A Fast Data Structure for Disk-Based Audio Editing

Dominic Mazzoni and Roger B. Dannenberg

School of Computer Science, Carnegie Mellon University
email: [dmazzoni, rbd]@cs.cmu.edu

Abstract

Computer music research calls for a good tool to display and edit music and audio information. Finding no suitable tools available that are flexible enough to support various research tasks, we created an open source tool called Audacity *that we can customize to support annotation, analysis, and processing. The editor displays large audio files as well as discrete data including MIDI. Our implementation introduces a new data structure for audio that combines the speed of non-destructive editing with the direct manipulation convenience of in-place editors. This paper describes the data structure, its performance, features, and its use in an audio editor.*

1 Introduction

Audio editors can be classified as either in-place or non-destructive, depending on whether they modify original samples on disk or not. The first audio editors, as described by Kirby and Shute (1988) and Moorer (1990) were non-destructive, and they were modeled after tape-based editors, with similar control panels and basic operations, with the main advantage that edits could be changed or reversed later. However, these types of editors force users to keep all of the original audio clips that are used to create the final mix, and once the editing is complete, an additional step is required to actually produce the output audio file from the originals. Another disadvantage is that as edits build up, more and more processing must be done in order to play the audio, at some point hurting real-time performance. In this sense, non-destructive editors are not well-suited to large numbers of edits or to displaying the results of arbitrary sound manipulations. Often, processed samples must be written to a new file before they can be viewed, and usually this implies some extra file management tasks for the user.

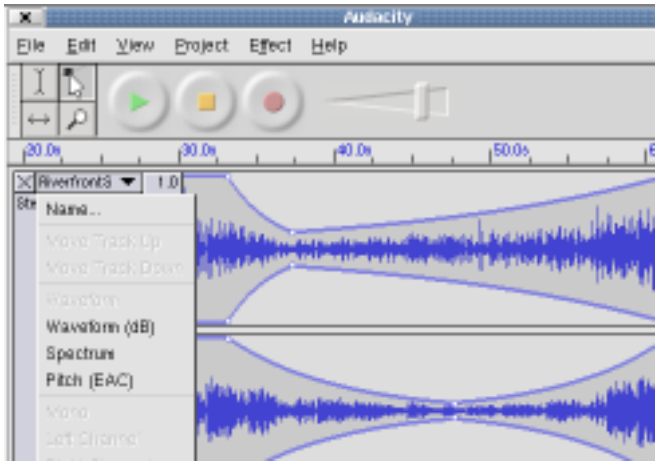

Figure 1. Using Audacity on Linux.

As personal computers have grown faster and more powerful, new audio editors have emerged that more closely resemble a computer word-processor or computer painting program than a reel-to-reel tape editor. These editors allow users to perform many effects on their audio files in place, with all changes affecting the original waveform data on disk. This makes editing much simpler and faster, especially for small files, and eliminates the extra step at the end, since the current copy of the entire project is always stored on disk. Modified waveforms can be viewed immediately since computed samples are directly available. Unfortunately, in-place editors do not scale with the size of the file: editing operations such as cut, paste, and undo take more time as the size of the file grows. Today both types of audio editor are popular.

Our work adopts the in-place editor model, where computation is performed immediately on stored samples and where the results of computations are immediately viewable on the screen. However, we introduce an implementation with performance that compares to a non-destructive editor. Unlike in-place or non-destructive editors, our approach scales well with both the size of the file and the number of edits. We have incorporated this approach in a cross-platform audio editor named *Audacity*. Because of its data structures, it can perform insertions and deletions extremely quickly, and it also supports multiple undo, which is also nearly instantaneous. Unlike a non-destructive editor, Audacity always has the current version of the audio file on disk, so it does not need to do any real-time processing in order to play the audio at any time. This also means that the user can safely throw away the original files or modify them in a different program without worrying that some audio project is depending on parts of

them. The core editing functionality of Audacity is finished, but we are continuing to develop the program under the open-source model to add features and improve its interface.

2 Data Structure

In order to achieve fast editing while writing changes to disk, Audacity uses a *Sequence* data structure, a generic structure that was recently described and analyzed by Charles Crowley (1996). A Sequence is an abstraction of an ordered array of small values (like samples in the case of audio) that supports the following operations:

Get(*i, l*): Retrieve *l* consecutive samples from the *i*th sample.
Set(*i, l*): Change *l* consecutive samples from the *i*th sample.
Insert(*i, l*): Insert *l* consecutive samples before the *i*th sample.
Delete(*i, l*): Delete *l* consecutive samples from the *i*th sample.

If the primary goal is to do disk-based editing, the strict algorithmic complexity of these operations (i.e. $O(l)$) is not nearly as important as the number and magnitude of *disk operations* that must take place for each editing operation. There may be many ways to implement a Sequence efficiently with respect to disk operations, but the way we chose is to store the samples in small blocks of size k to $2k$, where k is suitably chosen to be small enough that processing one block takes negligible time, but not so small that the audio is split into an unreasonable number of blocks. The limited range of block sizes is critical to guarantee good performance. Very large blocks make small edits take too long, and very small blocks reduce throughput. In our implementation, we store each block in a separate file on disk and we found that k=16K two-byte samples nearly maximizes efficiency (in terms of editing speed).

By making the restriction that all of the blocks in a Sequence must be between k and $2k$ samples, we guarantee that an insert or delete operation of any size involves reading or writing only a constant number of blocks on disk. The structure that holds the sorted list of pointers to the blocks and their relation to one another can be kept in memory, and it can be implemented as a binary tree or even as a simple dynamic array without sacrificing performance.

The intuition behind the k-2k restriction is that whenever you're given a block of at least k samples (but possibly much more), you can always split it into some number of blocks such that each block contains between k and $2k$ samples.

As an example, consider the operation Delete(*i, l*). By searching the list of blocks, we find that sample i is in block a, and that sample $i+l$ is in block b. (See Figure 2.) The blocks strictly between a and b contain all deleted samples, so we can remove them from the list immediately (deleting their associated files). Then we remove the deleted samples from the end of a and the beginning of b, taking time at most $4k$. Now one or both of blocks a and b may contain

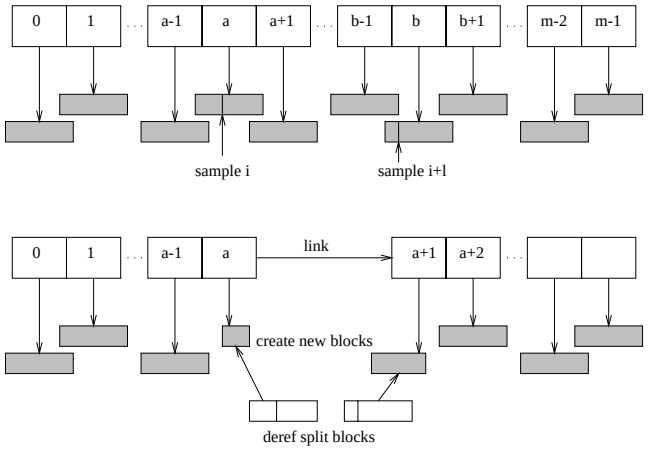

Figure 2. Deleting samples from a Sequence.

fewer than k samples. However, now combine block a with block a-1, and block b with block b+1. (If block a is the first, or block b is the last, then this step is not necessary because the first and last blocks have no minimum number of samples.) It is easy to see that the number of samples in block a plus a-1, and similarly in b plus b+1, is between k and $4k$. The samples can thus be reapportioned into either a single block or two blocks such that all blocks in the list now satisfy the k-$2k$ property, and while only modifying a constant number of blocks on disk. The operation Insert(*i, l*) is more complicated than Delete, but the idea is the same.

Splitting each audio track into blocks has many other benefits. By adding a reference count to each block, we make it possible for multiple tracks or even multiple places in the same track to share the same audio data, with minimal extra storage requirements. This also makes it easy to have multiple versions of an audio file around and implement an undo feature easily.

This data structure also lends itself very well to implementing other important features of an audio editor. Although it is occasionally useful to see the individual samples of a waveform, most of the time we want to see a few seconds, or even minutes, of audio on the computer screen at once. It is impractical to scan through minutes worth of audio just to render an image of the waveform on the screen, and thus audio editors need to cache a reduction of the audio somewhere in order to render it more quickly when the level of magnification is small. The most common type of reduction is to display the peaks – i.e. the minimum and maximum amplitudes of the samples represented by each pixel.

We take advantage of our Sequence structure and store these reductions in each block. By design, we have chosen our block size so that the time needed to process the samples in a single block is almost negligible. In our particular implementation, for every block we calculate the minimum and maximum of each group of 256 samples

within it, and store these numbers at the beginning of the file on disk for fast lookup.

3 Performance Measurements

In order to determine just how quickly this implementation of a Sequence performs in practice when doing many insertions and deletions, we set up the following benchmark: Using a fixed value for *k* (corresponding to blocks containing between 16K and 32K samples), we took audio files of sizes varying from 1MB to 512MB and imported them into the data structure. Then we performed 100 random edits, where each edit consists of cutting a random segment out of the file and inserting it at a different random location. Each benchmark was automated and run several times on our test computer, a Linux-based system containing a Pentium III at 500MHz and an Ultra ATA hard drive.

We had hoped to find that the total time to perform this sequence of operations was constant independent of total file size, and this is approximately true. We found actually it does take longer to perform operations as the data size increases (see Figure 3). The trend is not linear, but has a large jump, most likely due to disk caching: as the file size grows, less and less of the total file can be kept in the disk cache, so the probability that the samples at the boundary of a cut or paste are in the cache goes down. There are also some in-memory data structures that grow with the file size, but these structures are fairly small even for very large files.

For all practical purposes, though, editing operations do not take time proportional to the size of the edit. We are pleased that the average time per editing operation is only about an eighth of a second, even when the total file size is half a gigabyte. This is over 100 times faster than the conventional in-place editing approach and comparable to a non-destructive editor as we had hoped. Users never suffer from long delays that are common with in-place editors.

We also did benchmarks to verify that storing audio data in small files did not affect performance. By comparing different block sizes, we discovered that for block sizes below a certain threshold, performance seriously degrades, as the time overhead required to find and open each file dominates the time to read the data. On our test system, this minimum size is 32K bytes (16K two-byte samples), but this number is dependent on the speed of the host computer and the operating system used. However, as long as we choose block sizes larger than this minimum, performance is roughly the same. This is consistent with the observation by Abbott (1984) that by using large enough buffers, one can achieve high bandwidth reading blocks of audio that are scattered all over the disk.

Our only remaining cause for concern is that it is annoying to deal with large numbers of small data files manually. While any modern operating system can handle tens of thousands of files in a single directory without any problems, the user interfaces that people use to interact with their file systems often have issues with this number of files, whether you are using UNIX command-line tools or Microsoft Windows Explorer. It is also noticeably slower to delete a thousand files than one big one. These concerns should be considered when choosing whether or not to implement this structure, and if so, when choosing what block size to use.

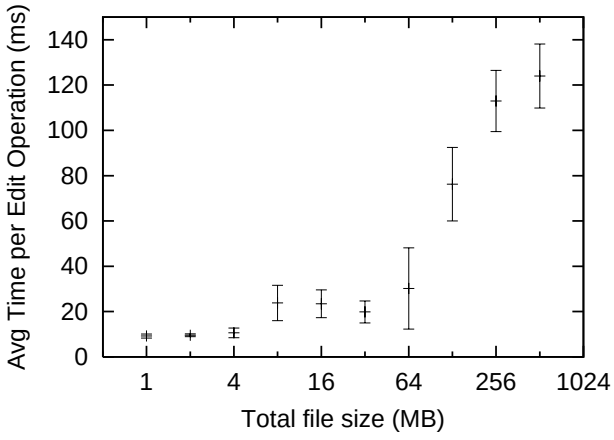

Figure 3. These performance measurements show that the time to perform one editing operation, while small, is affected by the total size of the file, probably due to less disk caching.

4 Current Status

Audacity was originally created as an audio research tool. We added tools for interactive labeling of audio, MIDI display, and spectral analysis to support various research projects, including labeling and annotation of audio recordings, and automatic pitch extraction.

We have also released the code to Audacity under an open-source license, encouraging others to study the source code for ideas and donate their contributions to the community. So far contributors have added many custom effects and customization options. It was our intention that Audacity be useful for general editing tasks in addition to specialized research goals. By accepting commercial plug-ins (in the VST format), Audacity has access to high-quality digital effects. Audacity uses the wxWindows toolkit and runs natively on Linux, Win32, and Macintosh systems.

5 Conclusions

We have found that by storing audio data in small blocks, we can achieve the speed and responsiveness of non-destructive editors, with the convenience, simplicity of design, and waveform visualization advantages of in-place editors. Not only can cuts and pastes be done in near

constant time, but unlimited undo and redo can also be implemented with very little extra space overhead. The entire data structure can be abstracted by a C++ class that allows the programmer to treat the structure as if it were a flat file while the class handles the internal details of the structure itself.

Our approach also allows disk blocks holding portions of an audio file to be reused after a certain number of editing operations have been applied, conserving disk space while still allowing a certain number of steps to be undone. Furthermore, our approach automatically merges small adjacent edits into a reasonably large, single block of data to improve playback performance. Finally, it encourages editors where users are never required to keep track of and manage the underlying files to get good performance or to minimize disk space. Because of all this, we believe that many can benefit from using this data structure as an alternative to storing audio data in a large flat file or using edit decision lists.

6 Acknowledgments

The authors are grateful to Bernard Mont-Reynaud and Andy Moorer for offering perspectives on digital audio editors. Additional thanks go to Eli Brandt for valuable feedback on this paper. This material is based upon work supported by NSF Award #0085945, an IBM Faculty Partnership Award, and an NSF Graduate Research Fellowship.

References

Abbott, Curtis. 1984. "Efficient editing of digital sound on disk." *Journal of the Audio Engineering Society*, June, pp. 394 - 402.

Crowley, Charles. 1996. "Data Structures for Text Sequences." http://www.cs.unm.edu/~crowley/papers/sds/sds.html

Kirby, D.G. and Shute, S.A. 1988. "The exploitation and realization of a random access digital audio editor." *IEEE Broadcasting Convention*, pp. 368 – 371.

Mazzoni, Dominic *et. al*. *Audacity*. 2001. (Software). http://www.cs.cmu.edu/~music/audacity/

Moorer, James A. 1990. "Hard-Disk Recording and Editing of Digital Audio." *89th AES convention*, September.