a pipeline. Pipelining is the cheaper form of parallelism because expensive hardware, like a floating point multiplier, is needed only once, and therefore, pipelining is found in most general purpose processors. The regular flow of instructions through the pipeline, however, comes to a halt, if the execution of an operation needs the result of a previous operation that is still in the pipeline and has not yet progressed to a stage where this result is already available. The instruction (and all that follow) is then delayed until the required operand is computed. This situation is called a "pipeline stall". It is best avoided by scheduling a mix of unrelated instructions for execution.

Producing fast code is a very hard problem. It requires to find a balance between contradicting requirements that depend very much on the target processor and its facilities for caching and pipelining. This task is best left to the compiler that generates the machine code and different compilers achieve different results.

We compare the GNU C compiler version 2.95.3[12], the Intel C Compiler version 7.0 [16], and the Optimizing Microsoft 32-Bit C/C++ compiler version 12.00.8804 [23], as well as four different implementations: The DCT used in `mpg123` version pre0.59s[22], the DCT used in `mad` version 0.14.2b[21], the DCT derived with spiral (not used), and the CosDFT derived with spiral actually used with `mp32pcm`. In Table 10, the run times are given in μs. We postpone a complete discussion of performance until section 3.6.

C Compiler	`mad`	`mpg123`	DCT	CosDFT
GNU	0.84	1.47	1.09	1.13
Intel	0.62	0.43	0.31	0.31
Microsoft	0.71	0.79	0.56	0.49

Tab. 10: Run times in μs for different implementations and compilers

3.4 Eliminating the Shifting

The vector $\boldsymbol{v}$ contains 16 blocks of 32 subband samples each. The shifting is needed because the windowing step uses each of these blocks successively in all 16 positions within $\boldsymbol{v}$ (Fig. 11). If we do not want to move the data, we can instead move the start point of $\boldsymbol{v}$ inside a larger vector $\boldsymbol{w}$ (Fig. 12). Of course, we can not move the start point indefinitely since the amount of memory available for $\boldsymbol{w}$ is limited. When we reach the left end of $\boldsymbol{w}$ we have to start over at the right end and at this point, we have to make a copy of $\boldsymbol{v}$, which is now at the left end, because next it will be needed at the right end of $\boldsymbol{w}$ (Fig. 13).

Increasing the storage by n blocks, therefore implies that we can skip the copying n times before copying is necessary again. Any value of $n > 0$ is possible. With just one block more, we can cut the amount of copying in half. Another good choice could be 16 additional blocks—doubling the amount of memory. This reduces the time spent with copying quite drastically (to 1/16). For very different reasons—we want to keep a certain amount of "historic" data from the stream—as explained in section A.1, we choose to have $n = 2 * 18 + 15 = 51$.

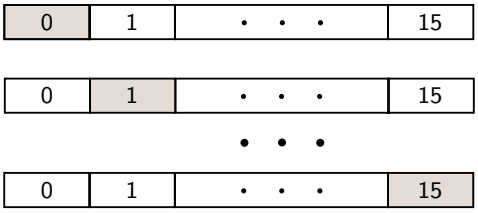

Fig. 11: Shifting a block of subbands through all positions

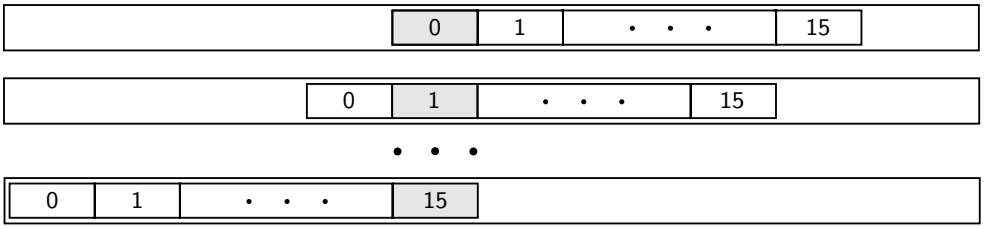

Fig. 12: Shifting the start point of $\boldsymbol{v}$, keeping subbands in place

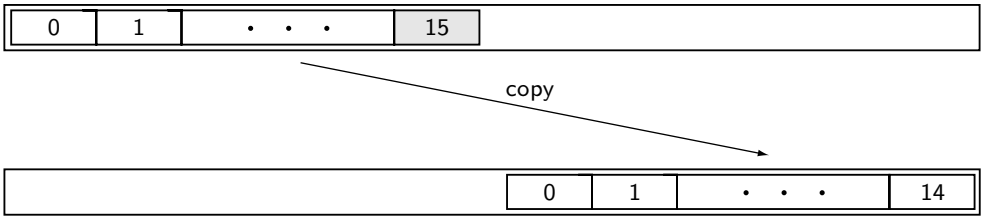

Fig. 13: Copy the subbands and start over at the right end

⟨ private declarations 1 ⟩ ≡ (1)
#define WINDOWBLOCKS 16
#define SHIFTBLOCKS (2 ∗ BLOCKS + WINDOWBLOCKS − 1)
#define SHIFTSIZE (SHIFTBLOCKS ∗ SUBBANDS)
#define CHANNELS 2 Used in 6, 7, and 130.

⟨ stream data 2 ⟩ ≡ (2)
double $\boldsymbol{w}$[CHANNELS][SHIFTSIZE]; Used in 42.

Now instead of copying $\boldsymbol{v}$ each time, we need to copy it only every 36th time. The shifting requires moving the beginning of $\boldsymbol{v}$ inside $\boldsymbol{w}$. For this purpose, we store as part of the ⟨ stream data 2 ⟩ for each channel an *offset* that indicates the intended start of $\boldsymbol{v}$ inside of $\boldsymbol{w}$.

⟨ stream data 2 ⟩ +≡ (3)
int *offset*[CHANNELS];

In each shifting step, the intended beginning of $\boldsymbol{v}$ moves to the left to make space for new subband samples. If the *offset* becomes negative, we have to copy $16-1$ blocks and start over 16 blocks left of the right end of $\boldsymbol{w}$. This is the C code to

```
⟨ shift vector v 4 ⟩ ≡                                                    (4)
  s→offset[ch] = s→offset[ch] − SUBBANDS;
  if (s→offset[ch] < 0)
  { s→offset[ch] = SHIFTSIZE − WINDOWBLOCKS ∗ SUBBANDS;
    memmove(&(s→w[ch][s→offset[ch] + SUBBANDS]), &(s→w[ch][0]),
        sizeof(double) ∗ (WINDOWBLOCKS − 1) ∗ SUBBANDS);
  }
  v = s→w[ch] + s→offset[ch];                          Used in 70, 75, and 408.
```

After that, $\boldsymbol{v}$ is shifted and ready to receive the next block of data.

3.5 The Core Algorithm

What remains, after the elimination of shifting, as the core of the algorithm is captured by the functions *dct32* and *windowing*.

```
⟨ private declarations 1 ⟩ +≡                                             (5)
  extern void windowing(const double ∗v, mp3_sample ∗x);
  extern void dct32(const double ∗y, double ∗v);
```

dct32 takes a vector $\boldsymbol{y}$ of scaled samples performs the ⟨ 32 point CosDFT 411 ⟩ on $\boldsymbol{y}$, with the output ending up in vector $\boldsymbol{v}$. The *windowing* function takes $\boldsymbol{v}$ and writes output samples to vector $\boldsymbol{x}$.

The code for the ⟨ 32 point CosDFT 411 ⟩ can be found in appendix A.2 and the code to ⟨ apply windowing 412 ⟩ is in appendix A.3.

Since the performance of the decoder depends largely on these two routines They are written to a separate file, `perform.c`, to facilitate further optimization.

```
⟨ perform.c 6 ⟩ ≡                                                         (6)
#include "mp32pcm.h"
  ⟨ private declarations 1 ⟩
  ⟨ conversion from double to mp3_sample 138 ⟩
  void windowing(const double ∗v, mp3_sample ∗x)
  { ⟨ apply windowing 412 ⟩ }
  void dct32(const double ∗y, double ∗v)
  { ⟨ 32 point CosDFT 411 ⟩ }
```

3.6 Performance

After all these optimizations, we return to the question of performance. The distribution of the runtime of `mp32pcm` is shown in Tab. 14. It is now dominated by the windowing step, which inevitably requires about 500 additions and multiplications, while the CosDFT needs only a reasonable fraction of the overall time. The reading of bit is only critical inside the main loop, where we read the subband samples. Decoding the header or reading the bit allocation and scalefactor information contributes

almost nothing to the overall runtime. Still, the reading of the subband samples is surprisingly expensive, since apart from some bit manipulation and table lookup, only a single multiplication is required per sample.

Activity	Intel[16]	GNU[12]	References
read bit allocation	1 %	1 %	see section 7.4
read scalefactors	1 %	1 %	see section 7.9
read subband samples	15 %	25 %	see section 7.6
CosDFT	25 %	30 %	see appendix A.2
windowing	56 %	41 %	see appendix A.3
rest	2 %	2 %	

Tab. 14: Distribution of runtime for different compilers

In Tab. 15, the times are given in ms for running the different decoders on the layer I file `fl1.mpg`, provided with Part 4 of the standard[2].

	`mad`[21]	`mpg123`[22]	`mp32pcm` 32 bit	`mp32pcm` 16 bit
GNU[12]	5.83 ms	3.29 ms	3.11 ms	3.55 ms
Intel[16]	5.27 ms	3.23 ms	2.30 ms	2.70 ms
Microsoft[23]	6.02 ms	4.49 ms	4.39 ms	4.29 ms

Tab. 15: Run times in ms for different implementations and compilers

Some remarks may help to interpret this table:

- `mad` uses fixed point binary fractions with operations that can be implemented through integer arithmetic and shifting. Different formats of these numbers can be used to adjust the speed accuracy tradeoff. But even if optimized for speed, the decoder runs slower than `mpg123` or `mp32pcm`. Due to the extensive use of hand-optimized code, the differences between various compilers are relatively small.
- `mpg123` is commonly regarded as the fastest decoder available. Again there are various options to influence the speed/quality tradeoff. The influence of the compiler on the speed is minor because critical parts (the DFT) are written in assembly language.
- `mp32pcm` is slightly faster than `mpg123` when compiled with the Intel compiler and slower when compiled for 16 bit with the GNU compiler. The difference is due to the faster code that the Intel compiler produces for CosDFT and windowing. For this reason, the distribution of `mp32pcm` supplies the assembly output of the Intel compiler for these parts (in file `perform.s`) as an alternative to the C sources (in `perform.c`). If using the file `perform.s`, the GNU compiler will produce a decoder as fast as the Intel compiler.