HALMILTON SEGMENTER FOR R PEAK DETECTION

Robin Berner, Hanjing Gao, Tom Kuchler, Xiaying Wang

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Our project aims to improve the performance of an ECG analysis system of detecting beat and classifying arrhythmias [1]. With the performance gain, the application can potentially process more user data in a shorter amount of time without compromising on accuracy or precision. We employed optimization techniques such as blocking, vectorization, unrolling, pipelining, etc to achieve a 3.46x gain in terms of flops per cycle.

1. INTRODUCTION

Motivation. Electrocardiogram (ECG) signals represent the electrical activities of the heart. Abnormal ECG signals are usually a sign of cardiac diseases or failures. Being able to correctly detect the ECG signals and classify the arrhythmic patterns is crucial in diagnosing a patient especially with life-threatening conditions. Most ECG analysis tools are proprietary and are part of a larger medical diagnostic system. The base implementation of the project is one of the very few open source ECG analysis programs that aims to help the medical industry and research reduce duplicated effort on developing ECG analysis software. Not only does this benefit companies who can now shorten their development-to-market cycle, but also researchers who can afford to spend more effort in creating new diagnostic and corresponding treatment methods.

With the advancement of Internet of Things and embedded chips, portable heart rate monitors running ECG analysis software are now made possible. The performance of real-time recording, analyzing and diagnosing one's ECG signals using portable devices plays a vital role in a patient's health in case of emergencies. The faster and more accurate the analysis runs, the quicker the patient in critical condition can get the suitable treatment. This motivates us to optimize the baseline implementation to reduce the total program runtime in order to provide better care for the targeted users.

Related work. Part of the baseline implementation can be traced back to an assembly-version of a QRS detector run on a Z80 microprocessor [2]. The original QRS detector was then ported and improved to C with modified sampling rate by Hamilton and Tompkins [3]. There have been various python implementations of the Hamilton QRS detector [4] all of which run orders of magnitude slower than their C/C++ counterparts.

2. BACKGROUND

ECG signal is a semi-periodic signal which reflects the electrical activities of the heart. It features specific patterns which are shown in Fig. 1.
The baseline implementation of our application consists of three main phases. As illustrated in Fig. 2. The first phase is an IIR notch filter which wasn't in the original code base. We implemented the notch filter from scratch, using the filter design tools from MATLAB. The notch filter gets rid of the 50 Hz noise which is present due to the humming of the grid power. The second stage is the Hamilton Segmenter which consists of a few other filters to clean up the ECG signal, a peak detector to detect both QRS peaks and non-QRS peaks and store the eight most recent ones in the corresponding buffer, and a variety of detection rules. Last but not least a morphology-based classifier is used to classify each beat as either normal, premature ventricular contraction (PVC), or unknown.

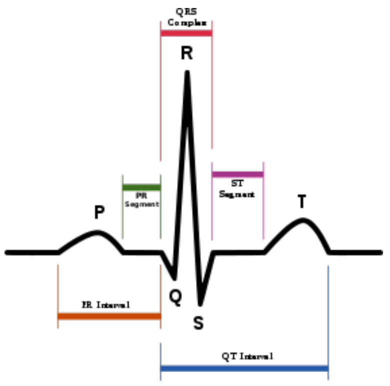

Fig. 1. Main features of ECG signal. (Source: Wikipedia)

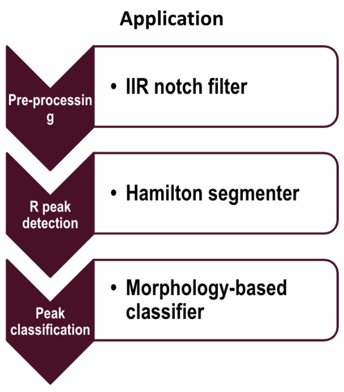

Fig. 2. Baseline Pipeline

IIR notch filter. The IIR notch filter is implemented as a 4-stage second-order infinite-impulse-response (IIR) digital filter which involves a feedback loop that creates data dependencies among the four stages.

QRS detector. QRS detector as shown in Fig. 3 consists of a low-pass filter with 16-Hz cutoff, a high-pass filter with 8-Hz cutoff, a derivative filter, a 80 ms averaging window and a set of detection rules to uncover the QRS complexes. A total of 5 detection rules is applied, the details of which can be found in [1].

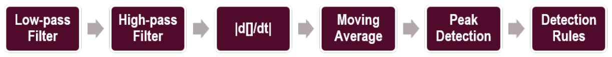

Fig. 3. QRS Detector Operations

Morphology-based classifier. The classification algorithm is based on morphological comparisons. It first checks for muscular noise, estimates the cardiac rhythm based on R-to-R interval, and analyzes the beat to eliminate the baseline shift. Secondly, it finds the best morphological match to a maximum of 8 stored templates, which are continuously updated in case a new template is found and it replaces the template which has occurred the fewest number of times in the last 500 beats. Consequently, it adapts the match limits to the matched type and the noise level, fetches the dominant type and classifies the new beat. Finally, it performs post-classification updates of the buffers.

Cost Analysis. For the first stage, the notch filter, and the filter part of the second stage, later called QRS filter, we can do a complete analysis of computational cost. We define the cost measure as the total of floating point additions, multiplications and divisions. The basic implementation of the Notch filter is implemented as four stages of second order systems (SOS). A single stage is illustrated in Fig. 4. The first stage takes the original sample as input and will feed its output as input to the second stage and so on.

From a cost standpoint it can be clearly seen, that one stage needs 6 floating point additions and 4 floating point multiplications to compute its output. As we have 4 stages, we need a total of 40 flops to compute the output of one sample. The total cost of n samples is accordingly 40n flops.

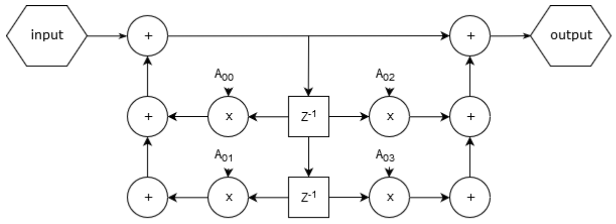

Fig. 4. General second order system

For the QRS filter we consider the first 4 blocks from Fig. 3. As already mentioned, the low- and high-pass filters are implemented as IIRs. The given design only used very simple filters, meaning few coefficients, resulting in a total cost per sample of 17flops for all 4 stages. This means for the application the QRS filter will amount to a total of 17n flops over an input of n samples.

For the application total cost, we will consider a different cost measure. Additionally to floating point additions, multiplications and divisions, we also count compare operations. In the QRS filter we already have one comparison for the moving average, this comparison was not accounted for in the above cost analysis. The reason for accounting comparisons for the overall application is that the parts after the notch and QRS filters depend very heavily on branching operations which are data dependent.
In order to get the overall cost, we instrumented our code with counters for floating point additions, multiplications, divisions and comparisons. We ran the application on 7200 samples of real ECG data provided by the online database of physionet[1]. It was necessary to use data from a real ECG signal and not a random signal, as many branches depend on the data and thus a random signal would not produce the cost that would be expected during real application. The results we got can be found in Table 1. This means that we have an average cost of

$$\frac{631'267 \text{ flops}}{7200 \text{ samples}} = 87 \text{ flops per sample}$$

3. PROPOSED METHOD

In this section we will explain the starting approach to the project, then we will split the whole application into parts, as we tried to optimize them separately. This approach was chosen to allow the work to be split between the members

[1]http://www.physionet.org/physiobank/database/mitdb

operation	counted during execution
additions	304’960
multiplications	209’668
divisions	41’015
comparisons	75’624
total	631’267

Table 1. table with overall cost

of the group. Finally, we re-combined every part and did further analyses.
Our starting point is the original open-source code provided by [1]. The code relies on the library provided by the Physionet dataset to fetch data from the website. For the sake of a future embedded application, we had to consider resource constraints, therefore our first step was to eliminate the dependency on the online library by saving the data locally. Another optimization step we used to improve locality was to block the input data, i.e. instead of processing end-to-end on one single sample, we process a whole block of data. Furthermore, the original source code consists of many small functions, therefore we proceeded with inlining these functions. Finally, we split the work and the details of each part are explained in the following paragraphs.

Notch filter. In the following we will go through the most relevant versions of Notch filter and explain what techniques from the lecture we applied.

The base version is a naive implementation of the 4 stage second order system. It is implemented using two for loops, the outer one over all samples and the inner one over the different filter stages. The inner loop multiplies the previous filter state with coefficients and adds them to the current filter input to produce the output of that stage. The filter state is marked as boxes with z^{-1} in Fig. 4. The inner loop also needs to update these state variables, to be able to compute the next sample.

As the different stages depend on the output of the last one, one idea was to change the loop order and block over some number of samples. This meant, we now had four independent loops over all samples computing the output of one stage and saving it into a buffer for the next stage to read or as output. We name this version the ’reversed’ implementation. Although the reversed loop order helped with the data dependencies between the filter stages, it was worse for the dependencies within each of the filter stages.

The next idea was to precompute the first filter stage once before the loop started. This can be seen like filling a pipeline, hence we give it the name ’pipelined’. This would allow the second stage to start computing before the first one is finished. This can also be done for the third stage by precomputing the first two outputs of the first and then the first output of the second, with the same logic we can extend this to the fourth stage. In concept this combined with scalar replacement should allow the compiler to achieve more instruction level parallelism.

We also tested SIMD to eventually get more speedup. The first idea was to use the pipelined version and expand it to SIMD (hence we named this version ’pipelined1’). As we have 4 filter stages, we tried to compute all 4 stages in paralell with 4-way SIMD instructions.

The last interesting approach was to compute the filter as matrix vector multiplication. Each filter can be represented as a matrix vector multiplication and also the total filter can be expressed as such. Since the filter was relatively small we wanted to try to express the entire filter in a 8x8 matrix that is multiplied with a 8 entry vector. We did this again with 4-way SIMD first. As the matrix had many zero entries and could be more nicely distributed into vectors with 4 entries.

QRS filter. As previously mentioned in the cost analysis, the QRS filter has two IIR filter in it, the low-pass and high-pass filters. Therefore we’re once again data dependent and face similar problems as in optimizing the notch filter in the previous subsection. In a first step we optimized the single loop iteration, by precomputing constants and inverse to turn divisions into multiplications. Next up there are multiple pointers to different ring buffer which get incremented every iteration and reset to 0 upon reaching the end of the buffer. We changed this logic from four if-statements to a vectorized version to remove some overhead in the computation of the pointers.
A different approach to get rid of this overhead was to unroll the loop. The calculation can be substituted by unrolling the loop and using fixed values. Since the ring buffers have length of 2, 5, 16 and 25, the smallest common multiple is 400. In order to not blow up the code and not restrict the input size to be a multiple of 400 to reach better performance, we tried unrolling only for three ring buffers. We took the three ring buffers with sizes 2, 5 and 25, meaning we had to unroll 50 times. We further tried to unroll 400 times in order to have a comparison.
In our last approach we vectorized the conditional statement of the moving average window since it is only dependent on an intermediate result and does not affect further computations. Therefore by unrolling the loop we can put multiple intermediate results into a vector and replace multiple conditional statements with it. Even though this yielded correct results we ran into bugs, which prevented us from measuring the performance, therefore it will not appear in the discussion of the results.

Classifier. As explained in Section 2, the classifier is heavily dependent on the current and past data and it consists of many if-statements. Furthermore, the number of buffers and the buffer sizes make branch predictions difficult and worthless. Overall, we applied scalar replacements and pre-computations and we further identified few for-loops where we could benefit from AVX optimizations.

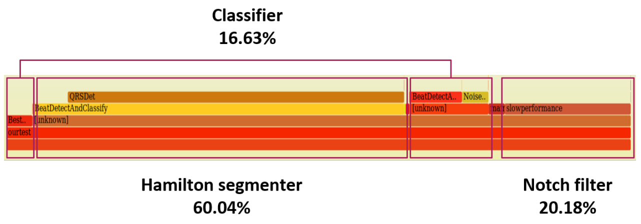

Fig. 5. Flame graph showing the percentage of cpu time.

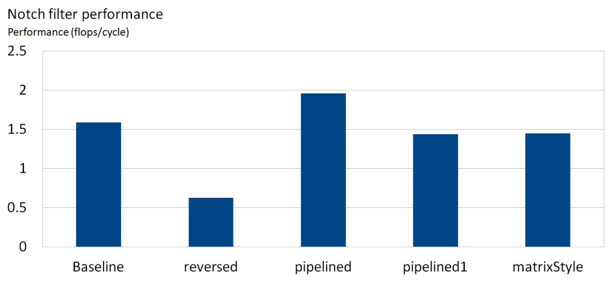

Fig. 6. Performance for different versions of the notch filter

However, the classifier counts for around 16% of the total computation time as shown in Fig. 5, therefore it's not the main focus in the optimizations.

Overall application. Finally, we combined all the stages of the application and our final evaluation process mainly consists of three parts. First we looked closely at how individual optimization steps impact the overall performance. Then we moved onto examining how combined optimization techniques interact with each other and how they work together to affect the performance. Last but not least, we tried to diagnose some performance degradation when optimization techniques were applied incrementally.

4. EXPERIMENTAL RESULTS

Experimental setup. All results form the report were produced on a Intel Core i7-2600 CPU. This a processor with the Sandy Bridge architecture. We ran it at 3.4 GHz and disabled Turbo-Boost. The L1 data cache and L1 instruction cache have a size of 32 kByte each and the L2 cache 256 kByte. For compiler we used gcc version 7.4.0 for Ubuntu. For flags we used were -O3 and -mavx, as well as -g for debugging.

For testing we had three different setups. We built benchmarks for both filters based on the ones from the homework exercises. For the notch filter we generated input and expected output in MATLAB, using the filter tools to simulate the notch filter. We made sure the notch filter in MATLAB used the same coefficients as out implementation. For the QRS filter we extracted the input and output from the original implementation for the correctness check. Both benchmarks first checked for correctness and then measured using the same method of running the test multiple times, measuring the time using the Time Stamp Counter and then using the average as result.
To measure the entire application we also ran it multiple times using the ECG signal already mentioned in Sec. 2. We made sure to flush the cache between each run, to avoid speedup from reuse of data. We then also average over multiple runs to make sure we get a more representative number.

Results. In the following paragraphs, we will first report individual results on the filters benchmarks and then the performance results on the whole application after combining all the stages. We don't plot the performance against the input size, as we compute one sample after the other in any case and there is no change of performance with input signal length.

Notch filter. Fig. 6 demonstrates the performance for the most interesting versions of the Notch filter. The best performing version is the pipelined implementation reaching almost the peak performance of 2 flops per cycle on the Sandy Bridge processor we used to run our tests. Whilst, the worst performing method was the 'reversed' one. Despite the idea of reducing the dependency between the different loops, the feed-back loop within the single stages became critical, as the computation to be done on one sample was not enough to hide the latency of the result before it was used as state for the next sample. As a result the performance degraded significantly, in fact it barely reached 0.6 flops per cycle as shown in Fig. 6.

The 'pipelined1' bar shows the performance of the SIMD based on 'pipelined' implementation. The problem with this approach was that we reintroduced dependencies we had alleviated in the 'pipelined' step. As all the stages were computed simultaneously, the computation of the next step was now again directly dependant on the previous computation. This together with the fact, that we needed almost half as many shuffle instructions as actual computation, made the vectorized version perform worse than the scalar version.

Finally, the matrix-vector-multiplication version again performed as slow as the vectorized pipeline version, because it again introduced additional operations for the shuffles. Additionally, the input vector for the next matrix vector multiplication directly depends on the last computation, which reduced instruction level parallelism.

QRS filter. Optimizing the single loop stage only yields a small increase in performance as we can see for the first two optimization steps in Fig.7. As for the unrolling approach, it turns out that unrolling 50 times is more efficient, even when using an input size (7200 samples) which is a multiple of 400.
Therefore we use the 50 times unrolled version which uses as cleanup code the vectorized approach for the ring buffers,

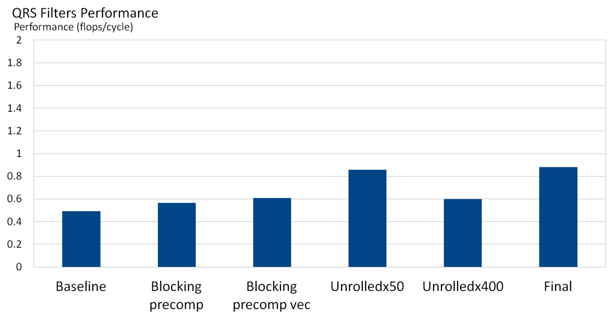

Fig. 7. Performance for different versions of the QRS filter

as it yields the greatest increase in performance when applied as individual optimization step.

Whole application. We first applied the individual optimization steps, the performance of which is shown in Fig. 8. The baseline used is the functions inlined version of the original source code. With blocking we obtained the highest performance speedup thank to improved locality. From the bar plot we can see that every optimization step yields improvement in performance with respect to the baseline. Finally, we applied each optimization step in a cumulative fashion and measure the performance demonstrated in Fig. 9.

At this point we also want to mention, that the initial application was only based on integer computation. If we take the run time of that version and divide the number of operations we counted for the baseline we get

As we can see the results are not as expected and the performance does not keep increasing when adding more optimizations. We suspected that the instruction cache could be a bottleneck, since certain optimization steps such as the FastQRSfilt come at the price of increasing the code's size. Measuring the actual cache misses we've proven this assumption to be wrong[2] and we currently have no explanation why this degradation of performance happens, even though each step increases the performance on its own.

5. CONCLUSIONS

At the end, we were able to obtain a 3x performance gain comparing to the baseline. The runtime reduction achieved through our optimization can greatly benefit potential users in determining and diagnosing their cardiac abnormalities, especially during critical conditions. Unfortunately convoluted data dependencies have prevented us from squeezing out more compute power from our current hardware system even further. In the future, we would like to investigate more into why performance degradation happened when certain optimization steps were applied one after another as well as the stability and consistency of our testing framework.

[2]Data can be found in our git under performance/output/cache_2019-06-13_07:22:35

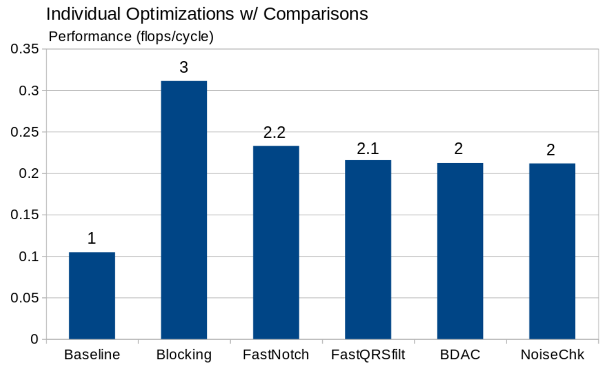

Fig. 8. Performance with individual optimization steps. The number on the bar are the speedup wrt. the baseline.

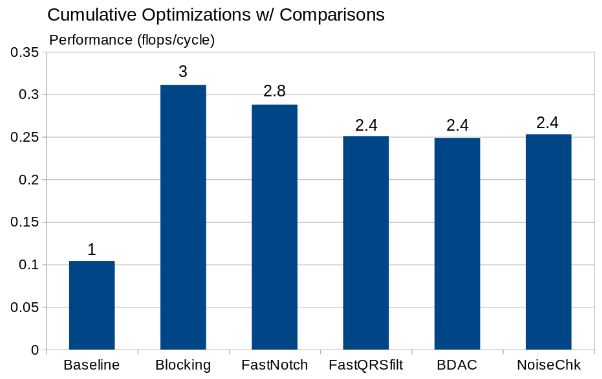

Fig. 9. Performance with accumulative optimization steps. The number on the bar are the speedup wrt. the baseline.

6. REFERENCES

[1] P. Hamilton, “Open source ecg analysis,” in *Computers in Cardiology*, Sep. 2002, pp. 101–104.

[2] J. Pan and W. J. Tompkins, “A real-time qrs detection algorithm,” *IEEE Transactions on Biomedical Engineering*, vol. BME-32, no. 3, pp. 230–236, March 1985.

[3] P. S. Hamilton and W. J. Tompkins, “Quantitative investigation of qrs detection rules using the mit/bih arrhythmia database,” *IEEE Transactions on Biomedical Engineering*, vol. BME-33, no. 12, pp. 1157–1165, Dec 1986.

[4] Luis Howell, *ECG-Detectors*, https://github.com/luishowell/ecg-detectors, Feb 2019.