Self Repairing Processors Through Software and Hardware.

York Prado, *Ungraduate Student, CNU,* Giang Hoang, *Graduate Student, NU,*
and Dr. Russ Joseph, *Professor/Advisor, NU*

***Abstract*—As hardware gets smaller and much more sophisticated, the likelihood of faulty hardware being produced increases dramatically. Due to this, current research is being done in trying to reduce the severity of these errors. As hardware faults cannot be fixed on the fly, methods have to be designed to minimize the error in the output and get a result that is good enough to use in software. Since this method does not produce a completely error-free output and is not 100 percent accurate, this is not ideal for anything that needs exact values. This includes programs that deal with statistical or data analysis. Instead, efforts will be focused on multimedia, where a slight alteration of a pixel or a drop in frames will be barely noticeable to the human observer. This leads to a higher fault tolerance in multimedia applications, which is a good starting point in testing new methods of self repair. The main focus now will be to concentrate on the MMX and SSE 1/2 instructions used in x86 CPUs when encoding and decoding different multimedia formats using the open source program FFMPEG.**

The goal of the paper will be to analyze and profile FFMPEG using the profiling tool OProfile to see what exact lines of code and assembly instructions are being used when encoding and decoding pictures and videos into different formats. Profiling using Oprofile works by taking constant samples of everything being run on the system. This program then lists the procedures and calls that were being run while the profiler daemon was running and lists them according to the length of time each procedure took. The procedures that will be concentrated on are those that will be filled with SIMD instructions. Once given, several software and hardware techniques will be simulated in order correct faults in hardware.

***Index Terms*—DCT, IDCT, SIMD, SSE, MMX, x86, OProfile.**

I. Introduction

As hardware gets smaller and much more sophisticated, the likelihood of faulty hardware being produced increases dramatically. Because the VLSI processes are reaching physical limits in nanoscale production, manufacturers are having a harder time to meet production yields. These sources of hardware faults are often attributed to soft errors, wearout, and process variation.[1] Today, a vast amount of research is being done in order to increase production yields. In [4], the concept of error tolerance in hardware that deals with multimedia applications is introduced. A circuit is defined as error-tolerant in its application if it contains defects in its circuitry that can cause errors and if the circuit maintains an acceptable result in the end. Multimedia is especially suited for error tolerance in hardware because of the standard encoding and decoding of multimedia to lossy formats, and the lack of perception to the human observer.[3] Humans tend to be insensitive to slight variations in sound and colors. Also, converting to a lossy multimedia format may increase fault tolerance of an acceptable level because of the loss of data.[5]

In [3], it is shown that conventional ATPG proves to give unacceptable yield improvement because it detects circuits with acceptable faults along with those with unacceptable faults. A new test methodology is provided which introduces a generalized test pattern and then performs a masking technique on boards that are deemed acceptable. Results show that circuitry with acceptable faults are accepted more often, thereby increasing yields.

[2] shows that for many applications, tolerance can be varied for many components inside that application. This is due to the fact that some components are less vital in producing an acceptable result than others. This is especially seen in color interpolation filtering which reproduces missing color samples at every pixel.

The paper is organized as follows. In section 2, a brief overview of the fault simulation and methods used to correct this will be given. Then, a brief overview of how multimedia is encoded and decoded will be be described in section 3. Section 4 will then be used to describe the profiling of FFMPEG and its results. Finally, section 5 will be used to draw conclusions.

II. Single Stuck-at-fault model

One of the biggest increasing problems in hardware today are hard faults. The single stuck-at-fault model is one such example problem and most test patterns are generated with this fault model in mind.[7] Below is a Kogge-Stone adder used to simulate a single stuck-at-fault model.

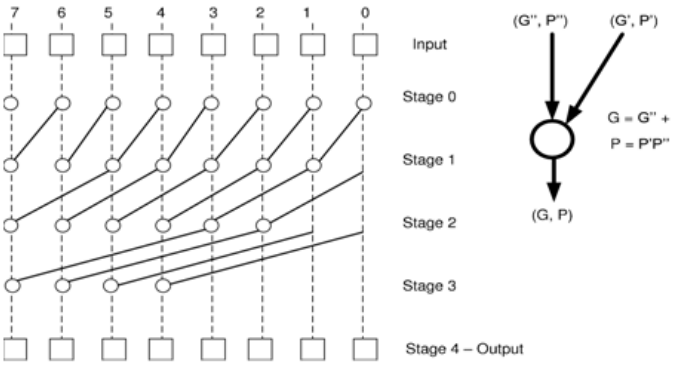

Fig. 1. Kogge-Stone adder used to implement SIMD addition instruction

The fault model can be described as having a gate stuck at a signal value that is independent of other signals and values. For example, any one of these adders in the diagram can be stuck at a 1 or stuck at a 0 value in which it cannot be changed.

This problem can give a computationational error depending on which bit the stuck at fault adder is located. If the stuck-at-fault adder is located on the LSB, there will be very little error, but if it were stuck at the MSB, it has the potential to give a tremendous amount of error.

A. Solutions

As it stands now, there are a variety of ways to correct a single stuck-at-fault error. Manufacturers often disable parts of logic in circuitry that is faulty. Even though this increases yield, overall yield is becoming smaller due to the complexity and small size of new circuitry. This causes chips to increase in price, slow down the move to new technology, and reduces incentive to add more logic.[4] Redundancy in a circuit can also be added in order to maintain yields but this proves costly and effectively doubles the circuitry of a design.

The new idea presented is to have different levels of self-repair in order to severely lower the effect of this fault on the output. Since this will be mainly focused on multimedia, a low amount of error will seem perceivably gone. below shows two different pictures of Lena where a fault was injected when converting this image from a PPM format to JPEG. As one can see, the final quality depends on where the fault is located. Errors injected into the higher bit values obviously gave a clearer distortion in the picture while errors injected into lower bit values seemed negligable.

Fig. 2. Image of Lena with faults injected at different places.

Looking at these figures, it seems obvious there should be different paths to repairing the hardware errors. When errors are small enough to have the encoded image be indescernible from the orginal image, nothing should be done. For larger errors, the use of hardware shifters could be used internally to shift the additions so that the fault can only effect the LSB and nothing larger. Lastly, if the error proves too large, loop unrolling could be done in software to move the computations from SSE units to general purpose units. This, of course, has the trade-off between speed and error correction.

III. Multimedia Encoding

The underlying lossy methods in multimedia encoding along with a human's insensitivity to small variation in color and sound make these type of applications very resilient to errors. These methods include chroma subsampling, DCT, and quantization for video and image encoding, and framing and motion vectors in video encoding.

The first step in JPEG and MPEG encoding is to change the color space from RGB to YUV.[8] This first step is neccesary to lower the amount of data held in the image or video frames while keeping quality relatively high. While RGB splits each pixel into seperate values of red, green, and blue, YUV splits pixels in terms of luma(Y) and chrominance(UV). Luma describes the light differences within a picture while chromninance describes the color within the picture. Since the human eye perceives brightness at a higher level than color changes, chrominance can be downsampled.[10] The most popular YUV format is 4:2:0, which keeps luma values for every pixel but reduces chrominance values to just one sample every 4 pixels. Below is a sample of a 4:2:0 YUV photograph split into Y, U, and V values respectively.

Fig. 3. Y samples are kept at higher sampling rates than U and V with little visual loss.

Most mainstream multimedia encoders also use the discrete consine transform (DCT) to futher reduce the size of the multimedia file. The DCT converts data from the time space domain to the time frequency domain. JPEG and many video encoders like H.264, MPEG-1, MPEG-2, and WMV2 use the DCT before quantization. In addition, DCTs are relatively effective at pushing the vital signal information to small areas in the macroblock. The image is usually split in 8x8 pixel blocks to which the DCT is applied. Pushing all the information to the smaller frequencies helps when applying a quantization. While JPEG applies this for the whole image, video formats only need to apply this to any residual frames that are left after getting rid of spacial and temporal redundancy.

After the DCT is applied, the data can then be quantized inside a macroblock. Quantization maps the range of possible values to a smaller set of values. Because the DCT shifts most of the data to the lower frequencies, the higher frequencies often have values small enough to map to 0. [10] This is helpful to reduce to amount of data afterwards when applying entropy encoding because it reduces huge pools of zero values to just a pair showing how many 0's are inbetween two non-zero numbers.

Before the DCT is applied to video frames, temporal and spatial redundancy is removed. The temporal model exploits redundancy because there is a high correlation between frames in a video that are close to each other. This is especially true in video with high framerates since, usually, little changes occur within 1/25th of a second. Below, the third frame shows the differences between two sequential frames in a video. In addition, the spatial model exploits redundancy because there is a high correlation between pixels close to each other within a frame.

Fig. 4. The differences between two sequential frames in a video is shown in the third frame.

In video, frames are first split into different types of frames. These frames are I-frames, P-frames, and B-frames. [10] I-frames are very much like normal JPEG images as the encoder stores all the data of the frame. On the other hand, P-frames only store data that is changed from the I-frame preceding it. Also, B-frames only contain data different from frames preceding and following it. B-frames are allowed to take data from I-frames and P-frames. The changed data in I and B frames are stored in the form of a residual image and motion vectors. Since there is movement within sequential frames, images go through motion estimation. Motion estimation takes macroblocks within a seperate frame and tries to match this to a macroblock within the current frame. This method tries to match a macroblock as concisely as possible and, once found, only stores a motion vector from the previous macroblock to the current one.

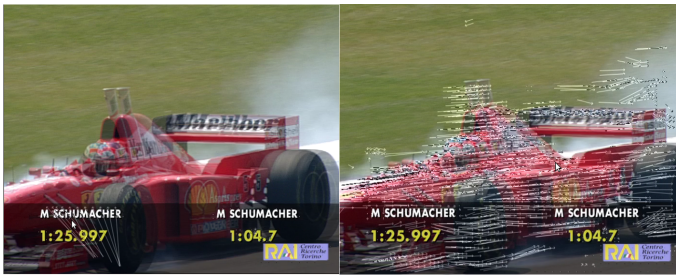

Fig. 5. The first frame is an I-frame and the second is the P-frame following it. The arrows shown are motion vectors stored inside the P-frame.

These motion vectors only contain data of the direction and distance of one macroblock to another. Above, two sequential images are shown, one of an I-frame, and one of a P-frame with motion vectors shown. Of course, not all parts of an image can be compensated this way because 3D objects can move out of the frame or turn. Because of this, the differences that cannot be compensated for are stored within a residual image.

IV. Results

A bash script for Ubuntu was created to profile the encoding and decoding of different multimedia formats to another. The open source program used to decode and encode videos and images is FFMPEG. The profiling tool used is OProfile. This tool is used to find SIMD heavy procedures within the program. The different conversions tested are shown in Table 1.

The bash script was set up to go through a loop of a specific encoding or decoding procedure 1000 times in order to capture a concise picture of the routines used within the conversions. The results were then redirected into a text file. Since the profiler took in all instances of all the applications being run at the current time, the profiler was set up to only record samples taken from FFMPEG. In order to profile FFMPEG's procedures, Debug symbols had to be enabled. A debug symbol contains information of where the give code within the program was executed and ties it to the source files. This is helpful when trying to debug a program and finding faults within it. Also, in a seperate directory, all the assembly code that is tied with the debug symbols are redirected into seperate files.

TABLE I
Conversions Profiled for finding SIMD heavy Procedures

Conversion Type	Converting From	Converting To
Encoding	JPEG	PPM
Decoding	PPM	JPEG
Encoding	YUV	DRC
Encoding	YUV	MPEG1
Encoding	YUV	MPEG2
Encoding	YUV	WMV 8
Encoding	YUV	H.264
Encoding	YUV	MPEG4 Part 2

A python script is then run to compare each debug symbol caught by the compiler to a list of debug symbols that are known to carry SIMD assembly instructions. Since these conversions were done on an x86 machine, MMX and SSE instructions were the SIMD instructions looked for. The python script then redirects this content into a seperate file. The procedures that were tagged with the debug symbols were then listed out and a brief description was given to what they did in the encoding/decoding steps.

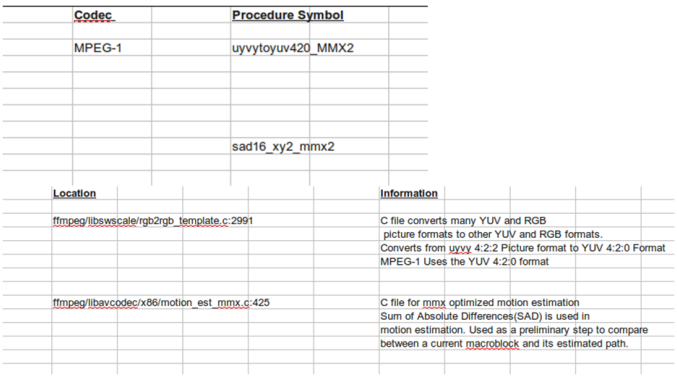

Fig. 6. An example of the final output given through the scripts. The codec used is first listed, then the debug symbol, then the location and any information pertaining to the procedure.

V. Conclusion

The hope for these results is to use it to further guide the development of an auto-correcting algorithm to use within software to correct any hardware faults that may occur. This will help greatly in the hardware industry because hardware that may have been considered faulty can be still used with

slight modification inside the code or adding extra non-trivial hardware. Using simulations and a better understanding of multimedia encoding, the hope is to first extend these improvements to multimedia applications. The reason for this is that multimedia has a good fault tolerance due to a human's visual perception. The ultimate goal is to further extend this into other areas that have a lower fault tolerance once these processes have matured.

REFERENCES

[1] Li, Xuanhua and Donald Yeung. Application-Level Correctness and its Implact on Fault Tolerance. *IEEE DSD '05 Conference*, 2005.
[2] Roy, Kaushik and Georgios Karakonstantis. Design Methodology to trade off Power, Output Quality and Error Resiliency:Application to Color Interpolation Filtering. *IEEE*, 2007.
[3] Lee, Kuen-Jong and Tong-Yu Hsieh. Reduction of Detected Acceptable Faults for Yield Improvement via Error-Tolerance. *EDAA*, 2007.
[4] Breuer, Melvin. Multi-media Applications and Imprecise Computation. *High-Performance Computer Architecture*, Feb. 2007.
[5] Lu, Chia-Lin and Melvin A. Breuer. A systematic methodology to employ error-tolerance for yield improvement. *IEEE*, 2008.
[6] Feig, Ephraim and Shmuel Winograd. Fast Algorithms for the Discrete Cosine Transform. *IEEE Transactions on Signal Processing*, 1992.
[7] Mccluskey, Edward and Chao-Wen Tseng. Stuck-Fault Tests vs. Actual Defects. *IEEE International Test Conference*, 2000.
[8] Kerr, Douglas. Chrominance Subsampling in Digital Images. *Pumpkin Issue 1*, November 2, 2005.
[9] cohen, William. Tuning Programs with Oprofile. *Wide Open Magazine*, 2004.
[10] Richardson, Iain. H.264 and MPEG-4 Video Compression. *Wiley* England, 2003.