Performance Engineering of Software Systems
Massachusetts Institute of Technology 6.172
Prof. Charles E. Leiserson and Prof. Julian Shun Handout 3

Homework 2: Profiling Serial Merge Sort

[Note: This assignment makes use of AWS and/or Git features which may not be available to OCW users.]

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 correct 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.

1 Getting Started

Code structure

The code for the recitation is in the `recitation` directory. You'll be looking at the files `isort.c`, which contains code for an insertion sorting routine, and `sum.c`, which allocates a large array of integers and sums a sample of them. For the homework, 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. Remember that to build with debugging symbols and assertions (useful for debugging with `gdb`), you must build with

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

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

Submitting your solutions

Remember to explicitly add new files to your repository before committing and pushing your final changes.

```
$ 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 the 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 slower than it should.

2.1 Perf

`perf` is a profiler tool for Linux 2.6+ based systems. It uses sampling to gather data about important software, kernel, and hardware events in order to locate performance bottlenecks in a program. It generates a detailed record of where the time is spent in your code.

Note: If `perf` is not installed on your AWS VM, run

```
$ sudo apt install linux-tools-common linux-tools-aws
```

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

Note: For `perf` to annotate your code properly, 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`, which allow tools like `perf` and `gdb` to associate lines of machine code with lines of source code. The provided Makefile will do this if you run it with DEBUG=1.

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

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

You can then view the results by running `perf report` from the same directory:

```
$ perf report
```

Note: You may see a warning screen that says something like "Kernel address maps were restricted." This is OK, just press any key to continue to the report.

The personal AWS machines are generally less performant (and less consistent) than the dedicated AWSRUN machines, which are accessed via `awsrun`. You can also profile performance in the cloud:

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

This will generate a report on an AWS cloud queue machine rather than on your personal instance, so you need to pass some extra arguments to `perf report` in order to view it. We provide the command `aws-perf-report` to do this for you:

```
$ aws-perf-report
```

Note: `aws-perf-report` should be run **without** `awsrun`.

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

You can see both the C code and the assembly instructions in the annotated output. 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` contains an insertion sort routine. Compile the program with

```
$ make isort DEBUG=1
```

Now, running `./isort n k` will sort an array of `n` elements `k` times. Run:

```
$ awsrun perf record ./isort 10000 10
$ aws-perf-report
```

Identify branch misses, clock cycles and instructions. Diagnose the performance bottlenecks in the program.

Checkoff Item 1: Make note of one bottleneck.

2.2 Cachegrind

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

On virtual environments like those on AWS, 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.

Here is an example on how to identify cache misses, branch misses, clock cycles, and instructions executed by your program using Cachegrind:

```
$ valgrind --tool=cachegrind --branch-sim=yes <program_name> <program_arguments>
```

Note: Although `valgrind --tool=cachegrind` measures cache and branch predictor behavior using a simulator, it bases its simulation upon the architecture on which it is run. You should expect different results when running on different machines, e.g. when running on your personal instance vs. the awsrun 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 it with:

```
$ make sum
```

Checkoff Item 2: Run `sum` under `cachegrind` to identify cache performance. It may take a little while. In the output, look at the D1 and LLd misses. D1 represents the lowest-level cache (L1), and LL represents the last (highest) level data cache (on most machines, L3). 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: to find information about your CPU and its caches, use `lscpu`.

Checkoff: Explain to a TA your responses to the previous two Checkoff Items.

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 make the code for all sort implementations by just typing `make`. After making, run the code with `./sort <num_elements> <num_trials>`.

Write-up 1: Compare the Cachegrind output on the `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.

We want to identify performance bottlenecks and incrementally improve the performance of the merge sort as explained in the tasks below. 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, i.e., each sort routine you make should have the same type as `sort_a` provided in `sort_a.c`. In addition, **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**.

3.1 Inlining

You would like to see how much inline functions can help. Copy over the code from `sort_a.c` into `sort_i.c`, and change all the routine names from `<function>_a` to `<function>_i`. Using the `inline` keyword, inline one or more of the functions in `sort_i.c` and `util.c`. 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 `inline` keyword. Use the annotated assembly code to help you verify whether the compiler really inlined the functions you wanted to inline.

Try to inline the `sort_i` recursive function (You may find forward declarations useful). When you inline a recursive function, it will expand the recursion only a fixed number of times, and then will recurse.

Write-up 3: Explain the possible performance downsides of inlining recursive functions. How could profiling data gathered using cachegrind help you measure these negative performance effects?

3.2 Pointers vs Arrays

You learn that pointer access can be more efficient than array indexing when manipulating arrays, and you want to implement this efficiency in the sorting algorithm, in addition to the inlining changes you did in the previous task. To accomplish this, you first copy your sort routine in `sort_i.c` into `sort_p.c` and update the function names in `sort_p.c` to `<function>_p`. Then you modify the code in `sort_p.c` to use pointers instead of arrays and include `sort_p` into the test suite.

Write-up 4: Give a reason why using pointers may improve performance. Report on any performance differences you observed in your implementation.

3.3 Coarsening

The base case in the recursion of `sort_p` routine in `sort_p.c` is sorting just one element. You want to coarsen the recursion so that the recursion 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 base 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 5: 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.

3.4 Reducing Memory Usage

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

Write-up 6: 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?

3.5 Reusing Temporary Memory

Though you have reduced temporary memory usage in `sort_m.c` by half, you find that it is unnecessary to allocate and deallocate the temporary memory scratch space in the `merge_m` function every time it is called. Instead, you would like to allocate the required memory once in the beginning and deallocate it at the end after sorting is done. To do this final 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 7: Report any differences in performance in your `sort_f.c`, and explain the differences using profiling data.