

Senior Project

Utilization of Automated GCC Optimization For Dual-Width Instruction Sets on the ARM Architecture.

Shane Watson
Dec 3, 2010
Professor Lupo

Abstract	2
Related Work	4
Benefits and Sacrifices.....	6
Thumb Instruction Set.....	7
Setting Up The Board	11
Test Considerations.....	13
Industry Standard Component	16
Results.....	18
bzip2.....	18
x264.....	21
Conclusions.....	22
bzip2.....	22
x264.....	23
Overall.....	24
Future Work.....	24
Appendix.....	26
Works Cited	48

ABSTRACT

One of the most important considerations in embedded systems is code size. This consideration is obviously imposed by external factors such as cost and physical space, but what it boils down to is that we want our devices to be as powerful as they can within a (typically limited) specific form factor. This limits the amount of space we have for memory and as such we should always be considering the code size of our application and making sure it's as efficient as possible. We also then need to consider other factors such as performance and power consumption.

This is where the ARM architecture comes into place. This architecture supports a dual-width instruction set meaning that it can switch between 16- and 32-bit instructions on the fly with a few different instructions. This allows us to reduce our code size with an expected hit to performance. Now that these processors are increasing in popularity, there is a lot of support for them including GCC (GNU C Compiler) support. GCC has two flags that allow us to tell the compiler to use these Thumb mode optimizations where possible. The purpose of this project is to measure what code size reduction we see against how much increase in processing time there is for a few different processor intensive applications.

Introduction

In embedded systems, there are many restraints put on the design of the system. The most important restriction is cost. This restriction imposes limitations on every area of the planned project. The designer/developer must make sure that the solution solves the necessary problems in a robust and efficient way while ensuring that the cost will not exceed the budget. This means that most embedded systems deal with a limited amount of available memory and system resources (as this tends to reduce both cost and physical size).

An interesting solution to this has been the advent of processors that support mixed width instruction sets, which is the ability to have instructions of different sizes. One of the most popular and prevalent examples of this is the ARM processor which is being used more and more every day in devices such as smart phones and hand held gaming consoles. The ARM processor has what is called a "Thumb mode" which allows the processor to switch from a 32-bit instruction set to a 16-bit one on the fly.

With the increased popularity of mobile and handheld devices, I wanted to see what can be done to quickly optimize applications for these platforms. The ARM architecture has been improving rapidly and there is a lot of community support for the platform. There is also extensive compiler and cross-compiler support which is very easy to set up and start working with. GCC now has built in flags to allow the user to utilize Thumb mode without needing to go in and optimize any of the code by hand. My project is based on measuring how much code space reduction these flags provide as well as how it affects performance when using processor intensive applications.

Related Work

One previous report dated Aug 2003 used many different forms of code translation to find varying degrees of improvement in code size. This experiment started off with profiling the code for execution-time-heavy functions. They targeted any function that took more than 5% of the overall execution time using `gprof` and modified the code in those segments based on four separate heuristics (Krishnaswamy, 58). They generated two versions of each of these functions, one based on the Thumb instruction set, the other based on the ARM instruction set. They chose which set to use based on the aforementioned heuristics. The heuristics were as follows:

- 1) Compare the cycle counts of the Thumb version of the function to the ARM version and chose the one with fewer cycles. This allows them to take into account both the absolute number of instructions and the behavior of the cache
- 2) Choose the instruction set for the function based solely on instruction counts. This allowed them to save time in simulating as it is not required for this heuristic.
- 3) Using a predetermined threshold $T\%$, they determined which instruction set to use based on whether or not the Thumb code was $T\%$ smaller than the ARM code.
- 4) The final heuristic was a combination of two. They would choose Thumb code if the Thumb instruction count was less than the ARM instruction count or if the Thumb instruction count was higher by no more than $T1\%$ and the Thumb code size was smaller by at least $T2\%$ (both of these being separate predetermined thresholds). (Krishnaswamy, 60)

The aforementioned method was provided as a method of coarse tuning the generation of ARM vs. Thumb code. That is, on a function level this is an all or nothing approach. If the given heuristic determines Thumb code to be used, the Thumb instruction set is used for the entire

function, and likewise for ARM instructions. They offered a solution for a fine grained approach as well which was based on targeting the instruction types (Branch, ALU, MOVE, etc) which were most to blame for increased instruction count in Thumb code. They used the functions which Heuristic 4 determined to be best suited with Thumb code and went in by hand to change some of the targeted instructions back to ARM instructions to hopefully save execution time (Krishnaswamy, 63).

They found that Heuristic 4 worked best for the coarse grained solution. They were able to achieve lower code size and instruction cache energy while still offering acceptable performance. They also determined that the fine grained option was not a significant improvement due to the addition of bx statements to switch back and forth between the two instruction sets (Krishnaswamy, 64). This project was very relevant to my project since I am using solely pre-generated code that required no fine tuning on the ASM level. They were able to find a worthwhile improvement using automatically generated code.

Another related project that was relevant to my work was based on UniCore32 as opposed to an ARM architecture which provides an insightful contrast. It is still closely related to the work that I am doing because they both have dual-width instruction sets that allow for both 16-bit and 32-bit instructions. This is an interesting comparison to see if a different architecture might perform better than the ARM processor when it comes to utilizing a dual-width instruction set. The other interesting contrast this project provides is that all of their optimization was fine grained and done at link time (Xianhua, 670).

They compiled each program using the gcc -O2 flag to begin generating compact code. This was interesting as I ran into problems with optimization flags (as I will go into in the test portion of the paper), though it is ultimately unrelated because they chose to fine grain optimize

the code at link time. They were able to find a significant reduction in code space without a significant impact to performance (Xianhua, 672).

Benefits and Sacrifices

This feature, however, introduces a small problem. With the Thumb mode we are going to save code space by using 16-bit instructions in places that previously had 32-bit instructions but the trade off for code space is going to be execution time. As we translate instructions from 32-bit into their 16-bit equivalents, some of what were previously single instructions will now take two depending on the specific operation. In the best case we are going to see a 1:1 translation which would be ideal as it would cut code space in half and keep the same general execution time and in the worst case we are going to see a 1:2 translation which would be the same code size and worse execution time. The average case, however, is going to lie somewhere between the two, we are going to have to weigh code space against desired execution time.

Code space and power consumption are both very important considerations in the current embedded systems market as we want to fit as much functionality as we can into as little space as possible (both physical and in terms of memory as they are correlated) and consuming as little power as possible. The power consideration is twofold, it allows our device to function for longer periods of time on the same battery capacity, or it allows us to keep the same amount of usage time while making the package smaller by using a more compact battery/power source.

There are a few different sacrifices we encounter given these optimizations, however. The first is a more limited instruction set. ARM instructions have a three address format and allow us access to all sixteen registers in the processor, however Thumb instructions (being 16-bits as opposed to their 32-bit counterparts) only supports a two address format and restricts us to accessing only eight of the registers (Krishnaswamy, 57). This means that we have many

considerations to make when we are deciding when and where to utilize the Thumb mode of the ARM architecture.

Thumb mode instructions also have other benefits that aren't immediately apparent. Since the instructions and their respective instruction set are smaller, the spatial proximity of these instructions is also reduced as compared to a standard instruction set (the standard ARM instruction set, for example). This means that we have better spatial locality in the instruction cache and fewer cache misses overall. In addition to this, the ARM processor also still fetches 32 bits at a time which means it is fetching two instructions at a time when we are in Thumb mode.

Thumb Instruction Set

The operation of Thumb mode hinges on a few important instructions in the ARM Instruction Set. These operations tell the processor what mode the instructions that follow the branch will be in based on a bit in the provided register. There are four instructions that support switching between ARM and Thumb instruction sets, two of them are standard and have full support, while the other two are more limited in scope and only supported by a limited number of variants.

The two that have full support are branch instructions. The first of these is the "branch and exchange" instruction which is of the form `BX Rm`. The `BX` instruction sets the program counter to `Rm` and checks the `Rm[0]` bit. If this bit is a 1, that instruction is going to be a Thumb instruction, if it is a 0 then it will be an ARM instruction. The second is the "branch with link and exchange" instruction which is of the form `BJX Rm`. This instruction assigns the address of the next instruction to the link register and assigns `Rm` to the program counter, it also evaluates `Rm[0]` in the same way that the `BX` instruction does.

The next two instructions are available in all versions of the ARM processor however they have special functions for certain variants. These will only exchange and evaluate the corresponding bit representing ARM vs. Thumb following instruction in the 5T and up variants of the ARM architecture (that is to say the instruction itself is supported more widely, but they can only be used to switch modes in the aforementioned variants). The first of these instructions is the "Block data load return (and exchange)" instruction which is classified with the "Load multiple" instructions. This instruction is of the form $LDM\{IA | IB | DA | DB\} Rn\{!\}, <reglist+PC>$ <see Appendix I>. It loads the given registers, sets the program counter to $[address][31:1]$, and evaluates $[address][0]$ in the same way as the branch instruction. The other limited use instruction is "Block data load return (and exchange) and restore CPSR" (the CPSR is the current program status register). It is of the form $LDM\{IA | IB | DA | DB\} Rn\{!\}, <reglist+PC>^{\wedge}$ and adds to the previous instruction the functionality of updating the current program status register with the value from the saved process status register.

(Arm.com)

BOARD

For the testing portion of my project, I am using a revision C4 Beagleboard. It is an all-inclusive ARM based development board that is sufficiently powerful to allow me to do my testing for this project as well as being compact (3.5"x3.5") and having an established community for support if necessary. It utilizes POP technology (Package on Package) which allows both NAND and SDRAM memory to be mounted directly onto the processor. It is also a fanless board which saves space and energy.

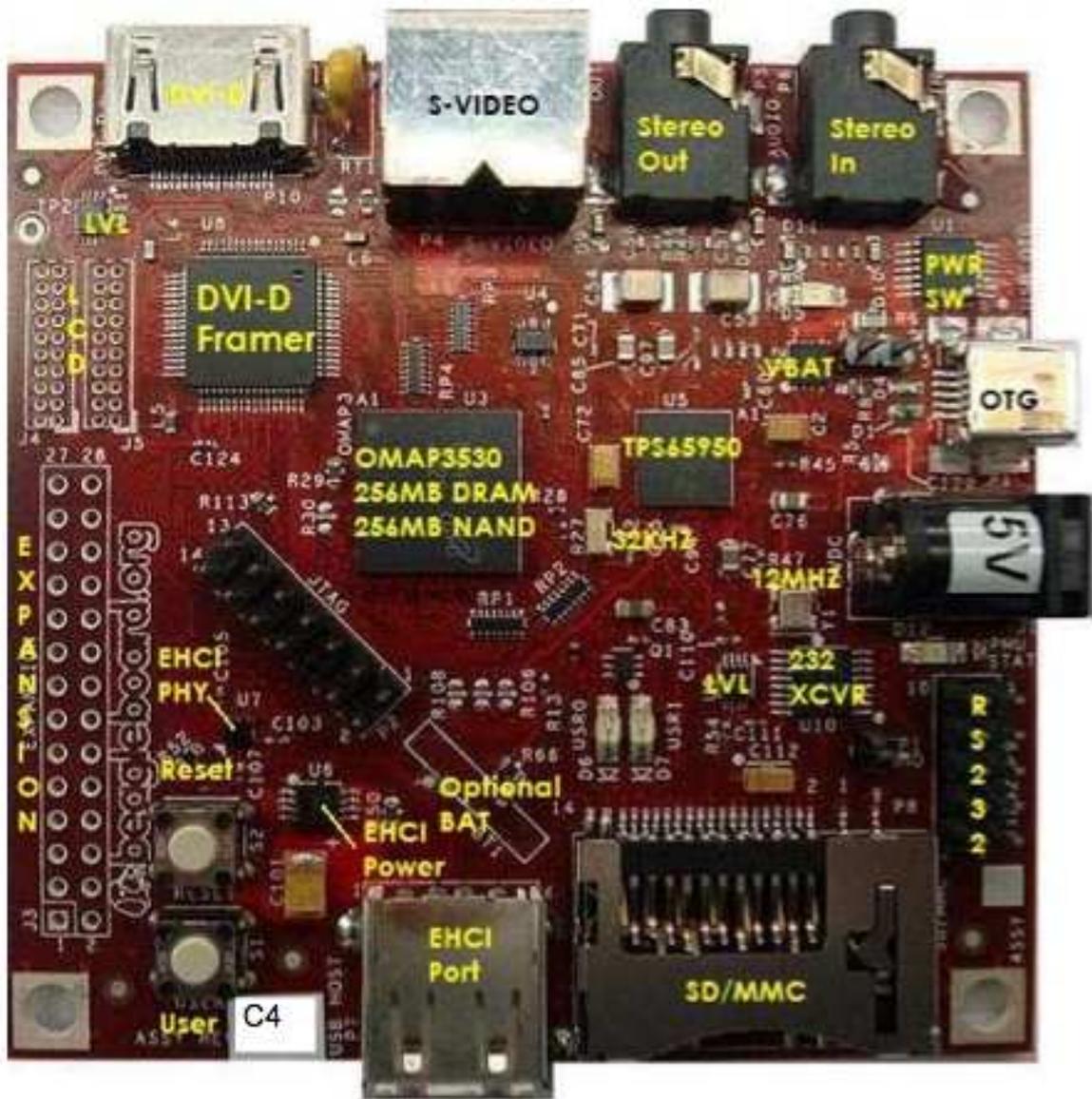


Figure 1. Picture of BeagleBoard with labels. (Beagleboard.org, 42)

The board comes with the TI OMAP3530DCBB72 processor which runs at 720MHz (Beagleboard.org, 18). This is an ARM Cortex-A8 processor (ARMv-7A architecture) which supports Thumb-2 instructions which is necessary for my project. It has a 32KB L1 cache and a 256KB L2 cache (TI page). The board comes with 256MB NAND and 256MB SDDR SDRAM running at 166MHz, it also has an onboard SD/MMC slot and USB port which can be used for expanded memory (via SD/MMC card or USB flash drive, respectively). The Beagleboard also

uses the TI TPS65950 IC to regulate power. Other features of this IC include stereo audio in/out, USB 2.0 OTG, and a status LED.

The board also provides a DVI-D interface via an HDMI connector. It uses the HDMI connector so as to conserve physical space and it does not support the full HDMI interface (Beagleboard.org, 20). This can be used with an HDMI to DVI-D cable to provide output to a standard monitor. Also provided on board is an IDC-10 interface which allows the use of an IDC-10 to DB9 (RS-232) adapter for serial communication. In addition to the USB 2.0 USB (mini USB) connection it also contains a Type A USB connection.

The USB Type A connection allows for a great deal of expandability. It can be used to plug in a powered USB Hub for extra USB slots, to provide power to the board, to connect any USB device, or to connect a USB to ethernet adapter. The board has a standard 5.0V power adapter that can be purchased for it. The set up that I used it as follows. At first I started with a mini USB to female Type A USB adapter. I purchased a powered USB hub which contains three Type A USB adapters, an ethernet port, a Type B USB adapter (used to connect to the board) and its own power adapter. I plugged this into an outlet and obtained a USB Type A male to 5.5mm power plug adapter cable which I plugged in to one port on the hub and used the power plug to provide power to the entire board. I connected the Type B adapter on the hub to the mini USB to Type A adapter I purchased for the board. This presented booting issues at first as on occasion, the board could not pull enough power from the mini USB connection. Once I pulled that out of the equation and plugged the hub into the Type A directly on the board I didn't face any other power issues. This left me with two Type A USB adapters on the hub as well as the USB mini to Type A adapter which was more than enough for my purpose.

The board contains a host of other adapters and connections as well. It has a 14 pin JTAG header which allows for advanced debugging with a JTAG emulator. This was out of the scope of this project, however it's a useful feature. The board also contains an S-Video out adapter, a few status LEDs, and User and Reset buttons. Finally, the board has a spot to connect two 2x10 LCD headers to access the LCD signals. All in all, it is a very complete and self contained board.

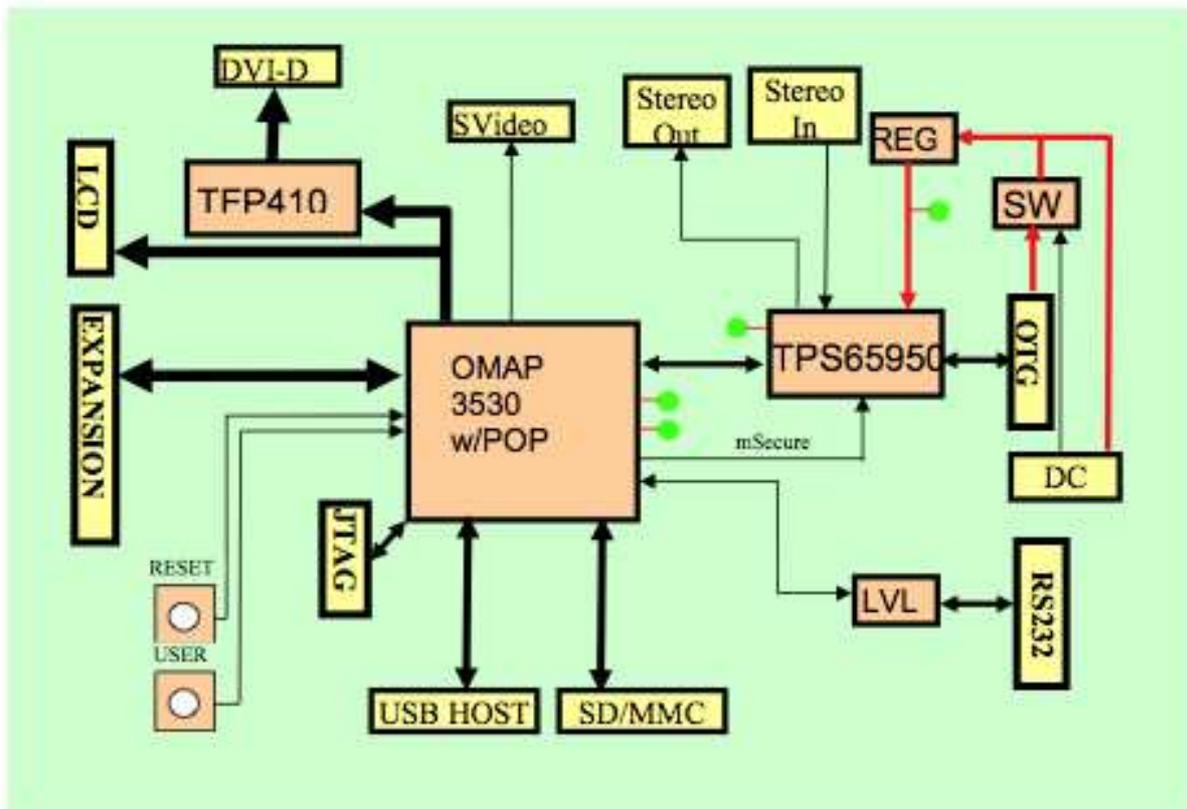


Figure 2. High Level Block Diagram of BeagleBoard. (Beagleboard.org, 41).

Setting Up The Board

The first consideration for this project was to get an operating system installed on the board. My original searches turned up with mostly information about an image of the Angstrom OS ported for the ARM architecture. I started off using their provided demo image on a partitioned SD card, there was a boot partition and a data partition. The SD card I used has

roughly 1 GB of free space, and the demo image itself took about 650mb of that. The demo image included everything necessary to use the board as a fully functioning computer. It had prepackaged libraries and a large amount of software included. It had a nice, simple GUI which would work well with the Beagleboard's HDMI to DVI-D adapter mentioned above.

I ran into some rather interesting issues for the first few days of trying to get Angstrom up and running properly. Every time it would boot, it gave me an ambiguous mini USB error with a 3 minute timeout. After this, it would boot properly each time. At this point I was using a mini USB to USB Type A female adapter which the hub was plugged into (which was then powering the board, as mentioned before). I discovered that the board was not able to pull enough power from the mini USB port, so I switched the hub to being plugged directly into the Type A adapter on board. Once I did this, that problem went away.

After spending some time getting used to Angstrom, I decided that this image was much heavier than what I needed to use. I decided that to give myself flexibility with testing (since available space was also a concern with this SD card), I should switch to a more compact image as the GUI and software were unnecessary for my purposes. Using Angstrom's online image builder, I was able to put together a minimal image with only necessary packages and applications which lowered the image size to around 150mb. This package did not include the GUI as all I needed as a console. This image also proved to be troublesome at first. It was trying to load drivers for an expansion device that I don't actually have, and it was crashing on those drivers. I built another image that trimmed some unnecessary add on modules (as I had no expansion devices connected to the board) and finally got the entire system running smoothly.

Test Considerations

My main considerations when deciding on the test cases were that I wanted applications which were processor intensive but also ones that are commonly used. I decided on using file compression and decompression as well as some form of video encoding or decoding. My initial plans were to use bzip and ffmpeg, however due to issues with ffmpeg that will be addressed later, I ended up using the x264 codec instead. When it came to deciding the exact application to use, I went with the supplied source for each of these applications and compiled them directly on board.

Originally I wanted to cross-compile for the Beagleboard but that proved to be a little more difficult than anticipated. I was unable to find a reliable tool chain for a Windows platform, even considering Cygwin. I went on to then try four or five tool chains for OS X. Some of these were "pre-built" tool chains and some of these were instructions on what all was needed to effectively build your own tool chain. I ran into complications with these as well mostly including the compiler not being able to find certain standard libraries. I decided the easiest solution would be to just compile directly on-board so I could begin testing as soon as possible. I started with downloading the standard source code from bzip to begin testing both compression and decompression. After transferring the source tarball to the Beagleboard, I untarred it and compiled it without issue. After transferring the necessary files to a separate directory, I edited the Makefile to utilize GCC's Thumb compatibility. Bzip was a great application to get started with as it does not require configuration prior to compiling, and the Makefile is laid out very neatly in order to better assist with cross-compilation. However, due to aforementioned problems I was unable to cross-compile. I added the necessary flags to the CFLAGS var, `-mthumb` and `-mthumb-interwork`, and compiled the second version for testing. On the first run I had

actually done so incorrectly, I only included the `-mthumb-interwork` flag, compiled and went through an entire round of testing before realizing the two versions I compiled were in fact identical.

For test data, I wanted a few tests with varying file types. I chose a 300mb image file for the binary tests, and I made text only files for plaintext testing (script text at Appendix II A). Each test was run with a single file so as to not need a tarball. The text files I used were chunks of repeated text in 10mb, 100mb, and 400mb sizes (scripts at Appendix II C-D). The last of those was the largest I could conceivably fit on the SD card when considering the added space of the compressed file alongside the standard file. The text files contained the same 10kb block of text repeated throughout the file. My test plan was to compress the file then decompress the file fifteen times each with the regular version, then another fifteen times with the Thumb version. I wrote a script to automate this portion of the testing and output the data to a file for parsing. I chose to run each test fifteen times so that I could take an average of test cases for use in the final results calculations. It is an amount that will give me a fairly accurate idea of the execution time without taking far too long to run all of the tests I had planned.

After doing the testing for zip, I moved on to my second application. My initial plans were to use ffmpeg to encode and decode various video files, and follow the same general guidelines as with the previous application. After downloading the source and extracting it onto the SD card for the beagleboard, I moved on to compiling it. The regular version compiled with no issues. The source for this application uses a configure file with a makefile so I added the necessary flags to the `$(CFLAGS)` variable in the `config.mak` file so that those would get used at compile time. When I started compiling the Thumb enabled version of ffmpeg, I started getting compiler errors. There were problems in the generated assembly code, and the compiler error

was "Unshifted register required". After doing some research, I learned that there was no good way around this besides digging into the assembly itself which is beyond the scope of this project. I also read that they decided specifically to not build Thumb support into ffmpeg because their tests showed no significant gains by doing so.

As a replacement for ffmpeg, I used the x264 codec for my second and final application. The standard compile of this application went without issues, but I ran into a problem with the Thumb mode again. Much like the ffmpeg source, the x264 codec also used a configure and makefile. After adding the flags to the config.mak \$(CFLAGS) variable again, I compiled only to receive an "Offset out of range error" on multiple files. The existing \$(CFLAGS) contained the `-O3` optimization flag which I had read sometimes interferes with the generation of Thumb code. I attempted to compile the codec with the `-O2` flag, the `-O1` flag, and the `-Os` flag each separately, but they all produced the same error. Interestingly, the amount of files which produced the error went down from `-O3` to `-O2` and decreased again from `-O2` to `-O1` where only one file gave the error. The `-Os` flag gave the same errors as `-O2`. Once I removed the optimization flags completely, it compiled without issue.

Considering each optimization flags caused errors, I believe that the core issue involves some of the flags that are set by `-O1` (as `-O2` and `-O3` build off of `-O1`). This flag turns on `-fthread-jumps` and `-fdefer-pop` always, and if supported it turns on `-fdelay-branch` and `-fomit-frame-pointer`. I think that the error involves `-fthread-jumps` and (if enabled) `-fdelay-branch`. As we have seen from the ARM and Thumb instruction sets, the branch statements are the most important as they control the switching between the ARM and Thumb instructions, so I think something gets messed up when combining that with `-mthumb` and `-mthumb-interwork`. I do, however, think that this is an edge case and it is

specific to x264 (and possibly other applications, but not a general large scale bug) as other applications compile with `-mthumb`, `-mthumb-interwork` and an optimization flag without problems.

For this test, I used three sizes of raw video files and encoded them using the x264 application. Due to hardware and software limitations, I was unable to add decoding to my tests (the application would compile with only encoding support). I used file sizes of 230mb, 720mb, and 1.15gb. The first test was run in the same way as the bzip testing: original file stored on the SD card, scripted to run 15 times (scripts can be seen at Appendix E-G). The latter two files were too large to fit on the SD card, so I used a USB flash drive plugged into the USB hub to store the original files, and the output files were saved onto the SD card. I originally connected the flash drive to the mini USB to USB adapter however it suffered from the same problems as when I tried to use that to power the board, namely that it could not provide enough power to the flash drive. The flow of this test set was to compress the file, record the total compression time, then delete the compressed file. Again I did this fifteen times for each version of the application.

Industry Standard Component

The last tests I ran were for the industry standard component. I obtained standard I/O for bzip from a SPEC benchmark and modeled some tests to replicate the data and check my output file sizes against theirs. The spec benchmarks seemed to take each sample input file and duplicate it a varying number of times to obtain a certain file size, then run the bzip compression and decompression on that enlarged file. Their increments didn't appear to be exact duplications, so I was unable to perfectly mimic their test files. To make up for this, I created a script that used `cat` to duplicate each file enough times to put it over their end file size, then used `truncate` to get the file size to be the same (script located at Appendix II I). This introduces problems in that

there's no way of knowing that my data for these test files is identical to the data that was used for the SPEC benchmark after the duplications. I then wrote a script to mimic the output of the benchmarks, running zip compression then decompression on each file three times (one each at levels 5, 7, and 9), and outputting the filesizes to verify at each step for each of the two versions of the application.

The following table shows the expected result from the benchmark in addition to the results obtained by each of my bzip applications for each test file. The script used to generate these results can be seen at Appendix II H, and the raw output of the script can be seen at Appendix V to ensure the correct filesize at each point in the testing.

Level 5 Compression				
Filename	Initial Filesize	Expected	Actual (Regular)	Actual (Thumb)
byoudoin.jpg	5242880	5189497	5189497	5189497
chicken.jpg	31457280	31306985	31306985	31306985
dryer.jpg	2097152	1434910	1434910	1434910
input.source	293601280	54403299	54403299	54403299
liberty.jpg	31457280	21917976	21917976	21917976
text.html	293601280	22290700	22290700	22290700

Figure 3. Level 5 compression chart.

Level 7 Compression				
Filename	Initial Filesize	Expected	Actual (Regular)	Actual (Thumb)
byoudoin.jpg	5242880	5170636	5170636	5170636
chicken.jpg	31457280	30427899	30427899	30427899
dryer.jpg	2097152	1127895	1127895	1127895
input.source	293601280	53599136	53599136	53599136
liberty.jpg	31457280	17697398	17697398	17697398
text.html	293601280	17811990	17811990	17811990

Figure 4. Level 7 compression chart.

Level 9 Compression				
Filename	Initial Filesize	Expected	Actual (Regular)	Actual (Thumb)
byoudoin.jpg	5242880	4554254	4554254	4554254
chicken.jpg	31457280	25949409	25949409	25949409
dryer.jpg	2097152	1067335	1067335	1067335
input.source	293601280	52960821	52960821	52960821
liberty.jpg	31457280	13755071	13755071	13755071
text.html	293601280	14427351	14427351	14427351

Figure 5. Level 9 compression chart.

Results

bzip2

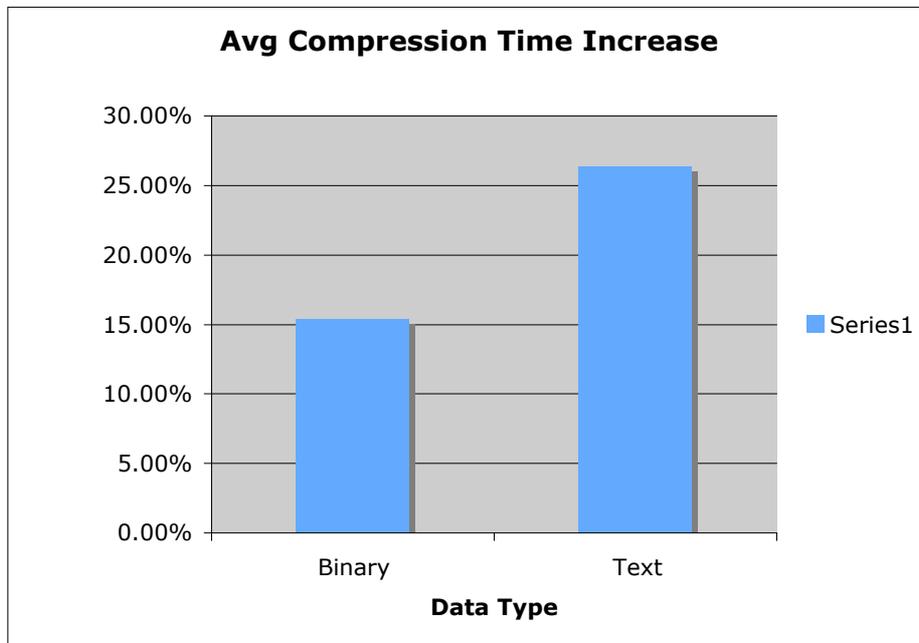


Figure 6. Average increase in compression time comparison.

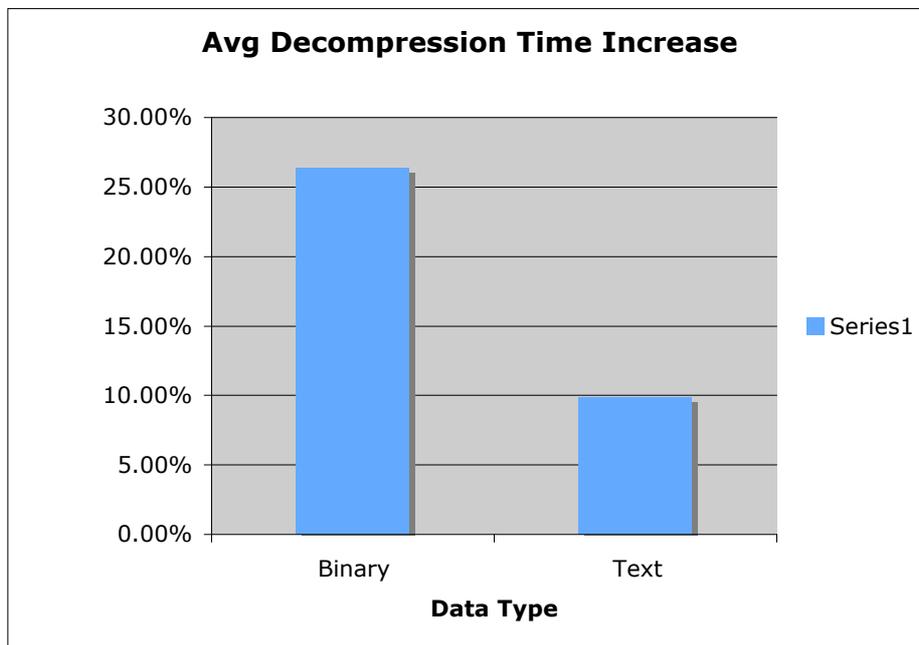


Figure 7. Average increase in decompression time comparison

The original application size for the compiled bzip2 binary is 204,305 bytes, and the compiled size of the bzip2 binary using Thumb mode is 170,605 bytes. This gives us an overall reduction of 16.5% (all percentage calculations use the formula found in Appendix VI). This is significantly less than what is expected by the related work which tended to see a decrease of around 30%. Most of those optimizations were done by hand as opposed to being automated.

Starting with the 300mb binary file, we see an increase of 15.4% in the execution time of the compression algorithm and an increase of 26.4% in the execution time of the decompression algorithm. That's a fairly large difference, however in terms of absolute execution time it is not very significant considering the average execution time for compression and decompression were 26 minutes 6 seconds and 11 minutes 36 seconds respectively for the standard ARM application. The raw data for this can be seen in Appendix III A.

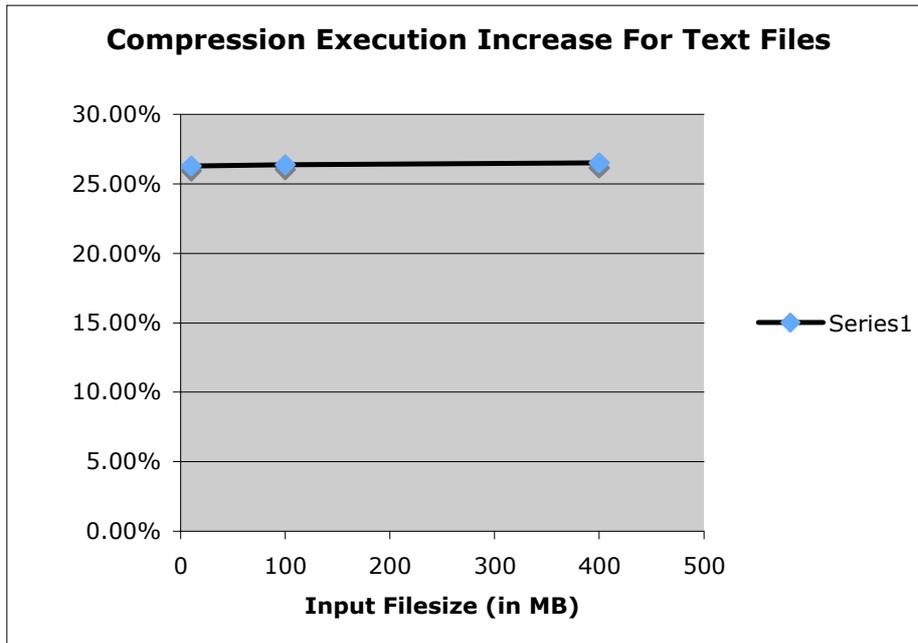


Figure 8. Increase in compression time for text files.

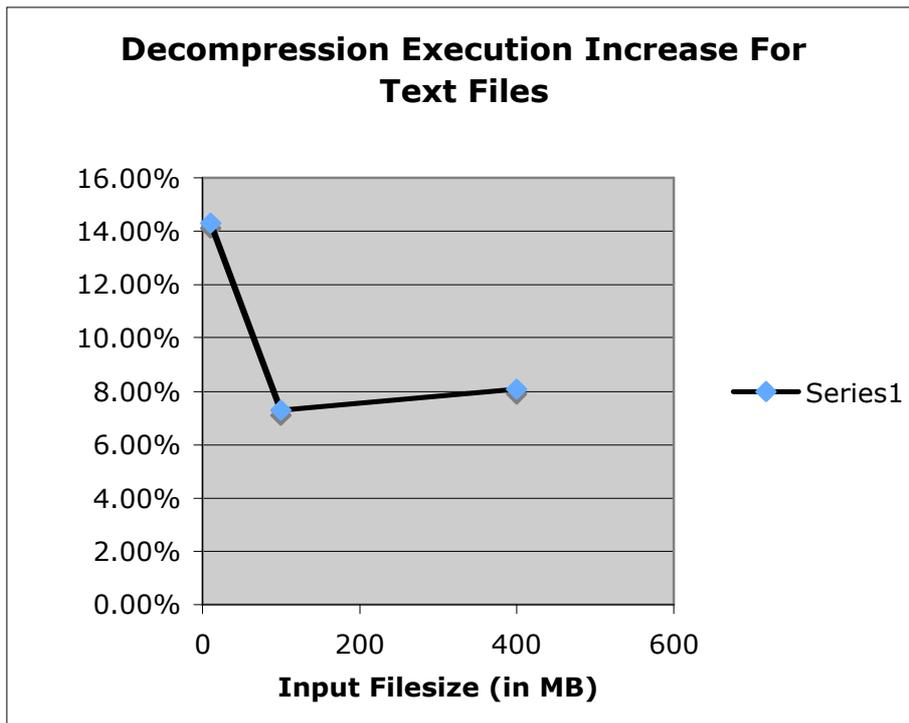


Figure 9. Increase in decompression time for text files.

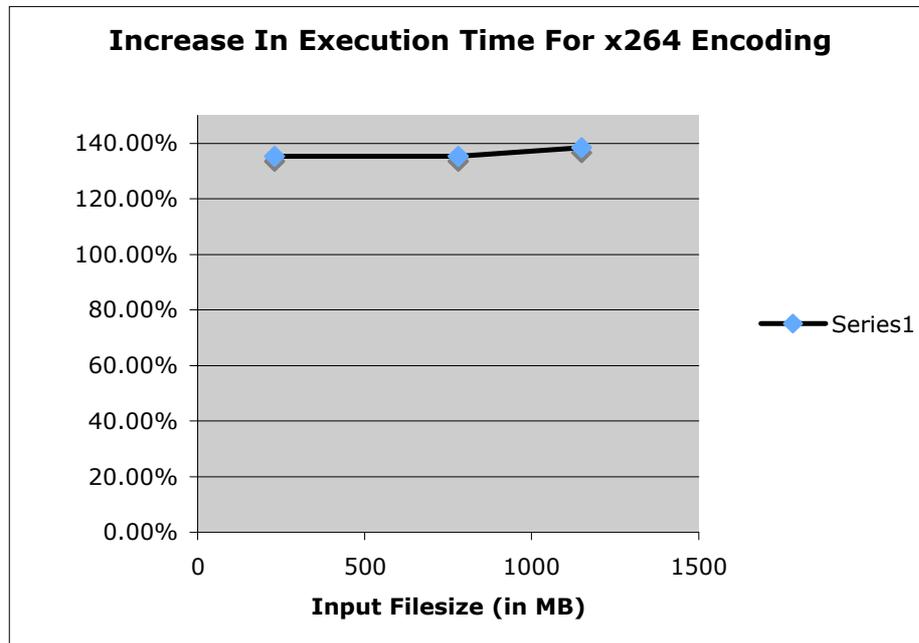
As I moved on to testing plaintext, I started seeing a slightly different trend. For the 10mb text file, there was an increase of 26.3% in the execution time of the compression algorithm and an increase of 14.3% in the execution time of the decompression algorithm (see Appendix III B). This is almost exactly the opposite of the results that were seen with the binary files. For the 100mb text file, there was an increase of 26.4% in the execution time of the compression algorithm and an increase of 7.3% in the execution time of the decompression algorithm see Appendix III C). The compression increase was identical to the smaller text file but there was a much more acceptable increase for the decompression algorithm. Finally, for the 400mb text file there was an increase of 26.5% in the execution time of the compression algorithm and an increase of 8.1% in the execution time of the decompression algorithm (see Appendix III D). These are much closer to the increases of the 100mb file. The discrepancy between the 10mb file and the other files with respect to the execution time of the decompression algorithm is rather interesting.

x264

The filesize for the standard x264 binary compiled on the beagleboard is 699,899 bytes, and the filesize for the Thumb mode x264 application is 857,051 bytes. This is an **increase** of 22.5% in our code size. This is an unexpected result. I believe that is a direct result of the fact that I had to remove all optimization flags in order to get this application to compile properly.

The x264 analysis is a little simpler than the previous application. This is due to the fact that I only used one file type as input (raw video files) and only performed one task with the application (encoding at 720p) due to limitations. Starting with the 230mb raw video file, there was an increase of 135.2% in the encoding time (see Appendix IV A). The tests on the 780mb raw video file showed an increase of 135.3% in the encoding time (see Appendix IV B). Finally,

the 1.15gb raw video file showed an increase of 138.4% in the encoding time (see Appendix IV C). These are all very close together and are significantly higher than the results seen with the bzip application. One thing that's worth noting in regards to the absolute execution time data for the x264 encoding tests is that the 230mb file was read from and the encoded file was written to the SD card whereas the 780mb and 1.15gb files were read from the USB flash drive and had the



encoded files written to the SD card.

Conclusions

bzip2

On average, bzip2 sees an increase of around 25% (for text files, less for binary) in compression execution and an increase of around 10% (for text files, greater for binary in this case) in decompression execution. We can expect that the average input data case will be some combination of the two. I believe that this is an acceptable increase in execution time alone, however we are only getting around 15% reduction in code size which, while nice, isn't as

significant as we would like for most embedded system designs. I would really only use it if the 15% decrease was absolutely necessary for my system.

There was, however, an outlier in the text decompression data. The increase for the 10mb decompression was much more significant (almost double) than the other two data points. Looking at the raw data, the comparison for this was Actual – 7 seconds and Experimental – 8 seconds. I believe the true value is much smaller and closer if not lower than the 7% and 8% we saw with the larger text files, however, due to the resolution being one second, I was unable to get detailed enough data to prove this.

x264

The outcomes of the x264 tests were surprising. I have so far been unable to find published works which share the significant increase in code size for the Thumb version of this application. I understand that, if done in an entirely suboptimal way, the code size **can** increase. In theory this would be caused by using too many `bxr` type functions for code segments that are too short or incorrectly formed to actually benefit from switching to the Thumb instruction set. Incorrectly formed code segments would be areas where the amount of Thumb instructions used in that segment plus one 32-bit instruction and one 16-bit instruction (a `bxr` type instruction to enter thumb mode and another to exit) is greater than two times the number of ARM instructions it would take to perform the task of that code segment. This would cause the overall code size of the necessary Thumb segment to be larger than the code size of the equivalent ARM segment.

The overall data itself left much to be desired as well. An increase of 135% in execution time is unacceptable unless being balanced by a significantly large decrease in code size as well as being in a system where both 1) the code size decrease was absolutely necessary and 2) the execution time was not critical. Another downside to the large increase in execution time is that

it will consume more power as well. This makes the current automated Thumb code generation for this application entirely undesirable.

Overall

The data appears to be highly volatile overall. It's dependent on a very large number of factors including, but not limited to, application type and input data type for the given application. I do not feel that the current state of GCC support for automatic generation of Thumb code is optimal. Given the large number of factors present, it's unlikely to be a good general solution to use if you are looking for significant improvements and really getting the most out of this architecture. I believe that at this point in time, fine tuned and hand based optimizations will be much more beneficial.

Future Work

There are a lot of things that this project can lead to. First of all I would like to see results for similar applications running in a more common embedded system such as the iPhone. I know there is some Thumb support but I have not done ample research to know the extent of that support.

I would also like to further investigate exactly how the compiler makes its decisions for which code segments it decides to translate to Thumb code in hopes of being able to improve on what is already there. Having GCC support for Thumb is really a great thing because it makes these optimizations more accessible, and if those are improved upon it could mean great things. Reaching the point where you could be confident that using these flags would provide a reasonable decrease in code size without a significant increase in execution time would be great for the embedded system community (of course, "reasonable" and "significant" in and of themselves are subjective terms and mostly dependent on the specific project). Finally, I would

love to research more into hand optimization specifically on a somewhat automated level such as the related work with heuristics to on which to form decisions regarding which areas of the code should be translated.

APPENDIX

I. ARM ISA Instruction Key

Key to Tables	
Rm {, <opsh>}	See Table Register, optionally shifted by constant
<Operand2>	See Table Flexible Operand 2 . Shift and rotate are only available as part of Operand2.
<fields>	See Table PSR fields .
<PSR>	Either CPSR (Current Processor Status Register) or SPSR (Saved Processor Status Register)
C*, V*	Flag is unpredictable in Architecture v4 and earlier, unchanged in Architecture v5 and later.
<Rs sh>	Can be Rs or an immediate shift value. The values allowed for each shift type are the same as those shown in Table Register, optionally shifted by constant .
x, y	B meaning half-register [15:0], or T meaning [31:16].
<imm8m>	ARM: a 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits. Thumb: a 32-bit constant, formed by left-shifting an 8-bit value by any number of bits, or a bit pattern of one of the forms 0xXYXYXYXY, 0x00XY00XY or 0xXY00XY00.
<prefix>	See Table Prefixes for Parallel instructions
{IA IB DA DB}	Increment After, Increment Before, Decrement After, or Decrement Before. IB and DA are not available in Thumb state. If omitted, defaults to IA.
<size>	B, SB, H, or SH, meaning Byte, Signed Byte, Halfword, and Signed Halfword respectively. SB and SH are not available in STR instructions.

<reglist>	A comma-separated list of registers, enclosed in braces { and }.
<reglist-PC>	As <reglist>, must not include the PC.
<reglist+PC>	As <reglist>, including the PC.
+/-	+ or -. (+ may be omitted.)
§	See Table ARM architecture versions .
<iflags>	Interrupt flags. One or more of a, i, f (abort, interrupt, fast interrupt).
<p_mode>	See Table Processor Modes
SPm	SP for the processor mode specified by <p_mode>
<lsb>	Least significant bit of bitfield.
<width>	Width of bitfield. <width> + <lsb> must be <= 32.
{X}	RsX is Rs rotated 16 bits if X present. Otherwise, RsX is Rs.
{!}	Updates base register after data transfer if ! present (pre-indexed).
{S}	Updates condition flags if S present.
{T}	User mode privilege if T present.
{R}	Rounds result to nearest if R present, otherwise truncates result.

II. Scripts for automation.

A. Testing bzip with 300mb binary file

```
#!/bin/bash
```

```

LIMIT=15

echo "300mb ISO Regular"
for ((a=0; a < LIMIT; a++))
do
    echo "Compress"
    echo -n "`date +%T` |"
    ./applications/bzip-reg/bzip2 -zq data.iso
    echo "`date +%T`"
    echo "Decompress"
    echo -n "`date +%T` |"
    ./applications/bzip-reg/bzip2 -dq data.iso.bz2
    echo "`date +%T`"
done

echo "300mb ISO Thumb"
for ((a=0; a < LIMIT; a++))
do
    echo "Compress"
    echo -n "`date +%T` |"
    ./applications/bzip-thumb/bzip2 -zq data.iso
    echo "`date +%T`"
    echo "Decompress"
    echo -n "`date +%T` |"
    ./applications/bzip-thumb/bzip2 -dq data.iso.bz2
    echo "`date +%T`"
done

exit 0

```

B. Testing bzip with 10mb text file

```

#!/bin/bash

LIMIT=15

echo "10mb Text File Regular"
for ((a=0; a < LIMIT; a++))
do
    echo "Compress"
    echo -n "`date +%T` |"
    ./applications/bzip-reg/bzip2 -zq 10mb_text.txt
    echo "`date +%T`"
    echo "Decompress"
    echo -n "`date +%T` |"
    ./applications/bzip-reg/bzip2 -dq 10mb_text.txt.bz2
    echo "`date +%T`"
done

echo "10mb Text File Thumb"
for ((a=0; a < LIMIT; a++))
do

```

```

    echo "Compress"
    echo -n "`date +%T` |"
    ./applications/bzip-thumb/bzip2 -zq 10mb_text.txt
    echo "`date +%T`"
    echo "Decompress"
    echo -n "`date +%T` |"
    ./applications/bzip-thumb/bzip2 -dq 10mb_text.txt.bz2
    echo "`date +%T`"
done

exit 0

```

C. Testing bzip with 100mb text file

```

#!/bin/bash

LIMIT=15

echo "100mb Text File Regular"
for ((a=0; a < LIMIT; a++))
do
    echo "Compress"
    echo -n "`date +%T` |"
    ./applications/bzip-reg/bzip2 -zq 100mb_text.txt
    echo "`date +%T`"
    echo "Decompress"
    echo -n "`date +%T` |"
    ./applications/bzip-reg/bzip2 -dq 100mb_text.txt.bz2
    echo "`date +%T`"
done

echo "100mb Text File Thumb"
for ((a=0; a < LIMIT; a++))
do
    echo "Compress"
    echo -n "`date +%T` |"
    ./applications/bzip-thumb/bzip2 -zq 100mb_text.txt
    echo "`date +%T`"
    echo "Decompress"
    echo -n "`date +%T` |"
    ./applications/bzip-thumb/bzip2 -dq 100mb_text.txt.bz2
    echo "`date +%T`"
done

exit 0

```

D. Testing bzip with 400mb text file

```

#!/bin/bash

LIMIT=15

echo "400mb Text File Regular"

```

```

for ((a=0; a < LIMIT; a++))
do
    echo "Compress"
    echo -n "`date +%T` |"
    ./applications/bzip-reg/bzip2 -zq 400mb_text.txt
    echo "`date +%T`"
    echo "Decompress"
    echo -n "`date +%T` |"
    ./applications/bzip-reg/bzip2 -dq 400mb_text.txt.bz2
    echo "`date +%T`"
done

echo "400mb Text File Thumb"
for ((a=0; a < LIMIT; a++))
do
    echo "Compress"
    echo -n "`date +%T` |"
    ./applications/bzip-thumb/bzip2 -zq 400mb_text.txt
    echo "`date +%T`"
    echo "Decompress"
    echo -n "`date +%T` |"
    ./applications/bzip-thumb/bzip2 -dq 400mb_text.txt.bz2
    echo "`date +%T`"
done

exit 0

```

E. Testing x264 encoding with 230mb raw video file

```

#!/bin/bash

LIMIT=15
FILENAME=/home/root/testout

echo "230mb, Regular"
for ((a=0; a < LIMIT; a++))
do
    echo -n "`date +%T` |"
    ./applications/x264-reg/x264 --crf 24 --input-res
1280x720 -o testout_raw_230mb.mov
    echo "`date +%T`"
    FILESIZE=$(stat -c%s "$FILENAME")
    echo "Compressed filesize: $FILESIZE"
    rm testout
done

echo "230mb, Thumb"
for ((a=0; a < LIMIT; a++))
do
    echo -n "`date +%T` |"
    ./applications/x264-thumb/x264 --crf 24 --input-res
1280x720 -o testout_raw_230mb.mov
    echo "`date +%T`"

```

```

        FILESIZE=$(stat -c%s "$FILENAME")
        echo "Compressed filesize: $FILESIZE"
        rm testout
done

exit 0

```

F. Testing x264 encoding with 780mb raw video file

```

#!/bin/bash

LIMIT=15
FILENAME=/home/root/testout

echo "REGULAR, NON THUMB"
for ((a=0; a < LIMIT; a++))
do
    echo -n "`date +%T` |"
    ./applications/x264-reg/x264 --crf 24 --input-res
1280x720 -o testout /mnt/USB/raw_780mb.mov
    echo "`date +%T`"
    FILESIZE=$(stat -c%s "$FILENAME")
    echo "Compressed filesize: $FILESIZE"
    rm testout
done

echo "THUMB MODE"
for ((a=0; a < LIMIT; a++))
do
    echo -n "`date +%T` |"
    ./applications/x264-thumb/x264 --crf 24 --input-res
1280x720 -o testout /mnt/USB/raw_780mb.mov
    echo "`date +%T`"
    FILESIZE=$(stat -c%s "$FILENAME")
    echo "Compressed filesize: $FILESIZE"
    rm testout
done

exit 0

```

G. Testing x264 encoding with 1.15gb raw video file

```

#!/bin/bash

LIMIT=15
FILENAME=/home/root/testout

echo "REGULAR, NON THUMB"
for ((a=0; a < LIMIT; a++))
do
    echo -n "`date +%T` |"
    ./applications/x264-reg/x264 --crf 24 --input-res
1280x720 -o testout /mnt/USB/raw_1_15gb.mov

```

```

        echo "`date +%T`"
        FILESIZE=$(stat -c%s "$FILENAME")
        echo "Compressed filesize: $FILESIZE"
        rm testout
done

echo "THUMB MODE"
for ((a=0; a < LIMIT; a++))
do
    echo -n "`date +%T` |"
    ./applications/x264-thumb/x264 --crf 24 --input-res
1280x720 -o testout /mnt/USB/raw_1_15gb.mov
    echo "`date +%T`"
    FILESIZE=$(stat -c%s "$FILENAME")
    echo "Compressed filesize: $FILESIZE"
    rm testout
done

exit 0

```

H. Script for industry standard testing

```

#!/bin/bash

BYOUDOINJPG=/home/root/byoudoin.jpg
BYOUDOINBZ=/home/root/byoudoin.jpg.bz2
CHICKENJPG=/home/root/chicken.jpg
CHICKENBZ=/home/root/chicken.jpg.bz2
DRYERJPG=/home/root/dryer.jpg
DRYERBZ=/home/root/dryer.jpg.bz2
LIBERTYJPG=/home/root/liberty.jpg
LIBERTYBZ=/home/root/liberty.jpg.bz2
INPUT=/home/root/input.source
INPUTBZ=/home/root/input.source.bz2
TEXT=/home/root/text.html
TEXTBZ=/home/root/text.html.bz2

# byoudoin.jpg block
echo "byoudoin.jpg, regular"
FILESIZE=$(stat -c%s "$BYOUDOINJPG")
echo "Starting filesize: $FILESIZE"
echo "Level 5 compression"
./applications/bzip-reg/bzip2 -zv5 byoudoin.jpg
FILESIZE=$(stat -c%s "$BYOUDOINBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv byoudoin.jpg.bz2
FILESIZE=$(stat -c%s "$BYOUDOINJPG")
echo "End filesize: $FILESIZE"
echo "Level 7 compression"
./applications/bzip-reg/bzip2 -zv7 byoudoin.jpg
FILESIZE=$(stat -c%s "$BYOUDOINBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv byoudoin.jpg.bz2

```

```

FILESIZE=$(stat -c%s "$BYOUDOINJPG")
echo "End filesize: $FILESIZE"
echo "Level 9 compression"
./applications/bzip-reg/bzip2 -zv9 byoudoin.jpg
FILESIZE=$(stat -c%s "$BYOUDOINBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv byoudoin.jpg.bz2
FILESIZE=$(stat -c%s "$BYOUDOINJPG")
echo "End filesize: $FILESIZE"

# byoudoin.jpg thumb block
echo "byoudoin.jpg, thumb"
FILESIZE=$(stat -c%s "$BYOUDOINJPG")
echo "Starting filesize: $FILESIZE"
echo "Level 5 compression"
./applications/bzip-thumb/bzip2 -zv5 byoudoin.jpg
FILESIZE=$(stat -c%s "$BYOUDOINBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv byoudoin.jpg.bz2
FILESIZE=$(stat -c%s "$BYOUDOINJPG")
echo "End filesize: $FILESIZE"
echo "Level 7 compression"
./applications/bzip-thumb/bzip2 -zv7 byoudoin.jpg
FILESIZE=$(stat -c%s "$BYOUDOINBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv byoudoin.jpg.bz2
FILESIZE=$(stat -c%s "$BYOUDOINJPG")
echo "End filesize: $FILESIZE"
echo "Level 9 compression"
./applications/bzip-thumb/bzip2 -zv9 byoudoin.jpg
FILESIZE=$(stat -c%s "$BYOUDOINBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv byoudoin.jpg.bz2
FILESIZE=$(stat -c%s "$BYOUDOINJPG")
echo "End filesize: $FILESIZE"

# chicken.jpg block
echo "chicken.jpg, regular"
FILESIZE=$(stat -c%s "$CHICKENJPG")
echo "Starting filesize: $FILESIZE"
echo "Level 5 compression"
./applications/bzip-reg/bzip2 -zv5 chicken.jpg
FILESIZE=$(stat -c%s "$CHICKENBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv chicken.jpg.bz2
FILESIZE=$(stat -c%s "$CHICKENJPG")
echo "End filesize: $FILESIZE"
echo "Level 7 compression"
./applications/bzip-reg/bzip2 -zv7 chicken.jpg
FILESIZE=$(stat -c%s "$CHICKENBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv chicken.jpg.bz2
FILESIZE=$(stat -c%s "$CHICKENJPG")
echo "End filesize: $FILESIZE"

```

```

echo "Level 9 compression"
./applications/bzip-reg/bzip2 -zv9 chicken.jpg
FILESIZE=$(stat -c%s "$CHICKENBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv chicken.jpg.bz2
FILESIZE=$(stat -c%s "$CHICKENJPG")
echo "End filesize: $FILESIZE"

# chicken.jpg thumb block
echo "chicken.jpg, thumb"
FILESIZE=$(stat -c%s "$CHICKENJPG")
echo "Starting filesize: $FILESIZE"
echo "Level 5 compression"
./applications/bzip-thumb/bzip2 -zv5 chicken.jpg
FILESIZE=$(stat -c%s "$CHICKENBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv chicken.jpg.bz2
FILESIZE=$(stat -c%s "$CHICKENJPG")
echo "End filesize: $FILESIZE"
echo "Level 7 compression"
./applications/bzip-thumb/bzip2 -zv7 chicken.jpg
FILESIZE=$(stat -c%s "$CHICKENBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv chicken.jpg.bz2
FILESIZE=$(stat -c%s "$CHICKENJPG")
echo "End filesize: $FILESIZE"
echo "Level 9 compression"
./applications/bzip-thumb/bzip2 -zv9 chicken.jpg
FILESIZE=$(stat -c%s "$CHICKENBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv chicken.jpg.bz2
FILESIZE=$(stat -c%s "$CHICKENJPG")
echo "End filesize: $FILESIZE"

# dryer.jpg block
echo "dryer.jpg, regular"
FILESIZE=$(stat -c%s "$DRYERJPG")
echo "Starting filesize: $FILESIZE"
echo "Level 5 compression"
./applications/bzip-reg/bzip2 -zv5 dryer.jpg
FILESIZE=$(stat -c%s "$DRYERBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv dryer.jpg.bz2
FILESIZE=$(stat -c%s "$DRYERJPG")
echo "End filesize: $FILESIZE"
echo "Level 7 compression"
./applications/bzip-reg/bzip2 -zv7 dryer.jpg
FILESIZE=$(stat -c%s "$DRYERBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv dryer.jpg.bz2
FILESIZE=$(stat -c%s "$DRYERJPG")
echo "End filesize: $FILESIZE"
echo "Level 9 compression"
./applications/bzip-reg/bzip2 -zv9 dryer.jpg

```

```

FILESIZE=$(stat -c%s "$DRYERBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv dryer.jpg.bz2
FILESIZE=$(stat -c%s "$DRYERJPG")
echo "End filesize: $FILESIZE"

# dryer.jpg thumb block
echo "dryer.jpg, thumb"
FILESIZE=$(stat -c%s "$DRYERJPG")
echo "Starting filesize: $FILESIZE"
echo "Level 5 compression"
./applications/bzip-thumb/bzip2 -zv5 dryer.jpg
FILESIZE=$(stat -c%s "$DRYERBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv dryer.jpg.bz2
FILESIZE=$(stat -c%s "$DRYERJPG")
echo "End filesize: $FILESIZE"
echo "Level 7 compression"
./applications/bzip-thumb/bzip2 -zv7 dryer.jpg
FILESIZE=$(stat -c%s "$DRYERBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv dryer.jpg.bz2
FILESIZE=$(stat -c%s "$DRYERJPG")
echo "End filesize: $FILESIZE"
echo "Level 9 compression"
./applications/bzip-thumb/bzip2 -zv9 dryer.jpg
FILESIZE=$(stat -c%s "$DRYERBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv dryer.jpg.bz2
FILESIZE=$(stat -c%s "$DRYERJPG")
echo "End filesize: $FILESIZE"

# liberty.jpg block
echo "liberty.jpg, regular"
FILESIZE=$(stat -c%s "$LIBERTYJPG")
echo "Starting filesize: $FILESIZE"
echo "Level 5 compression"
./applications/bzip-reg/bzip2 -zv5 liberty.jpg
FILESIZE=$(stat -c%s "$LIBERTYBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv liberty.jpg.bz2
FILESIZE=$(stat -c%s "$LIBERTYJPG")
echo "End filesize: $FILESIZE"
echo "Level 7 compression"
./applications/bzip-reg/bzip2 -zv7 liberty.jpg
FILESIZE=$(stat -c%s "$LIBERTYBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv liberty.jpg.bz2
FILESIZE=$(stat -c%s "$LIBERTYJPG")
echo "End filesize: $FILESIZE"
echo "Level 9 compression"
./applications/bzip-reg/bzip2 -zv9 liberty.jpg
FILESIZE=$(stat -c%s "$LIBERTYBZ")
echo "Compressed filesize: $FILESIZE"

```

```

./applications/bzip-reg/bzip2 -dv liberty.jpg.bz2
FILESIZE=$(stat -c%s "$LIBERTYJPG")
echo "End filesize: $FILESIZE"

# liberty.jpg thumb block
echo "liberty.jpg, thumb"
FILESIZE=$(stat -c%s "$LIBERTYJPG")
echo "Starting filesize: $FILESIZE"
echo "Level 5 compression"
./applications/bzip-thumb/bzip2 -zv5 liberty.jpg
FILESIZE=$(stat -c%s "$LIBERTYBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv liberty.jpg.bz2
FILESIZE=$(stat -c%s "$LIBERTYJPG")
echo "End filesize: $FILESIZE"
echo "Level 7 compression"
./applications/bzip-thumb/bzip2 -zv7 liberty.jpg
FILESIZE=$(stat -c%s "$LIBERTYBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv liberty.jpg.bz2
FILESIZE=$(stat -c%s "$LIBERTYJPG")
echo "End filesize: $FILESIZE"
echo "Level 9 compression"
./applications/bzip-thumb/bzip2 -zv9 liberty.jpg
FILESIZE=$(stat -c%s "$LIBERTYBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv liberty.jpg.bz2
FILESIZE=$(stat -c%s "$LIBERTYJPG")
echo "End filesize: $FILESIZE"

# input.source block
echo "input.source, regular"
FILESIZE=$(stat -c%s "$INPUT")
echo "Starting filesize: $FILESIZE"
echo "Level 5 compression"
./applications/bzip-reg/bzip2 -zv5 input.source
FILESIZE=$(stat -c%s "$INPUTBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv input.source.bz2
FILESIZE=$(stat -c%s "$INPUT")
echo "End filesize: $FILESIZE"
echo "Level 7 compression"
./applications/bzip-reg/bzip2 -zv7 input.source
FILESIZE=$(stat -c%s "$INPUTBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv input.source.bz2
FILESIZE=$(stat -c%s "$INPUT")
echo "End filesize: $FILESIZE"
echo "Level 9 compression"
./applications/bzip-reg/bzip2 -zv9 input.source
FILESIZE=$(stat -c%s "$INPUTBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv input.source.bz2
FILESIZE=$(stat -c%s "$INPUT")

```

```

echo "End filesize: $FILESIZE"

# input.source thumb block
echo "input.source, thumb"
FILESIZE=$(stat -c%s "$INPUT")
echo "Starting filesize: $FILESIZE"
echo "Level 5 compression"
./applications/bzip-thumb/bzip2 -zv5 input.source
FILESIZE=$(stat -c%s "$INPUTBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv input.source.bz2
FILESIZE=$(stat -c%s "$INPUT")
echo "End filesize: $FILESIZE"
echo "Level 7 compression"
./applications/bzip-thumb/bzip2 -zv7 input.source
FILESIZE=$(stat -c%s "$INPUTBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv input.source.bz2
FILESIZE=$(stat -c%s "$INPUT")
echo "End filesize: $FILESIZE"
echo "Level 9 compression"
./applications/bzip-thumb/bzip2 -zv9 input.source
FILESIZE=$(stat -c%s "$INPUTBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv input.source.bz2
FILESIZE=$(stat -c%s "$INPUT")
echo "End filesize: $FILESIZE"

# text.html block
echo "text.html, regular"
FILESIZE=$(stat -c%s "$TEXT")
echo "Starting filesize: $FILESIZE"
echo "Level 5 compression"
./applications/bzip-reg/bzip2 -zv5 text.html
FILESIZE=$(stat -c%s "$TEXTBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv text.html.bz2
FILESIZE=$(stat -c%s "$TEXT")
echo "End filesize: $FILESIZE"
echo "Level 7 compression"
./applications/bzip-reg/bzip2 -zv7 text.html
FILESIZE=$(stat -c%s "$TEXTBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv text.html.bz2
FILESIZE=$(stat -c%s "$TEXT")
echo "End filesize: $FILESIZE"
echo "Level 9 compression"
./applications/bzip-reg/bzip2 -zv9 text.html
FILESIZE=$(stat -c%s "$TEXTBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-reg/bzip2 -dv text.html.bz2
FILESIZE=$(stat -c%s "$TEXT")
echo "End filesize: $FILESIZE"

```

```

# text.html thumb block
echo "text.html, thumb"
FILESIZE=$(stat -c%s "$TEXT")
echo "Starting filesize: $FILESIZE"
echo "Level 5 compression"
./applications/bzip-thumb/bzip2 -zv5 text.html
FILESIZE=$(stat -c%s "$TEXTBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv text.html.bz2
FILESIZE=$(stat -c%s "$TEXT")
echo "End filesize: $FILESIZE"
echo "Level 7 compression"
./applications/bzip-thumb/bzip2 -zv7 text.html
FILESIZE=$(stat -c%s "$TEXTBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv text.html.bz2
FILESIZE=$(stat -c%s "$TEXT")
echo "End filesize: $FILESIZE"
echo "Level 9 compression"
./applications/bzip-thumb/bzip2 -zv9 text.html
FILESIZE=$(stat -c%s "$TEXTBZ")
echo "Compressed filesize: $FILESIZE"
./applications/bzip-thumb/bzip2 -dv text.html.bz2
FILESIZE=$(stat -c%s "$TEXT")
echo "End filesize: $FILESIZE"

exit 0

```

I. File resize script for industry standard tests

```

#!/bin/bash

#Build and truncate files
cat chicken.jpg chicken.jpg > 1
cat 1 1 > 2
cat 2 2 > 3
cat 3 3 > 4
cat 4 4 > 5
cat 5 5 > 6
truncate -s 31457280 6
rm chicken.jpg 1 2 3 4 5
mv 6 chicken.jpg

cat input.source input.source > 1
cat 1 1 > 2
cat 2 2 > 3
truncate -s 293601280 3
rm input.source 1 2
mv 3 input.source

cat liberty.jpg liberty.jpg > 1
cat 1 1 > 2
cat 2 2 > 3

```

```

cat 3 3 > 4
cat 4 4 > 5
cat 5 5 > 6
cat 6 6 > 7
truncate -s 31457280 7
rm liberty.jpg 1 2 3 4 5 6
mv 7 liberty.jpg

cat text.html text.html > 1
cat 1 1 > 2
cat 2 2 > 3
cat 3 3 > 4
cat 4 4 > 5
cat 5 5 > 6
cat 6 6 > 7
cat 7 7 > 8
cat 8 8 > 9
cat 9 9 > a
cat a a > b
cat b b > c
truncate -s 293601280 c
rm text.html 1 2 3 4 5 6 7 8 9 a b
mv c text.html

cat dryer.jpg dryer.jpg > 1
cat 1 1 > 2
cat 2 2 > 3
truncate -s 2097152 3
rm dryer.jpg 1 2
mv 3 dryer.jpg

cat byoudoin.jpg byoudoin.jpg > 1
cat 1 1 > 2
cat 2 2 > 3
truncate -s 5242880 3
rm byoudoin.jpg 1 2
mv 3 byoudoin.jpg

exit 0

```

III. Data tables for bzip execution times.

A. 300mb binary file

i. Regular application

300mb binary file					
Regular application					
Compression			Decompression		
Start	End	Duration	Start	End	Duration
1:08:10	1:34:12	0:26:02	1:34:12	1:45:34	0:11:22
1:45:34	2:11:38	0:26:04	2:11:38	2:23:16	0:11:38
2:23:16	2:49:20	0:26:04	2:49:20	3:00:48	0:11:28
3:00:48	3:26:53	0:26:05	3:26:53	3:38:38	0:11:45
3:38:38	4:04:42	0:26:04	4:04:42	4:16:21	0:11:39
4:16:21	4:42:24	0:26:03	4:42:24	4:54:06	0:11:42
4:54:06	5:20:15	0:26:09	5:20:15	5:31:54	0:11:39
5:31:54	5:58:02	0:26:08	5:58:02	6:09:39	0:11:37
6:09:39	6:35:48	0:26:09	6:35:48	6:47:21	0:11:33
6:47:21	7:13:33	0:26:12	7:13:33	7:25:12	0:11:39
7:25:12	7:51:15	0:26:03	7:51:15	8:02:52	0:11:37
8:02:52	8:28:56	0:26:04	8:28:56	8:40:33	0:11:37
8:40:33	9:06:42	0:26:09	9:06:42	9:18:18	0:11:36
9:18:18	9:44:22	0:26:04	9:44:22	9:55:58	0:11:36
9:55:58	10:22:05	0:26:07	10:22:05	10:33:39	0:11:34
	SUM	6:31:27		SUM	2:54:02
	AVG	0:26:06		AVG	0:11:36

ii. Thumb application

Thumb application					
Compression			Decompression		
Start	End	Duration	Start	End	Duration
10:33:39	11:03:48	0:30:09	11:03:48	11:18:20	0:14:32
11:18:20	11:48:25	0:30:05	11:48:25	12:03:00	0:14:35
12:03:00	12:33:02	0:30:02	12:33:02	12:47:43	0:14:41
12:47:43	13:17:36	0:29:53	13:17:36	13:32:09	0:14:33
13:32:09	14:02:13	0:30:04	14:02:13	14:16:39	0:14:26
14:16:39	14:46:54	0:30:15	14:46:54	15:01:33	0:14:39
15:01:33	15:31:33	0:30:00	15:31:33	15:46:06	0:14:33
15:46:06	16:16:29	0:30:23	16:16:29	16:31:14	0:14:45
16:31:14	17:01:23	0:30:09	17:01:23	17:15:56	0:14:33
17:15:56	17:45:56	0:30:00	17:45:56	18:00:36	0:14:40
18:00:36	18:30:38	0:30:02	18:30:38	18:45:15	0:14:37
18:45:15	19:15:21	0:30:06	19:15:21	19:30:00	0:14:39
19:30:00	20:00:11	0:30:11	20:00:11	20:15:32	0:15:21
20:15:32	20:45:52	0:30:20	20:45:52	21:00:42	0:14:50
21:00:42	21:30:36	0:29:54	21:30:36	21:45:10	0:14:34
	SUM	7:31:33		SUM	3:39:58
	AVG	0:30:06		AVG	0:14:40

iii. Results

Increase in execution time	
Compressio	15.3531741
Decompress	26.3934112

B. 10mb text file

i. Regular application

10mb text file								
Regular application								
Compression						Decompression		
Start	End	Duration				Start	End	Duration
0:30:27	0:34:36	0:04:09				0:34:36	0:34:43	0:00:07
0:34:43	0:38:51	0:04:08				0:38:51	0:38:58	0:00:07
0:38:58	0:43:05	0:04:07				0:43:05	0:43:12	0:00:07
0:43:12	0:47:19	0:04:07				0:47:19	0:47:26	0:00:07
0:47:26	0:51:33	0:04:07				0:51:33	0:51:40	0:00:07
0:51:40	0:55:47	0:04:07				0:55:47	0:55:54	0:00:07
0:55:54	1:00:01	0:04:07				1:00:01	1:00:08	0:00:07
1:00:08	1:04:15	0:04:07				1:04:15	1:04:22	0:00:07
1:04:22	1:08:29	0:04:07				1:08:29	1:08:36	0:00:07
1:08:36	1:12:43	0:04:07				1:12:43	1:12:50	0:00:07
1:12:50	1:16:57	0:04:07				1:16:57	1:17:04	0:00:07
1:17:04	1:21:11	0:04:07				1:21:11	1:21:18	0:00:07
1:21:18	1:25:25	0:04:07				1:25:25	1:25:32	0:00:07
1:25:32	1:29:39	0:04:07				1:29:39	1:29:46	0:00:07
1:29:46	1:33:53	0:04:07				1:33:53	1:34:00	0:00:07
	SUM	1:01:48					SUM	0:01:45
	AVG	0:04:07					AVG	0:00:07

ii. Thumb application

Compression			Decompression		
Start	End	Duration	Start	End	Duration
1:34:00	1:39:12	0:05:12	1:39:12	1:39:20	0:00:08
1:39:20	1:44:32	0:05:12	1:44:32	1:44:39	0:00:07
1:44:39	1:49:55	0:05:16	1:49:55	1:50:04	0:00:09
1:50:04	1:55:16	0:05:12	1:55:16	1:55:24	0:00:08
1:55:24	2:00:35	0:05:11	2:00:35	2:00:43	0:00:08
2:00:43	2:05:55	0:05:12	2:05:55	2:06:03	0:00:08
2:06:03	2:11:15	0:05:12	2:11:15	2:11:23	0:00:08
2:11:23	2:16:35	0:05:12	2:16:35	2:16:44	0:00:09
2:16:44	2:21:56	0:05:12	2:21:56	2:22:04	0:00:08
2:27:24	2:32:37	0:05:13	2:27:16	2:27:24	0:00:08
2:22:04	2:27:16	0:05:12	2:32:37	2:32:45	0:00:08
2:32:45	2:37:57	0:05:12	2:37:57	2:38:05	0:00:08
2:38:05	2:43:17	0:05:12	2:43:17	2:43:24	0:00:07
2:43:24	2:48:36	0:05:12	2:48:36	2:48:44	0:00:08
2:48:44	2:53:56	0:05:12	2:53:56	2:54:04	0:00:08
	SUM	1:18:04		SUM	0:02:00
	AVG	0:05:12		AVG	0:00:08

iii. Results

Increase in execution time	
Compression	26.3214671
Decompression	14.2857143

C. 100mb text file

i. Regular application

100mb text file						
Regular application						
Compression			Decompression			
Start	End	Duration	Start	End	Duration	
1:04:59	1:46:32	0:41:33	1:46:32	1:48:04	0:01:32	
1:48:05	2:29:35	0:41:30	2:29:35	2:31:07	0:01:32	
2:31:07	3:12:37	0:41:30	3:12:37	3:14:14	0:01:37	
3:14:14	3:55:42	0:41:28	3:55:42	3:57:13	0:01:31	
3:57:13	4:38:39	0:41:26	4:38:39	4:40:13	0:01:34	
4:40:13	5:21:38	0:41:25	5:21:38	5:23:07	0:01:29	
5:23:07	6:04:35	0:41:28	6:04:35	6:06:07	0:01:32	
6:06:07	6:47:39	0:41:32	6:47:39	6:49:09	0:01:30	
6:49:09	7:30:36	0:41:27	7:30:36	7:32:22	0:01:46	
7:32:22	8:13:49	0:41:27	8:13:49	8:15:23	0:01:34	
8:15:23	8:56:50	0:41:27	8:56:50	8:58:26	0:01:36	
8:58:26	9:39:52	0:41:26	9:39:52	9:41:26	0:01:34	
9:41:26	10:22:51	0:41:25	10:22:51	10:24:17	0:01:26	
10:24:17	11:05:44	0:41:27	11:05:44	11:07:16	0:01:32	
11:07:16	11:48:45	0:41:29	11:48:45	11:50:20	0:01:35	
	SUM	10:22:00		SUM	0:23:20	
	AVG	0:41:28		AVG	0:01:33	

ii. Thumb application

Compression			Decompression			
Start	End	Duration	Start	End	Duration	
1:34:00	1:39:12	0:05:12	1:39:12	1:39:20	0:00:08	
1:39:20	1:44:32	0:05:12	1:44:32	1:44:39	0:00:07	
1:44:39	1:49:55	0:05:16	1:49:55	1:50:04	0:00:09	
1:50:04	1:55:16	0:05:12	1:55:16	1:55:24	0:00:08	
1:55:24	2:00:35	0:05:11	2:00:35	2:00:43	0:00:08	
2:00:43	2:05:55	0:05:12	2:05:55	2:06:03	0:00:08	
2:06:03	2:11:15	0:05:12	2:11:15	2:11:23	0:00:08	
2:11:23	2:16:35	0:05:12	2:16:35	2:16:44	0:00:09	
2:16:44	2:21:56	0:05:12	2:21:56	2:22:04	0:00:08	
2:27:24	2:32:37	0:05:13	2:27:16	2:27:24	0:00:08	
2:22:04	2:27:16	0:05:12	2:32:37	2:32:45	0:00:08	
2:32:45	2:37:57	0:05:12	2:37:57	2:38:05	0:00:08	
2:38:05	2:43:17	0:05:12	2:43:17	2:43:24	0:00:07	
2:43:24	2:48:36	0:05:12	2:48:36	2:48:44	0:00:08	
2:48:44	2:53:56	0:05:12	2:53:56	2:54:04	0:00:08	
	SUM	1:18:04		SUM	0:02:00	
	AVG	0:05:12		AVG	0:00:08	

iii. Results

Increase in exeution time	
Compression	26.3451233
Decompression	7.28571429

D. 400mb text file

i. Regular application

400mb text file						
Regular application						
Compression			Decompression			
Start	End	Duration	Start	End	Duration	
0:28:10	3:14:32	2:46:22	3:14:32	3:20:36	0:06:04	
3:20:36	6:06:37	2:46:01	6:06:37	6:12:59	0:06:22	
6:12:59	8:58:52	2:45:53	8:58:52	9:04:47	0:05:55	
9:04:47	11:50:38	2:45:51	11:50:38	11:56:23	0:05:45	
11:56:23	14:42:12	2:45:49	14:42:12	14:48:15	0:06:03	
14:48:15	17:34:07	2:45:52	17:34:07	17:40:16	0:06:09	
17:40:16	20:26:08	2:45:52	20:26:08	20:32:21	0:06:13	
20:32:21	23:18:08	2:45:47	23:18:08	23:23:53	0:05:45	
23:23:53	2:09:38	2:45:45	2:09:38	2:15:34	0:05:56	
2:15:34	5:01:27	2:45:53	5:01:27	5:07:12	0:05:45	
5:07:12	7:53:06	2:45:54	7:53:06	7:58:54	0:05:48	
7:58:54	10:44:42	2:45:48	10:44:42	10:50:43	0:06:01	
10:50:43	13:36:30	2:45:47	13:36:30	13:42:18	0:05:48	
13:42:18	16:28:08	2:45:50	16:28:08	16:33:55	0:05:47	
16:33:55	19:19:48	2:45:53	19:19:48	19:25:35	0:05:47	
	SUM	17:28:17		SUM	1:29:08	
	AVG	2:45:53		AVG	0:05:57	

ii. Thumb application

Thumb application						
Compression			Decompression			
Start	End	Duration	Start	End	Duration	
19:25:35	22:55:11	3:29:36	22:55:11	23:01:52	0:06:41	
23:01:52	2:31:30	3:29:38	2:31:30	2:37:41	0:06:11	
2:37:41	6:07:19	3:29:38	6:07:19	6:13:45	0:06:26	
6:13:45	9:43:20	3:29:35	9:43:20	9:50:00	0:06:40	
9:50:00	13:19:41	3:29:41	13:19:41	13:25:59	0:06:18	
13:25:59	16:55:38	3:29:39	16:55:38	17:01:51	0:06:13	
17:01:51	20:31:27	3:29:36	20:31:27	20:37:48	0:06:21	
20:37:49	0:07:24	3:29:35	0:07:24	0:13:51	0:06:27	
0:13:51	3:43:25	3:29:34	3:43:25	3:49:54	0:06:29	
3:49:54	7:19:28	3:29:34	7:19:28	7:26:01	0:06:33	
7:26:01	10:55:38	3:29:37	10:55:38	11:01:53	0:06:15	
11:01:53	14:31:34	3:29:41	14:31:34	14:38:06	0:06:32	
0:56:34	4:27:18	3:30:44	4:27:18	4:33:40	0:06:22	
4:33:40	8:04:09	3:30:29	8:04:09	8:10:25	0:06:16	
8:10:25	11:40:37	3:30:12	11:40:37	11:47:15	0:06:38	
	SUM	52:26:49		SUM	1:36:22	
	AVG	3:29:47		AVG	0:06:25	

iii. Results

Increase in execution time	
Compression	26.4653677
Decompression	8.11518325

IV. Data tables for x264 execution times.

A. 230mb raw video file

230mb raw video file					
Regular application			Thumb application		
Start	End	Duration	Start	End	Duration
1:02:48	1:21:32	0:18:44	5:42:42	6:26:33	0:43:51
1:21:32	1:40:04	0:18:32	6:26:33	7:10:23	0:43:50
1:40:04	1:58:41	0:18:37	7:10:23	7:54:23	0:44:00
1:58:41	2:17:23	0:18:42	7:54:23	8:38:15	0:43:52
2:17:23	2:36:05	0:18:42	8:38:15	9:22:15	0:44:00
2:36:05	2:54:38	0:18:33	9:22:15	10:06:03	0:43:48
2:54:38	3:13:13	0:18:35	10:06:03	10:49:52	0:43:49
3:13:13	3:31:53	0:18:40	10:49:52	11:33:46	0:43:54
3:31:53	3:50:26	0:18:33	11:33:46	12:17:33	0:43:47
3:50:26	4:09:03	0:18:37	12:17:33	13:01:22	0:43:49
4:09:03	4:27:37	0:18:34	13:01:23	13:45:29	0:44:06
4:27:37	4:46:12	0:18:35	13:45:29	14:29:23	0:43:54
4:46:12	5:04:48	0:18:36	14:29:23	15:13:18	0:43:55
5:04:48	5:24:09	0:19:21	15:13:18	15:57:11	0:43:53
5:24:09	5:42:42	0:18:33	15:57:11	16:41:00	0:43:49
	SUM	4:39:54		SUM	10:58:17
	AVG	0:18:40		AVG	0:43:53
Increase in execution time					
Encoding	135.185185				

B. 780mb raw video file

780mb raw video file					
Regular application			Thumb application		
Start	End	Duration	Start	End	Duration
0:39:13	1:41:57	1:02:44	16:32:04	19:01:17	2:29:13
1:41:57	2:45:45	1:03:48	19:01:17	21:30:23	2:29:06
2:45:45	3:48:16	1:02:31	21:30:23	23:59:48	2:29:25
3:48:16	4:50:47	1:02:31	23:59:48	2:29:39	2:29:51
4:50:47	5:58:08	1:07:21	2:29:39	4:58:34	2:28:55
5:58:08	7:03:15	1:05:07	4:58:34	7:28:53	2:30:19
7:03:15	8:06:16	1:03:01	7:28:53	9:57:57	2:29:04
8:06:16	9:08:48	1:02:32	9:57:58	12:27:20	2:29:22
9:08:48	10:12:25	1:03:37	12:27:20	14:56:25	2:29:05
10:12:25	11:17:13	1:04:48	14:56:25	17:25:34	2:29:09
11:17:13	12:20:00	1:02:47	17:25:34	19:56:04	2:30:30
12:20:00	13:23:07	1:03:07	19:56:04	22:25:52	2:29:48
13:23:07	14:26:05	1:02:58	22:25:52	0:54:59	2:29:07
14:26:05	15:28:58	1:02:53	0:54:59	3:24:21	2:29:22
15:28:59	16:32:04	1:03:05	3:24:22	5:53:50	2:29:28
	SUM	15:52:50		SUM	37:21:44
	AVG	1:03:31		AVG	2:29:27
Increase in execution time					
Encoding	135.270247				

C. 1.15gb raw video file

End filesize: 5242880

chicken.jpg, regular

Starting filesize: 31457280

Level 5 compression

Compressed filesize: 31306985

End filesize: 31457280

Level 7 compression

Compressed filesize: 30427899

End filesize: 31457280

Level 9 compression

Compressed filesize: 25949409

End filesize: 31457280

chicken.jpg, thumb

Starting filesize: 31457280

Level 5 compression

Compressed filesize: 31306985

End filesize: 31457280

Level 7 compression

Compressed filesize: 30427899

End filesize: 31457280

Level 9 compression

Compressed filesize: 25949409

End filesize: 31457280

dryer.jpg, regular

Starting filesize: 2097152

Level 5 compression

Compressed filesize: 1434910

End filesize: 2097152

Level 7 compression

Compressed filesize: 1127895

End filesize: 2097152

Level 9 compression

Compressed filesize: 1067335

End filesize: 2097152

dryer.jpg, thumb

Starting filesize: 2097152

Level 5 compression

Compressed filesize: 1434910

End filesize: 2097152

Level 7 compression

Compressed filesize: 1127895

End filesize: 2097152

Level 9 compression
Compressed filesize: 1067335
End filesize: 2097152

liberty.jpg, regular
Starting filesize: 31457280
Level 5 compression
Compressed filesize: 21917976
End filesize: 31457280
Level 7 compression
Compressed filesize: 17697398
End filesize: 31457280
Level 9 compression
Compressed filesize: 13755071
End filesize: 31457280

liberty.jpg, thumb
Starting filesize: 31457280
Level 5 compression
Compressed filesize: 21917976
End filesize: 31457280
Level 7 compression
Compressed filesize: 17697398
End filesize: 31457280
Level 9 compression
Compressed filesize: 13755071
End filesize: 31457280

input.source, regular
Starting filesize: 293601280
Level 5 compression
Compressed filesize: 54403299
End filesize: 293601280
Level 7 compression
Compressed filesize: 53599136
End filesize: 293601280
Level 9 compression
Compressed filesize: 52960821
End filesize: 293601280

input.source, thumb
Starting filesize: 293601280
Level 5 compression
Compressed filesize: 54403299
End filesize: 293601280
Level 7 compression

Compressed filesize: 53599136
End filesize: 293601280
Level 9 compression
Compressed filesize: 52960821
End filesize: 293601280

text.html, regular
Starting filesize: 293601280
Level 5 compression
Compressed filesize: 22290700
End filesize: 293601280
Level 7 compression
Compressed filesize: 17811990
End filesize: 293601280
Level 9 compression
Compressed filesize: 14427351
End filesize: 293601280

text.html, thumb
Starting filesize: 293601280
Level 5 compression
Compressed filesize: 22290700
End filesize: 293601280
Level 7 compression
Compressed filesize: 17811990
End filesize: 293601280
Level 9 compression
Compressed filesize: 14427351
End filesize: 293601280

VI. Percentage formula

$$\frac{|\text{Actual} - \text{Experimental}|}{\text{Actual}} * 100$$

Actual - pertaining to the standard binary for the given application

Experimental – pertaining to the Thumb mode binary for the given application

Works Cited

- Arm.com. "ARM Limited Reference Card." *ARM Limited*. Arm.com. Web. 15 July 2010.
<http://infocenter.arm.com/help/topic/com.arm.doc.qrc00011/QRC0001_UAL.pdf>.
- Beagleboard.org. "BeagleBoard System Reference Manual." *BeagleBoard*. Beagleboard.org. Web. 15 July 2010. <http://beagleboard.org/static/BBSRM_latest.pdf>.
- Ebner, Dietmar, Bernhard Scholz, and Andreas Krall. "Progressive Spill Code Placement." *CASES '09 Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (2009). ACM. Web. 15 July 2010.
<<http://portal.acm.org/citation.cfm?id=1629408>>.
- Krishnaswamy, Arvind, and Rajiv Gupta. "Mixed-width Instruction Sets." *Communications of the ACM - Program Compaction* 46.8 (2003). ACM. Web. 15 July 2010.
<<http://portal.acm.org/citation.cfm?id=859697>>.
- Lee, Sheayun, Jaejin Lee, Chang Yun Park, and Sang Lyul Min. "Selective Code Transformation for Dual Instruction Set Processors." *ACM Transactions on Embedded Computing Systems (TECS)* 6.2 (2007). ACM. Web. 15 July 2010.
<<http://portal.acm.org/citation.cfm?id=1234675.1234677>>.
- Xianhua, Liu, Zhang Jiyu, and Cheng Xu. "Efficient Code Size Reduction without Performance Loss." *SAC '07 Proceedings of the 2007 ACM Symposium on Applied Computing*(2007). ACM. Web. 15 July 2010. <<http://portal.acm.org/citation.cfm?id=1244002.1244154>>.
- Yang, Fu-Ching, and Ing-Jer Huang. "An Embedded Low Power/Cost 16-Bit Data/Instruction Microprocessor Compatible with ARM7 Software Tools." *ASP-DAC '07 Proceedings of the 2007 Asia and South Pacific Design Automation Conference* (2007). ACM. Web. 15 July 2010.
<<http://portal.acm.org/citation.cfm?id=1323397>>.