_{C0} = 2, p_0 = p_2 = q_0 = 0, p_1 = 1$ can be used a test case: the correct code would generate an output $p_1' = -1,$ while the faulty code with a logical right shift would generate a different output. However, it is not so obvious to design test cases for *Incorrect constant and Incorrect rounding* faults, which are shown to be difficult to detect. The designing process for these two fault types are described as below.

A. Incorrect Constant

Another example is incorrect implementation of instruction 'pavgb xmm1, xmm2', which calculates the average of xmm1 and xmm2 with rounding. A programmer creates a fault by mistakenly implementing it as an average instruction without rounding.

Incorrect constant faults will be generated in the quantization defined in (2) if an incorrect multiplication factor (denoted by *EMF*) is used. Experiments in Tab.Ⅶ and Tab.Ⅻ demonstrate that such faults are difficult to detect by random testing. A test case is *X* can be used to detect this fault if $\exists i, j$, the following inequality holds:

$$(|W_{ij}|.EMF + f) >> qbits \neq (|W_{ij}|.MF + f) >> qbits \tag{5}$$

We observe that if W_{ij} is found to detect the fault with *EMF* = *MF* - *1*, i.e.,

$$(|W_{ij}|.EMF + f) >> qbits < (|W_{ij}|.MF + f) >> qbits \tag{6}$$

then all faults for *EMF* < *MF* can be detected by this test case. Similarly, all faults for *EMF* > *MF* can be detected by the test case for *EMF* = *MF* + 1. Therefore, we only need to design test cases for *EMF* = *MF* - 1 and *EMF* = *MF* + 1.

The first step is to find two values for each W_{ij} so that the inequality (5) and (6) hold respectively. Then the input matrix *X* can be estimated through $W = C_f X C_f^T$. For each *i,j*, we use a brute-force search with W_{ij} varying from zero to the upper bound and QP varying from 0 to 51. Proper values W_{ij} and QP would thus be found so (5) and (6) are satisfied. Although the upper bound for each W_{ij} is not explicitly given in [21] (only the upper bound 9180 for *all* elements in *W* is given), we could easily derive them as below:

$$\begin{pmatrix} 4080 & 6120 & 4080 & 6120 \\ 6120 & 9180 & 6120 & 9180 \\ 4080 & 6120 & 4080 & 6120 \\ 6120 & 9180 & 6120 & 9180 \end{pmatrix} \quad (7)$$

Since *MF* is decided by QP as shown in Tab.Ⅳ, W_{ij} should be searched for each QP. For simplicity, here we only consider the case when $QP\%6 = 0$. By searching through the values from 0 to the upper bounds defined in (7), *W* could be obtained for *EMF* = *MF* - 1 as below:

$$\begin{pmatrix} 3657 & 495 & 3657 & 495 \\ 495 & 249 & 495 & 249 \\ 3657 & 495 & 3657 & 495 \\ 495 & 249 & 495 & 249 \end{pmatrix} \quad (8)$$

The QP matrix for the above *W* is given by:

$$\begin{pmatrix} 18 & 0 & 18 & 0 \\ 0 & 0 & 0 & 0 \\ 18 & 0 & 18 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (9)$$

In the same manner, *W* could be obtained for *EMF* = *MF* +1 as below:

$$\begin{pmatrix} 1367 & 109 & 1367 & 109 \\ 109 & 980 & 109 & 980 \\ 1367 & 109 & 1367 & 109 \\ 109 & 980 & 109 & 980 \end{pmatrix} \quad (10)$$

All QPs are zero for this case.

Now that we find W_{ij} for (5) and (6), it is not difficult to find a proper input matrix *X* using $W = C_f X C_f^T$. For example, at position (0, 0), if the correct matrix element 13107 is replaced by 13106, the test case W_{00} =3657 is found for (6). Since $W_{00} = \sum_{i=0,j=0}^{i=4,j=4} X_{ij}$, the following *X* could be used as a test case:

$$\begin{pmatrix} 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 \\ 255 & 255 & 87 & 0 \end{pmatrix} \quad (11)$$

That is to say, using X defined in (11), *QP*=18 (as given in (9) and INTER block mode as input to the optimized function Trans&Quant4x4, we obtain W_{00} = 3657, *qbits* = 15 + *QP*/6 = 18, $f = 2^{qbits} / 6$ =43690 and have the following equalities:

$$3657 \times 13106 + 43690 >> 18 = 182 \quad (12)$$

$$3657 \times 13107 + 43690 >> 18 = 183 \quad (13)$$

The result obtained with the wrong constant 13106 in (12) is not equal to the result obtained with the correct constant in (13), meaning the fault is detected.

B. Incorrect Rounding

As for the incorrect rounding fault, test cases could be designed based on the following theorem.

Theorem 1: Assume the correct code is (x + y + 2n-1) >> n and the faulty code is (x + y + r) >> n, where x, y, r, n are non-negative integers and r≠2n-1, n > 0. The fault could be detected by the test cases x = 2n-1, 2n-1-1, y = 0.

Proof: The original faulty code becomes (x + r) >> n with y = 0. Obviously, we have the following inequalities:

$$(2^{n-1} + r) >> n \neq (2^{n-1} + 2^{n-1}) >> n \ (r < 2^{n-1}) \quad (14)$$

$$(2^{n-1} - 1 + r) >> n \neq (2^{n-1} - 1 + 2^{n-1}) >> n \ \ (r > 2^{n-1}) \quad (15)$$

This shows that the test cases $x = 2^{n-1}$, 2^{n-1}-1, $y = 0$ could be used to detect all the faults.

Theorem.1 provides an efficient way of designing test cases for incorrect rounding faults. For instance, suppose the correct equation is $(x+y+2)>>2$, test cases $x=2,y=0$ and $x=1,y=0$ are able to detect faulty equations with incorrect rounding, e.g., $(x+y+1)>>2$, $(x+y)>>2$, $(x+y+4)>>2$, etc.

VII EXPERIMENTS

Experimental results are outlined in this section. Three test techniques: conformance testing, random testing and manual testing are compared. Simulations are conducted in a Lenovo notebook with 2.53GHz Intel Core 2 Dual, 2.99GB memory and Windows XP Home Edition 2002 (Service Pack 3). In conformance testing, there are totally 135 bitstreams for baseline profile. We test those bitstreams in the order listed in the specification [7]. Both regular and SIMD faults are used. For conformance testing, a fault is considered not detectable if the faulty DUT does not report nonconformity for any of the 135 bitstreams. For random testing, a fault is considered not detectable if the faulty function does not report error after a maximum number of test runs, which is set to 10^5 in the experiment. The seed value for the pseudo-random number generator is set to 1 to make the experimental results replicable. As for the manual testing technique, two fault types are analyzed in detail and the design procedure is demonstrated. Functions under test include: forward transform and quantization, inverse transform, deblock filter across horizontal boundaries and luma interpolation. Forward transform and quantization is an encoder side function, so it is not implemented for conformance testing. For each function, a correct C

version, a correct SIMD version, and several faulty SIMD versions are created based on the fault types listed in Sec.Ⅴ. Without loss of generality, all SIMD instructions are implemented with SSE2 intrinsic for Intel CPUs.

To simulate faults in real applications, a number of faulty versions for the aforementioned four functions are created. For convenience, each version only contains a single fault. For each function, 0~3 faults are manually created for every fault type. As shown in Tab.Ⅶ---Tab.Ⅹ, the first column denotes the fault name, e.g., *Incorrect constant #2* denotes the second *Incorrect constant* fault. The second column contains experimental results for random testing with the format of *t/n*, where *t* and *n* denote the required fault detection time in million seconds and the number of required test runs, respectively. For example, 12.17/1100 means it takes 1100 runs and 12.17ms to detect the fault. The third column contains experimental results for conformance testing with the format of *m/n/t*, where *m* is the total number of bitstreams (out of total 135) that reveal the fault, *n* represents the bitstream firstly reveals the fault, *t* stands for the fault detection time in seconds. For instance, 98/5/5.15 means that, the DUT is reported non-conformant by 98 bitstreams (out of 135). The first bitstream that reveals the fault is the 5th bitstream, and it takes 5.15 seconds to finish decoding the first 5 bitstreams.

These experiments show that all the faults are detectable via random testing except for the *Incorrect constant #1* in the Trans&Quant4x4 function. Conformance testing is capable of detecting all the faults in the decoder (faults in the encoder functions are marked as 'N/A'). Although most faults are detectable, the difficulty levels are not identical. Faults planted in Trans&Quant4x4 and IDCT4x4 are easier to detect in most cases, either with random testing or conformance testing. For random testing, it only takes one run to reveal faults in most cases, and all the 135 bitstreams report nonconformity (Tab.Ⅶ,Tab.Ⅷ). For Luma Interpolation and Luma Deblock, it takes much longer to reveal faults. For example, it takes 50.53s to detect *Incorrect sign/unsigned right shift fault #1* for Luma Interpolation using conformance testing (Tab.Ⅸ), and it takes random testing 51033 runs to detect *Incorrect constant #2* for Luma Deblock (Tab.Ⅹ).

The metrics are averaged in Tab.Ⅺ, where *Incorrect constant* faults are not counted for Trans&Quant4x4 function because *Incorrect constant* #1 is not detectable. It shows that Luma Deblock requires the largest number of runs in average for random testing. As for conformance testing, it takes 4.90 and 7.83 seconds for fault detection in Luma Deblock and Luma Interpolation, respectively. Overall, in most cases, conformance testing takes much longer (in the order of seconds instead of million seconds) to detect a fault compared with random testing.

Code coverage analysis could be used to analyze the sufficiency of the generated test cases. Here we use a tool 'gcov' to calculate two metrics: statement coverage and branch coverage to measure the fraction of statements and branches in the function that are covered by executed test cases. Achieving high coverage can be seen as a requirement for test case generation. Fig.3 and Fig.4 report that, the studied four functions increase their coverage by executing more test cases. Trans&Quant 4x4 and Inverse Transform 4x4 achieve full coverage with only a few test runs, Luma Interpolation and Luma Deblock require more test runs. It is consistent with the experimental results.

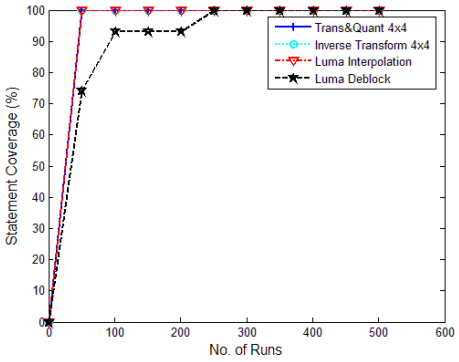

Figure 3. Statement Coverage with Random Testing.

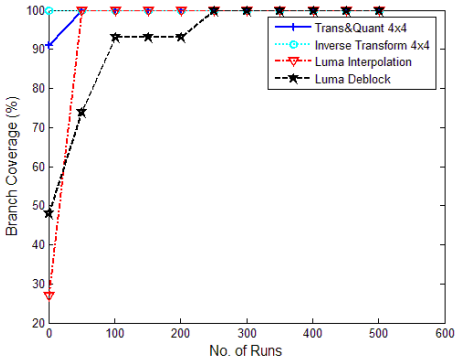

Figure 4. Branch Coverage with Random Testing

Manual test case design could be used to attack difficult faults listed in Sec.Ⅵ, i.e., Incorrect constant in the Trans&Quant4x4 function and Incorrect rounding in Luma Deblock. To verify the effectiveness of the designed test cases, eight faults for each type are planted as shown in Tab.Ⅻ. Moreover, manual testing is compared with random testing in terms of fault detection effectiveness. With random testing, Incorrect Constant #1 and #3 are not detectable within 105 test runs, and it takes random testing more than 104 runs to reveal Incorrect Constant #5, #7. For Incorrect rounding faults, most faults require more than 1000 random test runs. For manual testing, all faults are detected within the maximum number of designed test cases (e.g., 2 runs for Incorrect rounding faults). The results demonstrate that manual test case design is more efficient in terms of fault detection rate and speed for these difficult faults.

VIII Conclusion and Future Work

TABLE VII.
FAULT DIAGNOSIS FOR TRANS&QUANT4X4

Fault	Random Testing	Conformance Testing
Relational operator replacement #1	0.28/1	N/A
Logical operator replacement #1	0.19/1	N/A
Logical operator replacement #2	0.18/1	N/A
Incorrect initialization #1	0.17/1	N/A
Incorrect constant #1	$>10^5$	N/A
Incorrect constant #2	7.29/9118	N/A
Incorrect array element reference #1	0.17/1	N/A
Incorrect array element reference #2	0.17/1	N/A
Incorrect array element reference #3	0.18/1	N/A
Incorrect pointer operation #1	0.18/1	N/A
Same type variable replacement #1	0.18/1	N/A
Same type variable replacement #2	0.18/1	N/A
Arithmetic operator replacement #1	0.18/1	N/A
Arithmetic operator replacement #2	0.18/1	N/A
Delete a complete statement #1	0.17/1	N/A
Delete a complete statement #2	0.17/1	N/A
Delete a part of a statement #1	0.18/1	N/A
Delete a part of a statement #2	0.18/1	N/A
Incorrect number of loop iterations #1	0.18/1	N/A
Incorrect rounding #1	0.18/1	N/A

TABLE VIII.
FAULT DIAGNOSIS FOR IDC4X4

Fault	Random Testing	Conformance Testing
Incorrect array element reference #1	0.20/1	135/1/0.26
Incorrect array element reference #2	0.18/1	135/1/2.07
Same type variable replacement #1	0.21/1	135/1/0.80
Arithmetic operator replacement #1	0.18/1	135/1/0.59
Arithmetic operator replacement #2	0.18/1	135/1/0.71
Delete a complete statement #1	0.19/1	135/1/0.69
Delete a complete statement #2	0.16/1	135/1/0.65
Delete a part of a statement #1	0.18/1	135/1/0.66
Delete a part of a statement #2	0.18/1	135/1/0.85
Incorrect number of loop iterations #1	0.16/1	135/1/0.99
Incorrect signed/unsigned right shift #1	0.18/1	135/1/0.66
Incorrect signed/unsigned right shift #2	0.17/1	135/1/0.68

The issue of testing optimized H.264 functions is presented and studied in this paper. Two software complexity and code coverage metrics are used to measure the function complexity and help decide testing priorities. By analyzing those functions and their SIMD implementations, we firstly present a new category of SIMD faults and use three techniques to attack those faults. The results show that the conformance bitstreams can be used to efficiently test the optimized functions in decoders. Random testing is capable of testing easy faults at both encoders and decoders. As for difficult faults, manual testing is proposed and proved very effective.

TABLE IX.
DIAGNOSIS FOR LUMA INTERPOLATION

Fault	Random Testing	Conformance Testing
Missing path #1	0.27/7	114/3/2.04
Missing path #2	0.29/11	113/3/1.86
Relational operator replacement #1	0.28/11	113/3/1.82
Logical operator replacement #1	0.27/6	113/3/1.71
Logical operator replacement #2	0.17/1	113/3/1.86
Incorrect initialization #1	0.23/5	96/3/1.73
Incorrect array element reference #1	0.27/9	95/3/1.73
Incorrect array element reference #2	0.20/2	113/3/1.67
Incorrect pointer operation #1	0.52/14	110/3/1.87
Same type variable replacement #1	0.42/14	110/3/1.94
Same type variable replacement #2	0.31/14	110/3/1.81
Arithmetic operator replacement #1	0.25/7	96/3/1.79
Arithmetic operator replacement #2	0.25/7	96/3/1.77
Delete a complete statement #1	0.25/7	96/3/1.71
Delete a complete statement #2	0.26/7	96/3/1.79
Delete a part of a statement #1	0.25/7	96/3/1.82
Delete a part of a statement #2	0.25/7	96/3/1.67
Incorrect signed/unsigned right shift #1	0.26/7	39/25/50.53
Incorrect signed/unsigned right shift #2	0.29/7	46/25/49.70
Incorrect sign/unsigned extension #1	0.21/2	54/25/49.72
Incorrect rounding #1	0.33/15	96/3/1.75
Incorrect rounding #2	0.20/2	113/3/1.76
Incorrect saturation #1	0.23/5	96/3/1.96
Incorrect saturation #2	0.27/7	96/3/1.84

TABLE X.
FAULT DIAGNOSIS FOR LUMA HORIZONTAL DEBLOCK

Fault	Random Testing	Conformance Testing
Relational operator replacement #1	12.17/1100	98/5/5.15
Relational operator replacement #2	1.17/94	85/5/4.89
Relational operator replacement #3	0.22/3	98/5/4.77
Logical operator replacement #1	0.94/72	85/5/4.81
Logical operator replacement #2	0.25/5	85/5/4.79
Logical operator replacement #3	0.20/3	98/5/4.79
Incorrect constant #1	38.38/3415	85/5/4.81
Incorrect constant #2	554.23/51033	85/5/5.14
Incorrect array element reference #1	0.20/3	98/5/4.68
Same type variable replacement #1	0.23/5	85/5/5.13
Same type variable replacement #2	0.22/5	85/5/4.93
Arithmetic operator replacement #1	0.21/4	98/5/4.90
Arithmetic operator replacement #2	15.10/1425	85/5/4.89
Delete a complete statement #1	15.74/1425	85/5/4.78
Delete a complete statement #2	0.70/50	98/5/4.80
Delete a part of a statement #1	16.32/1476	98/5/4.99
Delete a part of a statement #2	0.69/50	98/5/4.85
Missing clause in predicates #1	0.20/3	98/5/4.87
Missing clause in predicates #2	0.20/3	98/5/4.70
Incorrect signed/unsigned right shift #1	0.75/54	98/5/4.88
Incorrect signed/unsigned right shift #2	2.38/208	98/5/5.21
Incorrect rounding #1	105.31/9325	85/5/5.05
Incorrect rounding #2	74.42/5778	85/5/4.99

TABLE XI.
INTERPOLATION AVERAGED TEST RESULTS

MODULE NAMES	Random Testing	Conformance Testing
Forward Tansform4x4	0.18/1	N/A
Inverse Transform4x4	0.18/1	135/1/0.80
Luma Interpolation	0.27/7.54	96.5/5.75/7.83
Luma Deblock Filter	36.53/3284.3	91.78/5/4.90

TABLE XII.
COMPARISON BETWEEN RANDOM TESTING AND MANUAL TESTIN

Fault	Random Testing	Manual Testing
Incorrect constant #1	$>10^5$	0.34/1
Incorrect constant #2	7.29/9118	0.34/2
Incorrect constant #3	$>10^5$	0.36/3
Incorrect constant #4	1.61/1863	0.36/4
Incorrect constant #5	26.87/37043	0.38/5
Incorrect constant #6	1.68/1993	0.38/6
Incorrect constant #7	19.75/24220	0.39/7
Incorrect constant #8	0.62/562	0.42/8
Incorrect rounding #1	1.70/141	0.41/1
Incorrect rounding #2	74.42/5778	0.42/1
Incorrect rounding #3	23.32/2086	0.42/1
Incorrect rounding #4	15.58/1425	0.54/2
Incorrect rounding #5	15.86/1425	0.52/2
Incorrect rounding #6	15.72/1425	0.55/2
Incorrect rounding #7	15.57/1425	0.44/1
Incorrect rounding #8	23.16/2086	0.42/1

Based on these results, we recommend managers/engineers to choose one or all of these three methods based on their unique needs (e.g., engineer resource limitation, time limitation, client requirements, etc.). It is also worth mentioning that, although only four optimized functions in H.264 are thoroughly analyzed and tested, we see the proposed test techniques valuable and potentially extendable to other video codec modules and standards.

Test case design for SIMD faults are sometimes platform specific. For example, the signed/unsigned extension issue introduced in Sec. Ⅴ-B3 could be avoided in Intel/Amd CPUs with SSE4.1 support. SSE4.1 contains signed extension instructions such as PMOVSXWD, PMOVSXBW, etc. Algorithm implementations also matter. The 16-bit Luma Interpolation described in [20] requires no signed extension. Furthermore, other SIMD instruction sets such as Altivec may generate different faults. In our future work, we would like to explore these topics.

Our future work would also include video coding algorithm comparison and selection based on complexity measures, where the efficiency of various video coding algorithms (e.g., fast mode decision algorithms, rate control algorithms, etc.) are not only judged by the improved PSNR and coding speed, but also functional complexity. Furthermore, test data adequacy criteria could be used to help improve the efficiency of test case generation [10], [22]. Lastly, we made an assumption in this paper that C versions are error free. How to effectively verify their correctness based on the specification and reference software is not discussed in this paper. We are currently investigating this important issue and will report our results in the near future.

REFERENCES

[1] G. Sullivan, P.Topiwala, and A.Luthra, “The h.264/avc advanced video coding standard: overview and introduction to the fidelity range extensions”, in Conference on Applications of Digital Image Processing. SPIE, 2004, vol. XXVII, pp. 1–22.

[2] J.Ostermann, J.Bormans, P.List, D.Marpe, M.Narroschke, F.Pereira, T. Stockhammer, and T.Wedi, “Video coding with h.264/AVC: Tools, performance and complexity”, IEEE Circuits and Systems Magazine, vol. 4, pp. 7–28, First Quarter 2004.

[3] Gary J. Sullivan and Jens-Rainer Ohm, "Recent developments in standardization of high efficiency video coding (HEVC)", Proc. SPIE 7798, 2010.

[4] M. Horowitz, A.Joch, F.Kossentini, and A. Hallapuro, “H.264/avc baseline profile decoder complexity analysis”, IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, pp. 704–716, July 2003.

[5] Y.L. Lai, Y.Y.Tseng, C.W. Lin, Z. Zhou, and M.T.Sun, “H.264 encoder speed-up via joint algorithm/code-level optimization”, in Visual Communications and Image Processing. SPIE, 2005, vol.5960, pp. 1089–1100.

[6] J.Lee, S. Moon, and W. Sung, “H.264 decoder optimization exploiting simd instructions”, in Asia-Pacific Conference on Circuits and Systems. IEEE, 2004, vol. 2, pp. 1149–1152.

[7] Joint Video Team, “Conformance specification for itu-t h.264 advanced video coding”, ITU-T Rec. H.264.1, April 2010.

[8] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, “An experimental evaluation of data flow and mutation testing”, Software - Practice and Experience, 26(2):165–176, February 1996.

[9] P. Thévenod-Fosse, H. Waeselynck and Y. Crouzet, “An experimental study on software structural testing: deterministic versus random input generation”, Proc. 21st International Symposium on Fault-Tolerant Computing, Montreal, Canada, pp. 410-417, June, 1991.

[10] P.G. Frankl, S.N. Weiss, and C. Hu, “All-uses versus mutation testing: An experimental comparison of effectiveness”, Journal of Systems and Software, vol. 38, pp. 235–253, September 1997.

[11] P. Meehan, N. Hurst, M. Isnardi, and P. Shah, “Mpeg compliance bitstream design”, in International Conference on Consumer Electronics, June 1995, pp. 174–175.

[12] C.M. Kim, B.U. Lee, and R.H. Park, “Design of mpeg-2 video test bitstreams”, IEEE Transactions on Consumer Electronics, vol. 45, pp. 1213–1220, 1999.

[13] J. Cho, S.Choi, and S.I. Chae, “Constrained-Random Bitstream Generation for H.264/AVC Decoder Conformance Test”, IEEE Transactions on Consumer Electronics, vol. 56, pp. 848–855, May 2010.

[14] Intel Architecture Software Developer’s Manual, Volume 2: Instruction Set Reference, 1999.

[15] S.Yu and S. Zhou, “A survey on metric of software complexity”, in Int. Conf. on Information Management and Engineering, April 2010, pp. 352–356.

[16] T.J.Mccabe, “A complexity measure”, IEEE Transactions on Software Engineering, vol. SE-2, pp. 308–320, December 1976.

[17] W.E. Wong, J.R. Horgan, S. London,and A.P. Mathur, “Effect of test set minimization on fault detection

effectiveness", in Proceedings of the 17th international conference on Software engineering, April 1995, pp. 41–50.
[18] I.E.G. Richardson, H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia, Wiley, 2003.
[19] Joint Video Team, "Advanced video coding for generic audiovisual services", ITU-T Rec. H.264 & ISO/IEC 14496-10 AVC, March 2005.
[20] F. Bossen, "Full 16-bit implementation of 1/4 pel motion compensation", in Doc. JVT-C37. Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, May 2002.
[21] L. Kerofsky, "Notes on jvt idct," in Doc. JVT-C24. Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, May 2002.
[22] H. Zhu, P.A.V. Hall, and J.H.R. May, "Software unit test coverage and adequacy", ACM Computing Surveys, Vol.29, pp. 366–427, December 1997.

Hao Zhang received the Ph.D. degree in electrical and computer engineering form Polytechnic University (now Polytechnic Institute of New York University), New York, USA, in 2006. He is currently an associate professor in the School of Information Science and Technology at Central South University, ChangSha, China.

His current research interests include software testing , evaluation system analysis and video coding.