CS 650, Assignment #1 Final Report

Keith Dailey, Kevin Lynch, Nick D'Andrea

1 Introduction

The purpose of this assignment is to compare the performance of various implementations of the Fast Fourier Transform (FFT). In particular, we compare the iterative implementation presented in *Numerical Recipes in C* [5], a naïvely implemented recursive function which assumes the length of the input vector is a power of 2, an optimized version of the recursive function, an implementation presented in Section 6 of "How To Write Fast Numerical Code: A Small Introduction" [3] which assumes the length of the input vector is a power of 4, and FFTW version 2.15.

In the following sections, we summarize the performance of each of the algorithms, fully describe the experiments that were run to measure performance, and provide information of the computing platform on which all measurements were calculated.

2 Summary of Results

The results of the experiments clearly show that FFTW is the most efficient of the implementations that we tested. This is not surprising given the amount of work which has been put into this library and the fact that it has the ability to take advantage of the platform on which it is run, whereas the other implementations do not take their environment into consideration.

Perhaps the most interesting result is the great difference in the performance of the two versions of the recursive algorithm which we ourselves implemented. The naïve implementation of the algorithm performs an explicit permutation on the input vector, uses temporary memory allocation for various operations, and does not merge any loops together. The "optimized" version of this algorithm uses stride parameters so that explicit permutation is not necessary, does not use any temporary allocation, and merges the loops that perform the "twiddle" and "vector butterfly" operations into a single loop.

The performance plots presented in Section 4 show that the naïve implementation is clearly the most inefficient of the implementations, but our optimized version performs nearly as well as both the radix four version from [3] and the iterative algorithm from [5]. This result can best be seen in Figure 4 which shows the total number of instructions performed by each of the implementations when called 10 consecutive times on each of the input vector lengths.

Another surprising result is that our optimized version actually outperforms all other implementations with respect to certain measures. In particular, Figures 6 and 7 show that our optimized recursive function usually results in fewer cache misses than any of the other implementations, especially when we look at relatively long input vectors.

3 Computing Platform

All timings were run on `kodiak.cs.drexel.edu`, a quad-core dual-processor system with Intel Xeon E5335 CPUs running at 2.00GHz with 16GB of RAM. The E5335 CPU is a Clovertown processor based on the Intel Core microarchitecture, which is able to perform a 128-bit SIMD double precision instruction per cycle per processing core [7] The processing unit has a unit to perform floating point multiplication and division and another unit for floating point addition.

It has a 32kB L1 data cache, a 32kB L1 instruction cache, and a 4MB L2 cache for each pair of cores, resulting in a total of 8MB L2 cache [6]. The results from calibrator [4], a program that analyzes the cache-memory system and extracts details are shown in Figure 1. These graphs clearly indicate that the core does indeed have a 4MB L2 cache and a 32kB L1 data cache. The TLB is also correctly identified as having 256 entries in Figure 2. `PAPI_mem_info` [2] was able to provide further information. The L2 cache is 16 way set associative and the L1 caches are 8 way set associative.

The Memory Organization Benchmark [1] was also run and further supported the other tools. All of the system information gathered is in the `system_info` directory submitted with this report, notably in `output.txt`.

4 Summary of Experiments

4.1 Source Code

The source code for each of the algorithms (with the exception of FFTW) can be found in the included files. The file `four1.c` contains the iterative implementation from *Numerical Recipes in C*. The file `fft.c` contains our implementations of both the naïve and optimized versions of the the radix-2 FFT algorithm. Note that even though this source contains multiple implementations of these algorithms, the functions named `kd_fftr2` and `kd_fftr2_opt` are the only ones we measured for comparisons. The file `fft4.c` contains the FFT radix-4 code from "How To Write Fast Numerical Code: A Small Introduction".

The included `Makefile` can be used to properly compile object files from each of the above mentioned source files. These object files are used along with `time_fft.c` to compile the timing program `time_fft`, which we describe in the next section.

4.2 Test Functions

The file `test.c` contains a suite of functions which can be used to ensure that the various implementations of the FFT are working properly. For each test case, there are two functions, named `test`i and `check`i, for some integer i. The `test`i function creates an input vector of the desired size for which the output of the FFT is known. The `check`i function compares the output vector from that FFT that it is given to the expected result. The `time_fft` program can be used to test the implementations by specifying an $i > 0$ as the final command line argument.

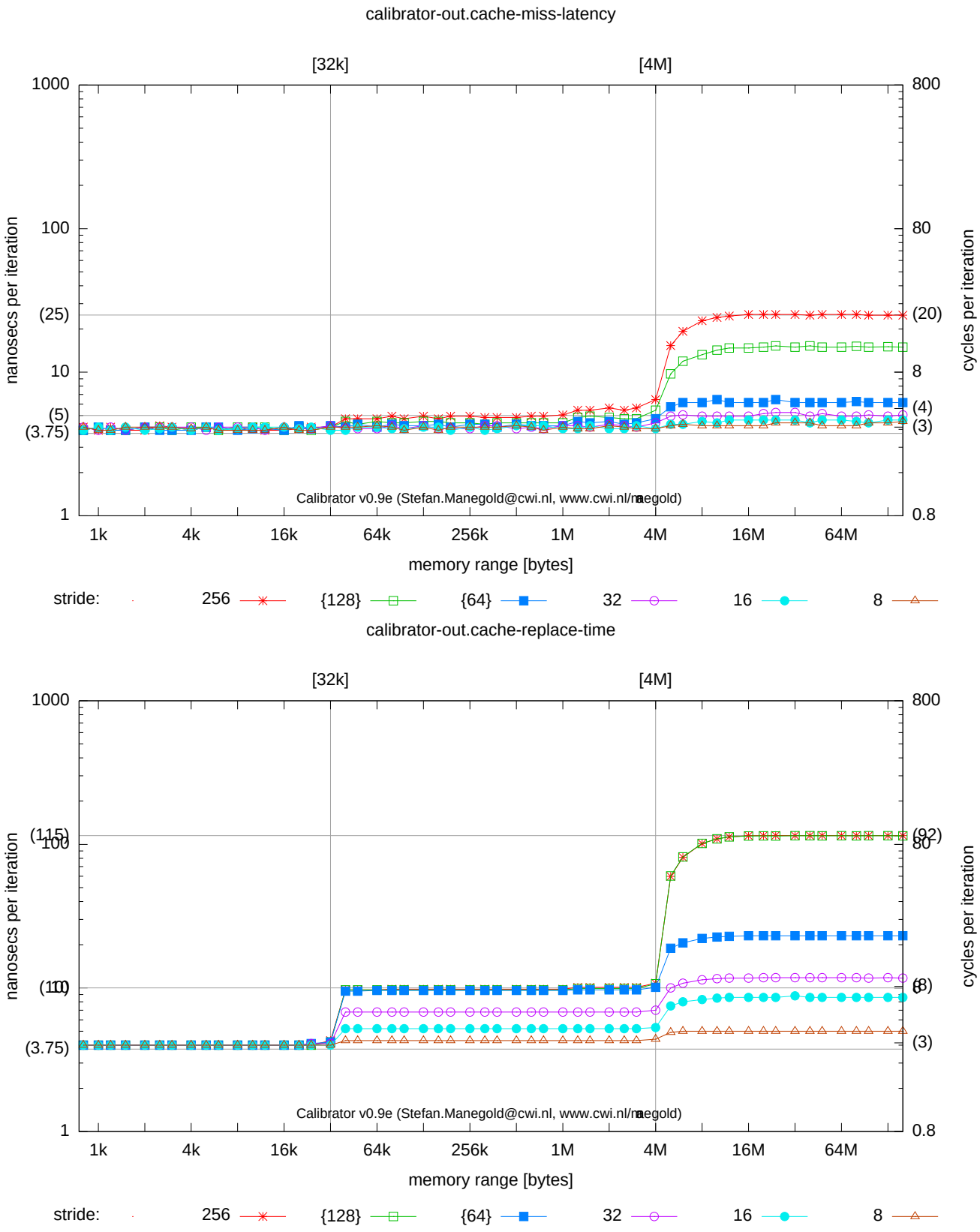

Figure 1: The Calibrator graphs indicating data cache sizes.

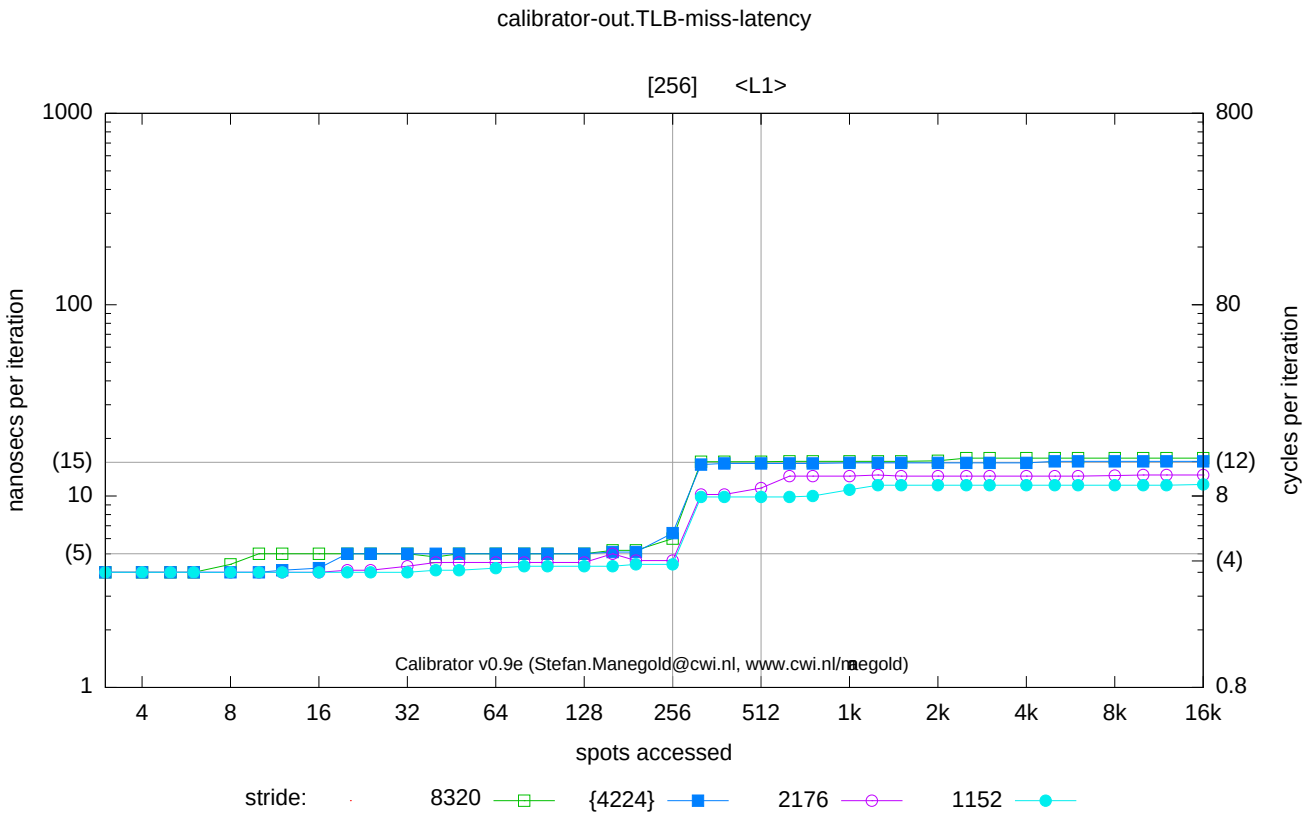

Figure 2: The Calibrator graph indicating TLB size.

4.3 Timing the Algorithms

In order to measure the performance of each of the algorithms, we used the Performance Application Programming Interface (PAPI) to record the following metrics for each experiment:

- `PAPI_TOT_CYC` - total cycles
- `PAPI_TOT_INS` - number of instructions completed
- `PAPI_FP_OPS` - floating point operations
- `PAPI_L1_DCM` - level 1 data cache misses

The file `time_fft.c` contains the main program which uses PAPI to compute and report these statistics. The included `Makefile` should properly compile all the source code and produce a program named `time_fft`. This program is used to generate the above statistics for the desired implementation, input vector size, and number of times to repeatedly call the indicated function. The program also performs a simple test on the implementation chosen to ensure that it has been implemented correctly. The output from the program is the average, minimum, and maximum values of each of the above statistics. For example, the following command generates these statistics when the naïve implementation of the FFT is called 10 times on a vector of length 2^5:

```
time_fft 5 10 4 0
```

The 4 in the above command tells the program which of the functions to call to perform the FFT. To obtain a usage statement for the program which details the allowable values for this argument, simply type `time_fft` at the command line. The 0 in the above command indicates which test function to run on the chosen function. This will be explained momentarily.

The timings which we present in this report were computed by running the above program on each of the implementations on input vectors of sizes 2^k, where k ranged from 1 to 24 (with the exception of the radix 4 code whose input sizes were of size 2^{2k}, where k ranged from 1 to 12). Each function was called 10 times on each input vector. The included shell script `time.sh` may be used to recreate the timing data, which is presented in its raw format in the file timings.txt.

4.4 Performance Plots

In addition to the raw data in timings.txt, we have also prepared performance plots for all of the data collected to give a visual comparison of the different algorithms. We include several of these plots here, but all plots we created have been included in the attached files.

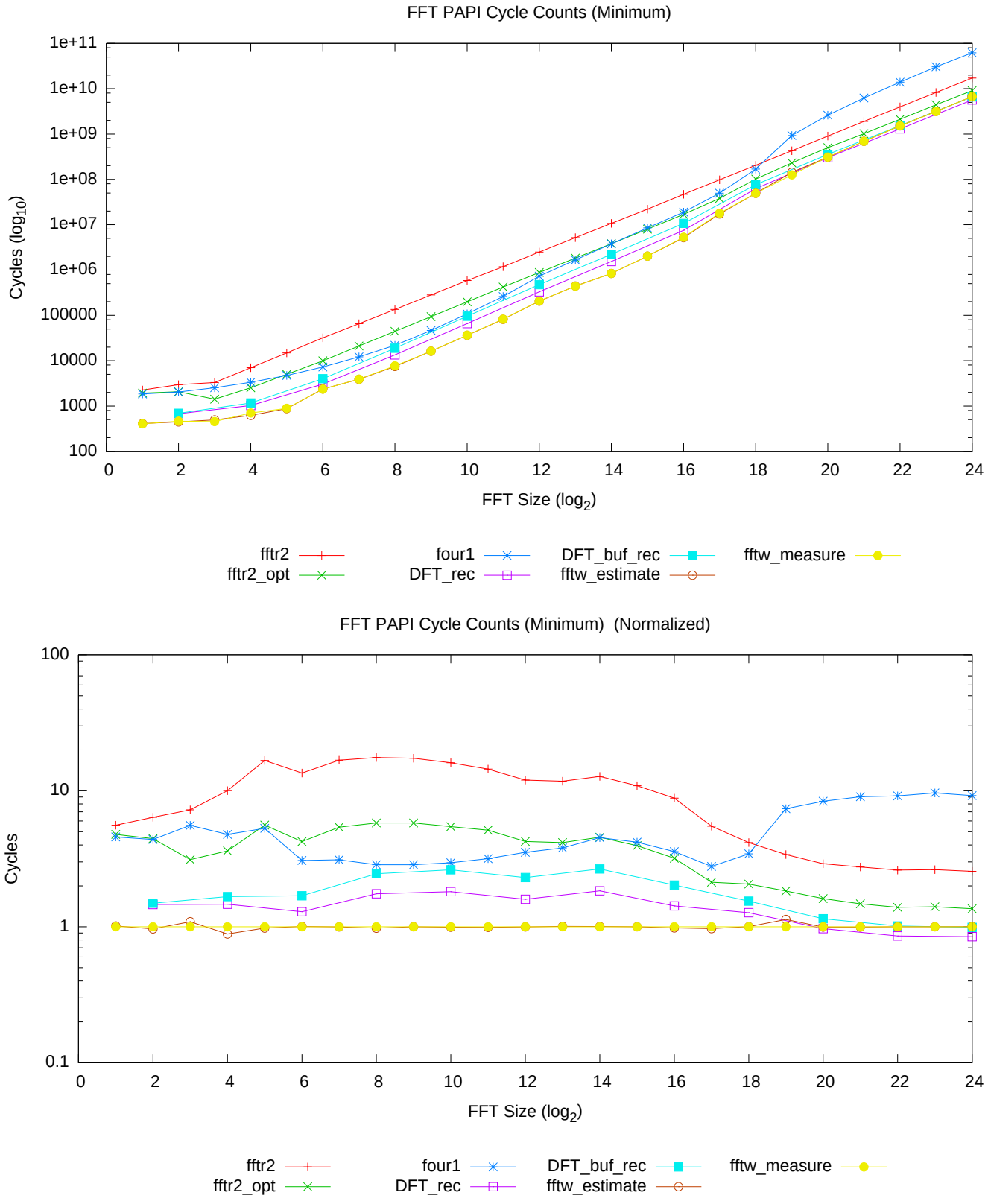

Figure 3: Plot summary of the minimum number of total cycles used by each algorithm when run on each input vector size 10 times.

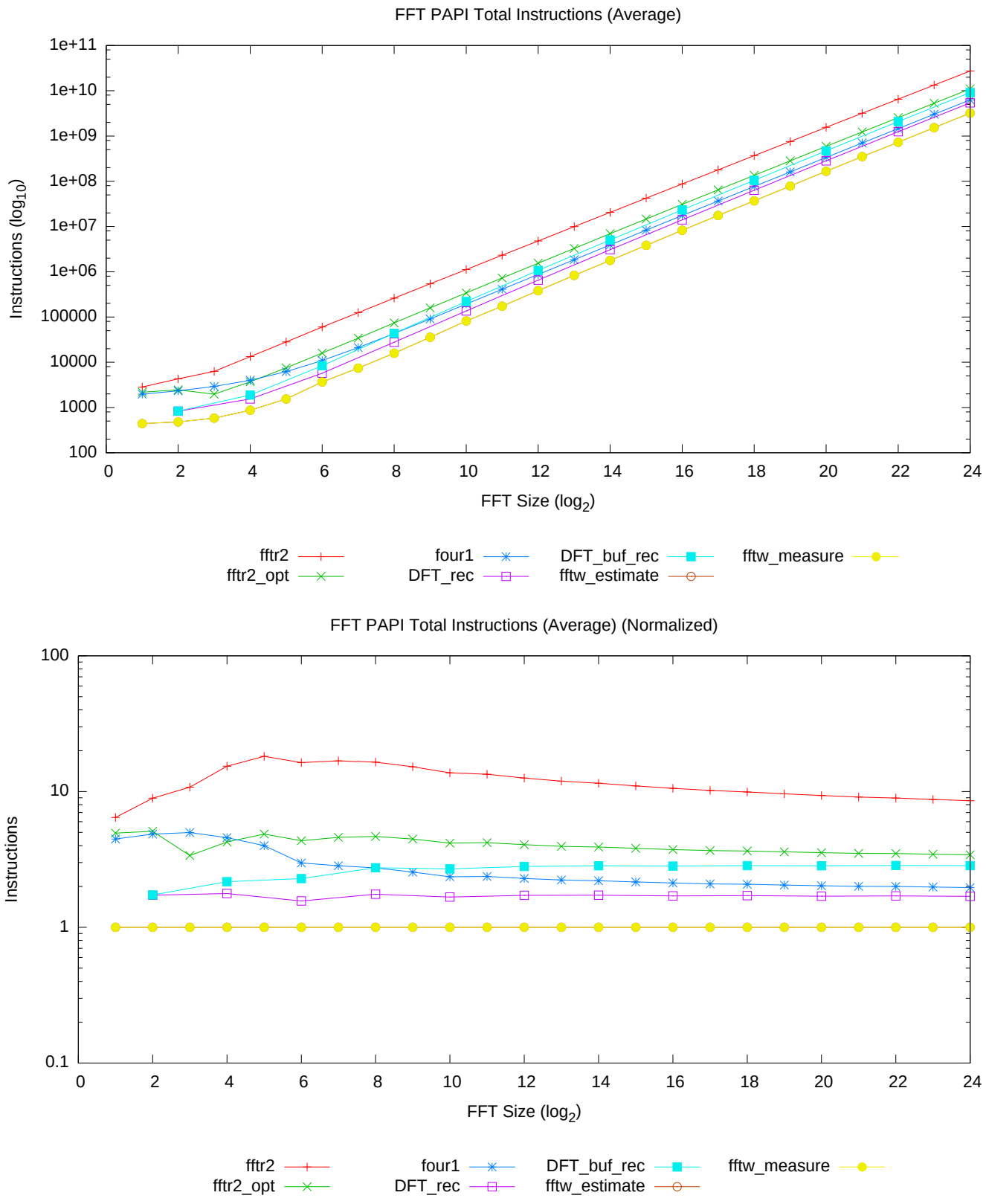

Figure 4: Plot summary of the average number of instructions issued by each algorithm over 10 iterations of each input size. The lower graph is normalized to the `fftw_measure`.

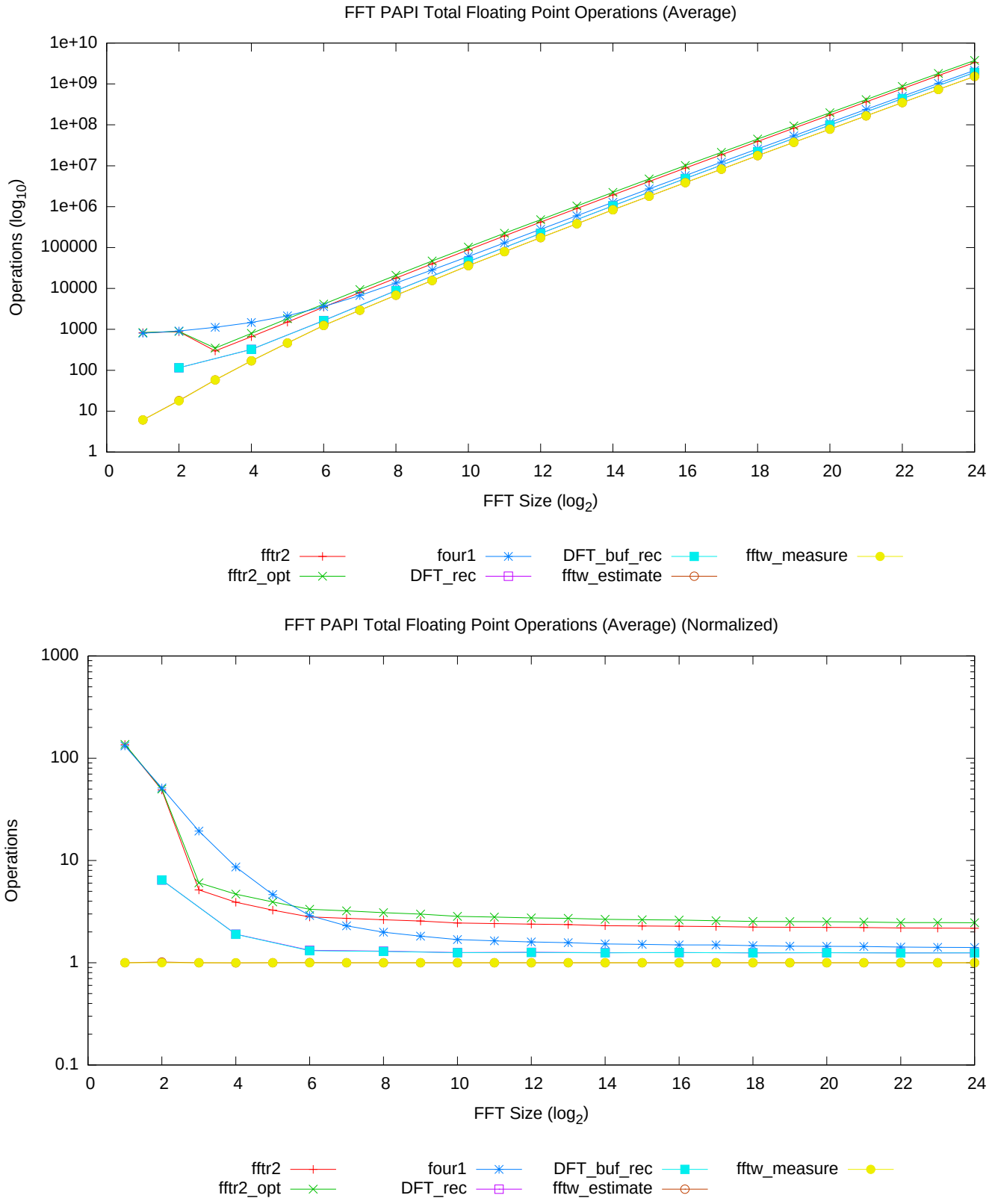

Figure 5: Plot summary of the average number of floating point operations issued over 10 runs of each algorithm. The lower graph is normalized to the `fftw_measure`. No major difference occurs when choosing the minimum instead of the average.

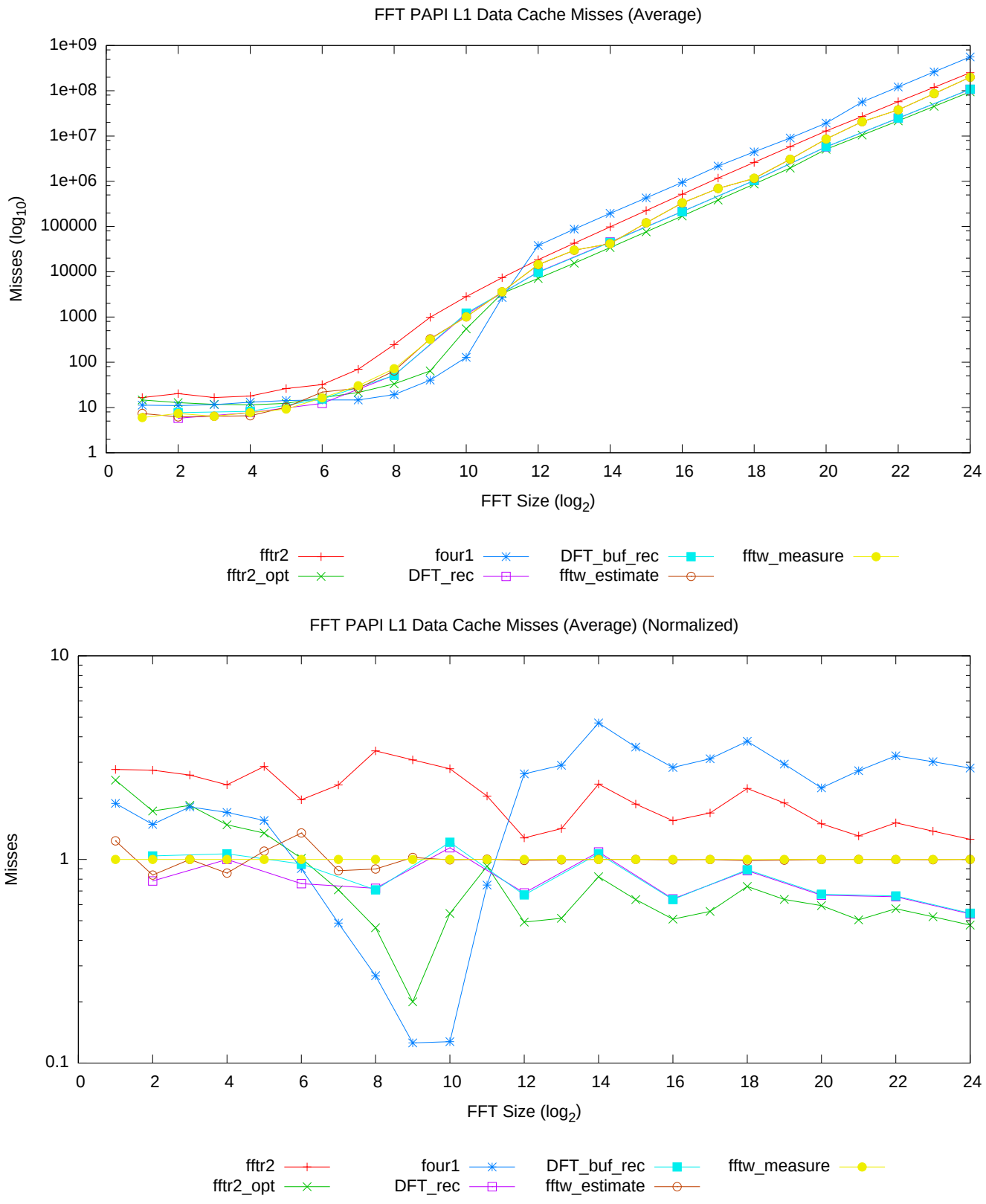

Figure 6: Plot summary of the average number of cache misses encountered over 10 runs of each algorithm on large input sizes.

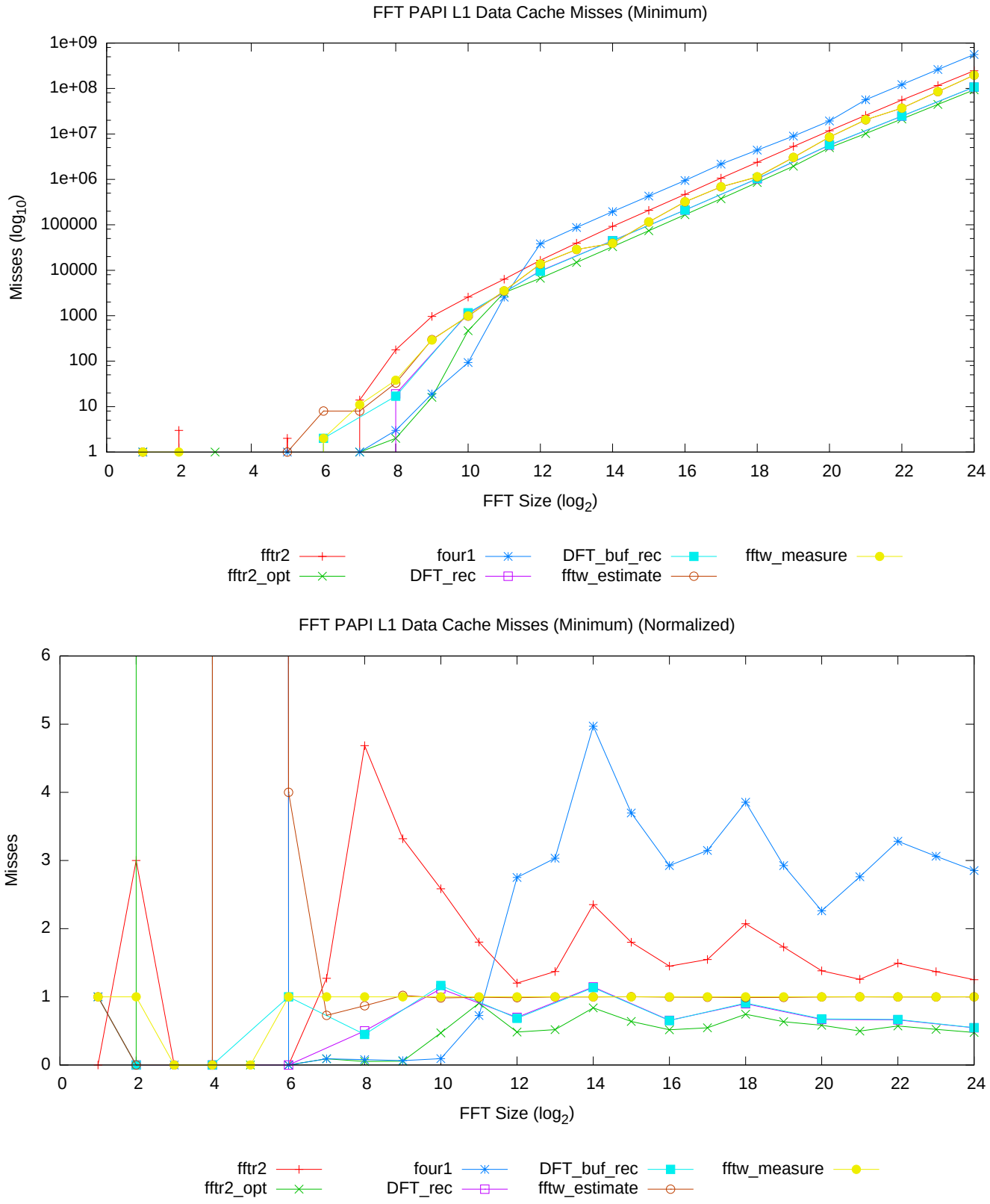

Figure 7: Plot summary of the minimum number of cache misses encountered over 10 runs of each algorithm on large input sizes. The spikes in the normalized graph are due to the FFTW algorithm getting 0 cache misses during a run.

5 Conclusion

As the performance plots in the previous section show, a straightforward implementation of the FFT performs poorly relative to existing implementations. However, using this implementation as a starting point, and then optimizing it by merging loops, removing all temporary allocation, and adding a stride parameter to remove the need for explicit permutation of the input vector results in a recursive algorithm whose performance is on par with the performance of the canonical iterative implementation presented in [5].

Furthermore, we have also seen how poorly the iterative algorithm performs against the finely-tuned recursive algorithm FFTW. This result shows the power of carefully implementing the algorithm from the Cooley-Tukey Factorization as it is more straightforward than understanding the iterative algorithm.

References

[1] MOB: Memory Organization Benchmark `http://www.nmsl.cs.ucsb.edu/mob/`.

[2] Performance Application Programming Interface `http://icl.cs.utk.edu/papi/`.

[3] Srinivas Chellappa, Franz Franchetti, and Markus Püschel. How To Write Fast Numerical Code: A Small Introduction. In *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 5235 of *Lecture Notes in Computer Science*, pages 196–259. Springer, 2008.

[4] Stefan Manegold. Calibrator `http://www.cwi.nl/~manegold/Calibrator/`.

[5] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[6] A.J. van der Steen. Evaluation of the Intel Clovertown Quad Core Processor. Technical Report HPCG-2007-02, Utrecht University, High Performance Computing Group, April 2007. (Also available through `http://www.euroben.nl`, directory reports).

[7] Ofri Wechsler. Inside Intel Core Microarchitecture – Setting New Standards for Energy-Efficient Performance Intel Whitepaper `http://download.intel.com/technology/architecture/new_architecture_06.pdf`. 2006.