Investigating memory usage and run times in Stock Synthesis compiled with different ADMB compilers

Ian Taylor
Allan Hicks

NOAA Fisheries, NWFSC, Seattle, WA

September 7, 2010

Abstract

ADMB includes options for controlling memory allocation for some of its calculations. Run time, memory use, and temporary file sizes were sampled for an example application over a range of inputs for these options and for four combinations of compiler and operating system. Although some results were puzzling, the patterns were consistent enough to provide some recommendations to ADMB users for memory settings and to ADMB developers for changes that could improve total memory use in the future.

Introduction

AD Model Builder, or ADMB, is a suite of libraries which will perform optimization of nonlinear models using auto-differentiation. Stock Synthesis (SS) http://nft.nefsc.noaa.gov/Download.html is a general stock assessment model which is compiled with the ADMB libraries and commonly used to assess fish stocks in the United States. The assessment models that SS are capable of can be rich with data including numbers-at-age, length-at-age, and numbers-at-age conditioned on length. This results in large matrices of data and population values (such as predicted numbers-at-age) which can occupy large amounts of memory. ADMB is capable of finding the maximum likelihood value given many parameters, and SS takes advantage of this with capabilities of estimating life-history parameters, fishing fleet parameters, survey parameters, and recruitment deviations. The combination of large data sets, fine resolution of the population, and a large number of parameters results in a need for large amounts of memory as well as numerous calculations which also need derivative information.

Recent assessments are approaching what appears to be limits of ADMB as well as current hardware and software configurations, and there have been reports of needing more memory than is available on a computer. ADMB has the ability to write temporary files to the hard drive instead of trying to allocate large amounts of memory when that memory may not be available, but writing to the hard drive is a much slower process than writing to dedicated memory. Increasing the amount of memory is possible in some computers, but many have already exceeded the maximum amount possible in a 32-bit operating system.

We describe the behavior of an ADMB model with different memory allocations when compiled with 32-bit and 64-bit compilers in Windows and Linux.

Run times to minimize some models often take an hour or more and Bayesian methods are impossible within the management time frame due to the time consuming Markov Chain Monte Carlo simulations needed.

Methods

A hypothetical model with many parameters and years modeled was ran in Stock Synthesis version 3.10c using different inputs of memory allocation for the cmpdiff.tmp and gradfil1.tmp files to determine the size of temporary files, memory usage, and speed between different compilers. The memory allocation inputs were entered on the command line using -gbs and -cbs. Systematic values of these inputs were not used because we wanted to cover a large space of combinations of values, yet focus on a small area where interesting results were expected to occur. Figure 1 shows the matrix of values used for -gbs and -cbs.

Four compilers (Table 1) were used to compile the SS version 3.10c code in optimized mode with ADMB 9.1 libraries released on December 31, 2009.

Runs were performed on a single computer with Windows and Linux operating systems set up in identical ways. The hardware is specified in Table 2. Windows 7 64-bit and Ubuntu 10.04 64-bit were installed at the beginning of identical, but separate hard drives. R 2.11.1 was used to run the different cases while recording memory allocated and size of the cmpdiff.tmp and gradfil1.tmp files every three seconds. The maximum of these recordings were used as a summary statistic. Additionally, the run time recorded in the Report.sso file written by SS was also recorded for each case. To limit the duration of the test, 25 function evaluations were used in each case and the Hessian matrix was not calculated.

Table 1: C++ compilers used to compile SS version 3.10c

MS Visual Studio 2008 32-bit
MS Visual Studio 2008 64-bit
GNU Compiler Collection version 3.4 for Windows (MinGW) 32-bit
GNU Compiler Collection version 4.4 for Linux 64-bit

Table 2: Hardware used to run the tests for each compiler.

Processor	Intel Core 2 Quad Q9550
RAM	2 $\times$ 2Gb DDR2 800 (PC2 6400)
Hard drive	Western Digital 750Gb Caviar Black
Motherboard	Asus P5G41-M LE
Video	Integrated Intel G41

Results

There were similarities and differences observed between compilers and operating systems. One of the biggest differences was that the 32-bit compilations failed to run at values near and above 1.9×10^9. This is can be seen as white areas in Figures 2–5. The four compiled programs behaved differently in this space (at least at the values run), with the Microsoft 32-bit compiler always failing to run, the Microsoft 64-bit compiler appearing to run normally, the MinGW 32-bit compiler failing to run at some intermediate values then appearing to run with odd behavior, and the Linux GNU 64-bit compiler not failing to run but exhibiting some odd behavior.

The four compilers showed similar patterns with regard to file sizes of the temporary files at values less than 1.9×10^9. Increases in `gbs` resulted in monotonic decreases in the size of gradfil1.tmp (Figure 7), and, when not of zero size, the gradfil1.tmp file was larger with 64-bit executables. As `cbs` increased, the cmpdiff.tmp file tended to decrease, but showed an undulating pattern before reaching a threshold value which resulted in a zero sized file (Figure 6). The compilers differed greatly in what happened once values greater than 1.9×10^9 were entered (and sometimes even smaller numbers when both `gbs` and `cbs` were large). The Microsoft 64-bit compiler always had file sizes at zero with large input values while the Microsoft 32-bit compiler failed to run (the application quit before minimization sometimes with a memory allocation error). The MinGW 32-bit compiled application failed to run after just exceeding 1.9×10^9, but ran at even larger values with varying file sizes. SS compiled with the Linux GNU compiler showed the same pattern as the MinGW 32-bit compiled program, except that it never failed to run.

Random access memory (RAM) allocated to SS increased with increasing values of `gbs` and `cbs` (Figure 4, although RAM allocated reached a plateau at small input values of `gbs` while memory increased linearly with `cbs` until a value near 1.9×10^9 when it the program either failed to run or exhibited odd behavior (Figures 4 and 8). The total memory allocated to RAM and temporary files (shown in Figure 9) mainly increases with `cbs` and is constant across inputs for `gbs`.

Run times varied across all compilers with 64-bit compilers showing faster run times than 32-bit compilers (Figure 10). The percent difference in run time compared between the 32-bit compilers, the MS compilers, and the 64-bit compilers is show in Figure 11. The two 32-bit compilers were similar in speed with a slightly faster run time with the Microsoft compiler when temporary files were not be written to. The 64-bit version of the Microsoft compiler greatly improved run times by about a 20% gain when temporary files were not be written to. The 64-bit Linux application outperformed the 64-bit Microsoft compiler by a large amount when small and intermediate input values for `cbs` and `gbs` were used. Once the inputs were large enough to eliminate writing to temporary files, Linux ran the minimization about 20% faster than the Windows Microsoft 64-bit compiler.

Discussion

Overall, the SS executable created with the Linux 64-bit GNU compiler outperformed the Microsoft and MinGW Windows compilers. Within the Windows environment, the Microsoft 64-bit compiler outperformed the 32-bit compilers. A 64-bit MinGW compiler was not tested, and it is likely that it will outperform its 32-bit counterpart. We are not certain why the 64-bit compilers ran the model faster, but evidence suggests that 64-bit ADMB executables outperform 32-bit executables, and although 64-bit executables use more memory for the same model ran with a 32-bit executable, a 64-bit system is able to address much more memory than a 32-bit system, making the extra memory a moot point with today's computer configurations.

Odd behavior occurred when input values neared and exceeded the 32-bit maximum for a signed integer ($2^{31} - 1 = 2.147483647 \times 10^9$). The 32-bit compilers failed at least part of the time, the Windows 64-bit compiler appeared to behave properly, and the Linux 64-bit compiler produce intermittent odd behavior. Furthermore, it was discovered that entering a `cbs` or `gbs` value in the tpl file of an ADMB program (via `CMPDIF_BUFFER_SIZE` and `GRADSTACK_BUFFER_SIZE`) produced different behavior than when entered via the command line. An brief investigation into the ADMB code revealed that the computations related to the gradstack and the compiled derivative code were coded used `long` integers, but the command line inputs were read in with an `int`. With all of these 32-bit compilers a `long` and an `int` are allocated 32-bits of memory, thus are similar. The Windows Microsoft 64-bit compiler also allocates a `long` and an `int` with 32-bits of memory, but the Linux GNU 64-bit compiler allocates 32-bits of memory to an `int` and 64-bits of memory to a `long`. This does not explain the difference in the odd behaviors seen with the compilers other than they all exceeded their integer limits.

Understanding the limits of the data types and how ADMB is currently coded gives insight into ADMB code changes which could allow large models to allocate more memory and increase the chance of eliminating any writing to temporary files. In the Linux environment, where the 64-bit compiler allocates 64-bits to a `long` data type, no change needs to be made to allocate more memory to the gradstack and precompiled derivative calculations when the `CMPDIF_BUFFER_SIZE` and `GRADSTACK_BUFFER_SIZE` are declared in the tpl file since the function that sets these sizes expects a `long` as an input. However, some minor changes must be made to properly use the command line inputs of `gbs` and `cbs` since these are converted to integers. We made these simple changes in the ADMB code in the file xmodelm3.cpp and were able allocate over 12 Gbytes of memory to SS on a large Linux server. The changes would be more difficult with Windows compilers since `long` data types are allocated 32-bits (`http://en.wikipedia.org/wiki/Integer_(computer_science)`) and changes would likely need to be made to the code for the gradstack and precompiled derivative calculations.

It is likely that the combinations of different compilers, ADMB versions, models, and system setups affect the optimal settings of input values for `cbs` and `gbs`. Only one model was used to test these runs, and that configuration in SS was specifically created for this study. We examined the catch-at-age model ("catage") that is included with the ADMB examples (with simulated data covering 40 ages and 200 years to slow it down) and after running it on a different setup we saw similar results, but on a different scale. If large enough values of `cbs` and `gbs` cannot be input to eliminate the writing of temporary files, then we suggest running a similar experiment of different input values to find what values may balance memory usage and run time on the system being used. Code written for R is available that can repeat this experiment for your particular setup.

Our recommendations are as follows:

- For all compilers (with the current version of ADMB) keep both `gbs` and `cbs` $\leq 1.8 \times 10^9$
  - This value is slightly below the $2^{31} - 1 \approx 2.14 \times 10^9$ numerical limit currently implemented in ADMB, but in practice it seems better to not push the limit too closely.
- Within this limit, if possible, increase `cbs` and `gbs` values until temporary files are of size zero
  - The total memory usage asymptotes with large values of `gbs`, but appears to continually increase with values of `cbs`. This should be tested for your particular setup to determine if it holds true.

- Start using 64-bit operating systems and compilers (either Windows or Linux) with 4Gb or more of RAM

This was a small study looking at a specific example and there are many other investigations that would prove useful to improving ADMB. Some topics worth investigating include the following:

- test additional compiler/OS combinations as well as different models,
- investigate the difference between entering `gbs` and `cbs` values in the tpl file vs. the command line,
- investigate the behavior of memory usage when the Hessian is estimated and inverted,
- consider the question of optimizing the run time for MCMC runs.

Additionally, our wish list for the ADMB project is

- better warnings when inputting values beyond limits of the compiler,
- recode ADMB to read in larger numbers and make use of 64-bit operating systems with additional RAM,
- release ADMB for the MinGW GNU 64-bit compiler,
- better document the details of how and why temporary files are written

Figures

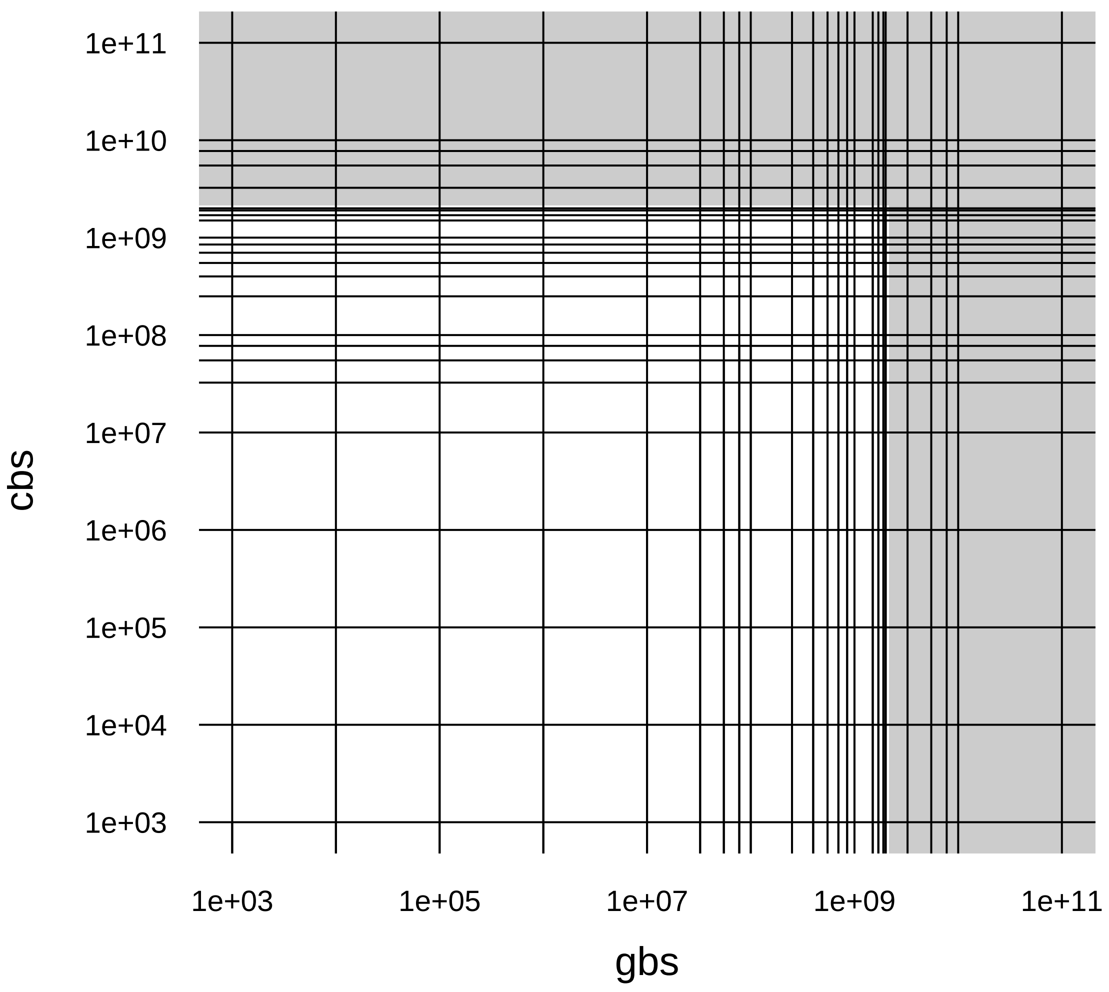

Figure 1: Values input for -gbs and -cbs. The grey area is where the input value exceeds 2^{31} (a signed 32-bit integer).

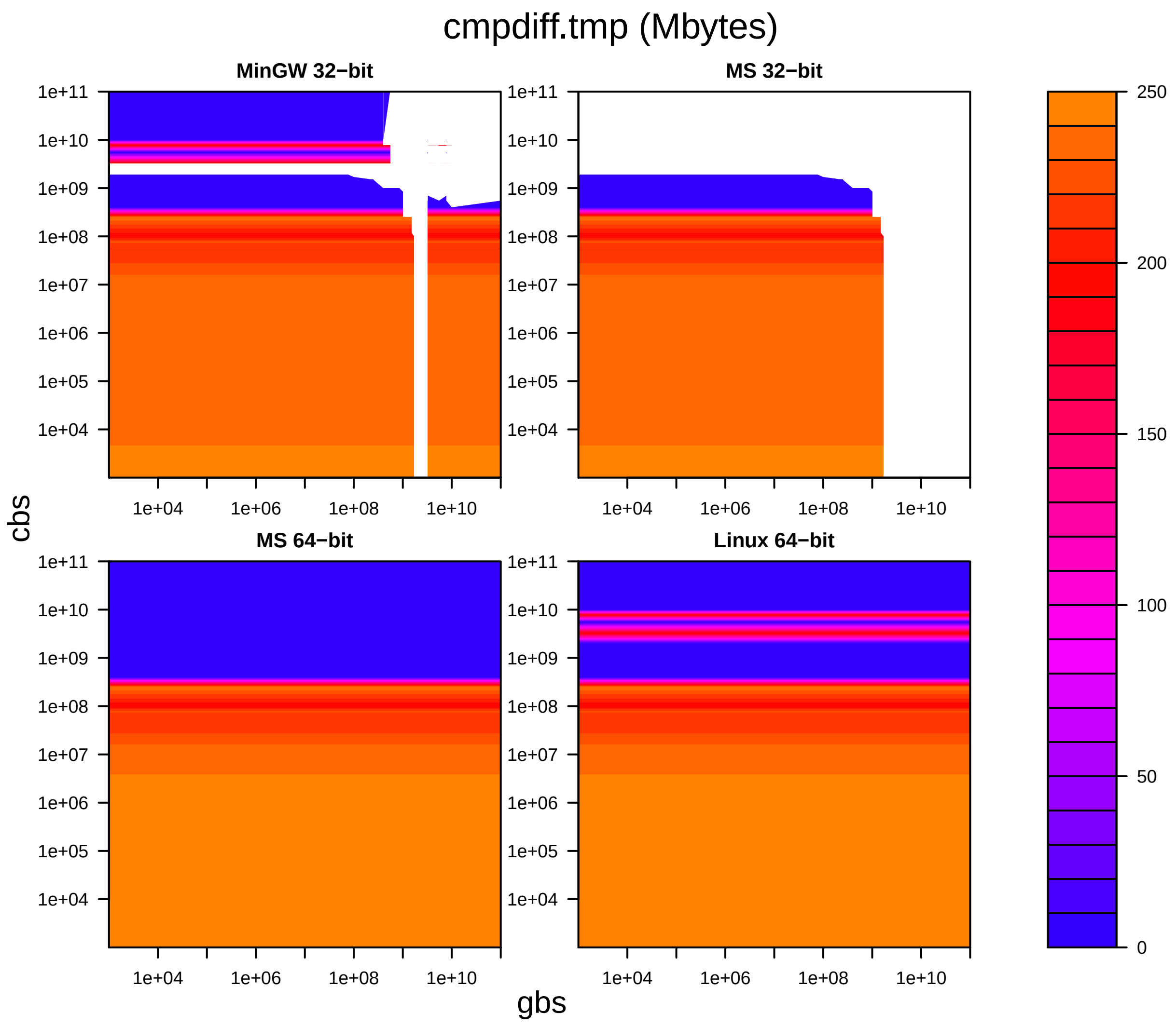

Figure 2: Run time of the Stock Synthesis model with different -cbs and -gbs inputs in log base 10 space.

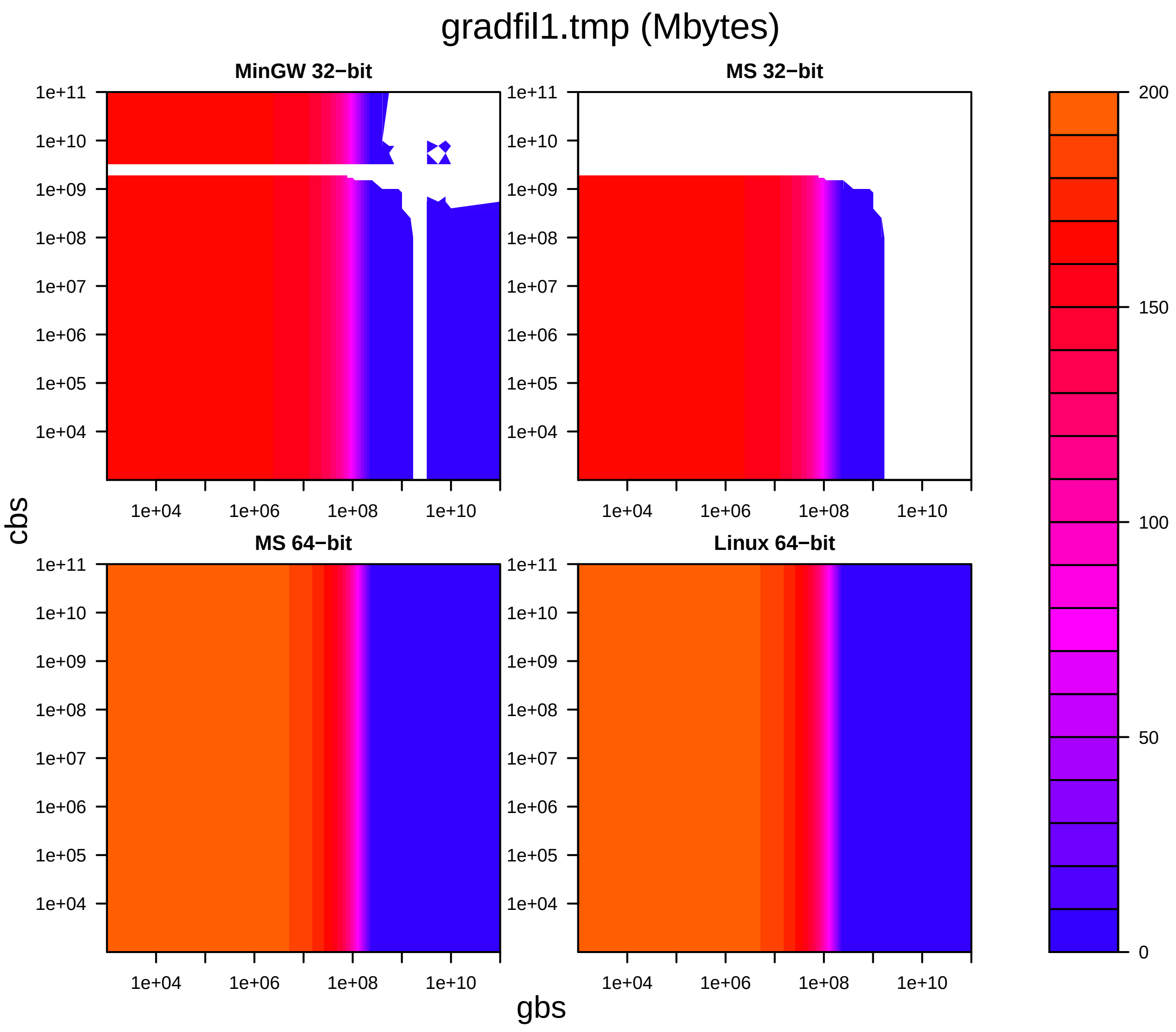

Figure 3: Run time of the Stock Synthesis model with different -cbs and -gbs inputs in log base 10 space.

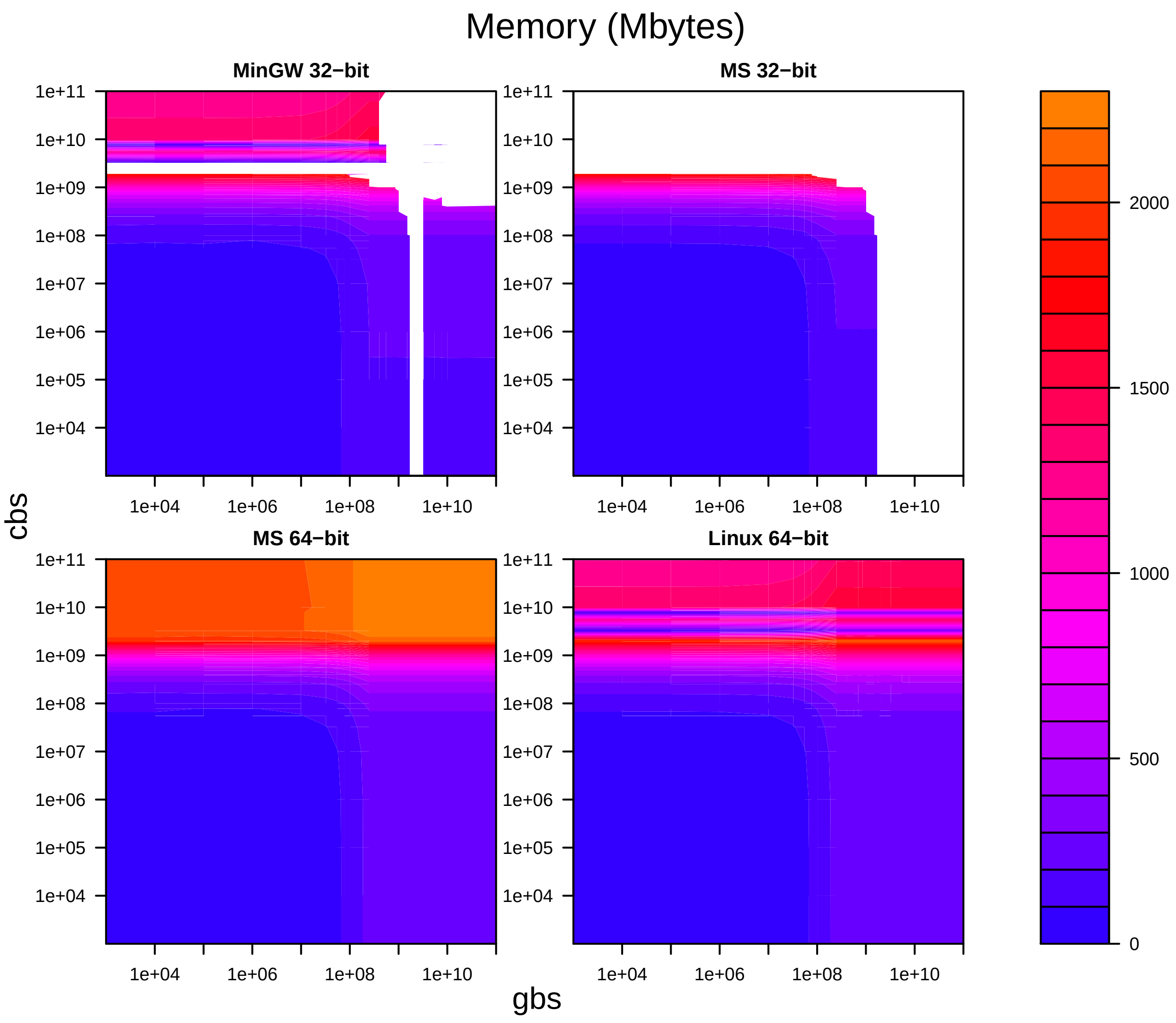

Figure 4: Run time of the Stock Synthesis model with different -cbs and -gbs inputs in log base 10 space.

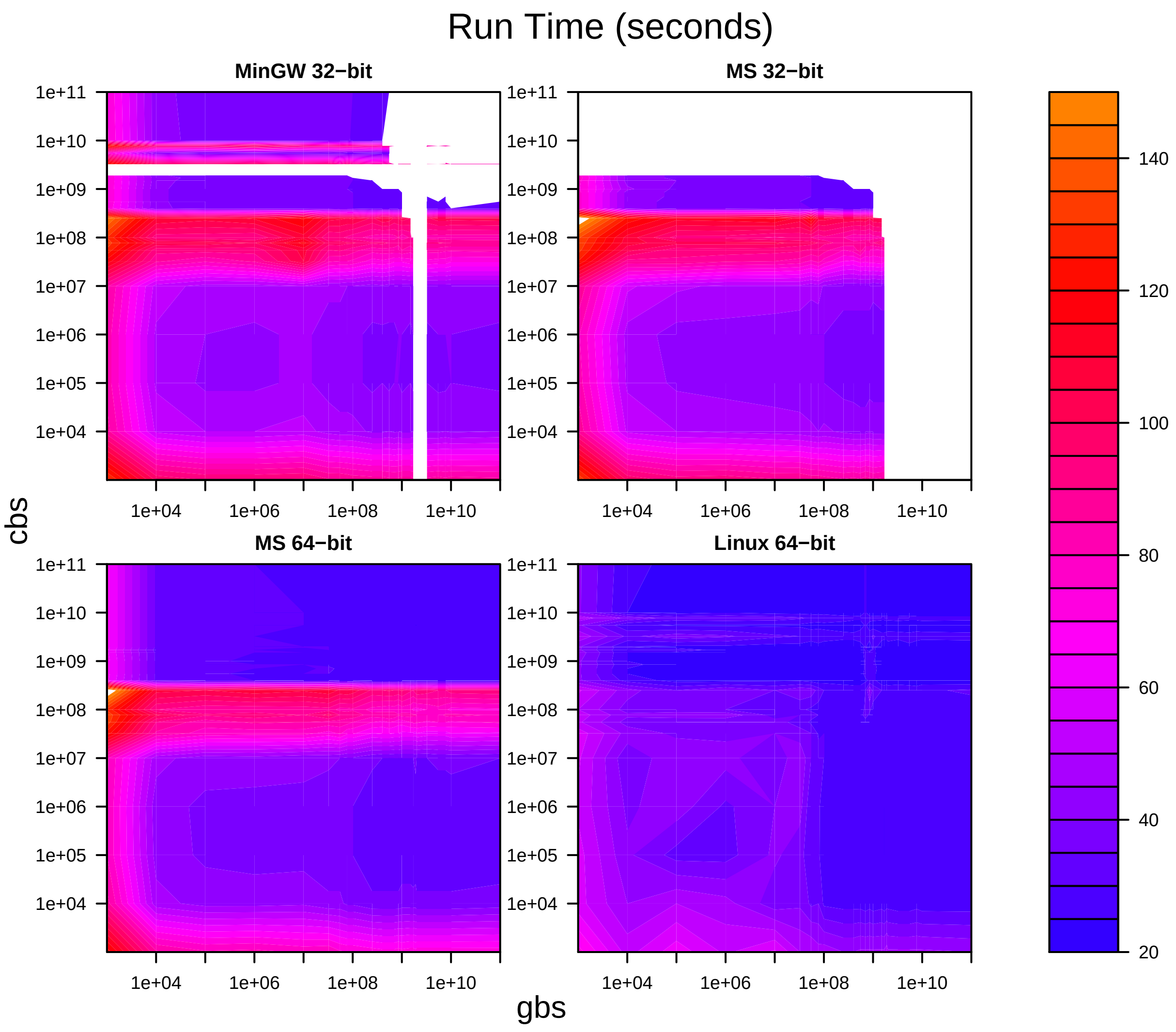

Figure 5: Run time of the Stock Synthesis model with different -cbs and -gbs inputs in log base 10 space.

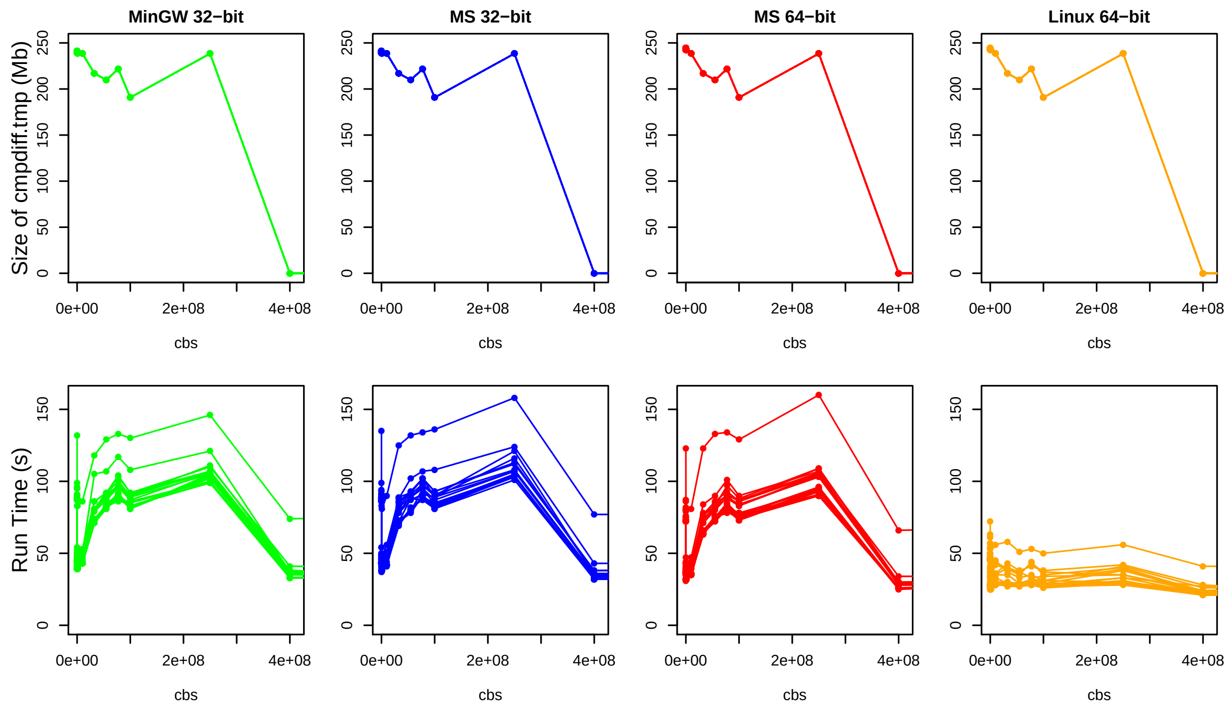

Figure 6: Size of cmpdiff.tmp and run time of the Stock Synthesis model over -cbs inputs in real space.

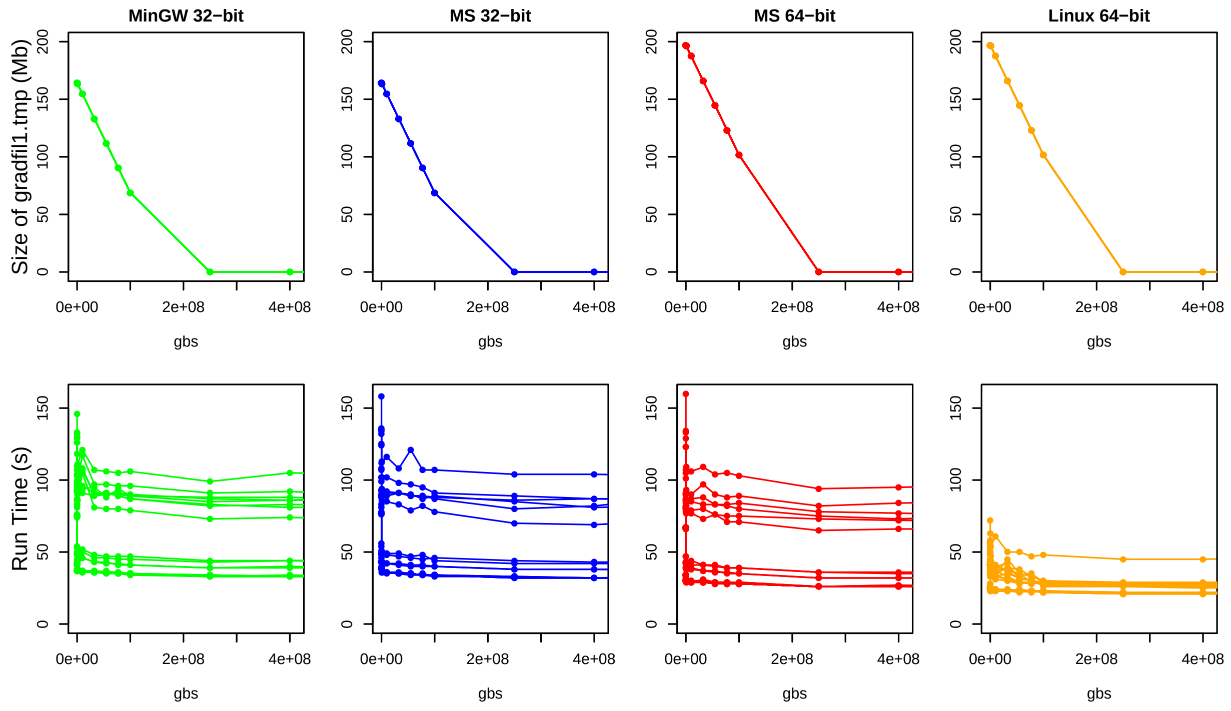

Figure 7: Size of gradfil1.tmp and run time of the Stock Synthesis model over -gbs inputs in real space.

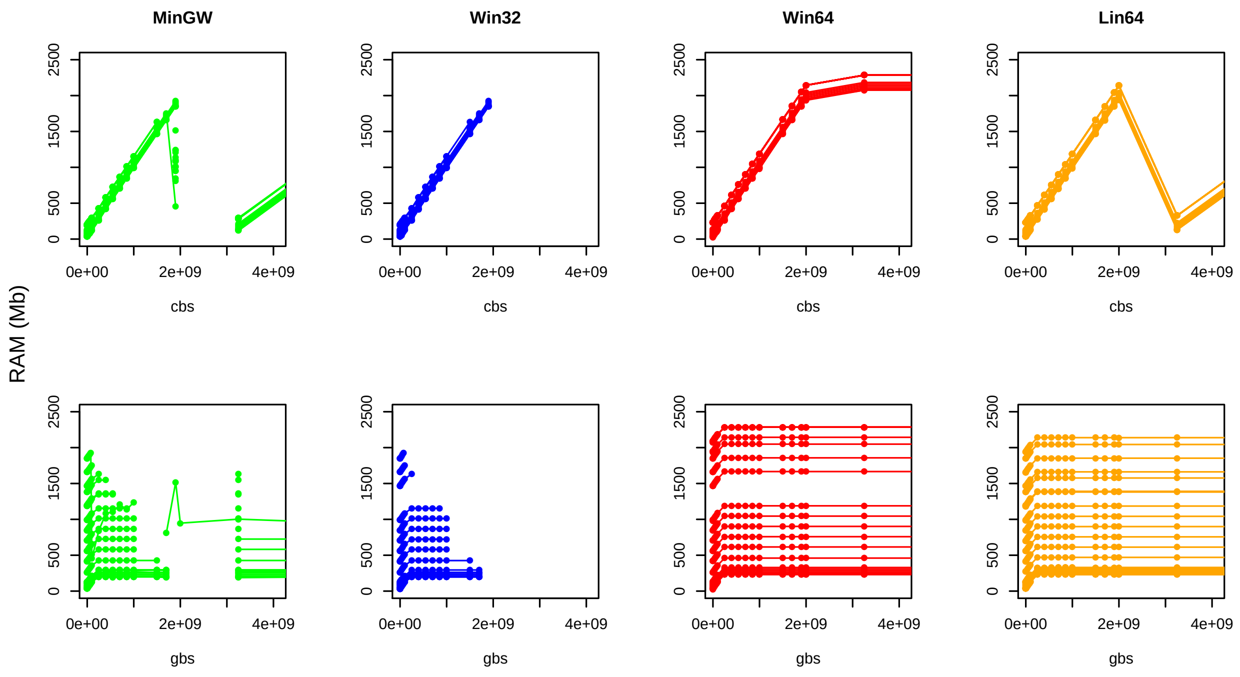

Figure 8: Random access memory (RAM) allocated to the Stock Synthesis model -cbs (upper row) and -gbs (lower row) inputs in real space.

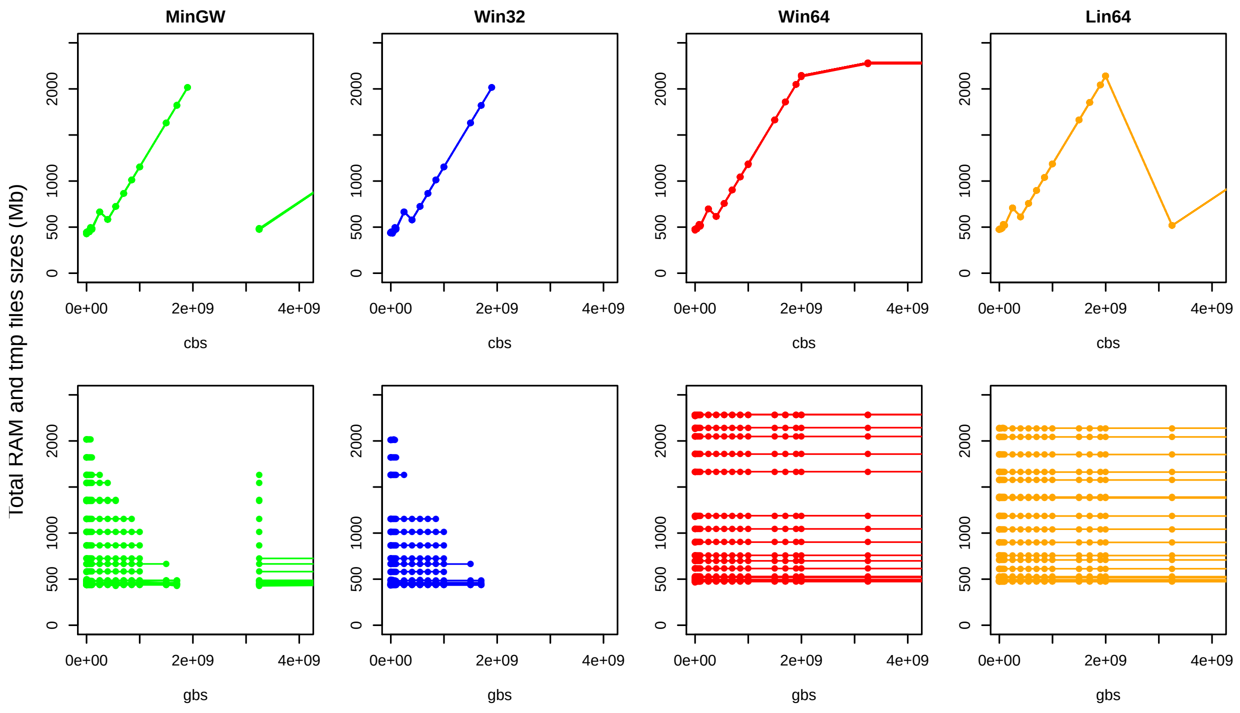

Figure 9: Random access memory (RAM) allocated to the Stock Synthesis model -cbs (upper row) and -gbs (lower row) inputs in real space.

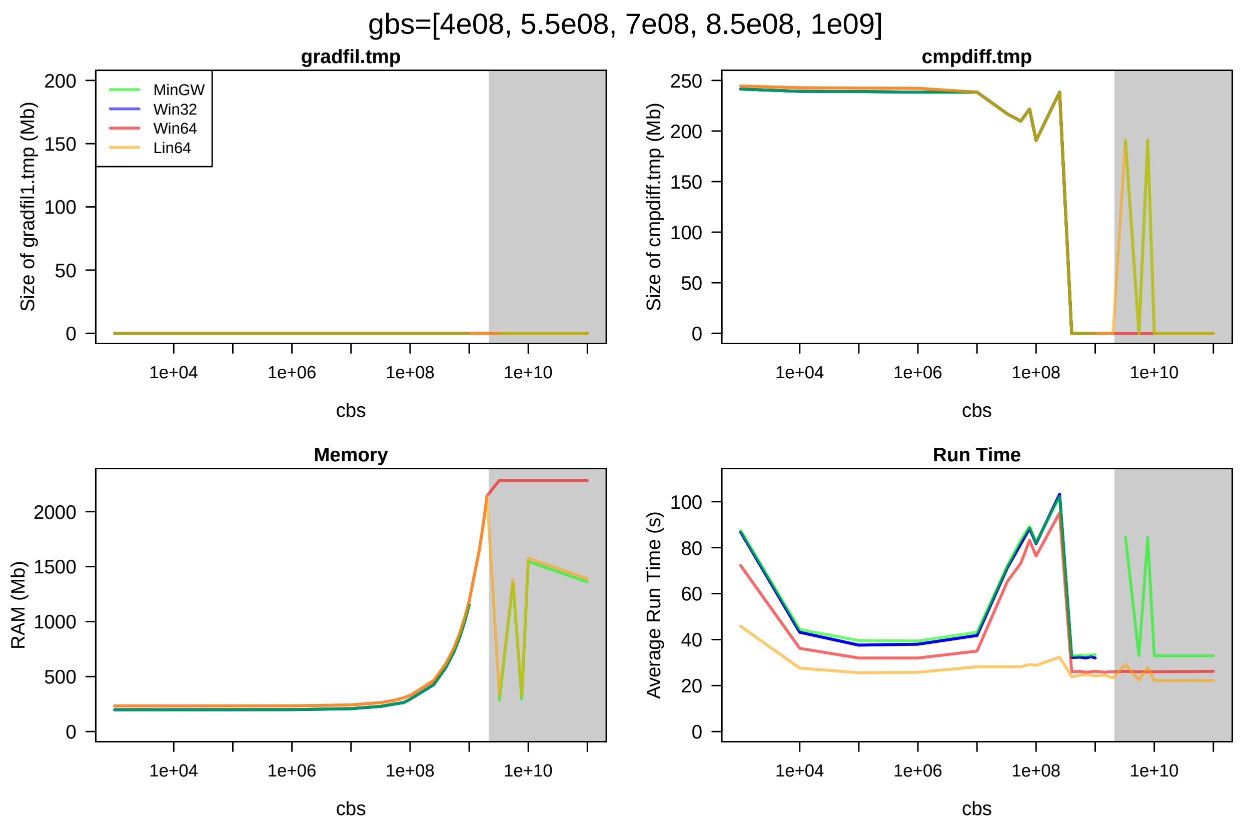

Figure 10: All four quantities measured for the Stock Synthesis model with -cbs and inputs in log base 10 space averaged over the results of five input values for gbs.

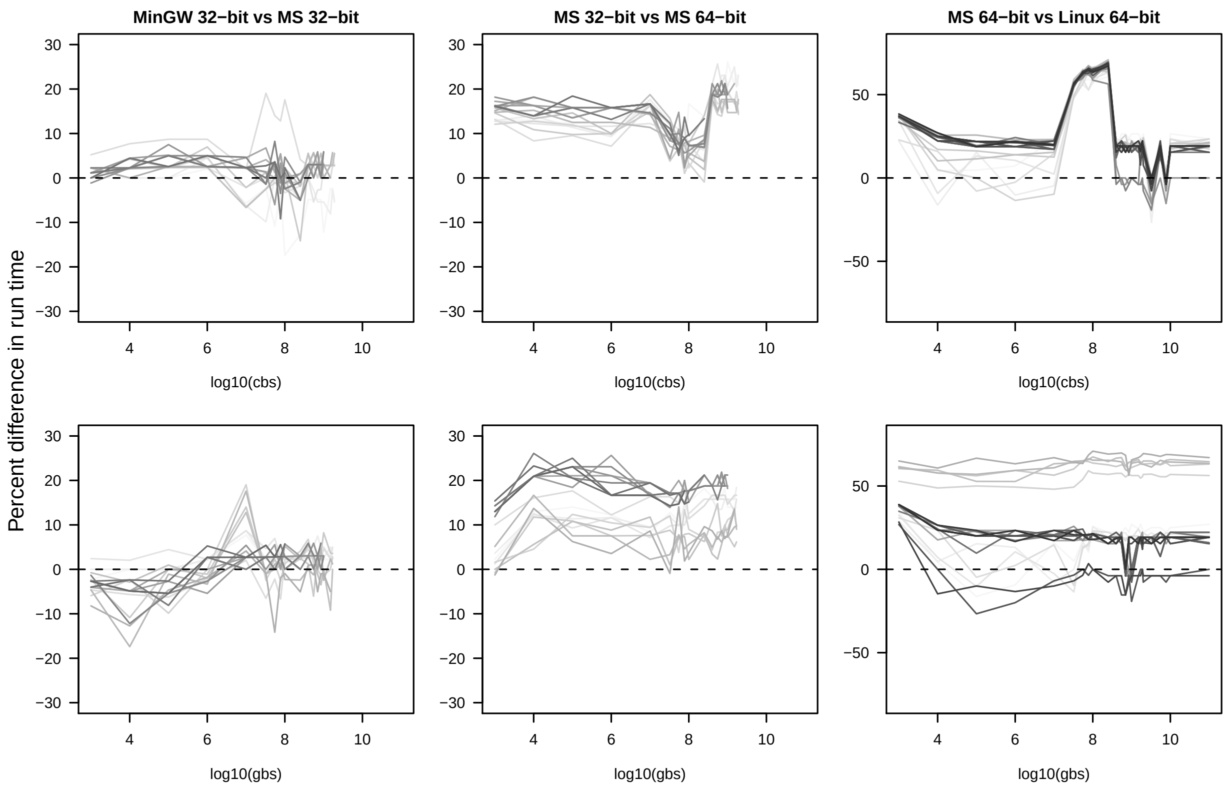

Figure 11: Percent change in the run time of the Stock Synthesis model with -cbs (upper row) and -gbs (lower row) inputs in log base 10 space. Light colored lines correspond to smaller values of the input value of -cbs or -gbs which is not plotted on the x-axis.