ABSTRACT
One of the few challenges facing face recognition systems is real time processing. In order for a face recognition system to be viable for real world implementations, it needs to be fast enough to track and identify facial images and do on-the-fly one-to-many identification from a multitude of database images. This is time consuming since face recognition requires a complex process of pre-processing and post-processing to be done along with the actual processing itself. In order to save precious processing time and increase the processing speed, specific compiler optimizations could be applied to the algorithms to help in reducing system processing time and increase total system performance.

In this paper, we explore different levels of compiler optimization techniques when applied to face recognition systems to help improve the algorithm’s performance and in turn improve the entire system performance when implemented in real time and real life situations. A multitude of different compiler optimization techniques are tested and implemented during the stages of development and the effects of such optimizations are logged and compared.

Keywords
Compiler optimization, Linux, face recognition

1. INTRODUCTION
Research on compiler optimization and its’ effects to system performance have been going on since the early stages of development of high level language programming. With the introduction of high level programming languages such as B, C and C++, developers are freed from the daunting task of development in complex machine language and assembly. This greatly reduces application and system development time whilst reducing development complexity.

While the introduction of high level programming languages speeds up development, it also brought particular discrepancies whereby the codes written by the developer are not exactly the codes executed by the machine. Since the codes written using high level programming languages goes through a compilation process before being assembled into a low level machine language, it generates discrepancies because of the multi-processes the codes have to go through before being generated into machine language.

Compiler optimizations have been introduced to reduce these discrepancies and help improve application performance. Various studies have shown that compiler optimizations have a large effect in reducing code overhead and thus improve total system performance. Furthermore, studies have also shown that compiler optimizations also have various effects in reducing power consumption [1, 2, 3], improve memory and cache subsystem performance [4, 5], and also have an indirect effect to I/O performance [6].

In this paper, we test different compiler optimizations and its effects on system performance, particularly to face recognition systems and its’ algorithms.

2. GCC COMPILER OPTIMIZATIONS
Compilers usually have some form of optimization routines implemented into the code generation process in order to help reduce code discrepancies when converted from high level programming languages into machine codes. Implementing a large set of different optimization options, GCC offers 54 different optimization routines which yields more than 2^54 different optimization settings [7].

These large 54 optimization options are grouped into 5 general optimization flags pre-selected to provide ease of choice for developers. The general optimization flags implemented in GCC are as following:

- O0 (Level 0)
  No optimization.
- O1 (Level 1)
  The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
- O2 (Level 2)
  The 2nd level optimization enables all optimization invoked by the 1st level optimization flag and a few other optimization sets tailored to increase the performance of the code. In this optimization level,
GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff.

- Os (Level 2.5) Optimizes for code size over performance. This special optimization level (-Os or size) enables all -O2 optimizations that do not increase code size.
- O3 (Level 3) The 3rd and highest level optimization level enables even more optimizations by putting emphasis on speed over size. This includes optimizations enabled at -O2 and rename-register.

Although optimization level 3 (-O3) can produce faster code, the increase in the size of the binary image can have adverse effects on its speed [7]. For example, if the size of the binary image exceeds the size of the available instruction cache, severe performance penalties can be observed. Therefore, it may be better simply to compile at -O2 to increase the chances that the binary image fits in the instruction cache. [7]

3. EVALUATION METHOD

In order to provide a comparable testing environment for performance evaluation, a single workstation is used throughout the entire evaluation process. Four test algorithms are selected and tested under different optimization levels. The execution time for each algorithm utilizing different optimizations are then logged and compared.

The algorithms and optimizations are tested on a same workstation which is a Dell Optiplex 740 workstation, consisting of an AMD Athlon X2 3800+ CPU, 1GB of system memory, 80GB of system storage and an integrated nVidia 640 display adapter running on OpenSUSE 10.2. A newer linux kernel version 2.6.16.27 is chosen to be used for the system with supporting system modules and utility programs for benchmarking purposes installed into the system storage utilizing an ext3 file system.

4. TEST ALGORITHMS

4 different test algorithms which are used in our face recognition system are tested under different optimization levels to seek out the impact that compiler optimizations have on them. The algorithms selected are widely used in image processing fields and are particularly used in our face recognition studies as pre-processing methods before the actual recognition process could be done. The algorithms, including its’ particular functions are briefly explained below:

4.1 Gross-Brajovic Light Normalization Algorithm

The Gross-Brajovic light normalization algorithm was developed by Ralph Gross and Vladimir Brajovic of the Robotics Institute, Carnegie Mellon University to compensate for illumination variation in images [8, 9]. This particular light normalization algorithm is used to help minimize lighting condition variation to the input images by normalizing the lighting condition brought by direct and indirect light sources. This reduces lighting affect to the recognition process by allowing the input images to have a similar lighting condition even when taken under different situations.

4.2 Histogram Equalization

Histogram equalization is a common algorithm used in image processing applications to improve contrast distribution. Histogram equalization works by normalizing light intensities across a grayscale image according to the pixel values from the entire image. The algorithm is used to reduce light variations and allows the recognition phases to work with less variation.

4.3 Neural Network Based Face Detection

A neural network based face detection provides a fast filtering mechanism to help reduce redundant data feeded to the recognition algorithm. By applying the input images through a validation process to make sure the images actually has a facial image in them, redundant data could be removed from being processed in later stages.

2 sets of neural network architectures are tested, using a single a multilayer perceptron (MLP) neural network, and using a cascaded MLP neural network.

4.4 Haar-classifier Based Face Detection

Providing the same functions as the neural network based face detection, a haar-classifier is used as a quick method of face detection in input images to validate the presence of faces within the input images. This is also used to remove redundant data from being processes which could bring to false detection errors.

5. RESULTS AND DISCUSSION

5.1 Gross-Brajovic Light Normalization Algorithm

![Performance Comparison on Gross Brajovic Light Normalization Algorithm](image)

Performance graph obtained from the light normalization shows that an unoptimized binary has the highest processing time compared to other optimized binary. By implementing an architectural specific optimizations, in this case the K8 optimization flag (specific for AMD K8 platform) and an i686 optimization flag (specific for Intel i686 platform), we could test specific architectural optimization effect on execution speed.

Previous studies [10] have shown that architectural specific optimizations does have an impact on the code execution.
performance and such optimizations tailored to a specific architecture could further improve code efficiency and allow for a better execution performance.

A brief study on the average processing time taken by each optimization shows that the unoptimized binary took the most time, followed by the 1st level optimization, 3rd level optimization and the least time are from the 2nd level optimized binaries. It is also confirmed that the architectural specific optimizations does have an effect on the execution speed whereby the average processing time for both the 2nd and 3rd level optimized binary for the AMD K8 platform shows marginally faster processing speed than the binaries optimized for the Intel i686 platform. This is largely because of the testing platform which are using AMD K8 architecture and are in line with the findings found from other studies [10].

Averaged, the 2nd level optimization with specific K8 optimizations have the largest average performance gain compared to any other optimization levels. Results also shows that 3rd level optimization have a minimal improvement compared to a general 2nd level optimization sequence.

### 5.2 Histogram Equalization

Processing time for histogram equalization sequence shows that 3rd level optimization with specific K8 optimizations have the shortest processing time which is marginally faster than 2nd level optimization. Though 3rd level optimization is faster, it shows inconsistent processing speed during initial processing.

The average processing time for histogram equalization shows that 3rd level optimized binary with K8 optimizations have the fastest processing time when compared to 2nd level optimization with a general flag and when compiled specifically for i686.
The average performance gain for 3rd level optimization is nearly 30% higher than 2nd level optimization. A specific i686 flag also brought some marginal improvement compared to general flag. This shows that a specific architectural optimization correctly applied according to the target architecture that the binary is targeted to run have a definite impact on the speed.

5.3 Neural Network Based Face Detection

5.3.1 Single Neural Network Performance

The average processing time for a single neural network (without optimization) requires more than double the processing time required when implemented with some optimization sequences. The performance graph prior shows that the unoptimized binary requires an average of 5 second processing time whereby when optimized; the average processing time is down to below 2 seconds. This shows a huge improvement of around 200% speed gain.

5.3.2 Cascaded Neural Network Performance

Cascaded neural network also shows a large decrease in processing time when compiled with optimization routines. An unoptimized binary took an average of 1.27 second to complete whereby a properly optimized binary took less than half the time to complete the same task. The 2nd level optimization is 146% faster than an unoptimized binary and 3rd level optimization applied with specific K8 tunings is 279% faster. The average processing time taken by all binaries is shown here:
On both neural network models, 3rd level optimization with K8 specific flags shows a large increase in execution speed with an average of 279% gain in performance. This is a large increase considering there’s no other changes applied to the binaries except for the optimization sequences.

5.4 Haar-classifier Based Face Detection

Performance comparison between the 3 different optimizations shows that a significant speed increase whereby an unoptimized binary took an average of 0.278 second to complete and a properly optimized binary successfully lowers the processing time to around half the time required. The average processing time required by the algorithm under different optimizations is shown next:

The average performance increase when compiled under 2nd level optimization sequences shows a significant 75% better performance and when compiled under 3rd level optimization with specific K8 routines doubles the performance obtained without any optimizations. This follows the other test results obtained beforehand where optimization sequences have a significant impact on the performance gained and code execution speed.

6. CONCLUSION

In this research, we have found that significant performance gain could be had by optimizing the algorithm and supporting libraries by issuing specific and generic optimization flags to the compiler during the algorithm’s compilation stages. Specific architectural optimizations following the target architecture of the system also have a large impact on the execution speed which in turn affects the entire system performance. By implementing a proper optimization sequences according to the correct target
architecture will help increase the system processing speed and help the system to deliver better performance.

7. ACKNOWLEDGEMENTS

The authors acknowledge Universiti Malaysia Perlis (UniMAP) and everyone else that help enabled the production of this research project.

8. REFERENCES


