Labs in EDAG01 Efficient C

November 9, 2022

1 Overall structure

- In labs 1 - 3 you will implement the simplex algorithm
- In labs 4 you will implement branch-and-bound
- In labs 5 you will use a simulator from IBM to visualize instruction execution on a superscalar processor (POWER8)
- In labs 6 you will experiment with optimizing compilers: gcc, clang, commercial ones from IBM and Nvidia

2 How to do the labs

- To make lab presentation more efficient, you must have a Discord username that shows your real name (but to ask questions, you can use any username).
- Except for Lab 6, you can do the labs on Mac, Linux, or Windows with the Linux subsystem.
- Some functionality will only work on Power.
- First download the Tresorit directory you got an email about.

3 Lab requirements and questions

The teachers will ask you questions including these but it is not required to write the answers in the pdf.

Lab 1: Introduction to C

1. File input and memory allocation in C The purpose of this lab is to learn about reading numbers from a text file, allocating memory, and printing numbers with nicely aligned columns. Below ”the book” refers to the course book printed in 2020, and in particular Appendix B which you can find as a pdf in the Tresorit directory you got a link to in an email. The first task is to create a hello-world program. Start an editor and a terminal window.

   Write a hello world program, such as on in Section 1.1 in the book, (if you don’t have the book, take e.g. the one from the slide in the first lecture), and save it in a file called e.g. intopt.c.

   Try to locate the file and cd to the same directory in the terminal (see e.g. Appendix A in the book).

   Type the following command in the terminal window:

   ```
   gcc intopt.c
   ```

   Run it with:

   ```
   ./a.out
   ```

   Next change your program to read two integer numbers from stdin. See first lecture. The numbers should be stored in two variables called m and n. When you run your program, you need to hit the key called "return" or "enter" after the two numbers.

   Print their values using the formatting string `"m = %d, n = %d\n"`.

   To avoid typing so much, put your input in a file, called i and run with this command:

   ```
   ./a.out < i
   ```

   Now it is time to allocate memory for a linear program.

   The easiest way to create the matrix is to use a double** pointer. See book or first few slides of F03. You can also get inspiration from the check function in main.c in the intopt directory from Tresorit (type tar xvf intopt.tar.bz2 to expand it).

   It is also a good idea to look at the video clips for Lecture 2.

But for lab 1, you don't have to understand anything about linear programs. It is only about reading the input and printing it.

See Appendix B, and use the following format for the input, where m is the number of constraints and n the number of decision variables.

As you can see in the youtube clips, there is a vector with n c-coefficients, a matrix with m rows and n columns, and a vector with m b-values. The matrix and vectors should have elements of type double.

The format of the input is:

$$\begin{array}{cccc} m & n & & \\ c_0 & c_1 & \ldots & c_{n-1} \\ & & & \\ a_{0,0} & a_{0,1} & \ldots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \ldots & a_{1,n-1} \\ & \ldots & & \\ a_{m-1,0} & a_{m-1,1} & \ldots & a_{m-1,n-1} \\ b_0 & b_1 & \ldots & b_{m-1} \end{array} \quad (1)$$

Use the following input (from Appendix B):

```
2 2
 1     2
-0.5   1
 3     1
 4    18
```

Print out the linear program to check that it is what you expect.

Try to print your linear program with nice columns. Hint: if you instead of only using `%lf` instead use `%10.3lf` with printf, it will print the numbers 10 characters wide and with 3 decimal digits. Compile and run the following program to get ideas about making nice columns!

```
#include <stdio.h>
#include <math.h>

int main()
{
        double  pi = 4 * atan(1.0);

        printf("%lf\n", pi);
        printf("%+lf\n", pi);
        printf("%-lf\n", -pi);
```

```
        printf("% lf\n", pi);
}
```

Also add ”max z = ” and ≤ and + between columns. You can use <= or Unicode in the C string such as `"a \u2264 b"` for $a \leq b$.

To make the output even neater, you can experiment with using some additional flags. See the course book or give the command: man printf

Now change your program to allocate too little data. Allocate memory for only one instead of two coefficients in the objective function. Run your program and see if anything strange happens.

Next compile with -g to tell gcc to add debug information to a.out:

```
gcc -g intopt.c
```

Use Valgrind to let it find the array-index-out-of-bounds error:

```
valgrind ./a.out < i
```

What does it say?

Next try the Google sanitizer:

```
gcc -g -fsanitize=address intopt.c

./a.out
```

It will say some obscure information but also some very useful things.

If you download the intopt program with the input files for your platform from Tresorit, you can try them as follows:

```
chmod 700 intopt
./intopt -f simplex.in
./intopt -p -a intopt.in
```

Lab 2: The GNU Debugger GDB

You can read about gdb in Section 6.5 (on page 212) in the book.

1. Create a `struct simplex_t` according to page 718 but with proper C syntax.
2. In this lab you should implement the following functions from Appendix B.
   - `simplex`
   - `xsimplex`
   - `pivot`
   - `initial` and assume $b_i \geq 0$ so skip the call to `prepare` and the rest of `initial`
   - `init`
   - `select_nonbasic`
3. It is expected that your will encounter various problems and to fix them you should use a combination of debugging with printing output and the GNU debugger gdb. Both are equally important to be familiar with.
4. When you have implemented these, your program should be able to solve the example input from Lab 1 (and Appendix B). The optimal value should be 16.
5. To use gdb your program should be compiled with the `-g` option.
6. If your program crashes, start the debugger:

   ```
   gdb a.out
   ```

   and type

   ```
   run < i
   ```

   or just

   ```
   r < i
   ```

   which will start the program with input from the file i.

   You should now see where in the source code the program crashed.
7. To print out the value of a variable, say x, type

   ```
   print x
   ```

8. Use commands such as:

```
break xsimplex
break 100
```

to set a so called breakpoint at the beginning of function or certain line number.

9. If you run the program again it should stop at the breakpoint.

10. Try to find out what the difference is between the two commands:

    ```
    next
    step
    ```

11. To continue execution, type

    ```
    c
    ```

12. Now call your function which prints your linear program from gdb. If that function is named print and your linear program is represented in a struct named s, you can type

    ```
    p print(s)
    ```

13. You will now learn a very useful trick to find out when a variable changes value. Add a global variable `int glob;` before any function.

    Then add for instance `glob += 1;` to any function that will be called (but use `main` on Ubuntu app in Windows — see below).

    On Windows and the Ubuntu app, you need to give the following command in gdb:

    ```
    set can-use-hw-watchpoints 0
    ```

    Normally watch uses a hardware feature which makes it fast but that does not work in the Ubuntu app so an alternative is to use so called software watchpoints. Note that they are very slow so it is probably better to do this task on Power or at least modify `glob` at the very beginning of `main`.

    Recompile, start gdb and type

    ```
    watch glob
    ```

    Run your program. It should stop when glob is modified.

    It is easy to use watch for global variables but to stop when data allocated with malloc or calloc is modified, you need to know the address of that data.

    Put a breakpoint in `pivot`

    Assuming you have a struct s with the array b, type

```
p &s.b[1]
```

That should give you the address of element one of the b array.

The address will be a hexadecimal number (base 16). Assume it is `1234567890abc0` and the type of the element is `double`.

Now type

```
watch *(double*)0x1234567890abc0
```

Continue the program and see what happens.

14. Find out what the following commands mean

    - `up`
    - `down`
    - `display`

Lab 3: Advanced Valgrind and Google Sanitizer

In this lab you should implement the following functions from Appendix B which will then make your linear program solver complete.

- `prepare`
- make `initial` complete (Appendix B, page 720)

1. When all elements of the b array are non-negative, we can solve the linear program starting in the origin, i.e. vertex 0.
2. Use the same input as in the previous labs but add the constraint that all feasible points should be on or above the line $x_1 = 5 - x_0$. See Figure B.1.

   How do you express this constraint in your linear program from the previous labs?

   Will this new constraint affect the value of the optimal solution?
3. Implement the function `prepare` and add the call to it in `initial`. Verify that your program reaches the `prepare` function.
4. Implement the rest of the function `initial` and make sure it works for the example input.
5. It is now time to use Valgrind to check that all memory allocated from the heap also is deallocated. Run your program with

   ```
   valgrind ./a.out < i
   ```

   If you use a Mac, do these tasks on a Linux computer instead since Valgrind does not work on macOS. For you own future use after the lab, the flag `-fsanitize=leak` can be used with the C compiler on macOS. Apple did not enable this so you need to install a different C compiler like this:

   ```
   brew install llvm@12
   PATH=/usr/local/opt/llvm@12/bin:$PATH
   ```

   It is a good idea to put the assignment to `PATH` in a file, such as `.zshrc` in your home directory so it will be set automatically when a new terminal is started.

   Valgrind will probably complain about memory leaks due to not freeing everything allocated.

   Try to figure out what was not freed and modify your program accordingly.

   If that turns out to be complicated, use instead

   ```
   valgrind --leak-check=full --show-reachable=yes ./a.out < i
   ```

6. Now destroy your program by commenting out the first for-loop in the function init If you allocated memory for the var array with calloc, then switch to malloc.

   The purpose is to see that Valgrind can warn you about using uninitialized memory. Run with

   ```
   valgrind ./a.out < i
   ```

   Does Valgrind complain?

   We would like to know where that memory was allocated, and therefore run with

   ```
   valgrind --track-origins=yes ./a.out < i
   ```

   This collects more information and is slower but can be crucial. In Section 6.9.1 memcheck / Reading uninitialized memory (on page 228) you can see that Valgrind keeps track of individual bits in memory.

7. Repair your program so it works again and then make another disaster: take a for-loop and skip initializing the loop index variable. Run again with the previous command.

   Does Valgrind keep track of local variables?

8. Now add the following to any function in your program:

   ```
   int     local_array[10];

   for (i = 0; i < 11; i += 1)
           local_array[i] = i;
   ```

   Does Valgrind check index-out-of-bounds for local variables allocated on the stack?

9. Repeat the last problem with a global array. What does Valgrind do?

10. Now compile with

    ```
    gcc -g -fno-common -fsanitize=address intopt.c
    ```

    Repeat the experiments with local and global arrays. What is the result?

Lab 4: Performance measurements: operf, gprof, and gcov

In this lab you should implement the remaining functions from Appendix B.

As a rule, compile your intopt.c with `-lm` at the end of the command line which tells the compiler to search for math functions when producing a.out (this needs to be done in addition to `include <math.h>`).

Note! To pass the lab your `intopt.c` must be sufficiently complete so that you get a score of at least 80 on Forsete. To get good performance measurements, it is useful to take an input case with 20 variables and inequalities. For this lab, even if your program computes the wrong answer for input with 20 variables, you can most likely get useful statistics with that large input anyway.

1. You need to do this lab on a computer with operf enabled, which means either power.cs.lth.se or your own Linux machine. See page 221 for how to enable operf on Linux. See Appendix A for how to work remotely with a Unix machine (Linux or macOS) and login to power.cs.lth.se.

2. You will use but don't have to implement `isfinite` from the pseudo code since it is part of the Standard C library.

3. When you have implemented the remaining functions your integer program solver can find the optimal solution for our example input.

4. In tresorit, there is a file `intopt.tar.bz2`. Unpack it by typing:

   ```
   tar xvf intopt.tar.bz2
   ```

   Now run with larger input, which you can find in the just unpacked directory `intopt/test`. The structure is: `intopt/test/`N contains directories with N decision variables and inequalities. Start for example with input from a subdirectoty of input/test/20.

   You will next use four different programs for performance measurements.

5. First type compile as usual with `-g` to produce debugging info (such as line numbers), and then type:

   ```
   operf -e CYCLES:100000:0:0:1 ./a.out < input
   ```

   See page 220. Your program is run with hardware counters enabled which count the number of clock cycles.

   Then use `opreport` and `opannotate` to figure out which functions take most of the time in your program.

   ```
   opreport -t 1 -l -g a.out
   ```

   which should list all functions which get at least 1% of all samples, and:

```
opannotate -s a.out
```

which takes the source file and print it with sample counts for each line.

```
opannotate -a -s a.out
```

also prints machine instructions.

If you want to count something other than clock cycles, you can type

```
ophelp
```

to see what can be counted on POWER8.

6. Next compile with

   ```
   gcc -pg -g intopt.c
   ```

   and run your program. Then give the command

   ```
   gprof -T a.out
   ```

   What does the output mean and how can you use it to better understand your program?

7. Next compile with

   ```
   gcc -fprofile-arcs -ftest-coverage -g intopt.c
   ```

   and run your program again. Then give the command

   ```
   gcov intopt.c
   ```

   This will create a file intopt.c.gcov. What does it contain how and can you use this information?

   To get even more detailed information about which cases in your program are most common, you can use

   ```
   gcov -b intopt.c
   ```

   What does it say?

8. In addition to finding memory errors, Valgrind can also be useful when doing performance measurements. Valgrind is slower than the other tools due to it simulates the computer so use a small input, at least initially. Recompile with -O3 and give the command

   ```
   valgrind --tool=cachegrind --I1=65536,1,128 --D1=32768,2,128 \
   > --LL=1048576,8,128 ./a.out < i
   ```

These options specify the cache parameters to match our POWER8 machine.

How can you see how many instructions in total, load instructions, and store instructions are executed?

What are the cache miss rates and are the cache likely to be a performance problem for this input?

Lab 5: the IBM POWER8 pipeline simulator

- In this lab you will study your intopt using a cycle-accurate pipeline simulator from IBM, which you can find the in `sim_ppc.tar` in the tresorit directory.
- The IBM pipeline simulator takes as input an instruction trace, which is a file with a sequence of:
  - instruction address
  - instruction opcode
  - optional data address

  Thus neither register nor memory contents are included in the trace.
- For efficiency, this file is in a binary format, but a human-readable ASCII file can also be created.
- The instruction trace is created by a special version of valgrind, using the IBM tool itrace.
- The output from itrace is then converted to another format for the simulator.
- Then the simulator is run, and it analyzes essentially all details relevant for performance of the pipeline and produces an output file which can be used by another IBM program, scrollpv to visualize the execution, with remarks about what is happening in the pipeline such as that an instruction is waiting for input operands.
- The `scrollpv` program is written in Java and is most easily used at home (but in theory you can login with ssh -Y to power.cs.lth.se but I suspect it will be far too slow). The `scrollpv` program can be downloaded from IBM, or alternatively, you can download it from the Tresorit directory (see email).
- You need to create the trace files on power.cs.lth.se so be sure you have setup ssh to avoid having to type your password (use ssh-keygen and then ssh-copy-id stilid@power.cs.lth.se — see appendix A (also in Tresorit)).
- Note that what we call reorder buffer in the course is called global completion table in IBM terminology. Lab instructions

1. Copy the following directory on power.cs.lth.se to your home directory on that machine:

   ```
   cd
   cp -r /usr/local/cs/edag01/labs/lab5 .
   ```

2. There you can find the following script as the file `t`

```
export FILE=$1
rm -rf a.out *.vgi *.qt *.dir &&
gcc $FILE.c -fno-tree-vectorize -O3 -mcpu=power8 -fno-inline &&
/opt/at13.0/bin/valgrind/valgrind --tool=itrace --fn-start=main --binary-outfile=$FILE.vgi --num-insns-to-collect=1k ./a.out &&
/opt/at13.0/bin/valgrind/vgi2qt -f $FILE.vgi -o $FILE.qt &&
/opt/ibm/sim_ppc/sim_p8/bin/run_timer $FILE.qt 1000 100 1 $FILE -inf_all -p 1 -b 1 -e 2000
```

   It can be used to compile a C file, making the trace and running the simulator.

   To analyze a file a.c, invoke the script with the parameter a, such as with ./t a.

   It is easier to learn about the simulator if only simple instructions (no SIMD) are used so we compile with -fno-tree-vectorization.

   Always tell valgrind in which function tracing should start as is done above with `--fn-start=main` (about 200,000 instructions are executed before main is reached with dynamic linking, and valgrind dislikes static linking).

   The number of instructions to collect can be specified with k, m, or g.

   The simulator is called `run_timer` and takes the following parameters above:

   - a qt-trace file (`$FILE.qt`),
   - the number of instructions to simulate (1000),
   - how often it should print on stdout its progress (once every 100 instructions),
   - when certain counters should be reset, (1 = not reset),
   - a name X, use for two files X.pipe and X.results, (`$FILE`),
   - `-inf_all` to ignore the memory hierarchy which assumes all cache accesses hit L1 caches. This avoids having hundreds of wasted cycles in the pipeline visualizer but must of course be used with caution.
   - `-p 1` visualize based on real instructions (and not e.g. cycles),
   - `-b 1` begin visualization at instruction 1,
   - `-e 2000` end visualization at instruction 2000 (which is more than enough).

3. Next look at the program a.c. First make a rough guess how many clock cycles each iteration takes, and then use the simulator and scrollpv to check in the next few tasks.

4. Create a trace file with ./t a

5. Disassemble the a.out with the command objdump -d a.out > x

6. Copy the directory lab5 (at least the files x a.config a.pipe and a.results) to your computer from Power.

   Open x in an editor and search for the instructions in the main-function. You can find them using the pattern:

   ```
   main>:
   ```

7. You should see the lines:

   ```
   10000420:   02 10 40 3c     lis     r2,4098
   10000424:   00 7f 42 38     addi    r2,r2,32512
   10000428:   64 00 c0 38     li      r6,100
   1000042c:   00 00 00 60     nop
   10000430:   78 87 e2 38     addi    r7,r2,-30856
   10000434:   00 00 00 60     nop
   10000438:   38 81 02 39     addi    r8,r2,-32456
   1000043c:   00 00 00 60     nop
   10000440:   58 84 42 39     addi    r10,r2,-31656
   10000444:   00 00 20 39     li      r9,0
   10000448:   a6 03 c9 7c     mtctr   r6
   1000044c:   14 00 00 48     b       10000460
   10000450:   00 00 00 60     nop
   10000454:   00 00 00 60     nop
   10000458:   00 00 00 60     nop
   1000045c:   00 00 42 60     ori     r2,r2,0
   10000460:   ae 4c 08 7c     lfdx    f0,r8,r9
   10000464:   ae 4c 8a 7d     lfdx    f12,r10,r9
   10000468:   2a 60 00 fc     fadd    f0,f0,f12
   1000046c:   ae 4d 07 7c     stfdx   f0,r7,r9
   10000470:   08 00 29 39     addi    r9,r9,8
   10000474:   ec ff 00 42     bdnz    10000460
   10000478:   00 00 60 38     li      r3,0
   1000047c:   20 00 80 4e     blr
   ```

   The loop starts at 10000460 and ends at 10000474 so six instructions per iteration.

8. Start `scrollpv` (it is located in `sim_ppc.tar` in tresorit) and in the upper left corner select the File menu and open the file a.pipe (you can have four different traces opened at a time).

   You will see a matrix filled with dots and some characters. The X-axis is time or more exactly clock cycles and the Y-axis is instructions.

9. In `scrollpv` and the column Mnemonic you should see the same instructions (except that `lis` and `li` are not seen since they are so called synthetic instructions and actually variants of the `or` instructions).

   You can also see that each store instruction `stfdx` appears twice and this is due to it is split into two internal operations: one for adding the registers to produce the data address and another for writing to memory.

   In the matrix, you can see what happens to each instruction each clock cycle. You can see lots of D which mean the instruction is in one of the decode pipeline stages.

   You can also see that the floating point add fadd has many E which mean execute, i.e. a floating point unit is actually working on the instruction.

   In the Info bar the character is explained which you can see when you hover the mouse over a character (such as an E).

   When the mouse is in the matrix and you press the left mouse button, you will see a crosshair that you can move around and easier find the clock cycle (at the top of the window) when an event happened.

   Use it to find the clock cycle when the first load lfdx left the reorder buffer of the pipeline, i.e. completed, which is marked as a C (it does not matter much which event you look for but the last is easy to find).

   Write down the cycle.

   Now calculate which internal operation the last floating point load instruction is (i.e. the Iop Id in the first column) using the number of iterations in the loop (see source code) and the number of internal operations in each iteration.

   Scroll down to this internal operation (i.e. instruction) and look a bit below to see that the main function is returning so that you know you found the last iteration (you should find the blr at address 1000047c — blr means branch-to-contents-of link register).

   Find the clock cycle when the load in the last iteration completed and calculate the number of cycles per iteration.

10. Now redo the same with the file b.c and explain why each iteration takes more clock cycles (not so easy as you might first think by counting the number of arithmetic operations in the source code, but not too difficult).

11. Now change the t script and use your intopt with the file in. Start tracing in pivot and view the result in scrollpv.

12. Look for the worst cases of pipeline stalls, i.e. red characters. Try to explain what they are caused by. This will differ between different programs, obviously, but the purposes are

- that you should see which arithmetic operations should be avoided, if possible, and
- you should get a better understanding of what happens in pivot.

Warning: an important general rule about tuning floating point software: even if you identify an optimization which is mathematically equivalent, it might not work with floating point types due to limited precision, which means testing on a few examples only is a bad idea.

13. Finally, what is a rename register, and do both speculative and non-speculative execution need rename registers and why?

Lab 6: Optimizing compilers

The purpose of this lab is to learn more about the optimization options of GCC and optimizing compilers in general. We will use a benchmark called the Stanford benchmarks, used for the MIPSX project in the beginning of the 1980's. The input sizes have been adapted for our POWER8 machine, however.

Online documentation about the optimization options of GCC is available at:

`https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

1. Login to power.cs.lth.se and start bash in your terminal (to make sure the script below work).
2. Copy lab6 from

   ```
   /usr/local/cs/edag01/labs/lab6
   ```

3. Go to lab6
4. The first task in the lab is to determine which optimization levels have any additional effect with gcc. Compile the program mipsx.c with different optimization levels and save each relocatable file, using the command:

   ```
   for x in s 0 1 2 3 4 5
   do
   gcc -c -O$x -o $x.o mipsx.c
   done
   ```

   The for loop is equivalent to writing the following commands:

   ```
   gcc -c -Os -o s.o mipsx.c
   gcc -c -O0 -o 0.o mipsx.c
   gcc -c -O1 -o 1.o mipsx.c
   gcc -c -O2 -o 2.o mipsx.c
   gcc -c -O3 -o 3.o mipsx.c
   gcc -c -O4 -o 4.o mipsx.c
   gcc -c -O5 -o 5.o mipsx.c
   ```

   Level zero is the same as no optimization. To determine the size of the instructions in a file, the command size is used:

   ```
   size --common *.o
   ```

   The size of the read-only section is printed below text and includes instructions and constants. Global variables which should be initialized to zero are listed under bss which means block started by symbol (for historical reasons).

   To see which variables bss refers to, type the command:

```
nm 0.o
```

Try to figure out what T, U, G (or D), and C mean.

The explanation is given in the manual page for nm. Type:

```
man nm
```

What does `-Os` mean? All files compiled at levels 3–5 should have identical sizes.

To compare whether two files are identical, use the command diff file1 file2

If diff prints no output, the files are identical.

5. Now run the following shell command:

```
for x in s 0 1 2 3
do
gcc -O$x -o $x mipsx.c tbr.s timebase.c
./$x
done
```

POWER processors have a special register called the timebase register which can be used for very accurate timing (avoid making a system call to ask the operating system kernel what time it is, which normally is what happens when you use the ISO C `clock` function).

You can check the current CPU frequences using:

```
cat /proc/cpuinfo
```

The files `tbr.s` and `timebase.c` are used for this. Note that the POWER8 processor sometimes changes its clock frequency but that is usually not a big problem. Run your programs a few times to see if it appears to remain the same.

Then run the programs optimized at the different levels and note whether they improve in speed.

6. The remaining compilations should use `-O3` and some other flags. It is sometimes useful to tell the compiler for which pipeline it should optimize that code such as with `-mcpu=power8`. It has no effect on gcc on our machine but the IBM xlc compiler can sometimes produce different code.

Try now to figure out if gcc could vectorize the mipsx.c program.

To understand what happended, type the following:

```
objdump -d mipsx > x
```

This disassembles the program and writes to the file x. Edit that file and search for `stvx` which is the machine instruction for storing a vector register to memory. Can you find any? Many? Can you see many other vector instructions? Examples of other vector instructions are `vmaddfp`, `vperm`, and `lvx`.

7. Feedback-directed optimizations uses statistics (usually about branches) from previous executions and exploits that information when optimizing a file.

   Read about the options -fprofile-generate and -fprofile-use online or on page 208 of the course book. Compile with:

   ```
   gcc -O3 -fprofile-generate mipsx.c tbr.s timebase.c
   ```

   then run the program and then recompile with:

   ```
   gcc -O3 -fprofile-use mipsx.c tbr.s timebase.c -mcpu=power8
   ```

   What is the effect and what can have happened? You need to run the programs a few times.

   Related to `-fprofile-generate` are the options used in the following command:

   ```
   gcc -g -fprofile-arcs -ftest-coverage mipsx.c tbr.s timebase.c
   ```

   Compile and run. These options do not give feedback to GCC but rather to you. The command:

   ```
   gcov mipsx.c
   ```

   will produce a file `mips.c.gcov` with statistics about how many times each line was executed. Adding -b will in addition print statistics about branches, as explained on page 224 in the book.

   A recent addition to GCC is link-time optimization. It means that optimization is performed during linking, i.e. when all relocatable files of a program are available. Issue the following commands:

   ```
   cat a.c b.c
   gcc -O3 a.c b.c
   objdump -d a.out > x (dump the file and disassemble)
   ```

   Then open the file x to see what the function main looks like.

   Search for

   ```
   main>:
   ```

Redo the same thing but add `-flto` to GCC. Check what main now was compiled to!

8. Now take the matrix multiplication program `clang-matmul.c` and try to get clang to vectorize the inner loop. Is there any array reference which would make it difficult to put the elements in a vector register? If so, what can you do about it? What else would be affected by that change?

   You should modify the `clang-matmul.c` file!

   Compile with:

   ```
   clang -O3 clang-matmul.c tbr.s timebase.c -mcpu=power8 && ./a.out
   ```

   Verify that the code was vectorized: again look for a "v" instruction and "mul", e.g. `xvmulsp`.

   Next compare clang, gcc, IBM xlc, and Nvidia pgcc on the original matmul.c

   ```
   clang -O3 matmul.c tbr.s timebase.c -mcpu=power8 -o matmul.clang

   gcc -O3 matmul.c tbr.s timebase.c -mcpu=power8 -o matmul.gcc

   xlc -O3 matmul.c tbr.s timebase.c -mcpu=power8 -o matmul.ibm

   pgcc -O3 matmul.c tbr.s timebase.c -tp=pwr8 -o matmul.nvidia

   echo -n "clang: "; ./matmul.clang

   echo -n "gcc: "; ./matmul.gcc

   echo -n "ibm: "; ./matmul.ibm

   echo -n "nvidia: "; ./matmul.nvidia
   ```

   You can read more about IBM xlc at:

   `https://www.ibm.com/products/xl-cpp-linux-compiler-power`

   You can learn more about optimizing compilers at:

   `https://cs.lth.se/edan75`

   This course will probably be given next time 2023 in LP3 but if you want to take it as a self-study course (with help from Jonas, youtube lectures, and discussions on Discord, you can do it in LP3 or LP4 in 2022).