Dominic Mazzoni and Roger B. Dannenberg
Computer Science Department
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213 USA
{dmazzoni,rbd}@cs.cmu.edu

A Fast Data Structure for Disk-Based Audio Editing

For some time, ordinary personal computers have been powerful enough to allow people to edit, filter, and mix digital audio without any special added hardware. The earliest editors, such as those described by Freed (1987), Kirby and Shute (1988), and Moorer (1990), were modeled after tape-based editors, with similar control panels and basic operations; the main advantage of this is that edits could be performed non-destructively and then changed or ''undone'' later. However, these types of editors still force users to keep track of 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. 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 operations on their audio files in place, with all changes affecting the original waveform data on disk. Furthermore, the visual display reflects the results of all edits, which is not always the case for non-destructive editors. This makes editing much simpler and faster, especially for small files, and eliminates the extra step at the end, because the current copy of the entire project is always stored on disk. However, these ''in-place'' audio editors are not usually able to provide more than a single level of undo, and they are often very slow in dealing with large files. Today, one can find a variety of both types of audio editors for personal computers. Some popular in-place editors are SoundEdit 16 from Macromedia, CoolEdit from Syntrillium, and Sound Forge from Sonic Foundry. Non-destructive editors include Cubase from Steinberg Media Technologies AG, Digital Performer from Mark of the Unicorn, Inc., and ProTools from Digidesign, a division of Avid Technology, Inc.

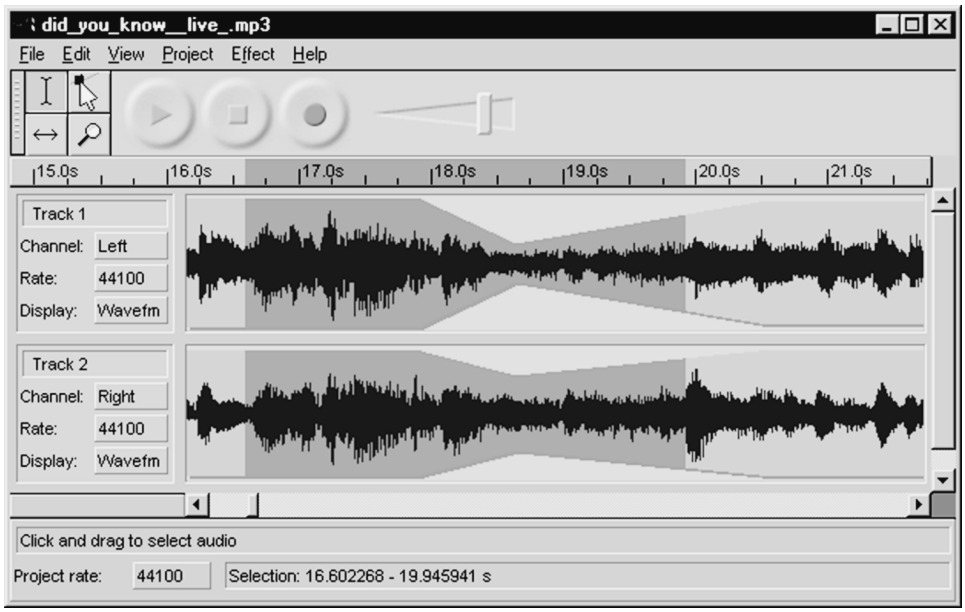
Computer Music Journal, 26:2, pp. 62–76, Summer 2002

Nomenclature is not always consistent; for example Peak from BIAS, Inc., is called a non-destructive editor, but it performs most operations by creating temporary files on disk to hold the changes, and it displays the actual values of edited samples. This is closer in behavior and performance to in-place editors. Also, most in-place editors are designed for single small files with the assumption that a separate, non-destructive editor will be used to combine many smaller files into a final mix. We think a well-designed editor ought to be able to perform well in both sorts of tasks.

This article examines how to combine the strengths of both in-place and non-destructive approaches to audio editing, yielding an editor that is almost as fast and reversible as a non-destructive editor, while almost as simple and space-efficient as an in-place editor. Although we create an interface that looks like that of an in-place editor, we also support multiple tracks with editable amplitude envelopes. This allows us to manipulate and combine many audio files efficiently.

We think this work is particularly interesting to the computer music community for several reasons. First, longer works of music require larger files; editors that suffice for 3-minute pop songs may not work well with larger works. Second, computer music composers may want to see the effects of signal processing effects or to apply effects that are too slow for real-time processing. The non-destructive editors do not support these capabilities directly. Finally, composers might want fast undo for many levels and quick redisplay to facilitate experimentation and creative exploration. We believe our approach offers good support for all of these features and suffers from fewer problems than other editors we have seen. Ultimately, the best approach is determined by the application and personal preferences, and we will compare all three approaches later. We believe computer music composers and researchers will find our approach especially attractive.

Figure 1. Audacity, *a free cross-platform digital audio editor that uses a novel data structure for fast editing.*



The idea for our approach came in part from the work of Charles Crowley of the University of New Mexico, in his examination of the various data structures used by word processors in a paper entitled ''Data Structures for Text Sequences'' (available online at www.cs.unm.edu/~crowley/papers/sds/sds.html). In designing a word processor, it would not make sense always to store an entire file in one contiguous block, because inserting a character at the beginning would be unreasonably slow. However, it would also make no sense for a word processor to use an entirely non-destructive approach, because as the number of edits grows, the time required just to render a page of text would also grow. Crowley showed that most of the approaches used by existing word processors were variations of a general data structure called a *sequence*, which is optimized to allow fast insertion and deletion of contiguous blocks of data. The next section describes the functional requirements of a sequence structure for digital audio editing.

In the Implementation section of this article, we propose a particular variant of the sequence data structure that is well suited for storing large audio tracks. Our basic idea is to store a large audio track as a set of small files of approximately equal size. We demonstrate that, by imposing certain simple constraints, only a small constant number of these files must be read or changed on disk to perform simple editing operations such as insertions and deletions of arbitrary size, or undoing the last operation. By reducing the number of disk operations to a small constant, editing operations can be made to seem almost instantaneous. The Undo section of this article discusses how our sequence structure can provide very fast multi-level undo. Following that, we discuss how to store reductions of the audio data for fast screen display, and how this is affected by storing the data in blocks.

To demonstrate the effectiveness of our solution, we measured its performance, as described in the Performance section. We also implemented a freeware cross-platform audio editor called Audacity (see Figure 1), and we invite readers to experience this approach first-hand.

After the detailed description of our approach, we present a discussion of the advantages and disadvantages of the three approaches we have identified: in-place editors, non-destructive editors, and sequence-based editors. This is followed by conclusions about some of the major trade-offs that seem inherent in these different approaches.

The original motivation for this work included the need for a music data-visualization tool. We want to display audio, continuous parameters, spectral information, and discrete information such as MIDI data and labels. A good tool should allow flexible display and editing of various forms of data. Our program, Audacity, fulfills many of our music data-visualization needs, and its clean design and open source allow researchers to build customized data visualizers, starting from an already powerful base. Although data visualization is not the focus of this article, we encourage readers to use Audacity as a visualization tool.

Data Structure Requirements

Suppose that we have a single sequence of consecutive audio samples that we would like to store in a data structure. The sequence data structure supports the following operations:

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

Let there be n samples in the entire sound file (noting that n is often orders of magnitude larger than l). A naïve implementation of Insert and Delete would require shifting $O(n)$ samples on disk every time. (A note on notation: we use $O(n)$ to mean that the worst-case computation time grows linearly with respect to the parameter n.) However, it is possible to implement these operations so that Delete changes only a constant number of samples on disk. Furthermore, it is possible to implement Insert so that only $O(l)$ samples are changed if the data being inserted is in an array, or a constant number if the data being inserted is already in another sequence structure on disk. In addition, this data structure and its associated algorithms can be augmented to support an undo history with very little space overhead.

A primary concern for us is that our solution is not only theoretically preferable but actually fast in practice. Specifically, we want to ensure that editing operations take much less than 1 sec to perform on a typical computer and that a large number of tracks can be played from the disk in real time.

Implementation

At first glance, it is tempting to think that a tree-like structure would be a good approach. Ignoring the memory overhead, suppose that we stored each sample in its own node in a balanced binary tree (such as a Red-Black Tree or a Splay Tree). Inserting a single sample would always take $O(\log n)$ time, which is reasonable enough, but then inserting l consecutive samples would take $O(l \log n)$ time, which is definitely wasteful. Storing samples in a tree this way ignores the fact that most operations tend to work on large consecutive chunks of samples.

The idea of the binary tree is fine, but instead of storing one sample per node, consecutive samples (maybe about 32 kB) could be stored in each node. In fact, this tree resembles what is known as an *interval tree*, with each node containing a pointer to the samples in that interval. While this idea is good, let us simplify it even more and instead consider a simple linked list of such nodes. If the number of samples pointed to by each node is allowed to vary, then inserting or deleting samples within one node would not have to affect any of the neighboring nodes. Unfortunately, if the number of samples in one node could grow without bounds, then there would be no way to achieve Insert and Delete in constant disk time. By cleverly enforcing a minimum and maximum size for most nodes, however, we can achieve our goals in the following manner.

Let k be the minimum number of samples that should be stored in each node. The constant k

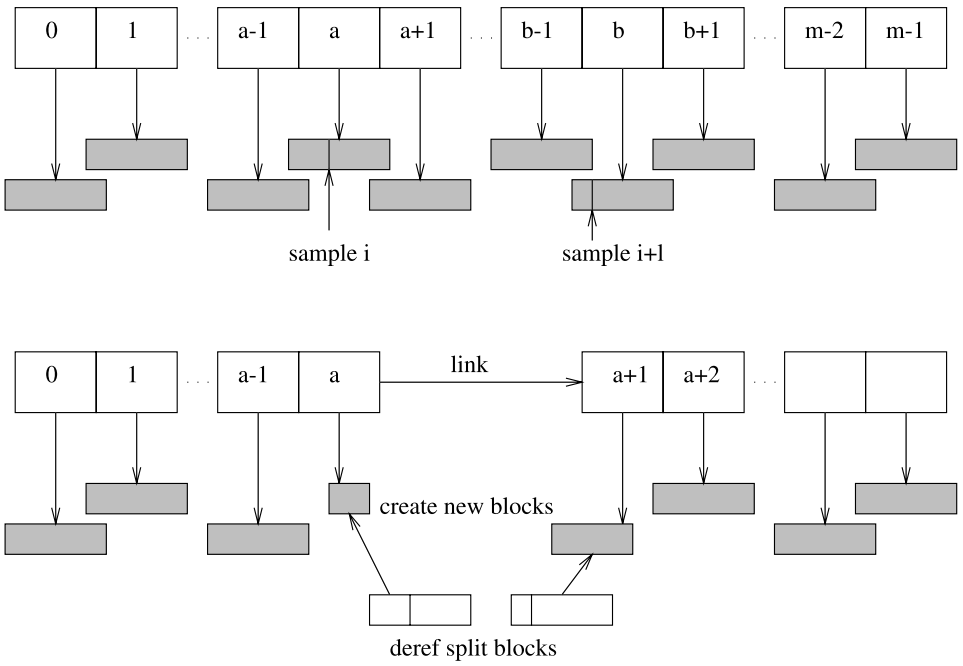


*Figure 2. An illustration of the operation Delete(*i, l*) where sample* i *is in node* a *and sample* i+1 *is in node* b. *To complete the operation, blocks may need to be reapportioned to satisfy the size constraints (described in the text).*

should be chosen as large as possible, while small enough that shifting $2k$ samples in memory or on disk still takes a negligibly short amount of time. Now make the restriction that all nodes in the linked list must have between k and $2k$ samples (inclusive), except for the first and last nodes, which are allowed to have fewer. We found that storing between 32 kB and 64 kB of data per block was ideal on our test system, which translates to $k = 16$ k, assuming 16-bit samples. However, this value is highly dependent on the efficiency of the implementation, the transfer rate to and from the hard disk, and the operating system used.

The intuition behind this restriction is that, given a block of at least k samples (but possibly many more), one can always split it into some number of nodes such that each node contains between k and $2k$ samples. This is the secret to how we will be able to perform a Delete or an Insert without ever modifying the samples in more than a small constant number of nodes. Essentially, we perform the Delete or Insert as normal, and then if a couple of nodes end up with too few or too many samples, we just combine them with several neighboring nodes and reapportion the samples between them so that they all obey the k-$2k$ restriction. What we have now is one of the variants of the sequence data structure described by Crowley for text editors. Let us examine the details of how to insert or delete samples from this structure.

Consider the operation Delete(i, l) illustrated in Figure 2. By traversing the linked list, we find that sample i is in node a, and that sample $i+l$ is in node b. We remove the deleted samples from the end of a and the beginning of b, taking time bounded by the maximum size of any one node, $2k$. The nodes strictly between a and b contain all deleted samples, so we can remove them from the list immediately. Note that a and b are just indices (not names), so after the deletion, node b becomes $a+1$. Now, as shown in the lower part of Figure 2, one or both of nodes a and $a+1$ may contain fewer than k samples. We now combine node a and node $a-1$, and then we combine node $a+1$ with node

$a+2$. (If node a is the first, or node $a+1$ is the last, then this step is not necessary, because the first and last nodes have no minimum number of samples.) It is easy to see that the number of samples in the sum of nodes a and $a-1$, and similarly in the sum of nodes $a+1$ and $a+2$, is between k and $4k$. The samples can thus be reapportioned into either a single node or two nodes such that all nodes in the list now satisfy the k-$2k$ property, and this has only taken constant time (plus the time searching through and manipulating the pointers in the linked list, which we will address below).

Next consider Insert(i, l). Suppose that the samples to be inserted come from another sequence structure, so they have already been grouped into nodes of length k-$2k$, except for the first and last nodes, which are allowed to contain fewer samples. First, we find the node containing sample i—say it is the ath node. We split this node in two so that node a contains the samples before i, and the new node $a+1$ contains those after i. We want to insert the new nodes between a and $a+1$, but we have a potential problem. Owing to the split, a and $a+1$ may be too small, and the first and last nodes we are inserting may also be too small. Thus, it is not clear how to join the nodes such that all of these nodes contain between k and $2k$ samples.

While there may be an optimal solution that involves examining a number of special cases, we can solve the problem very easily in just two cases while still only forcing a constant number of nodes to be reapportioned. The first case occurs when the number of nodes to be inserted is three or fewer. In this case, we temporarily join node a, all of the inserted nodes, and node $a+1$ into one contiguous block. This block has size at least k and at most $8k$, and therefore it can be split evenly into between one and four nodes, depending on the exact number of samples. Because all of the nodes before a and after $a+1$ already had an appropriate number of samples, we are finished.

The other case occurs when we are inserting four or more nodes. In this case, we join the first two inserted nodes with node a into one temporary block, and we join the last two inserted nodes with node $a+1$ into another temporary block. Then, we split these blocks into appropriate sets of nodes, just as in the previous case. The remainder of the inserted nodes can be inserted as is, preserving the property that we wanted, that only a constant number of nodes would have to be affected by the operation. In fact, this last case involved the largest number of nodes: six. Thus, an Insert operation will affect at most six nodes.

One minor detail is that a search of the entire linked list could be required just to find the nodes needed to modify. In practice, we have found that this is insignificant, as the number of nodes in even a large audio track is far smaller than the number of samples in one node. However, one could simply replace the linked list with a binary tree of nodes, still requiring the k-$2k$ property, and then finding a node would be even faster. There are a few special cases that have not been addressed in the above descriptions of Insert and Delete, such as inserting exactly between two nodes, inserting fewer than k samples, or deleting samples that appear in just one node, but these are all relatively straightforward to work out. The interested reader is referred to the source code to Audacity, available online at audacity.sourceforge.net/.

Undo

Fully non-destructive editors can easily implement an undo feature simply by "backtracking" in the edit decision list; no waveform data has to be changed on disk. It is more difficult for in-place editors, because they must save a copy of the samples before the edit, which can take time proportional to the size of the edit. With a simple modification, our sequence-based approach can be used to implement an unlimited undo mechanism that runs in constant time and wastes very little space. To do this, let each block of samples to which every node points be a reference-counted object instead of just a simple array. That way, two different nodes can point to the same set of samples, and the samples are not destroyed until both nodes pointing to them are deleted. Then, we make a copy of the en-

tire node structure (i.e., the linked list of nodes) before any undoable operation and push the copy onto a stack representing the undo history. To undo, we simply revert to the most recent node structure on the stack. If the user then performs a new operation, the node structure that was undone can be thrown away, and if any of the blocks's reference counts reach zero they can be disposed of.

We have found that this very simple implementation of undo is efficient in practice, despite the fact that copying the entire node structure is a linear-time operation. The number of nodes is a small fraction of the number of samples in the track, so it is much faster to make a copy of the nodes than it is to make a copy of all of the samples. In practice, we have found that it is fast enough not to be noticeable, even when the total number of samples exceeds 1 GB. However, if time or space considerations prohibit copying the entire node structure after every operation, one can imagine implementing the equivalent of an edit decision list for the node structure instead. This would require very little time or space for each operation. We did not investigate this potential optimization.

Compared to a non-destructive editor or an in-place editor, our solution wastes more disk space in order to support undo. Clearly, it wastes more than a non-destructive editor, because such editors do not change any samples on disk when one performs an editing operation. In-place editors do change samples on disk, so they save a copy of the samples that are being changed to allow undo. Our system requires that one save their files a little more frequently, because, in a sense, users are ''rounding'' data to the nearest block boundary. Because we used block sizes of at most 64 kB, a single operation could result in 128 kB of wasted disk space, plus an entire copy of the linked list of nodes in memory. However, the average wasted disk space per operation in practice is not 128 kB but 64 kB, which is quite negligible compared to the size of most audio files. Clearly, it does not prohibit supporting a dozen or more levels of undo.

Note that reference-counting the blocks provides an additional feature: if we copy a large chunk of data (large enough to contain many blocks) and then paste it into a different place in the same file, all of the blocks except the ones on the boundaries can be copied by reference instead of by value, saving valuable disk space. This could be very useful, for example, for a long passage that repeats itself, or for stereo channels that are actually identical copies of one another except for a few small segments. Note that if either the original or ''copied'' data are subsequently modified, the modification appears only in one place. For example, if a sample is inserted into one copy, a new node will be allocated to contain the modification, but the copied regions continue to share other nodes.

Storing Reductions for Screen Display

Although it is occasionally useful to see the individual samples of a waveform, most of the time users want to see a few seconds, or even minutes, of audio on the computer screen at once. Moorer (1990) noted that 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).

From a theoretical standpoint, a binary tree would be a perfect way to store these maxima and minima for easy retrieval. Suppose we have a track containing exactly 2^k samples. Imagine a tree where each node contains the minimum and maximum values of some range of samples, and its children contain the minima and maxima of the left and right halves of that range of samples. So the root would contain the minimum and maximum of the entire track, and there would be a total of $2^{k+1} - 1$ nodes. Of course, the tree could easily be truncated at some earlier depth, so that the leaves would actually store the minima and maxima of, say, 256 samples each. Then the tree would take up only about 1/128 of the memory of the entire track (assuming the tree is stored in a compact format

without pointers). However, the minimum and maximum of any range of samples could be calculated exactly in logarithmic time (i.e., time-proportional to k, which is small). The problem with this minimum/maximum tree is that, as defined above, it would need to be recalculated every time a change was made to the audio. Worst of all, deleting a single sample from the beginning would potentially force the entire tree to be recalculated.

Instead, we can take advantage of our sequence structure and store separate reductions in every node. 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, we calculate the minimum and maximum of each group of 256 samples within each block, and then we store these numbers at the beginning of the file on disk for fast lookup.

Next, we describe how to use this data to display a graphical image of the waveform peaks at some level of magnification (where each pixel represents more than 256 samples), using only the reductions and none of the original waveform data. Suppose there are P pixels visible on the screen, and we wish to display n samples in this space. From our assumption about the level of magnification, $n/P > 256$. Our goal is to calculate the function $\max(p)$ for $0 \le p \le P - 1$, which represents the maximum value attained by the waveform in the region represented by the pth pixel (and the corresponding min function, obviously):

$$\max(p) := \max\left(sample\left\langle\left\lfloor\frac{n \cdot p}{P}\right\rfloor\right\rangle \ldots sample\left\langle\left\lfloor\frac{n \cdot (p+1)}{P}\right\rfloor - 1\right\rangle\right)$$

However, because the floor of np/P and $n(p+1)/P$ are not likely to be multiples of 256, this function cannot be calculated given only the reductions. Instead, we define the approximate max as

$$approx\ \max(p) := \max\left(sample\left\langle 256\left\lfloor\frac{n \cdot p}{256P}\right\rfloor\right\rangle \ldots sample\left\langle 256\left\lfloor\frac{n \cdot (p+1)}{256P}\right\rfloor - 1\right\rangle\right)$$

This simply rounds down the left and right sides of our range to the nearest multiple of 256. However, because we have already stored the maximum of every set of 256 samples in our reduction, we can equivalently define the approximate max as

$$approx\ \max(p) = \max\left(reduction\left\langle\left\lfloor\frac{n \cdot p}{256P}\right\rfloor\right\rangle \ldots reduction\left\langle\left\lfloor\frac{n \cdot (p+1)}{256P}\right\rfloor - 1\right\rangle\right)$$

Clearly, it is faster to calculate this approximate maximum than it is to calculate the true maximum function for screen display, but how do we approximate it? Note that if $\max(p-1) \cdot \max(p)$, then $\max(p-1) \cdot approx\max(p) \cdot \max(p)$, and otherwise $\max(p-1) \cdot approx\max(p) \cdot \max(p)$. Thus, the approximate maximum function is never incorrect by more than one pixel, which seems like a reasonable compromise when the goal is fast screen display. (The user can always zoom in to get more precise information.) The algorithm for calculating the approximate maximum is complicated slightly by the fact that we must deal with block boundaries, but it can still be done in one pass. The following pseudocode illustrates how we calculate the min and max values to display for each pixel in the range $0 \ldots P - 1$ using pre-calculated reductions:

```
p = 0; // current pixel
i = 0; // current sample
b = 0; // current block
j = 0; // current sample within this block
k = j div 256; // current reduction within
  //this block
tmin = +Inf;
tmax =-Inf;
while(p < P) {
 tmin = min(tmin, block[b].min[k]);
 tmax = max(tmax, block[b].max[k]);
 k++;
 j += 256;
 i += 256;
 if (j > block[b].len) {
  // Go on to next block
  b++;
  j = 0;
  k = 0;
  i = block[b].start;
 }
```

```
 if ( (i*n/P) > p) {
  // Output min and max and go on to next
    //pixel
  output p, tmin, tmax;
  tmin = +Inf;
  tmax =-Inf;
  p++;
 }
}
```

Data Integrity

With audio data spread out among many files, a real concern is what happens if there is a computer system failure. It is tempting to think that we could store the filenames of the next and previous nodes inside each data file, so that even if the project file gets corrupted, the data files themselves contain enough information for the linked list of nodes to be reconstructed. Unfortunately, this is not possible, because we allow one data file to be pointed to by multiple nodes. Still, in designing Audacity, we took some steps that should make users feel safe that even in the event of a crash or other system failure, they will be able to revert back to the last saved version of their project.

First, Audacity project files are stored in a plain text, human-readable format, so that if all else fails, even a non-technical user could open the file and read the appropriate ordering of the data files. Second, the data files belonging to a saved project are never modified or deleted while the user is editing a project in place. To support undo, new data files are created very frequently to store new sample blocks, and so the ones belonging to the original saved project are marked. When the user saves a project, a new project file is created, and only when this has been written to disk will the old project file or old data files be deleted.

Performance Measurements

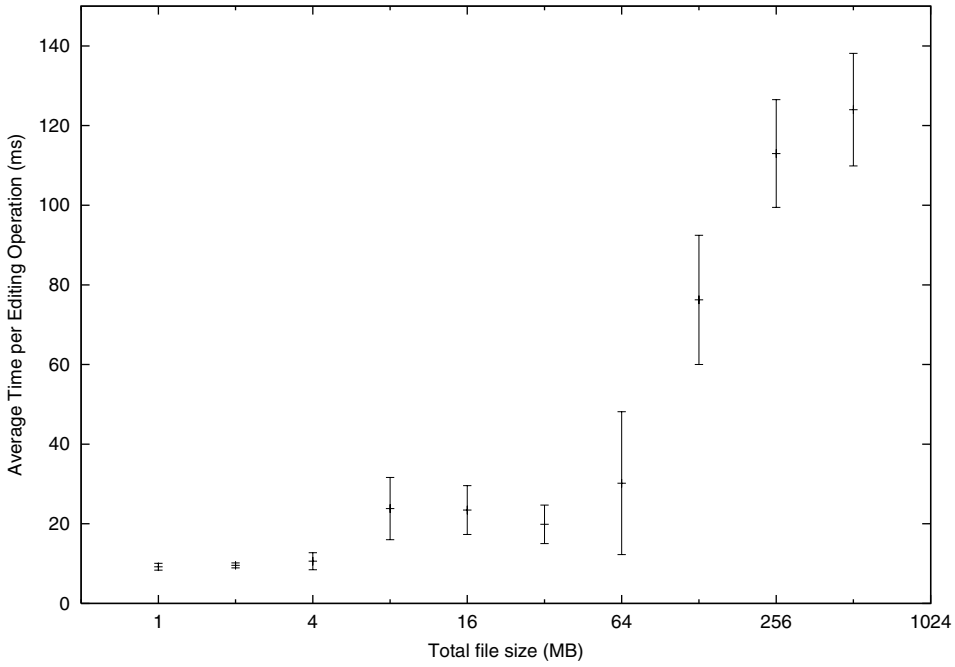
Once we had implemented this data structure and its associated algorithms, we tested the speed of editing operations and of playback. In order to verify that our implementation really can insert and delete quickly independent of the total file size, we set up the following benchmark: Using a fixed value for k (corresponding to nodes containing between 16K and 32K samples), we take audio files of sizes varying from 1 MB to 512 MB and import them into the data structure. Then we perform 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 500 MHz, and an Ultra ATA hard drive. The results of our experiment are shown in Figure 3.

We had hoped to find that the total time to perform this sequence of operations was constant independent of total file size. However, we found that there is a small trend toward a slight increase in time to perform the operations as the data size increases. Because the trend is not linear, but has a large jump, the most likely cause is disk caching: as the file size grows, less and less of the total file can be kept in the disk cache, and so the probability that the samples at the boundary of a cut or paste are in the cache goes down. The data points to the left of the large jump in the graph would thus represent files that fit entirely into the disk cache, and the points to the right are from files that are larger than the cache.

We are pleased to note, though, that the average time per editing operation is still only about an eighth of a second, even when the total file size is half a gigabyte. Note that if we stored our audio data in a single large file instead of smaller ones, then a single cut and paste would take longer than it takes to perform a hundred cuts and pastes with our sequence structure. Therefore, despite the fact that the performance decreases slightly as the data size increases, it still provides an enormous increase in speed over the conventional method for storing audio.

So far, we have been concerned with minimizing the number of disk operations needed to perform an edit, and we have been ignoring the amount of time it takes to work with the list of nodes in memory. If we were to store the nodes in a balanced binary tree instead of a linked list, we could

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.



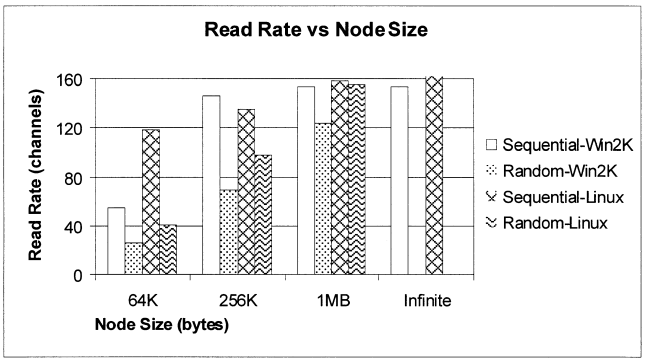
perform insertions and deletions in logarithmic time. However, if we are going to implement undo in the simplest possible manner, as discussed above, then we must make a copy of the whole node structure after every operation anyway. Even if we are working with 3,000 nodes in our structure (enough to store about 30 min of 44 kHz audio on average), our measurements show that it takes only about 12 msec to duplicate this structure in memory, including updating the reference counts of all of the blocks. In contrast, it takes over 100 msec on average to perform an editing operation (unless the files needed are already in the disk cache). Therefore, we feel justified in ignoring the cost of the memory-based operations, because they are dominated by the cost of operations that must go to disk.

Disk Bandwidth

Besides testing the speed of editing operations, we also need to test the speed of playback. Specifically, we want to make sure that storing audio data in small files does not limit performance. Small files have the advantage that less data needs to be copied for insertions and deletions, while large files have the advantage that less time is spent finding and opening files during playback. This is consistent with the discovery of Abbott (1984) that, by using large enough buffers, one can achieve high bandwidth reading blocks of audio that are scattered all over the disk. More recent work, for example by Anderson et al. (1992), has described designs for real-time file systems capable of supporting many streams of audio. It should be noted that we are working with "off-the-shelf" file systems that do not make any performance guarantees. The best we can do is to design applications that are likely to perform well.

As an example, Figure 4 illustrates an experiment comparing the disk performance in our approach with that of an in-place editor reading a single file sequentially. The experiment writes 1 GB of data in files of uniform size (s = 64 kB, 256 kB, and 1 MB) to disk. These files are read sequentially (in the same order written) and in random order. We also wrote a single large file (1 GB) to the

Figure 4. Data rates for different file sizes measured by reading 1 GB of data in the order the files were created, or in a random order that reads each file once. Results are expressed as a number of 16-bit, 44.1 kHz audio channels.



same disk and measured the time it takes to read it sequentially. (This is the ''Infinite'' category in Figure 4.) The test was performed using both Windows 2000 Professional and Linux running on the same hardware, an IBM A21p ThinkPad. The results are reported in average bytes per sec divided by 88,200. In other words, the vertical scale shows how many channels of audio (16 bit, 44.1KHz) are supported by the measured data rate.

According to these measurements, even random reads of 64 kB files supports 26 (Windows) to 41 (Linux) channels of audio. This number increases as the file size increases. Reading the files in the order written is faster than random order, indicating that the file system is allocating files in a roughly sequential manner on the disk, and that within files, blocks are allocated roughly sequentially. If the goal is to maximize the data rate, 1 MB nodes achieve 80% (Windows) to 93% (Linux) of the highest measured data rate.

It is important to realize that even if audio data is stored in one large file, there is no guarantee that it is stored contiguously on disk. In fact, any sufficiently large file on a hard disk is likely to be highly fragmented.

Editing Speed vs. Bandwidth

By using large block sizes, we can achieve excellent playback performance, but editing times suffer because more data is copied for each edit. Thus, there is a tradeoff between editing performance and playback performance. With 1 MB nodes, average editing times are about 1 sec, depending upon hardware and the operating system. Because even 1 sec can be annoying to users, block sizes should be chosen just large enough to guarantee sufficient data rates for the maximum number of simultaneous tracks anticipated. For example, if one wants to play eight 16-bit tracks, the block size can safely be set to 64 kB while allowing fast editing times. On the other hand, if one wants to mix 64 24-bit tracks at once, edits may take about one sec at a block size of 1 MB.

Further Optimizations

It is not a good idea for the editor to open too many files at once, so our implementation opens files as needed and closes files after reading them. This additional overhead is already included in all of our measurements.

To optimize disk performance during playback, the editor should issue many file read requests, allowing the operating system to minimize seeks and to allow files on separate disks to be read in parallel. We have not implemented this optimization, and in practice, we achieve good performance with sequential, blocking reads. Note that typical file systems automatically pre-fetch data from files that are accessed sequentially.

An early implementation of Audacity required that audio files be imported to our data structure before editing or even viewing them. This resulted in an unacceptable delay when files were opened. An optimization now allows ordinary audio files to be used by the editor without copying data. It is still necessary to read the data to compute the display, but this is true of other editors as well.

Random-access memory (RAM) is inexpensive enough now to consider an editor that is entirely RAM-based. Our data structures are ideal for a RAM-based implementation, because they avoid copying samples for insertion and deletion operations. At present, however, a RAM-based implementation would not support large audio files or many levels of undo on typical machine configurations, and a virtual memory scheme could lead to inconsistent performance.

Table 1. A comparison of three implementation strategies for audio editors

	In-Place	*Non-Destructive*	*Sequence-Based*
Insert/Delete	May copy whole file.	Fast.	Copies at most a small number of nodes.
Playback	Fast, sequential access.	Random access to source files and real-time DSP.	Predictable performance, many files.
Undo	Requires data copy on disk.	No disk copy.	No disk copy.
DSP/Effects	Overwrite old samples.	Deferred to playback time.	New samples written to disk.
Display	Based on edited data.	Based on source data.	Based on edited data.
Files and Fragmentation	One sequential file (plus undo data).	Source files plus edit list.	Many small files.
Quantization Error	Cumulative.	Non-cumulative.	Sometimes cumulative.

Another implementation possibility is to use a single file and allocate nodes from within the file. This is essentially the same task as implementing a file system, and we find it preferable to rely on the operating system to manage files.

Yet another consideration is the use of disk arrays. A disk array offers high disk bandwidth by distributing each file across many disks so that the file can be read in parallel (Chen et al. 1994). Because disk bandwidth is the limiting factor of in-place editors, disk arrays offer a way to increase in-place editing performance. However, the editing speed-up offered by disk arrays is small compared to the speed-up offered by the sequence data structure.

One could also use disk arrays with a sequence-based editor. This would provide some performance improvement by reading nodes faster, and it is fairly easy to configure a computer system with a simple disk array. Unfortunately, the total number of seeks in a disk array increases, because each file is spread across the array of disks. A better approach is to allocate whole audio tracks to each disk. That way, all operations to read a track can be performed in parallel, and performance will be even higher than in a disk array. Asynchronous or multi-threaded I/O would be used to keep disks operating in parallel. Of course, this would require a more complex implementation, careful configuration to use available disk parallelism, and perhaps new disk allocation strategies.

Comparison of the Models

Audio editors can be evaluated along many dimensions. In this section, we will concentrate on fundamental properties that derive from the underlying models, and we will ignore superficial features, that, while important, could generally be implemented within any model. Table 1 summarizes properties that we will discuss below.

Insert/Delete

As discussed earlier, in-place editors must copy disk data to perform insertions and deletions, and this can take many seconds if the file is large. Non-destructive editors simply modify edit decision lists, so the cost of insertion or deletion is negligible. Our sequence structure must update the disk, but this is typically very fast because a limited amount of data is copied.

Playback

In-place editors play directly from a sequential file, obtaining the best possible performance without implementing a custom file system. Non-destructive editors typically perform well because they read from sequential source files, but edits may require rapid disk seeks. Closely spaced edits

can arbitrarily degrade disk bandwidth. Furthermore, effects processing adds an unlimited computational load during playback. Both of these problems can be handled by computing samples and storing them on disk, but this complicates the model implementation and/or user interface. Our sequence-based model provides more predictable disk-performance by setting a lower bound on the length of sequential data so that most disk reads are sequential. A typical implementation will apply an envelope to each track and mix multiple tracks, so the load depends only upon the number of simultaneous tracks.

Undo

Undo operations require data to be copied from an undo buffer back to its original location with an in-place editor, so undo can be time consuming. With non-destructive editors and our sequence-based approach, undo merely modifies RAM-based data structures and updates the display. When reference counts go to zero, the sequence approach must free files on the disk, but this is faster than copying data.

DSP and Effects

With an in-place editor, effects such as equalization, reverb, and pitch shifting overwrite the file, requiring disk reads and writes. In contrast, non-destructive editors perform signal processing in real time, so effects can be applied and adjusted with no apparent computation, as long as there is sufficient processing power. Our sequence approach must compute effects and store them to disk, much like an in-place editor. However, because effects can usually be undone, and undo requires a copy, an in-place editor performs about twice as many disk operations as our sequence-based approach to apply an effect to data.

Typically, in-place editors apply amplitude controls and envelopes as effects; that is, the data on disk is modified immediately. On the other hand, non-destructive editors apply amplitude controls during playback. Our sequence-based implementation allows users to edit envelopes interactively and view the results on the display without actually changing data on disk. Because adjusting amplitudes, fading sounds in and out, and shaping sounds with envelopes are common operations, we decided to support them using a non-destructive approach.

Display

In-place editors and sequence-based editors always display the result of all editing operations, while non-destructive editors base the display on source files before any effects are applied. It is conceivable that non-destructive editors could compute the effects and render the results on the display, but this is not done in practice. Thus, there is a fundamental tradeoff here: to see the results of applying an effect, one must accept the delay of computing and storing the results.

Amplitude envelopes in our sequence-based editor do not actually modify samples until playback, so we scale the displayed values according to envelopes (see Figure 1). This has no impact on the display speed, because the computation is just one multiply per pixel, and it maintains our interface model that the user sees the results of applying each editing operation.

Files and Fragmentation

An in-place editor maintains one file of edited data and possibly some temporary files for undo data. This is the simplest format for users to manage. A non-destructive editor may refer to many source files. Typically, users must be aware of what source files have been incorporated into an editing project, because changes to any file may adversely affect the project. To copy the project, the user must copy all source files. In-place editors and non-destructive editors can typically open and edit a standard audio file quickly without copying to another format.

Our editing model maintains many files within a directory. These are managed automatically, al-

though it is slower to copy a directory of files than to copy a single large file. ''Standard'' file formats are typically copied to a sequence structure before editing, which makes files slower to open than with in-place or non-destructive editors. However, as noted earlier, it is also possible to be ''lazy'' about copying sound data into a sequence structure to improve the time it takes to open a standard audio file. In this case, users may need to be aware of file dependencies as with non-destructive editors. In addition, audio data and projects can be saved in the sequence structure format, which can be reopened very quickly.

Disk fragmentation occurs when disk blocks are partially filled. On average, each sizeable file is expected to have its last block only half full. Thus, the amount of fragmentation is proportional to the number of files. The sequence structure, with a file for each node, has a potential fragmentation of about 4% (assuming the average node size is 96 kB and the file block size is 8 kB), but even this would occur only after extensive insertions and deletions. To minimize fragmentation, files can be created initially without fragmentation simply by making their length an exact multiple of the block size. (One must be careful to allow for any file system overhead and storage for peak amplitude values.)

There are some file systems in use with large block sizes that would lead to greater fragmentation. Fortunately, small files are very common, and file systems are designed accordingly. The exceptions we know of arise when legacy file systems are used on modern (large) disk drives. This seems to be a temporary problem that we do not need to be too concerned about.

In contrast to internal fragmentation, where blocks contain unused space, we should also consider external fragmentation, where blocks between files are unused and unallocated. File systems eventually use all available blocks, but doing so may cause file blocks to be scattered randomly across the disk rather than sequentially. This in turn degrades file system performance. Disk defragmentation software offers one remedy. Alternatively, exporting to a single large file on another disk, deleting all nodes, and then recreating nodes should eliminate external fragmentation, but this requires space for the exported file. In-place and non-destructive editors should generate less disk fragmentation.

A final issue relating to files is the question of operating system support. It is possible to create many thousands of files in a directory. It is inconvenient and slower to make backups, copies, and even to view these files in a browser, and the single file representation of an in-place editor offers a much more manageable representation outside of the editor program.

Quantization Error

When samples are read, manipulated, and stored, quantization noise can be added. This matters particularly if data are stored as 16-bit samples. Non-destructive editors can minimize quantization error by reading data once, converting to a high-resolution format such as 32-bit floating point or integers, performing all processing at high resolution, and then writing the final samples to disk or directly to an audio output device. Because processing is performed in real time as files are read, the larger format samples do not need to be stored.

In-place editors typically read and write samples many times, once for each operation. If levels are adjusted many times, significant quantization noise is at least possible. This is especially true if 16-bit samples are stored on disk. Our sequence-based editor performs mixing and applies envelopes in real time, avoiding any accumulation of quantization error as audio levels are adjusted and refined. However, effects such as equalization could cause accumulated quantization error if applied repeatedly. The efficient undo mechanism can help avoid this. In addition, we are modifying Audacity to allow a choice of either 16-bit integers or 32-bit floating point numbers for the disk-based representation.

Conclusion

We have found that by storing audio data in small blocks, we can achieve the speed and responsiveness of non-destructive editors, with the conve-

nience and simplicity of design of in-place editors. Not only can cuts and pastes be done in near constant time, but also unlimited undo and redo can 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 and allow the class to handle the internal details of the structure itself. As a result, we believe that many people would benefit from using this data structure as an alternative to storing audio data in a large flat file or using edit decision lists.

The major difference between non-destructive editors and our sequence-based approach is the handling of effects and signal processing. Non-destructive editors have the advantage of instant changes to effects parameters. Our sequence-based approach has the advantages that effects need not run in real time, and the results of effects are visible on the display. Another advantage is greater predictability of playback performance, because files have a minimum length, and effects (other than simple mixing) are not computed in real time.

Both systems seem superior in almost all ways to in-place editors. The single advantage of in-place editors is that the state of audio is always maintained in a single contiguous file. A single file is easy for users to copy or import into another application. In contrast, non-destructive editors rely on a set of source files that the user must manage. Our sequence structure uses a directory of small files, and while it can be annoying to deal with large numbers of small data files manually, users do not need to manage source and intermediate files explicitly. Any modern operating system can handle tens of thousands of files in a single directory without any problems. However, the user interfaces by which people interact with file systems often have issues with this number of files, whether one uses UNIX command-line tools or Microsoft's Windows Explorer. It is also noticeably slower to delete one thousand files than one big one.

The best editor depends upon the application and user preferences. Non-destructive editors seem well-suited to routine studio production, where effects are applied after selecting, sequencing, and mixing. In many cases, it is more desirable to hear the results of many simultaneous mixer and effects settings than to see them on screen or to perform and audition each step in sequence. The ability to render equalization, panning, mixing, and reverberation effects all at once is important for studio production work. In more creative and experimental situations, we believe our new approach may be preferable, because it allows editing operations in any order and full display of the results of each step, all with reasonable efficiency.

Hybrid Approaches

Based on our observations of advantages and disadvantages in current editors and in our own approach, it may be that a hybrid approach could be even better. We are already implementing some non-destructive editor concepts in Audacity. In particular, we allow standard audio files to be edited without copying their data into a sequence structure. Instead, we build the structure with pointers to the original file. We also allow amplitude envelope editing without computing and storing samples, and we perform multiple-track mixing during playback. Further enhancements could allow for real-time post-processing effects, making Audacity more suitable for conventional studio production work.

In-place editors might consider using sequence structures to optimize performance with larger files. To avoid additional cost, the sequence structure could be constructed the first time the user performs a large undoable edit or an operation that requires copying most of the file. Alternatively, the sequence could be created incrementally as the user works or in the background as the user thinks. The edited structure could be ''flattened'' back into the original file when the editing session is closed.

Non-destructive editors could use our sequence structures to hold the results of non-real-time effects. The reference counting and undo mechanisms we describe might allow users to work with less explicit management of intermediate files. Sequence-like structures could also be useful for regions where there are many edits. For example, some non-destructive editors compute fades and

store them on disk (Moorer 1990). If there are many short edits and fades in succession, the computer may not be able to seek to and open files rapidly enough for real-time playback. In contrast, a sequence structure merges short edits and fades, maintaining a minimum node size on disk. This in turn provides more efficient playback.

Audacity

Audacity is quite fast and offers many useful editing functions. The editor's functionality has been enhanced by an interface to VST plug-ins so that Audacity can take advantage of many effects for multi-channel panning, equalization, compression, etc. It also calculates and displays spectrograms and supports text annotations, making it useful for research in sound analysis and synthesis. In the future, we plan to add an interface to Nyquist (Dannenberg 1997), allowing users to create their own effects in a high-level language.

We find this approach to be attractive for data visualization. It is fast, convenient, and always displays the actual results of signal processing operations. Our cross-platform implementation has already become popular (based on download statistics) for general-purpose audio editing, even though our implementation lacks an elaborate user interface such as those found in high-end commercial editors. We invite researchers to try Audacity and to collaborate on its further development. Even if Audacity does not become one's favorite editor, we think the combination of high performance with a simple interface will exert a positive influence on many commercial systems as they evolve. We hope that our analysis of editor performance and features, as well as the features of our new approach, will inspire others to make further improvements.

Acknowledgments

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

References

Abbott, C. 1984. "Efficient editing of digital sound on disk." *Journal of the Audio Engineering Society* 32(6):394–402.

Anderson, D. P., Y. Osawa, and R. Govindan. 1992. "A File System for Continuous Media." *ACM Transactions on Computer Systems* November:311–377.

Chen, P. M., et al. 1994. "RAID: High-Performance, Reliable Secondary Storage." *ACM Computing Surveys* 26(2):145–185.

Dannenberg, R. B. 1997. "Nyquist, a Language for Composition and Sound Synthesis." *Computer Music Journal* 21(3):50–60.

Freed, A. 1987. "Recording, Mixing, and Signal Processing on a Personal Computer." *Proceedings of the AES 5th International Conference: Music and Technology*. New York: Audio Engineering Society, p. 158.

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

Moorer, J. A. 1990. "Hard-Disk Recording and Editing of Digital Audio." *Proceedings of the 89th Convention of the Audio Engineering Society.* New York: Audio Engineering Society.