# Homework 2: Profiling Serial Merge Sort

**Due:** 10:00 P.M. (ET) on Friday, January 31, 2025

(Last Updated: January 20, 2025)

*In this homework you will be introduced to important profiling tools. You'll learn to use Perf, which gives you details about where time is being spent in your program, and Cachegrind, a cache and branch-prediction profiler in the Valgrind tool suite. These tools will help you identify parts of your code that would benefit from further optimization. Finally, you'll use these tools to optimize a serial merge sort routine.*

*Generally, when you are concerned about the performance of a program, the best approach is to implement something correctly and then evaluate it. In some cases, many parts of this initial implementation (or even the entire thing) may be adequate for your needs. However, when you need to improve performance, you must first decide where to focus your efforts. This is where profiling becomes useful. Profiling can help you identify the performance bottlenecks in your program.*

## Contents

# 1 Getting started

### *Getting the code*

You can obtain this assignment's code using Git to clone your repository from Github:

```
$ git clone git@github.com:CSE491-spring25/homework2_<your_netid>.git homework2
```

### *Updating tools*

**Do not skip this step!** Homework 2 requires some updated versions of scripts on your VM. Before moving forward with this lab, you should run:

```
$ cd /path/to/student_software/
$ git pull
$ ./scripts/install_telerun.sh
```

### *Code structure*

The code for the recitation is in the `recitation/` directory. You'll be looking at the source code in `isort.c`, which contains an insertion sorting routine, and `sum.c`, which allocates a large array of integers and sums a sample of them. The program you are responsible for improving is in the `homework/` directory.

### *Building the code*

In general, `make <target>` will build the code for a specific target binary. To build with debugging symbols and assertions — which are useful for debugging with `gdb` — you must build with

```
$ make <target> DEBUG=1
```

**Note:** Please ensure that the `Makefile` uses `clang-spe` as the compiler. This is done by changing the `CC` parameter to `CC := clang-spe` in the `Makefile`.

**Whenever a question asks about performance, we want you to run your programs on the job machines using `telerun`.**

### *Submitting your solutions*

Submit your write-up on Gradescope and your code via Git by the deadline stated at the top of this handout. For grading, we will use the code on the main branch of your personal Git repository at the time of the submission deadline. Remember to explicitly add new files to

your repository before committing and pushing your final changes, e.g., using the following Git commands:

```
$ git add
$ git commit -a
$ git status
$ git push
```

If `git status` shows any modified files, then you probably haven't checked your code into your repository properly. We recommend you commit often.

## 2 Recitation: Perf and Cachegrind

First, we will explore how to use two extremely useful performance profiling tools: Perf and Cachegrind. These tools allow you to identify performance bottlenecks in a program and measure salient performance properties, such as cache and branch misses. Used correctly, they can help you figure out exactly why your code is running slow.

For the checkoffs on Perf and Cachegrind (Checkoff Items 1 and 2), note your answers in the recitation 2 assignment on Gradescope and discuss them with your TA.

### 2.1 Perf

Perf is a performance profiler supported by Linux systems based on kernel version 2.6 and newer. Perf uses sampling to gather data about important software, kernel, and hardware events in order to locate performance bottlenecks in a program. Perf can generate a detailed record of where the time is spent in your code.

You can use the Perf commands `perf record` and `perf report` to analyze the performance of your program. The `perf record` command generates a record of the events that occur when you run your code, and `perf report` allows you to view this record interactively.

**Note:** For `perf` to relate its report to your source code, you must pass the `-g` flag to `clang` when compiling your program. This flag, which does not affect performance, tells the compiler to generate ***debugging symbols*** for your program, allowing tools like `perf` and `gdb` to associate lines of machine code with lines of source code. The provided Makefile will compile your program with the `-g` flag if you run `make` with `DEBUG=1`.

To record performance events on a program, run the following:

```
$ perf record <program_name> <program_arguments>
```

You can then view the recorded performance profile by running `perf report` from the same directory:

```
$ perf report
```

**Note:** You may see a warning screen that reads something like "Kernel address maps were restricted." This warning is safe to ignore, and pressing any key will let you continue to the report.

Your personal VM may run slower and provide less consistent performance results than the dedicated job machines, which are accessed via `telerun`. You can profile performance on the job machines by combining `telerun` with `perf` as follows:

```
$ telerun perf record <program_name> <program_arguments>
```

This command will generate a perf report on your personal instance, and you can use `perf report` to view these reports:

```
$ perf report
```

If you compiled your program with debugging symbols, using the `-g` flag, you can see both the C code and the assembly instructions in the annotated output. If you only see assembly instructions in the annotated output, then the program might not have been compiled with debugging symbols. The performance counter events are usually associated with the instruction right after or a few instructions below the one that causes extra stalls.

File `isort.c` in `recitation/` contains an insertion sort routine; `cd` into `recitation/` and compile the program using `make` as follows:

```
$ make isort
```

Now, running `./isort n k` will sort an array of `n` elements `k` times. Use Perf with the TELERUN utilities to profile the code as follows:

```
$ telerun perf record ./isort 10000 10
$ perf report
```

Spend some time exploring the contents of the report and try to figure out what it means.

Try to identify where the program is spending the most time. Note that, due to the interrupt-based sampling mechanism Perf uses to create the report, Perf only approximately measures the amount of time spent performing individual instructions. Hence, some interpretation will be required on your part.

**Checkoff Item 1:** Make note of one performance bottleneck.

## 2.2 Cachegrind

Cachegrind is a cache and branch-prediction profiler within the Valgrind tool suite. Recall from class that a read from the L1 cache can be around 100× faster than a read from RAM! Optimizing for cache hits is a critical part of performance engineering.

On virtual environments, hardware events providing information about branches and cache misses are often unavailable, so `perf` may not be helpful. Cachegrind simulates how your program interacts with a machine's cache hierarchy and branch predictor and can be used even in the absence of available hardware performance counters.

You can use Cachegrind to identify cache misses, branch misses, clock cycles, and instructions executed by a given program using the following command:

```
$ valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes <binary> <binary args>
```

**Note:** Although `valgrind --tool=cachegrind` measures cache- and branch-predictor behavior using a simulator, it bases its simulation on the architecture of the machine on which it is run. You should expect different results when running on different machines, e.g., when running on your personal VM versus the job machines.

File `sum.c` contains a program that allocates an array of $U = 10$ million elements, and then sums $N = 100$ million elements chosen at random. Make this program for your native machine (not `telerun`) as follows:

```
$ make sum LOCAL=1
```

**Checkoff Item 2:** Run `./sum` (with no program arguments) under `cachegrind` to identify cache performance. Cachegrind may take while to run. In the output, look at the D1 and LLd misses. D1 corresponds to L1, the lowest-level cache, and LL represents the last (highest) level data cache, which is L3 on most machines. Make note of the following. Do these numbers correspond with what you would expect? Try playing around with the values `N` and `U` in `sum.c`. How can you bring down the number of cache misses?

**Hint:** You can find information about your CPU and its caches using `lscpu`.

**Checkoff Item 3:** If you haven't already, take this time to discuss with your Project 1 partner about the team contract and how you plan to collaborate. Remember, all team members must submit a copy of the team contract to Gradescope.

## 3 Homework: Sorting

Profiling code can often give you valuable insight into how a program works and why it performs the way it does. In this exercise, we have provided a simple implementation of merge sort in `homework/sort_a.c`. You can build the code for all sort implementations by just typing `make`. After building, run the code with `./sort <num_elements> <num_trials>`.

**Write-up 1:** Compare the Cachegrind output on the unoptimized `DEBUG=1` code versus `DEBUG=0` compiler-optimized code. Explain the advantages and disadvantages of using instruction count as a substitute for time when you compare the performance of different versions of this program.

**Make sure you're evaluating the non-debug version for the rest of your experiments.**

In this homework, you will perform a sequence of tasks to identify performance bottlenecks and incrementally improve the performance of the merge sort routine. As you make improvements to the sort routine in each task, we would like you to keep a copy of the sort routine before the improvements and keep adding new versions of the sort routine in the testing suite (with different names), so we can profile and compare the performance of different versions. The

testing code is set up so that you can easily add new sorting routines to test, **as long as you keep the same function signature for the sort routine**. In particular, each sort routine you make should have the same type as `sort_a` provided in `sort_a.c`. Furthermore, **do not remove the `static` keyword in any of the internal functions, preserve the structure of the merge sort algorithm as coded, and do not change it radically for this homework**.

## 4 Function inlining

For the first task, we shall study how much function inlining can improve the performance of this sorting code. Copy over the code from `sort_a.c` into `sort_i.c`, and change all the routine names from `<function>_a` to `<function>_i`. We want you to have a chance to experiment with how inlining functions can affect code performance. However, the compiler will perform function inlining as an optimization by default. **In order to turn off this optimization, and allow you to inline functions explicitly, add `-fno-inline` to the `CFLAGS` variable in the Makefile**.

Use the `inline __attribute__((always_inline))` keywords to inline one or more of the functions in `sort_i.c`.

For example, to inline a function `void foo()`, change its prototype to:
`inline __attribute__ ((always_inline)) void foo()`.

To add the `sort_i` routine to the testing suite, uncomment the line in `main.c`, under `testFunc`, that specifies `sort_i`. Profile and annotate the inlined program.

> **Write-up 2:** Explain which functions you chose to inline and report the performance differences you observed between the inlined and uninlined sorting routines.

The compiler does not always inline functions with the `always_inline` attribute. For example, try to inline the functions in `util.c`. Profile and annotate your code, and check whether these functions have been inlined or not.

You'll notice that they haven't, because the compiler will not inline functions across translation units (i.e. different `.c` files) by default. In order inline across translation units, a feature called "link-time optimization" must be enabled. Add `-flto` to the `LDFLAGS` variable in your Makefile, and then recompile `sort`. Compare the performance of your sorts with link-time optimization enabled versus without. Note: normally you would add `-flto` to both `CFLAGS` and `LDFLAGS` as both the compilation and linking phases must be aware of the desire to do link time optimization, but in this case the Makefile creates an executable straight from the source files without saving intermediary object files and passes both `LDFLAGS` and `CFLAGS` in a single command.

**Write-up 3:** How much of a performance difference does link-time optimization make? Explain how you know that the functions in `util.c` have been inlined based on the annotated perf report.

Return your Makefile to normal for the rest of this assignment by removing the `-fno-inline` and `-flto` flags that you added.

# 5 Additional optimizations

**Note:** For this section, we only ask that you complete at least **two** of the three exercises below. You may disregard the write-ups for the exercises that you choose not to complete.

## 5.1 Coarsening recursion

The base case of the recursion of the `sort_a` routine in `sort_a.c` just sorts one element. You decide to coarsen the recursion so that the base case sorts more than one element. As before, copy all your changes into `sort_c.c`, and update your function names. Use the fastest correct sort routine so far as a basis for `sort_c.c`. Then coarsen the recursion in `sort_c.c`. We have provided an insertion sort routine in `isort.c`, which you can use as the sorting algorithm in your base case. You can also write your favorite sorting algorithm and use it for your base case. Remember that in order to use a function defined elsewhere, you first need to write a function prototype. Include `sort_c` into your test suite.

**Write-up 4:** Explain what sorting algorithm you used and how you chose the number of elements to be sorted in the base case. Report on the performance differences you observed.

## 5.2 Reducing memory usage

Observe that two temporary memory scratch spaces, `left` and `right`, are used in the `merge_a` function. You want to optimize memory by using just one temporary memory scratch space and using the input array to be sorted as the other memory scratch space during the merge operation. This change should reduce the total temporary memory usage by half. As before, copy your newest changes into `sort_m.c` and update the function names. Then, implement the memory optimization described above.

**Write-up 5:** Explain any difference in performance in your `sort_m.c`. Can a compiler automatically make this optimization for you and save you all the effort? Why or why not?

### 5.3 Reusing temporary memory

You find that it is unnecessary to allocate and deallocate the temporary memory scratch spaces `left` and `right` in the `merge` function every time it is called. Instead, you would like to allocate the required memory once at the beginning and deallocate it at the end after sorting is complete. To perform this optimization, copy your sorting routine into `sort_f.c`. Then, implement the memory enhancement described above in `sort_f.c` and include `sort_f` in your test suite.

**Write-up 6:** Report any differences in performance in your `sort_f.c`, and explain the differences using profiling data.

## 6 Turn-in

When you've written up answers to all of the above questions, turn in your write-up by uploading it to Gradescope and commit and push your code to your Git repository.

## 7 (Optional) Appendix: Other profiling tools

This section introduces you to a few tools that may make performance profiling easier and more productive.

Besides these, there are many more profiling tools that are available on the internet. We encourage exploring and trying out different tools to see what works best for you and your teammates. If you find anything that's particularly effective, please share it with the class on Piazza!

### 7.1 Google Perf Tools

Google has a suite of perf tools including `pprof`, a CPU profiler similar to Perf. The `pprof` tool provides various ways to view reports, including in graphical form.

#### *Installation*

To install, run the following in your VM and open a new shell:

```
$ sudo -i
$ su root -c  "apt-get install -y google-perftools libgoogle-perftools-dev"
```

#### *Usage*

To use `pprof`, you must add `-lprofiler` to your program's `LDFLAGS` in the Makefile (remember to remove it for performance benchmarking).

When you've compiled your program with this flag, you can dump profile data to a file like `pprof-data.out` like so:

```
$ CPUPROFILE=pprof-data.out ./<binary> <args>
```

Then, you can use `pprof` to visualize the output as graph in a PDF file:

```
$ google-pprof -pdf ./<binary> pprof-data.out <filename>.pdf
```

See https://gperftools.github.io/gperftools/cpuprofile.html for more information on how to use `pprof`.

### 7.2 Other tools

Other more sophisticated tools used to visualize profiling data include Hotspot (https://github.com/KDAB/hotspot) and KCachegrind (https://kcachegrind.github.io/html/Home.html).