MINIX 3
C Compiler Performance

Comparing the Amsterdam Compiler Kit to the GNU Compiler Collection on x86 systems

Feisal S. Ahmad fsahmad@few.vu.nl

9-6-2008

Table of Contents

1. Abstract 5

2. Introduction 5

3. Measurement set-up 5

Operating system 5

Compilers 5

Optimization options 6

Measurement methodology 7

Program selection 8

Test machines 9

4. Compilation 9

Compilation time factors 9

Program compilation time 11

5. Executable segment sizes 15

Results 15

6. Execution 17

Used benchmarks 17

Benchmark set 1 results 19

Benchmark set 2 results 23

Architecture optimization performance 25

7. Conclusion 26

8. References 27

1. Abstract

For years the default compilation toolchain of MINIX was the Amsterdam Compiler Kit. New to MINIX 3 is the port of the GNU Compiler Collection which brings an alternative toolchain, which is actively developed and enjoys a large user base. In this paper a comparison of C compiler performance is presented, focusing on three aspects of performance; compilation speed, executable segment sizes, and execution speed. It is shown that compilation speed is several orders of magnitude slower for with gcc compared to ACK's cc. Segment sizes vary with gcc generating larger segments on average, but with some programs which have smaller segment sizes in gcc. Execution speed is shown to be significantly faster with gcc in almost all cases, especially for CPU bound processes.

2. Introduction

With the release of MINIX 3, a number of applications that previously were unavailable to MINIX have been ported from other operating systems, like Linux. One of these ported applications was the GNU Compiler Collection. In the past, the main compiler which was available for MINIX was the Amsterdam Compiler Kit, originally written by Andrew S. Tanenbaum and Ceriel J.H. Jacobs. Because this compiler originated in the 1980s, and the continuous development and wide use of GCC, it is interesting to compare these compilers on MINIX. For this paper, the aspects that have been investigated and compared are the C compiler's performance in three areas; compilation speed, executable segment sizes, and execution speed. Other aspects like code checking features and debugging functionality have not been taken into consideration, to limit the scope of the paper.

The outline of this paper is as follows; the first section explains the measurement set-u, the three following sections deal with the measured aspects of performance, and finally the conclusions are presented.

3. Measurement set-up

This section describes what tools were used to measure the compilers performance in each field, and why these were selected. Also how the measurements were taken is described.

Operating system

All measurements were done on MINIX version 3.1.2a, with all binaries from the CD installed. The measurements were all run as user root, this is necessary because the scripts need to run the chmem command on various compiler components.

Compilers

For these measurements, the c compiler of ACK, cc, was compared to the c compiler of the GNU Compiler Collection, gcc. Two versions of gcc were used, version 3.4.3 and version 4.1.1. Because compilation fails for some of the selected programs when using high optimization settings, various components of the compiler needed more allocated memory. The following memory sizes were set with chmem to prevent compiling from failing;

Table 1 - Compiler component sizes

Compiler	File	Stack + malloc size
cc	`/usr/lib/ego/bo`	2 MB
	`/usr/lib/ego/ra`	5 MB
	`/usr/lib/ego/cs`	2 MB
	`/usr/lib/ego/cf`	2 MB
	`/usr/lib/ego/sp`	2 MB
	`/usr/lib/ego/sr`	2 MB
	`/usr/lib/ego/ud`	5 MB
	`/usr/lib/ego/lv`	5 MB
	`/usr/lib/em_opt`	512 kB
	`/usr/lib/em_opt2`	512 kB
	`/usr/lib/em_cemcom.ansi`	10 MB
gcc	`/usr/gnu/i386-pc-minix/bin/ld`	6 MB
	`/usr/gnu/libexec/gcc/i386-pc-minix/3.4.3/collect2`	6 MB

Optimization options

To make a selection in optimization options (or flags), research in the possibilities was necessary. For cc, the man-page tells us that optimization options are as follows;

Table 2 - ACK cc optimization options

-O	Optimize code. This option is a no-op, because all the compilers already use the -O1 optimization level to get code of reasonable quality. Use -O0 to turn off optimization to speed up compilation at debug time.
-Olevel	Compile with the given optimization level. (MINIX 3)
-OS **-OT**	Optimize for space or for time. (MINIX 3)

During testing I noticed that -O0 and -O1 do not seem to differ in practice. Both options output identical executables, and compilation time is the same for both settings. For higher settings, -O3 seems to be the highest setting, since higher settings produce the same executable for the tested programs. The next higher setting -O4 does seem to use different options for optimization, since compilation failed with an out of memory error, where -O3 compiled without error. The options -OS and -OT are not supported, cc prints an error message.

In contrast to cc, gcc provides a lot of specific optimization options (1) (2), providing fine grained control of the optimization used. Manually selecting these options requires a great deal of knowledge about compiler optimization strategies and their effectiveness on different types of code. Gcc also provides presets in the same form as cc with -O flags, namely -O0, -O1 (or -O), -O2, -O3 and -Os. These presets cannot be combined, only the last specified preset will be used, so it is not possible to combine for example -O2 and -Os. For a complete description of the options used in each preset, and what the options do, see (1) and (2).

In addition to these CPU independent optimization options, gcc also provides options to optimize the code for a specific CPU architecture (3). These options attempt to produce code that tries to tune the code (-mtune, -march) or allow the compiler to make use of special instructions available on the specific CPU architecture (-march). Using these settings can have consequences for the produced executable. The

-mtune setting produces code that does not use any CPU architecture specific instructions, so the produced executable can still run on other architectures. The -march setting makes CPU architecture specific instructions available to the compiler, and therefore can produce executables that fail to run on other architectures.

The following optimization options were used in the measurements;

Table 3 - Selected optimization options

Compiler	Optimization parameters

cc	`-O1 (default)`
	`-O2`
	`-O3`

gcc	`-O0 (default)`
	`-O1`
	`-O2`
	`-O2 -march=architecture`
	`-O2 -mtune=architecture`
	`-O3`
	`-O3 -march=architecture`
	`-O3 -mtune=architecture`
	`-Os`
	`-Os -mtune=architecture`
	`-Os -march=architecture`

For cc, -O0 was omitted because it does not behave differently from -O1 which is the default setting. Optimization options higher than -O3 were omitted as well because the produced executables did not differ from the -O3 setting. For gcc, all options were used, and architecture options were used for -O2, -O3 and -Os. To limit the duration of the tests, architecture options were not used for the other settings. This selection was made because these settings do not attempt to optimize a lot.

Measurement methodology

Three aspects of compiler performance were measured; compilation speed, segment sizes and the execution speed of the compiled program.

Used programs

The speed measurements were done using a modified version of the time program found on MINIX. The original time program runs the specified command once, and when the command is finished, it prints the real, user, and system time in seconds rounded to two decimals. The modified program, called ccb_time[1], times in clock ticks instead of seconds. Ccb_time performs timing in such a way that the specified command is run once, and is then looped for a number of times. By executing the command once before starting the loop, the command can take advantage of the file system cache in subsequent executions, thereby minimizing the effect of the disk speed on the timing. The command is looped for a minimum of N times (N is passed as parameter), and if after N times the real, user and system times are below 60 ticks, the loop continues to a maximum of 50 loops. For completely CPU bound commands, the 60 tick limit is optional for the system time to prevent unnecessary long loops. The output of ccb_time is appended in a comma separated format to a specified file.

Size measurements were performed using a modified version of the size command called ccb_size. It appends its output to a specified file, like ccb_time, instead of printing to standard output.

[1] ccb for c compiler benchmark

Compilation speed measurement

The compilation speed was measured for each combination of compiler and optimization option parameters. To measure this, the selected programs were set up in such a way that the Makefile of each program uses the environment variables CCB_CC and CCB_OFLAGS for the compiler command and the optimization parameters respectively. For CCB_CC the ccb_time command combined with the compiler command was set like the following example;

```
        /path/to/ccb_time/ccb_time 5 output_file “gcc” "-O3 -march=pentium3"
```

After the Makefile has been processed, the total time can be extracted by adding the results from the output file to give a total time spent by the compiler program. This means that the overhead of the make program, and other compilation related commands like ar (archiver) are not measured.

Segment size measurement

After compilation, the segment sizes can be easily measured by using the ccb_size program on the produced executable. This program measures the text, data, BSS, stack and memory sizes, but only the text, data and BSS segments are of interest to this test since stack and memory sizes can be adjusted by the chmem program.

Executable performance

Measuring executable performance has been done in a straight forward way. For each selected programs a test set was made which lets the program do some work which takes a relatively long time to complete. To measure the time these tasks take, the ccb_time program was used. The details of the test set for each program is explained in section 1.

Program selection

To be able to measure the performance of the compilers with the optimization options, a selection of programs was needed. These programs needed to compile without errors with both cc and gcc, and needed to be able to run without interactive input to make automated testing possible.

The following programs were used in the measurements;

- awk
- bc 1.02
- bzip2 1.03
- gzip 1.2.4
- python 1.5.2
- sed
- sorting algorithms (bubblesort, insertionsort, selectionsort and quicksort)
- whetstone

Awk accepts input instructions to usually transform input files or perform calculations on an input file. It can also be used to perform calculations without input. Bc is an arbitrary precision calculator program, which works interactively or uses input scripts. Bzip2 and gzip are lossless compression programs. Python is an interpreter program for the general-purpose, high-level programming language Python. Sed is a stream editor program which accepts regular expression commands to transform input streams. The sorting algorithms are implementations of different sorting algorithms in C (4), which are executed on randomly initialized arrays. Whetstone (5) (6) is a synthetic benchmark which was published in ALGOL in 1976. It was designed to provide a benchmark with which the performance of new hardware could be estimated for general programs. It runs for a specified amount of loops, and the time spent to perform these loops is used as a measure of performance. The test consists of 11 weighted modules, each measuring a certain aspect of performance.

To test some specific characteristics of compilation speed, two types of code were tested in the compilation speed part only. The effects of identifier length and function size have been tested using code files generated for different amounts of lines. To test the effect of identifier length, code files were generated only containing repeated assignments to zero of the same variable. The length of this variable name was set to a single character for the short identifier test, and to 624 for the long identifier test. To test the effect of function size, code files were generated by repeating the code for the insertion sort algorithm in one big function and in separate functions for the big function and small function cases respectively.

Test machines

The complete test set was run on two different computers, the specifications are listed below.

Table 4 - Test machines specifications

	Machine 1	Machine 2
CPU	Intel Pentium II	Intel Pentium III EB
Clock frequency	350 MHz	1 GHz
Memory	192 MB (3x64MB)	512 MB (2x256MB PC133)
Motherboard	Intel SE440BX-2	Gigabyte 6VX7-4X
Hard drive	Quantum Fireball CR4.3A (4.3 GB)	Maxtor D740X-6L (40GB)
Whetstone score (built with gcc 3.4.3, no optimization)	4.0 MIPS	11.6 MIPS

Machine 1 will be referred to as Pentium 2 350MHz, and machine 2 as Pentium 3 1GHz from now on.

4. Compilation

The results of the compilation speed test are presented and discussed in this section. First the separate tests of compiler behaviour are discussed, and then the compilation times of the tested programs are discussed.

Compilation time factors

To analyze how the compilation time is affected by several factors, several tests were conducted to measure these effects. To test the influence of the input file size, several code fragments were repeated to generate code files of varying sizes, which were then compiled as object files to exclude the linker from measurement. The code fragments used were three different types; integer assignment with a short identifier, integer assignment with a long identifier, and the insertion sort algorithm. These fragments are listed in the appendix, code listings 1, 2 and 3. The bold sections represent the part which is repeated, and the identifier in listing 1 is replaced with `i` for the short identifier, and 24 times the alphabet (624 characters) for the long identifier version. In listing 3, the `%%` part of the function identifier is replaced by a number for each repeat.

Figures 5 and 6 in the appendix show the results of cc for the integer assignment test. Because cc -O3 is much slower than the other presets for a high number of lines, the results are shown separately in figure 6 as well. Trend lines (generated by Microsoft Excel 2007) are shown in these figures, which are also used in other figures for this section. Figure 5 shows a linear relation for both -O1 and -O2 with the short identifier, and -O1 with the long identifier. -O2 with the long identifier shows a polynomial relation. Figure 6 shows a polynomial relation for -O3. Important to note here, is that this test compiles large functions instead of a lot of small functions. As will be seen further on, this influences the behaviour of the compiler in this aspect of performance. Comparing -O1 and -O2, it is clear that -O2 compiles slower than -O1, which is as expected. The compilation time of -O3 is extremely large compared to the other presets, especially for a large number of lines, but this is entirely caused by the polynomial relation this presets shows.

Figure 7 shows the results of gcc 3.4.3 for the same test. Compared to cc, gcc does not have a preset which shows a different relation than the other presets, only the coefficient of the lines differs. At first sight, the trend lines look linear, but in fact they are polynomial. The presets -O2 -O3 and -Os show the same times, with the exception of a peak for -O2 and -O3 at 3000 lines in the short identifier case. Why this peak is only present for these presets is not clear, perhaps some sort of caching in gcc causes this peak for this specific number of lines. In this case, compilation time is highest for the preset -O0 which is quite unexpected since this is supposed to give the lowest compilation time. Other than these cases, the results are as expected, and again there is a difference between the short identifier and the long identifier with the short identifier compiling faster.

The last figure concerning compilation user time for the integer assignment code is figure 8, which shows the results of gcc 4.1.1. Compared to version 3.4.3, the presets show the same ordering in compilation time, only the -O1 -O2 -O3 and -Os show a stronger polynomial relation. For a lower number of lines these presets use slightly less compilation time in version 4.1.1 than in version 3.4.3, but for a high number of lines the opposite is true. The preset -O0 uses less compilation time in version 4.1.1 than in version 3.4.3.

Figures 9, 10, and 11 in the appendix contain the results of the insertion sort compilation test for cc, gcc 3.4.3 and gcc 4.1.1 respectively. The figures combine the results of the separate functions and combined function code. For the combined function code, cc shows a polynomial relation for each preset. The separate functions code however, shows a linear relation for each preset. These results show that for cc, the time complexity of function compilation has an order of n^2 where n is the function size. The linear behaviour for the separate functions code proves that this order of n^2 is caused by the function size, and not the total input size. In the case of the integer assignment, this time complexity is only clearly shown for -O3. This is probably caused by the trivial code, in contrast to the insertion sort algorithm.

The time complexity of gcc has an order of n^2 for both the separate functions code and the combined function code, for every preset. The difference between the two types of code is only that the combined function code compiles faster than the separate function code. Compilation time goes from lowest to highest for both versions of gcc in the following order of presets; -O0, -O1, -Os, -O2, -O3. Comparing version 3.4.3 to 4.1.1 shows that the -O0 preset compiles in roughly the same time in version 4.1.1, but the other presets need more time than in version 3.4.3.

So far in this section, only the user time of the compiler has been studied. This only takes into account the amount of processing time the compiler, which is not the only factor which determines how much time a compiler needs to compile a complete program. Another factor is how long loading and initialization of the compiler takes before it starts to compile. To analyze this, the real times for the compilation tests above can be analyzed. These results are shown in figures 12-17 in the appendix. Using extrapolation, the value for 0 lines of code can be approximated for each compiler. For cc, the different presets show different values when extrapolating the lines, with a rough estimation of 7 ticks (~117 msec.) for -O1, 12 ticks (~200 msec.) for -O2, and 20 ticks (~333 msec.) for -O3. This difference can be explained by the way the cc compiler is constructed. For optimization, it uses sub processes like `em_opt` and `em_opt2`. Since these programs need to be loaded separately, depending on what optimization preset is used, the startup times vary for each preset. For gcc, every preset has the same startup time, roughly 40 ticks (~667 msec.) for version 3.4.3, and roughly 115 ticks (~1.92 sec.) for version 4.1.1. These startup times are higher than those of cc, especially for gcc 4.1.1 the startup time is much higher. High startup times become a large factor in total compilation times for build processes which make many calls to the compiler for small input files. In these types of build processes, the startup time is a very large part of each compiler call because the user time is low for small input files.

Program compilation time

The absolute compilation times of the selected programs for both test systems are shown in figures 1 and 2 of the appendix. Programs are sorted by number of compiler executions from left to right, high to low. The times are divided in three sections, real, user and system. The top rows indicate the program, the number of times the compiler was started during the build, and the total size of the source files, code and headers in kilobytes. The cases where compilation failed are empty cells. Python, gzip and awk all failed in the case of cc -O3, where the compiler returns the following type of error message;

```
/usr/lib/em_opt2: error on line 766: This is not allowed outside a
procedure
```

The em_opt2 program handles optimization of the intermediate EM code (7), so the error statement has to do with the code produced by the C front-end and the optimizations already performed on that code. Investigating and fixing the problem was outside the scope of this test, so the results for cc -O3 are incomplete. Awk failed to compile under gcc 4.1.1 because the behavior of this version is different from gcc 3.4.3. Version 4.1.1 fails with an error message (static declaration of [identifier] follows non-static declaration) where gcc 3.4.3 does not. Because the code needed more than some simple fixes to fix this error, and the code does compile in version 3.4.3, the code was left as it was.

From the absolute compilation times, the user/real ratios can be calculated. This ratio indicates how CPU bound or I/O bound a process is, where a high ratio is CPU bound, and low I/O bound. The results from the Pentium 2 system are shown in table 5, sorted by the ratio of the total source size and number of compiler executions.

This table shows that the compiler is less CPU bound when the ratio of size versus executions is low, because in this situation, the time spent loading the compiler (which does not count for user time) is a larger factor for the total time.

The lowest ratios are found for gcc -O0. This is explained by the lack of optimization and the longer startup time for the gcc compiler. Comparing the gcc versions shows that version 4.1.1 is even less CPU bound than 3.4.3, which can be explained by the higher startup time for version 4.1.1.

The gcc compiler becomes more CPU bound for higher optimization parameters, which can be expected since the compiler needs to do more processing for the same file. Cc however behaves in an opposite way, becoming less CPU bound at higher settings, with the exception of bzip2 with -O3 parameter. The user time does increase, so more processing is being performed as expected, however the compiler seems to do more I/O as well.

In order to compare compilation times independently from program size, figures 3 and 4 of the appendix show the times indexed to the default case of cc -O1. The formula used for this index is the time divided by the time of cc -O1, times 100. A value of 100 is thus equal to the time of cc -O1, 50 is half, 200 double, and so on. With these values, an average score is calculated for each case. This average excludes the values for awk, since this failed to compile in gcc 4.1.1. Cc -O3 does not have an average because two values are missing, so the result would not be comparable to the rest.

Table 5 - Compilation user/real ratio (Pentium 2)

		sed	bzip2	python	gzip	bc	awk	whet-stone	sorting	AVERAGE excl. awk
Total size / compiler executions (kB)		45.04	21.70	19.51	16.69	16.18	10.43	7.93	1.97	n/a
Compiler	**Optimization parameters**									
cc	-O1	76.87%	80.72%	67.64%	58.25%	60.99%	58.50%	54.14%	34.82%	**61.92%**
	-O2	69.39%	73.19%	59.07%	48.35%	53.81%	50.22%	48.25%	24.29%	**53.76%**
	-O3	67.61%	76.51%			46.23%		39.56%	19.22%	
gcc 3.4.3	-O0	33.37%	54.32%	44.58%	31.06%	36.80%	36.56%	17.83%	7.45%	**32.20%**
	-O1	49.21%	69.02%	56.61%	42.37%	47.44%	48.64%	26.10%	9.05%	**42.83%**
	-O2	66.03%	82.10%	69.77%	55.52%	60.23%	62.17%	37.31%	11.38%	**54.62%**
	-O2 -march=pentium2	66.69%	82.08%	69.81%	56.30%	60.47%	62.99%	39.00%	11.28%	**55.09%**
	-O2 -mtune=pentium2	66.69%	82.13%	69.83%	56.06%	60.48%	62.66%	38.49%	11.23%	**54.99%**
	-O3	72.64%	85.02%	76.82%	60.27%	71.42%	75.92%	41.35%	12.72%	**60.03%**
	-O3 -march=pentium2	73.17%	85.19%	76.72%	61.20%	71.08%	76.09%	43.14%	13.46%	**60.57%**
	-O3 -mtune=pentium2	72.85%	85.17%	76.87%	61.07%	71.52%	76.03%	42.87%	13.11%	**60.50%**
	-Os	64.84%	80.31%	67.83%	52.83%	57.39%	59.50%	34.24%	11.13%	**52.65%**
	-Os -march=pentium2	65.65%	80.73%	68.27%	53.56%	57.73%	60.86%	35.12%	10.86%	**53.13%**
	-Os -mtune=pentium2	66.35%	80.56%	68.38%	53.58%	57.78%	61.00%	35.64%	11.17%	**53.35%**
gcc 4.1.1	-O0	29.47%	31.28%	20.71%	14.24%	17.42%		16.43%	5.72%	**19.32%**
	-O1	56.90%	58.62%	38.49%	29.80%	31.21%		24.08%	8.54%	**35.38%**
	-O2	65.95%	67.75%	46.37%	36.68%	38.76%		28.94%	10.75%	**42.17%**
	-O2 -march=pentium2	67.38%	69.02%	47.33%	37.66%	39.50%		33.03%	11.12%	**43.58%**
	-O2 -mtune=pentium2	66.80%	68.43%	47.17%	37.82%	39.26%		33.03%	10.96%	**43.35%**
	-O3	67.93%	71.76%	53.08%	41.93%	51.16%		38.61%	12.87%	**48.19%**
	-O3 -march=pentium2	69.28%	72.70%	53.81%	43.09%	51.94%		48.72%	13.14%	**50.38%**
	-O3 -mtune=pentium2	69.67%	72.24%	53.84%	43.03%	51.65%		49.04%	13.04%	**50.36%**
	-Os	64.03%	58.21%	43.57%	34.26%	35.88%		25.40%	9.75%	**38.73%**
	-Os -march=pentium2	64.92%	59.84%	44.91%	35.37%	37.00%		29.05%	10.28%	**40.20%**
	-Os -mtune=pentium2	65.28%	59.11%	44.86%	35.39%	37.07%		27.92%	10.10%	**39.96%**

Used color scale	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%

The table shows that gcc is slower than cc for every combination of program and optimization parameters, only scoring better than cc -O3 for bzip2 with gcc 3.4.3 -O0. For stronger optimization parameters, gcc is severely slower than cc, with increases of a factor 10 and more for several programs. Cc shows an increase in real compilation time of about 50% for -O2 on the Pentium 2 system, and about 60% on the Pentium 3. In the cases where compilation with -O3 succeeded, increases of over 100% are seen on both systems, with the Pentium 3 showing slightly larger increases. Gcc shows more variation for the different programs, not only relative to cc, but also relative to its own optimization parameters. For example, on the Pentium2, compiling gzip takes 74.1% longer with -O3 than with -O0, but for bzip2 this increase is 206.5%. The sorting program shows almost no difference for each of the optimization settings, but this can be explained by the relatively long startup time of gcc, combined with the very small input size per execution (see table 5). Looking at the average values, gcc is a lot slower than cc, even when no optimization is applied. This can only partly be attributed to the longer startup time, since the user times show a big increase as well.

With the average values, an ordering can be made to show how the different compilers and optimization parameters perform on average. The results sorted by real values are shown in figure 1 and figure 2 below.

Comparing the results for the two systems, the ordering hardly differs. Only the positions 11 through 14 are reordered, and positions 23 and 24 are switched. The averages show that the cc compiler is fastest, followed by version 3.4.3 of gcc, and gcc 4.1.1 is slowest. Interesting is that gcc 4.1.1 is even slower without optimization options than all the other compiler / optimization parameter combinations. Both versions of gcc clearly show that compilation time increases for stronger optimization parameters -O1

-O2 -O3. The size optimization parameter (-Os) takes just a little shorter to compile than -O2. Gcc's architecture specific parameters increase compilation time slightly in every case.

In the system values, cc shows an increase for -O2, yet for gcc the values are the same for every tested optimization parameter. The figures show that the average system time for version 3.4.3 is a lot higher than version 4.1.1. Looking back at tables XXX and YYY, this difference only occurs for the programs on the left of awk. These programs have a much larger total source size than the programs to the right of awk which can be seen in the row "Total source size". Another difference between these sets of programs is that the larger programs consist of multiple .c and .h files. Somehow version 3.4.3 uses more system time to process these files.

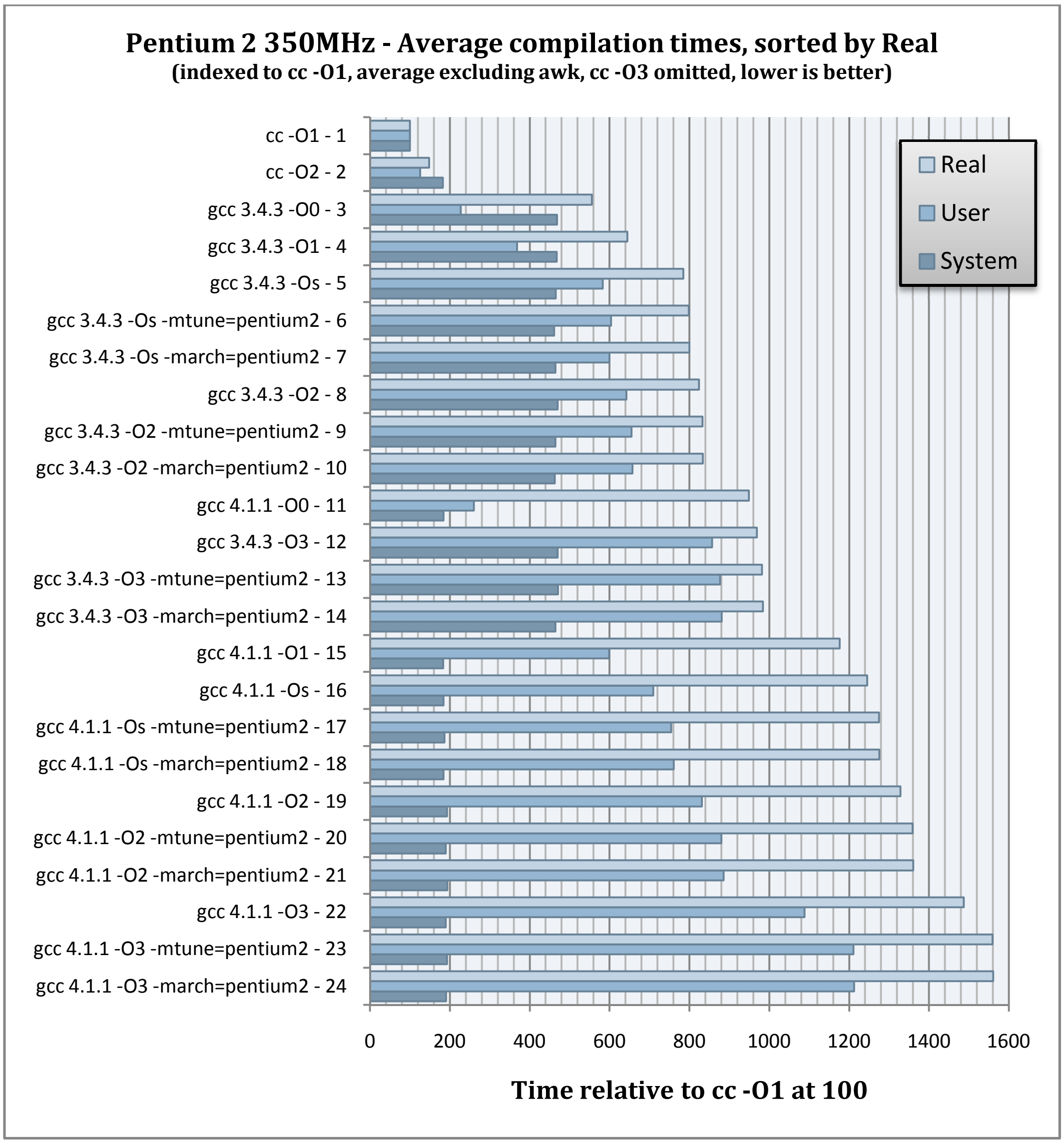

Figure 1 - Pentium 2 Average compilation times

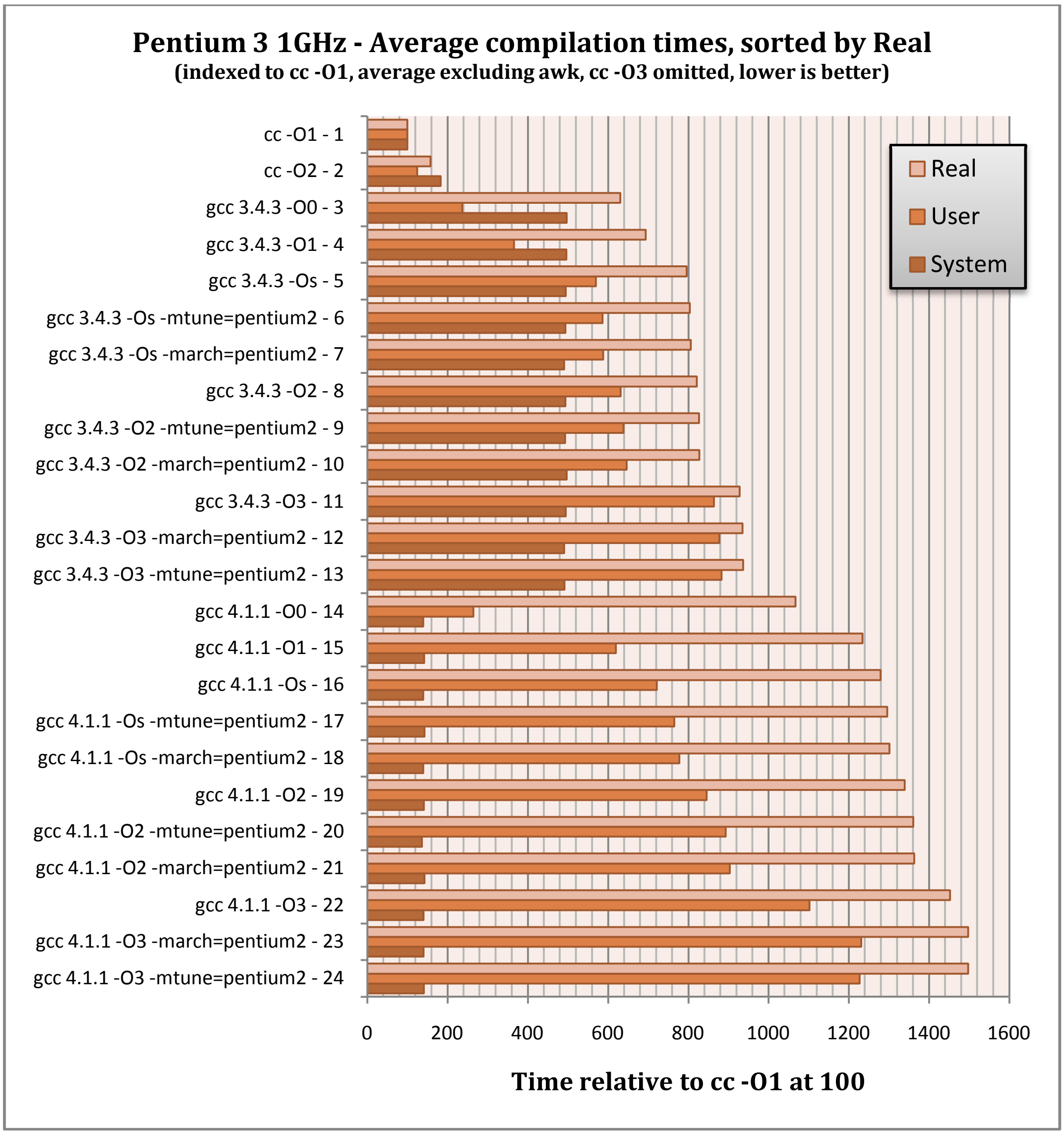

Figure 2 - Pentium 3 Average compilation times

5. Executable segment sizes

The second part of compiler performance measured is the segment sizes of the produced executables. As explained in section 1 - Segment size measurement, only the Text, Data and BSS segment are of interest. Because the Text segment contains the instructions, it is most interesting when looking from an optimization perspective, both for performance and for size. The Data segment and BSS segments contain the initialized and un-initialized global variables respectively.

Results

The results are presented as absolute values in figure 18 of the appendix, and indexed to the default case of cc -O1 in figure 19. Because the produced executables are identical for both systems, with the exception of those produced with architecture specific optimization parameters, the results are presented in single tables instead of separate tables per test system. The results of gcc with -mtune=pentium3 are not displayed because the sizes were equal to the executables compiled with -mtune=pentium2.

Although the set of programs is not elaborate enough to draw hard conclusions about the optimization strategies used when looking at the segment size, it does give an indication of what can usually be expected.

In the table showing the sizes in bytes, the sizes of each segment can be compared to see the share each segment has in the total size of the executable. For example, the BSS segment of the sorting programs is very large compared to the other segments. This is caused by the use of large uninitialized arrays which are filled and then sorted at run time. Gzip also has a relatively large BSS segment, because it was compiled using buffers which are declared at compile time. Python has Text and Data segments which are much larger than the other programs. This is not very surprising since Python is an interpreter for a high level language, and thus has to implement a lot more functionality than the other programs. Apart from these observations, this table also indicates how relevant each value in the indexed table is. For example, looking at the Data segment for the sorting program, gcc has an indexed value of 228.57 but the absolute value is 256 against 112, which is small compared to the size of the Text segment.

The values for the Text segment of the programs compiled with cc -O2 in the indexed table show that the optimization used by cc decreases the size of the segment in every case. The behaviour for cc with -O3 is less clear because three results are missing due to failed compilation, but the value for the sorting programs indicate that the optimization strategies used here do not only remove instructions. Gcc shows a similar behavior when comparing its performance optimization parameter results. All programs show a decrease in Text segment size for optimization parameters -O1 and -O2 when compared to -O0, except for the whetstone program when architecture specific optimization is added (gcc version 3.4.3). Size optimization in gcc always produces a smaller Text segment than gcc's other optimization parameters, for Python this difference is 33%[1] for version 3.4.3 and 35%[2] for version 4.1.1. In bytes these decreases are 168 kB and 185 kB respectively.

The Data segment is always the same size in cc, and shows small variation among the performance optimization parameters of gcc. Size optimization in gcc does not increase the segment size, and can decrease segment size up to 16%[3].

[1] Comparing -O3 to -Os

[2] Comparing -O0 to -Os

[3] Comparing gcc 3.4.3 -O3 and -Os for awk

The BSS segment is the same size for every optimization parameter, so the tested optimization parameters do not use any strategies which have effect on the BSS segment size.

Comparing cc to gcc, it is not possible to pick a clear winner with respect to segment sizes. Looking at the results for the Text segment, the compiler which generates the smallest segment depends on the program being compiled. For example, bzip2 has a smaller Text segment with gcc when optimization is used, especially when optimizing for size. On the other hand, gzip is 7% to 42% larger than cc -O1 when compiled by gcc. The results for the Data segment show a similar behaviour, where gcc produces a smaller segment for some programs, and a larger segment for others. One point where a clear distinction can be made is the BSS segment; cc produces a smaller segment for each tested program.

To make an ordering in compiler / optimization parameter combinations, the average sizes of the Text segment have been sorted and displayed in figure 3 below. The averages were calculated without awk, and cc -O3 has been omitted from the results because too many programs failed to compile.

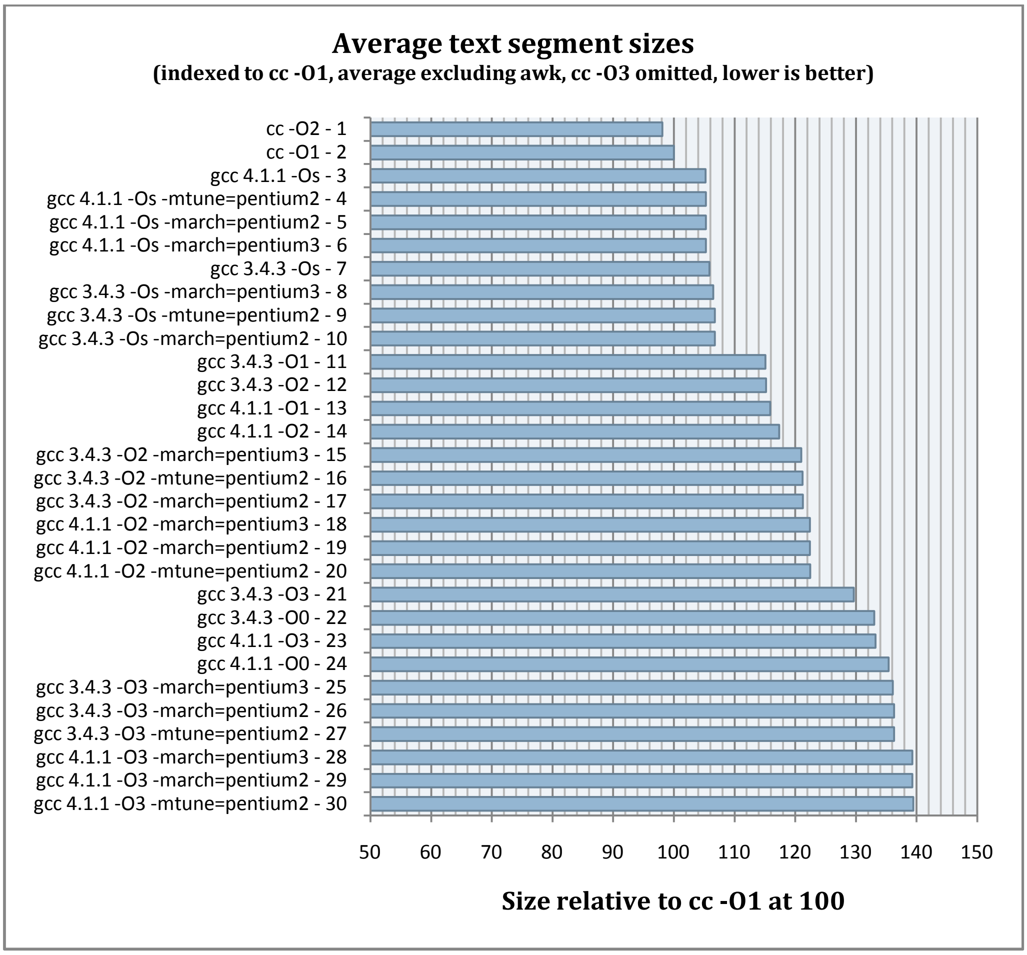

Figure 3 - Average text segment sizes

On average, cc produces a smaller Text segment than gcc, with cc -O2 producing a 2% smaller segment than the default cc -O1. Gcc generates the smallest segment when using optimization for size, with version 4.1.1 generating slightly smaller segments. Adding architecture specific optimization produces a marginally larger segment, but the difference is under 1% for version 3.4.3 and even under 0.1% for version 4.1.1. Optimization for size is followed by -O1 and -O2, and -O2 with architecture specific optimization produces a 5% larger segment than without these optimizations. The difference between the largest result for size optimization and the next smallest, gcc 3.4.3 -O1 is 7.8% which shows that the size optimization parameter is very effective compared to the other optimization options for gcc. At the bottom end of the chart, where the largest segment results are shown, -O3 produces a smaller segment than the -O0 of the same compiler version. Version 3.4.3 generates the smaller segment here, in contrast to the size optimization where version 4.1.1 generates the smaller segment. The difference is bigger here, with 4.1.1 being 1.8% larger for -O0 and 2.8% larger for -O3. The combination of -O3 and architecture specific optimization produces the largest segment. In these cases, the segments produced by version 3.4.3 are again smaller than version 4.1.1.

6. Execution

In this section, the results of the execution performance measurements are presented and discussed. This part of compiler performance is probably most interesting for most readers, since compilation time is only an issue during development in the vast majority of cases, and segment sizes are not very important on the majority of x86 based hardware today. Execution performance has been measured by timing each program as it executes a set of tests, to create a benchmark. Because some benchmarks perform I/O and others use almost none at all, the benchmarks have been split into two sets. Set 1 is the set of CPU bound benchmarks, so only user time is presented for this set. Real and system times are left out because the former is almost equal to the user time, and the latter is too small to measure in these benchmarks. Set 2 is the set of benchmarks which are less CPU bound, and therefore have a measurable amount of system time, so all times are included in their results.

This section is laid out as follows; first the used benchmarks are listed and explained, then the results for set 1 are presented and discussed, then the same follows for set 2. Gcc's architecture specific optimization is discussed last, so this subject will not be discussed in detail in the sections before it.

Used benchmarks

The benchmarks used in the measurements, and in which set they are included, are listed and explained in table 6. Where relevant, the used code is included in the appendix.

In most cases, the program was executed repeatedly using ccb_time (see *Used programs* in the section *Measurement set-up*). Some exceptions were made to this method, where the benchmark executed sufficiently long enough for a reliable measurement. The benchmarks that were not repeated are the sorting programs (except quicksort), whetstone, and pybench. Pybench has also been timed differently than the other benchmarks, because the program has its own way of calculating the performance of python. Instead, the minimum time total which pybench prints in the output has been parsed and used as real time in the data.

Table 6 - List of used benchmarks

Program	Benchmark	Set	Description
awk	duplicates	1	Detects duplicate words in the input (like “is is”).
	line_numbers	2	Prints the line number followed by a tab before each line of input.
	pi	1	Calculates π using the Bailey-Borwein-Plouffe formula with 500000 terms.
	word_count	2	Calculates the word count for each word found in the input and prints the results.
bc	e	1	Calculates e with 2000 decimals by calculating 8000 terms of the following series; $$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$
	pi	1	Calculates π using 1000 terms of the Bailey-Borwein-Plouffe formula. Uses 1000 decimals in the calculation.
bzip2	compression	2	Compresses a large xml file (~ 6 MB), a large text file (~ 1.8 MB), a MPEG 4 encoded avi file (~ 3.3 MB) and a jpeg file (~ 300 kB) at maximum compression (option -9).
	decompression	2	Decompresses the compressed files.
gzip2	compression	2	Compresses the same set of files as bzip2 again at maximum compression (option -9).
	decompression	2	Decompresses the compressed files.
python	pybench	1	Pybench is a benchmark suite currently included in the source tree of Python. The tested version 1.5.2 of Python did not contain this however, so the latest version (revision 61317) at the time of measurement was taken from the subversion repository at http://svn.python.org/projects/python/trunk/Tools/pybench. The file pybench.py was modified because it requires a module “platform” which was not present. Only code printing the platform information was removed. The program was run with the parameters -n 1 -w 20, which indicate one loop at warp 20. This was done to let the benchmark run for a shorter time than default.
	pystone	1	This is a translation to python of the dhrystone benchmark program (8). Pystone is included in the directory `Lib/test` of python 1.5.2.
sed	reverse	1	Reverses the character order per line.
	selective_print	2	Selectively prints lines containing the word “the”.
	substitute	2	Substitutes parts of input that match a certain format.
sorting	bubble	1	Sorts a randomly initialized integer array with 65536 items using bubblesort.
	insertion	1	Same but with insertionsort.
	selection	1	Same but with selectionsort.
	quick	1	Same, but with a larger array of 655360 items and quicksort. The larger array was necessary because quicksort is too fast for the smaller array.
whetstone	whetstone	1	Executes the whetstone program with 5000 loops (instead of the default 1000).

Benchmark set 1 results

Following the same approach as in section 4 dealing with compilation speed, the results are presented in three forms. First the absolute measurements are presented with time in clock ticks (appendix, figure 20), then the results indexed to cc -O1 (appendix, figure 21), and finally the averages of these indexed values are sorted and presented in a chart (figure 4 and figure 5). In addition to these results, the performance differences between the presets are also analyzed.

In the absolute results, the length of the benchmarks can be compared, which shows that of the sorting algorithms, bubblesort is the slowest followed by selectionsort, insertionsort, and quicksort. Quicksort is extremely fast in comparison, even though the benchmark for quicksort sorts a set which is 10 times larger than the other algorithm's benchmarks. Some large differences already can be seen between the different compilers and optimization parameters in these numbers, but the indexed table will show this more clearly.

From the indexed table, it is clear to see just looking at the colours (green and red indicate faster/slower than cc -O1 respectively) that gcc is faster in the vast majority of these benchmarks. The lowest value can be found for the *e* benchmark of bc compiled with gcc 3.4.3 with -O2, which is 26.1 on the Pentium 2 system and 25.8 on the Pentium 3. Even though these cases are the lowest points, these values show that when performance is important, it is worth investigating the performance of a program when compiled with different compilers and optimization parameters.

In general, one would expect that the execution performance of programs compiled with the more aggressive optimization presets to be higher than the less aggressive presets. To analyze this, the differences between each successive preset have been calculated for each benchmark. This calculation uses the following form; $\frac{\mathrm{O(n+1)} - \mathrm{On}}{\mathrm{On}}$. If the theory applies, for each successive preset this number will be below 0 to indicate a lower execution time. Tables 7, 8, and 9 show the results for the compilers cc, gcc 3.4.3, and gcc 4.1.1 respectively. Architecture specific optimization is ignored for these results, and empty cells indicate a failure of compilation. Awk is not shown in table 9, because it failed to compile in this compiler, gcc 4.1.1.

Looking at the results of cc, -O2 performs better than -O1 in most benchmarks, with a lowest value of -4.5% for the *pi* benchmark of bc. For -O3 the failure of compilation of awk and python limits the amount of data available, but the data that is available does show that -O3 is not necessarily faster than -O2. In some cases like quicksort or bubblesort the performance is substantially worse with 18% and 24% longer execution times for the benchmarks respectively. The *pi* benchmark of bc shows that -O3 can result in a big performance increase with 40% reduction in execution time.

For gcc, in most cases the values in the table are negative or only slightly positive, so the theory applies pretty well to this data. There are some exceptions however, with the *e* benchmark of bc showing a large degradation in execution performance for both version 3.4.3 and 4.1.1 of the compiler. In version 4.1.1, the *reverse* benchmark of sed also shows a significant decrease in execution performance. In both versions, the performance difference is largest between -O1 and -O0.

Overall it can be said that the performance optimization for gcc behaves as expected, with -O3 being faster overall than -O2, which is faster overall than -O1, and -O0 is slowest. For cc, the theory does not really apply very well, the preset -O3 shows mixed results, with some benchmarks performing faster while other benchmarks perform slower.

Table 7 - CC optimization ordering (values from Pentium 2)

CC		$\frac{O2 - O1}{O1}$	$\frac{O3 - O2}{O2}$
Program	**Benchmark**		
awk	duplicates	0.00%	
awk	pi	-2.31%	
bc	e	-2.63%	-7.24%
bc	pi	-4.50%	-40.31%
python	pybench	0.07%	
python	pystone	-1.77%	
sed	reverse	1.43%	0.06%
sorting	bubble	-1.04%	24.05%
sorting	insertion	4.80%	-9.81%
sorting	quick	-0.12%	17.69%
sorting	selection	-2.13%	0.46%
whetstone	whetstone	-2.70%	2.77%

Used color scale	-50%	-25%	0%	25%	50%

Table 8 - Gcc 3.4.3 optimization ordering (values from Pentium 2)

GCC 3.4.3		$\frac{O1 - O0}{O0}$	$\frac{O2 - O1}{O1}$	$\frac{O3 - O2}{O2}$
Program	**Benchmark**			
awk	duplicates	-8.62%	-2.80%	-9.41%
awk	pi	-6.05%	-2.10%	-0.95%
bc	e	-57.05%	-9.30%	58.44%
bc	pi	-57.99%	-9.06%	1.27%
python	pybench	-3.89%	-1.50%	0.15%
python	pystone	-22.67%	-8.82%	-4.88%
sed	reverse	-45.75%	-3.16%	-3.63%
sorting	bubble	-43.14%	0.70%	-11.59%
sorting	insertion	-42.76%	-0.03%	-16.21%
sorting	quick	-23.13%	1.72%	0.00%
sorting	selection	-31.32%	-2.04%	-6.34%
whetstone	whetstone	-0.39%	1.78%	-27.76%

Used color scale	-50%	-25%	0%	25%	50%

Table 9 - Gcc 4.1.1 optimization ordering (values from Pentium 2)

GCC 4.1.1		$\frac{O1 - O0}{O0}$	$\frac{O2 - O1}{O1}$	$\frac{O3 - O2}{O2}$
Program	**Benchmark**			
bc	e	-56.28%	-7.49%	40.70%
bc	pi	-51.29%	-8.65%	-8.57%
python	pybench	0.08%	-1.35%	-0.48%
python	pystone	-33.45%	-8.58%	0.97%
sed	reverse	-46.14%	-10.55%	7.40%
sorting	bubble	-41.95%	-4.11%	-3.17%
sorting	insertion	-51.53%	2.24%	-3.99%
sorting	quick	-21.09%	-1.76%	-1.80%
sorting	selection	-26.38%	-2.93%	-6.96%
whetstone	whetstone	0.10%	-2.10%	-26.86%

Used color scale	-50%	-25%	0%	25%	50%

As in the previous sections, to compare the different compiler / optimization parameter combinations to each other, an average was calculated for each combination except cc -O3. Again, this average excludes awk because it failed to compile in gcc 4.1.1. The best average execution performance is achieved with gcc 4.1.1 -O3 using -march architecture specific optimization with a value of 62 on the Pentium 2 and using -mtune on the Pentium 3 (because -march is missing the whetstone value due to failure in execution). Comparing both compilers shows that cc is only faster than gcc when no optimization is used for the latter. Disregarding architecture optimization for the moment, the optimization presets of gcc show a ordering of -O3 -O2 -O1 -Os -O0 from fast to slow for both versions of gcc. For each of these presets, version 3.4.3 is slightly faster than 4.1.1. On averages, architecture specific optimization is better for version 4.1.1, this is discussed in greater detail in the section Architecture optimization performance.

Concluding this benchmark set, when a program is very CPU bound, gcc can increase performance significantly compared to cc. Double the performance is no exception, especially when using the more aggressive optimization presets. Without architecture specific optimization, version 3.4.3 of gcc generally generates faster code than version 4.1.1, but when this optimization is used, version 4.1.1 generates faster code.

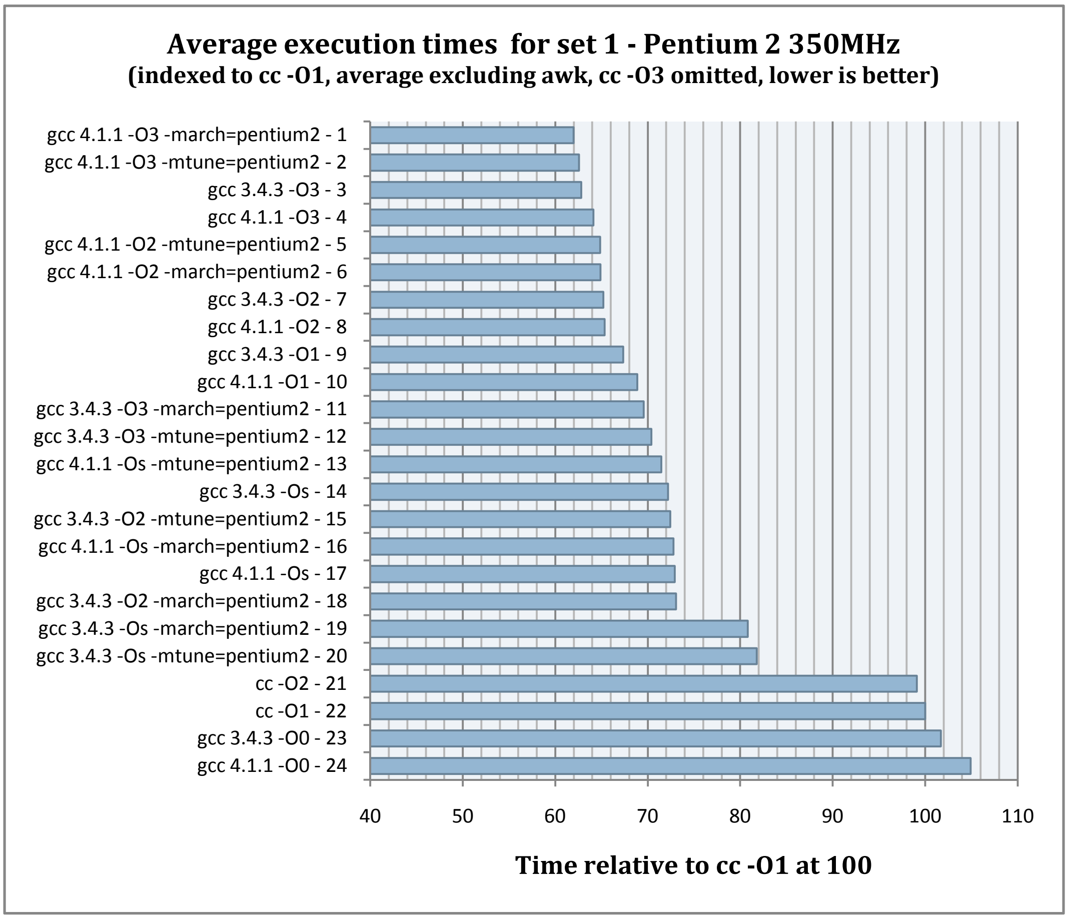

Figure 4 - Average execution times for set 1 (Pentium 2)

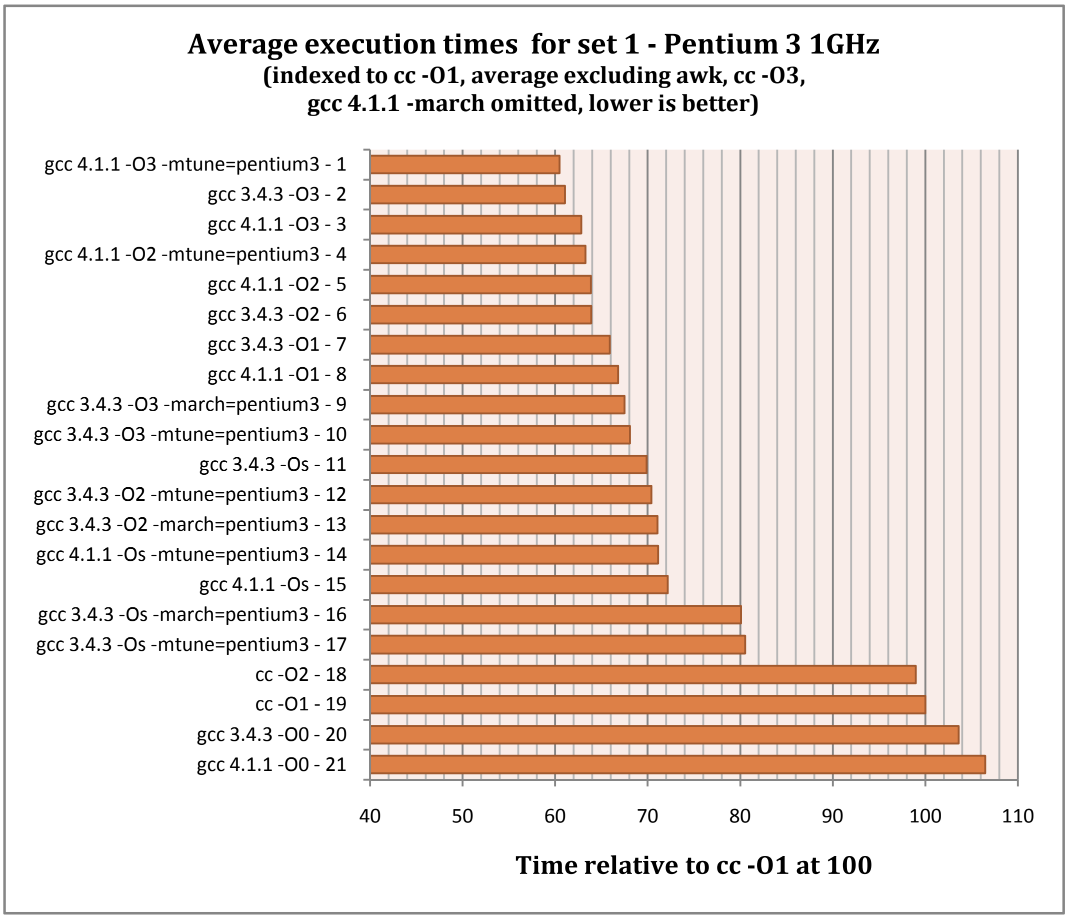

Figure 5 - Average execution times for set 1 (Pentium 3)

Benchmark set 2 results

The second set of results contains the benchmarks which use a measurable amount of system time. Their results are discussed much the same way as the first set, only gcc's optimization ordering is not discussed as elaborately as before.

Figures 22 and 23 of the appendix show the absolute results for this set of benchmarks on the Pentium 2 and Pentium 3 system respectively. The benchmarks are sorted from high to low by the User/Real ratio for cc -O1. The compression programs bzip2 and gzip both show asymmetric performance; compression is slower than decompression. Comparing the two programs to each other, it is clear that bzip2 is much slower than gzip (both programs compress and decompress the same set of files). Also bzip2 uses more system time, which could be caused by a different buffering strategy, but can also be caused by the different algorithms requiring a different access pattern. Again the absolute results show that gcc performs better than cc in a lot of cases, which is illustrated further in the indexed tables.

The indexed results are shown in figures 24 and 25 of the appendix. Comparing the results for the user times to the first set, the colours visually suggest that the differences between cc and gcc are not as large. Comparing the average user values confirms this. Bzip2 shows the biggest performance increase for gcc compared to cc, with roughly double the performance on the Pentium 2 when optimization is used, and a slightly smaller performance increase on the Pentium 3. The other benchmarks do not come close to this difference.

Moving on to the system values, the picture is very different. In this area, gcc is mostly slower than cc, with the exception of the bzip2 compression benchmark, the awk word_count benchmark, and some other benchmarks for certain optimization settings on the Pentium 2. On the Pentium 3, the two mentioned benchmarks also show mostly increased system time. The gzip benchmarks show a big increase in system time for gcc compared to cc. The Pentium 3 shows even greater differences than the Pentium 2; on the former, decompression is more than 4 times as slow with gcc compared to cc -O1, where on the latter the same benchmark is between 2 and 3 times as slow.

The indexed values for the real times show some differences with the user values. For example, the real values of gcc for the decompression benchmark of gzip are noticeably higher than the user values, especially on the Pentium 3. This is caused by the higher system times for gcc, which slow down the benchmark. The benchmarks with a high user/real ratio have real values which are almost the same as the user times because of the low impact of the system times on the total time.

Like the results of set 1, the results of set 2 have been graphically represented in figure 7 and figure 6 by taking the averages of the indexed values, and sorting these values low to high. In this case, both the user and real values are shown, and sorted by user. Compared to set 1, it is obvious that gcc's architecture optimization is more effective for this set of benchmarks, since all the top ten positions are taken by gcc with architecture specific optimization options. Version 3.4.3 of gcc shows an ordering in performance of -O3, -O2, -Os, -O1, and -O0 from high to low, for version 4.1.1 the same ordering applies, except -Os comes after -O1. Gcc version 3.4.3 is a bit faster than gcc version 4.1.1 in this benchmark set. This also holds when architecture optimization is used, in contrast to set 1 where version 4.1.1 was much faster than version 3.4.3. As mentioned before, gcc does not perform as good compared to cc as in set 1, nonetheless, in this set gcc generates faster code than cc when any optimization is used.

In conclusion for this set of benchmarks, for programs which are very IO bound, gcc can significantly decrease performance compared to cc. When the program does some IO but is more CPU bound, performance can benefit from the faster code gcc generates.

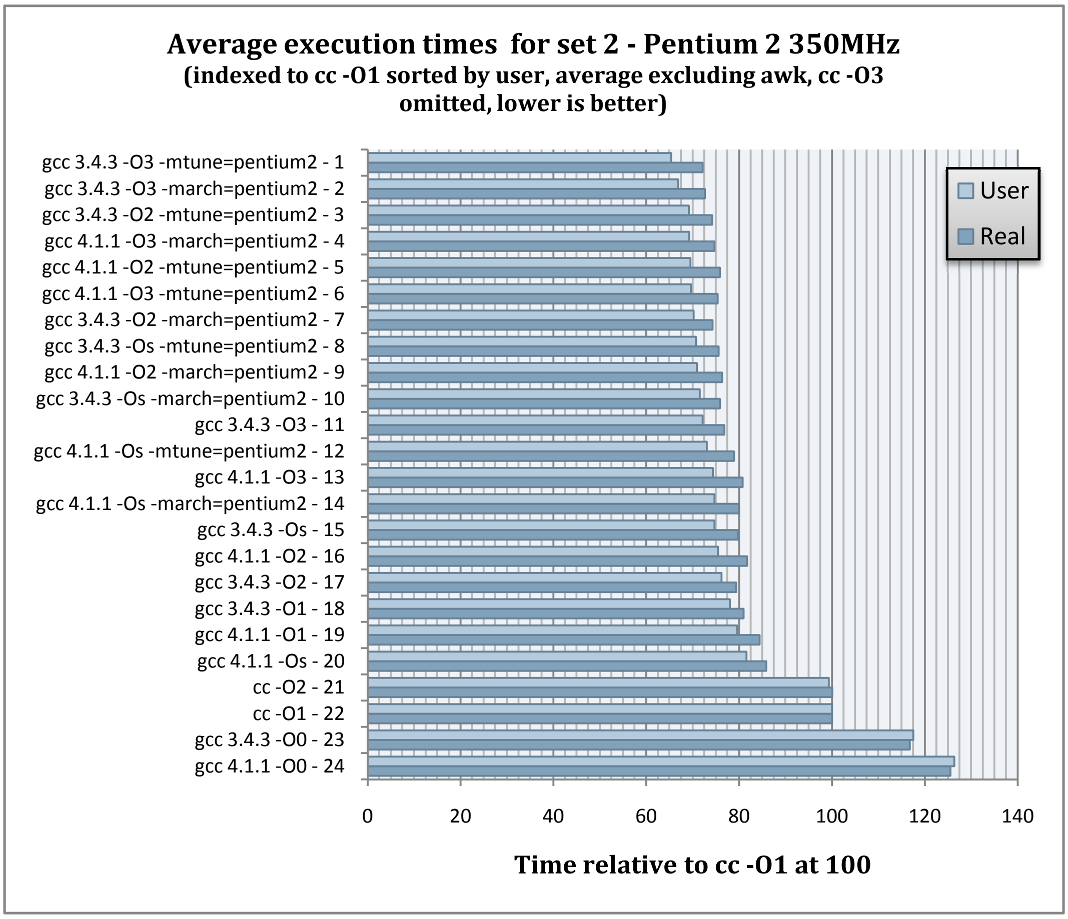

Figure 7 - Average execution times for set 2 (Pentium 2)

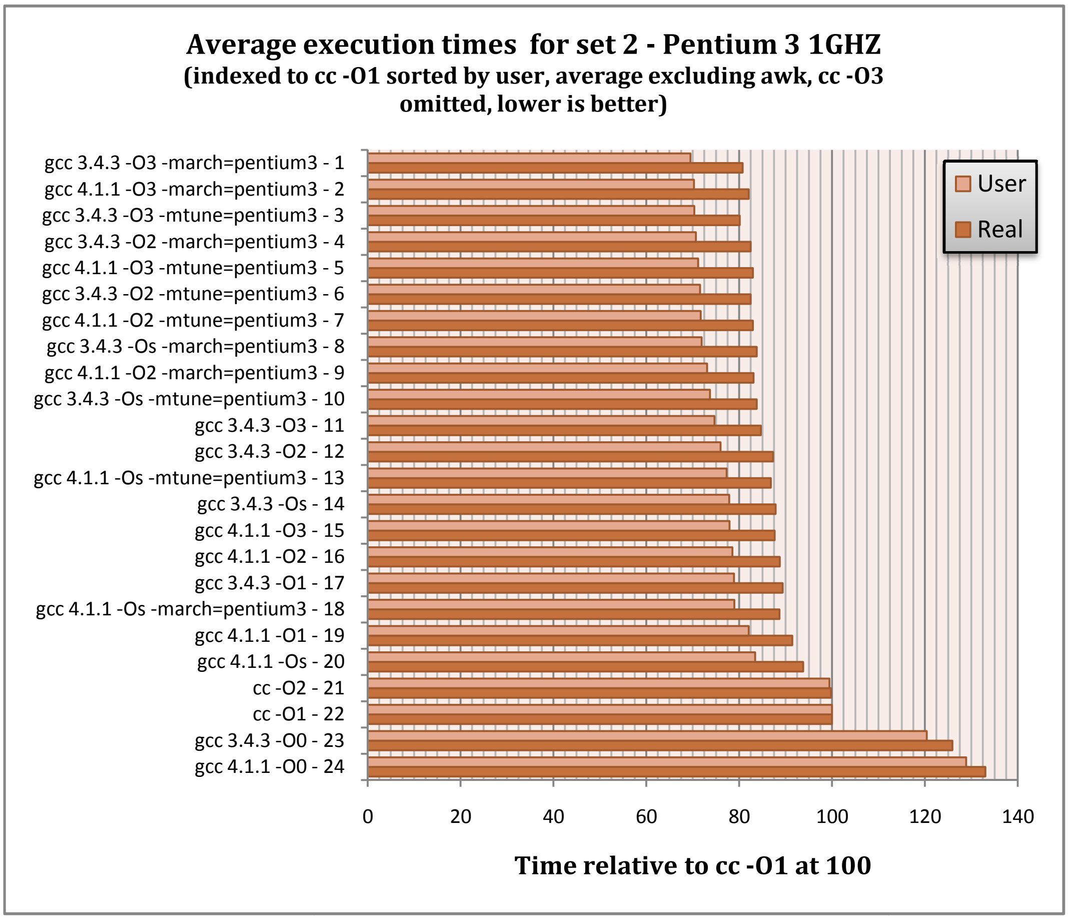

Figure 6 - Average execution times for set 2 (Pentium 3)

Architecture optimization performance

To judge the performance of the architecture specific optimization in gcc, I compared the benchmark times of the executables compiled with -march or -mtune to those that were compiled without architecture parameters but with the same optimization preset. These times were indexed in the same way as before, so for example -O2 was set to 100 and -O2 -march=pentium2 was indexed to this time. With these indexed values, the average was calculated for each setting. This average combines the benchmarks of set 1 and 2, and excludes awk, bc, and whetstone to make the comparison between version 3.4.3 and 4.1.1 fair. Awk was excluded like before because it fails to compile in version 4.1.1, whetstone was excluded because the program failed to execute properly when compiled with -march on the Pentium 3. Bc was excluded for a different reason. As can be seen in the indexed tables discussed before, bc's *pi* benchmark performs much worse when using architecture specific optimization in 3.4.3, and slightly worse in version 4.1.1. As none of the other programs show this behaviour, it was chosen to leave out this program for this comparison. This behaviour does show that architecture optimization does not necessarily improve performance, but can indeed degrade performance on the targeted architecture. Figure 8 shows the average values for both test systems. Comparing the two compiler versions, it is clear that gcc 4.1.1 shows better performance in this area with only the -O2 -march options and -O3 -mtune=pentium3 showing a slightly better result for version 3.4.3. The differences between the -march and -mtune options for version 3.4.3 are small in most cases, only the -O2 preset shows a relatively large difference. In most of these cases, the -march is faster, but for version 4.1.1 one cannot conclude that one generates faster code than the other. Only in the -O3 case does -march perform better than -mtune.

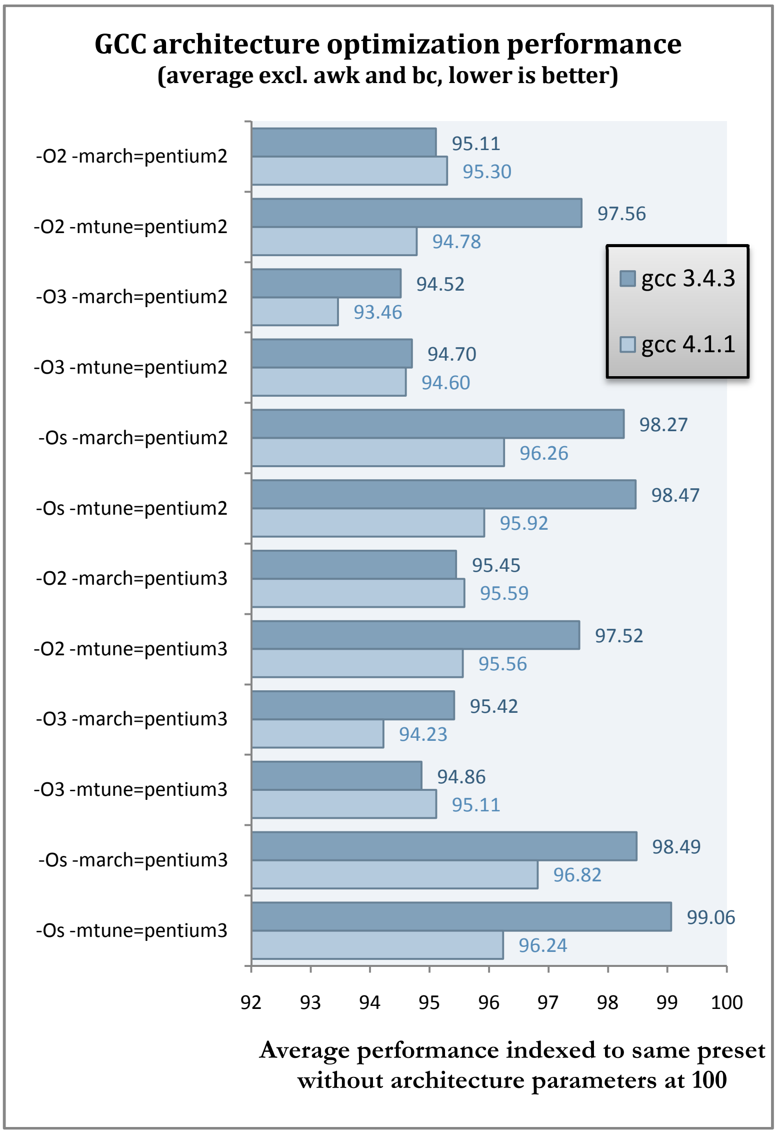

Figure 8 - GCC architecture optimization average performance

Using these architecture specific optimization parameters limits the distribution of the executables to the targeted executable, so the use of these options adds complications when distributing a program in binary form (one would need to compile and distribute separately for each architecture). Also, the effectiveness is not always guaranteed, as can be seen with the bc program, so for each program the effect on performance should be measured rather than applying these optimizations blindly.

7. Conclusion

When it comes to compilation speed, comparing ACK's cc to gcc shows that cc compiles a great deal faster for all programs. This difference is caused by two factors; cc's faster compilation when looking at the compiler's user times, and the faster loading and initialization of cc. During development therefore, cc has an advantage in reducing compilation time. In this paper other compiler features like code safety checking and debugging facilities have not been taken into consideration however, so these features should be taken into consideration as well when choosing a compiler for the development phase. Comparing the gcc versions to each other, version 4.1.1 compiles considerably slower than version 3.4.3, which can be attributed to the loading and initialization phase which takes almost 2 seconds for version 4.1.1 against 0.67 seconds for version 3.4.3.

Segment sizes are generally larger for gcc than for cc, especially when aggressive performance optimization is used. Gcc's size optimization preset -Os always generates smaller segment sizes compared to the other presets, but does not always generate smaller segment sizes than cc. In general, when segment sizes are an issue, using cc gives the best results, however for some programs gcc gives better results, so it is useful to compare the two for the program being compiled.

Comparing cc to gcc in the area of execution performance, gcc clearly performs better than cc. For some programs, gcc even outperforms cc when no optimization is used for gcc (-O0). On average, execution times can be reduced to 77% and 85% using gcc -O3 on the Pentium 2 and Pentium 3 respectively. Some programs like bc however are reduced to 26%, so the potential performance gain is quite large, depending on program and its usage. Architecture specific optimization does not necessarily improve performance, it can also degrade performance, and version 4.1.1 of gcc performs better in this area than version 3.4.3.

The results shown in this paper show that it could be useful to try to make use of the gcc compiler toolchain to build at least some of the programs released for MINIX 3. Also, it is worth investigating if the kernel can be built using gcc, and if so, what the differences are in kernel size and system performance. Also, the comparison between cc and gcc could be broadened to look at compiler features which were ignored for this paper, like debugging and code checking.

8. References

1. **GNU GCC Team.** Options That Control Optimization. *GCC 3.4.3 Online Documentation.* [Online] http://gcc.gnu.org/onlinedocs/gcc-3.4.3/gcc/Optimize-Options.html.

2. —. Options That Control Optimization. *GCC 4.1.1 Online Documentation.* [Online] http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Optimize-Options.html.

3. —. Intel 386 and AMD x86-64 Options. *GCC 3.4.3 Online Documentation.* [Online] http://gcc.gnu.org/onlinedocs/gcc-3.4.3/gcc/i386-and-x86_002d64-Options.html.

4. **Wang, Thomas.** Sorting Algorithm Examples. *Thomas Wang's Home Page.* [Online] http://www.concentric.net/~ttwang/sort/sort.htm.

5. **Painter, Rich.** C Converted Whetstone Double Precision Benchmark. *The Netlib.* [Online] Painter Engineering, Inc., March 22, 1998. http://www.netlib.org/benchmark/whetstone.c.

6. *A synthetic benchmark.* **Curnow, H J and Wichmann, B A.** 1, 1976, Computer Journal, Vol. 19, pp. 43-49. Available from http://freespace.virgin.net/roy.longbottom/whetstone.pdf.

7. **Bal, Henri E.** *The Design and Implementation of the EM Global Optimizer.* Vrije Universiteit. Amsterdam : s.n., 1985. Available from http://tack.sourceforge.net/olddocs/ego.html or http://tack.sourceforge.net/olddocs/ego.pdf.

8. **Weicker, Reinhold.** Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM (CACM).* October 1984, Vol. 27, 10, pp. 1013-1030.

9. **Beszédes, Á, Gergely, T, Gyimóthy, T, Lóki, G, Vidács, L.** *Optimizing for Space: Measurements and Possibilities for Improvement.* Research Group on Artificial Intelligence, University of Szeged. Szeged, Hungary : s.n., 2003. Available from http://www.inf.u-szeged.hu/gcc-arm/paper/summit2003-paper.ps.