Manipulating Lossless Video in the Compressed Domain

William Thies
Microsoft Research India
thies@microsoft.com

Steven Hall
Massachusetts Institute of Technology
shall@alum.mit.edu

Saman Amarasinghe
Massachusetts Institute of Technology
saman@mit.edu

ABSTRACT

A compressed-domain transformation is one that operates directly on the compressed format, rather than requiring conversion to an uncompressed format prior to processing. Performing operations in the compressed domain offers large speedups, as it reduces the volume of data processed and avoids the overhead of re-compression.

While previous researchers have focused on compressed-domain techniques for lossy data formats, there are few techniques that apply to lossless formats. In this paper, we present a general technique for transforming lossless data as compressed with the sliding-window Lempel Ziv algorithm (LZ77). We focus on applications in video editing, where our technique supports color adjustment, video compositing, and other operations directly on the Apple Animation format (a variant of LZ77).

We implemented a subset of our technique as an automatic program transformation. Using the StreamIt language, users write a program to operate on uncompressed data, and our compiler transforms the program to operate on compressed data. Experiments show that the technique offers speedups roughly proportional to the compression factor. For our benchmark suite of 12 videos in Apple Animation format, speedups range from 1.1x to 471x, with a median of 15x.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications–Dataflow languages; D.3.4 [**Programming Languages**]: Processors–Compilers, Optimization; I.4.2 [**Compression**]; H.5.1 [**Multimedia Information Systems**]

General Terms

Algorithms, Design, Experimentation, Languages, Performance

1. INTRODUCTION

In order to accelerate the process of editing compressed data, researchers have identified specific transformations that can be mapped into the compressed domain—that is, they can operate directly on the compressed data format rather than on the uncompressed format [4, 12, 22, 28]. In addition to avoiding the cost of the decompression and re-compression, such techniques greatly reduce the total volume of data processed, thereby offering large savings in both execution time and memory footprint. However, existing techniques for operating directly on compressed data are largely limited to lossy compression formats such as JPEG [6, 8, 14, 19, 20, 23] and MPEG [1, 5, 15, 27, 28]. While these formats are used pervasively in the distribution of image and video content, they are rarely used during the production of such content. Instead, professional artists and filmmakers rely on lossless compression formats (BMP, PNG, Apple Animation) to avoid accumulating artifacts during the editing process. Given the computational intensity of professional video editing, there is a large demand for new techniques that could accelerate operations on lossless formats.

In this paper, we present a technique for translating a specific class of computations to operate directly on losslessly-compressed data. We consider compression formats that are based on LZ77, a compression algorithm that is utilized by ZIP and fully encapsulates common formats such as Apple Animation, Microsoft RLE, and Targa. Our transformation applies to a restricted class of programs, termed *stream programs* [26], that operate on continuous streams of data. The transformation is most efficient when each element of the stream is transformed in a uniform way (e.g., adjusting the brightness of each pixel). However, it also applies to cases in which multiple items are processed at once (e.g., averaging pixels) or in which multiple streams are split or combined (e.g., compositing frames). The precise coverage of our transformation is defined in Section 2.

The key idea behind our technique can be understood in simple terms. In LZ77, compression is achieved by indicating that a given part of the data stream is a repeat of a previous part of the stream. If a program is transforming each element of the stream in the same way, then any repetitions in the input will necessarily be present in the output as well. Thus, while new data sequences need to be processed as usual, any repeats of those sequences do not need to be transformed again. Rather, *the repetitions in the input stream can be directly copied to the output stream*, thereby referencing the previously-computed values. This preserves the compression in the stream while avoiding the cost of decompression, re-compression, and computing on the uncompressed data.

In this paper, we extend this simple idea to encompass a broad class of programs that can be expressed in the StreamIt programming language [26]. We have implemented a subset of our general technique in the StreamIt compiler. The end result is a fully-automatic system in which the user writes programs that operate on uncompressed data, and our compiler emits an optimized program that operates directly on compressed data. Our compiler generates plugins for two popular video editing tools (MEncoder and Blender), allowing the optimized transformations to be used as part of a standard video editing process.

Using a suite of 12 videos (screencasts, animations, and stock footage) in Apple Animation format, our transformation offers a speedup roughly proportional to the compression factor. For trans-

MM'09, October 19–24, 2009, Beijing, China.

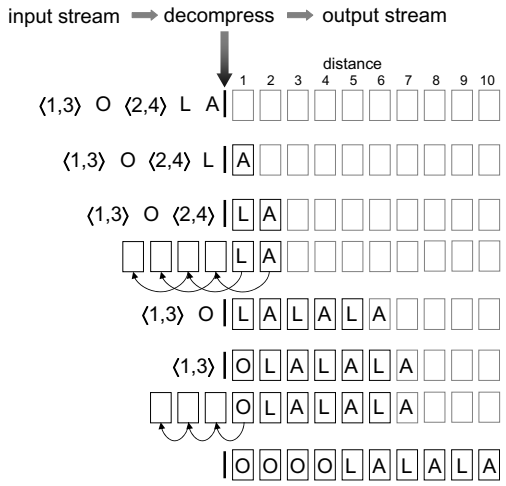


Figure 1: Example of LZ77 decompression.

formations that adjust a single video (brightness, contrast, color inversion), speedups range from 2.5x to 471x, with a median of 17x. For transformations that combine two videos (overlays and mattes), speedups range from 1.1x to 32x, with a median of 6.6x. We believe this is the first demonstration of compressed-domain techniques for losslessly compressed video content.

To summarize, this paper makes the following contributions:

- An algorithm for mapping an arbitrary stream program to operate directly on lossless LZ77-compressed data. In addition to transforming a single stream, programs may interleave and de-interleave multiple streams while maintaining compression (Sections 2-3).
- An implementation of a subset of our techniques in the StreamIt compiler. Starting from videos in the Apple Animation format, our implementation supports transformation and combination of individual pixel streams (Section 4).
- An experimental evaluation of our system, demonstrating that automatic translation to the compressed domain can speedup realistic operations in popular video editing tools. Across our benchmarks, the median speedup is 15x (Section 5).

The paper concludes with related work (Section 6) and conclusions (Section 7).

2. PROGRAM REPRESENTATION

Our transformation requires a specific representation for compressed data, as well as for programs that transform the data. The data must be compressed with LZ77, while the transformation must be expressed in the cyclo-static dataflow model (e.g., in the StreamIt language).

2.1 LZ77 Compression

LZ77 is a lossless, dictionary-based compression algorithm that is asymptotically optimal [29]. LZ77 forms the basis for many popular compression formats, including ZIP, GZIP and PNG, and also serves as a generalization of simpler encodings such as Apple Animation, Microsoft RLE, and Targa.

The basic idea behind LZ77 is to utilize a sliding window of recently encoded values as the dictionary for the compression algorithm. In the compressed data stream, there are two types of tokens: *values* and *repeats*. A value indicates a token that should be copied directly to the output of the decoded stream. A repeat $\langle d, c \rangle$ contains two parts: a distance d and a count c. It indicates that the decoder should start at offset d from the end of the decoded stream and copy a sequence of c values to the output. It is important to note that the count may exceed the distance, in which case some of the values produced by a repeat operation are also copied by that operation. For example, a value A followed by a repeat $\langle 1, 3 \rangle$ results in an output of "A A A". An additional example is given in Figure 1.

```
struct rgb {
    byte r, g, b;
}
rgb->rgb pipeline HalfSize {
    add splitjoin {
        split roundrobin(WIDTH,WIDTH);
        add Identity<rgb>();
        add Identity<rgb>();
        join roundrobin(1,1);
    }
    add AveragePixels(4);
}
rgb->rgb filter AveragePixels(int N) {
    work push 1 pop N {
        rgb out; int r, g, b;
        for (int i=0; i<N; i++) {
            rgb in = pop();
            r += in.r; g += in.g; b += in.b;
        }
        out.r = r/N; out.g = g/N; out.b = b/N;
        push(out);
    }
}
rgb->rgb filter InvertColor() {
    work push 1 pop 1 {
        rgb pixel = pop();
        pixel.r = 255 – pixel.r;
        pixel.g = 255 – pixel.g;
        pixel.b = 255 – pixel.b;
        push(pixel);
    }
}
void->void pipeline Toplevel() {
    add ReadRGB();
    add HalfSize();
    add InvertColor();
    add WriteRGB();
}
```

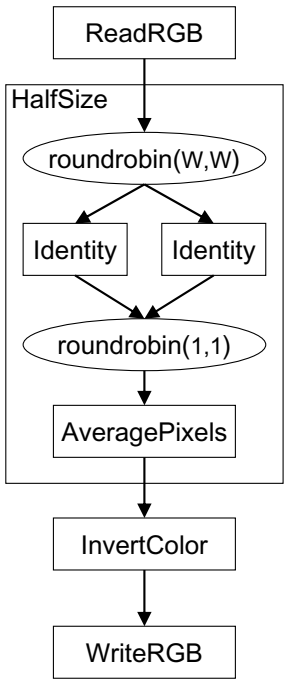


Figure 2: Example StreamIt program.

2.2 Cyclo-Static Dataflow

The basic idea behind cyclo-static dataflow [3, 10] is to represent a program as a block diagram, in which independent components (called *actors*) communicate over FIFO data channels (see Figure 2). One can think of an actor as a function (called repeatedly by the runtime system) that consumes items from the input tapes and produces items on the output tapes. Actors must declare how many items they will produce and consume on each execution; these I/O rates must either be constant across all executions, or must follow a fixed cycle. In this paper, actors may not retain internal state between executions.

We use the StreamIt language [26] to express cyclo-static dataflow programs. An example StreamIt program appears in Figure 2. It reads lines of an RGB image from a file, shrinks the image by a factor of two, inverts the color of each pixel, then writes the data to a file. There are three kinds of actors in StreamIt, and our analysis handles each one separately:

- **Filters** have a single input stream and a single output stream, and perform general-purpose computation. For example, in the `InvertColor` filter, the `work` function specifies the atomic execution step; it declares that on each execution, it pops (inputs) 1 item from the input tape and pushes (outputs) 1 item to the output tape.

Execute a filter in the compressed domain, given that it consumes n items and produces m items on each execution.

EXECUTE-COMPRESSED-FILTER (int n, int m) {
  while true {
    / pass-uncompressed */*
    if input **endswith** n values **then**
      execute one call to uncompressed filter

    / pass-compressed */*
    else if input **endswith** $\langle d, c\rangle$ **and** $d\%n = 0$ **and** $c \geq n$ **then**
      replace $\langle d, c\rangle$ with $\langle d, c\%n\rangle$ on input
      push $\langle m\ d/n, m\ (c - c\%n)/n\rangle$ to output

    else
      let $\langle d, c\rangle$ = last repeat on input

      / coarsen-repeat */*
      let $L = \text{LCM}(d, n)$
      if $d < L < c$ **then**
        replace $\langle d, c\rangle$ with $\langle c - (L - d)\rangle, \langle d, L - d\rangle$ on input

      / expand */*
      else if $c > 0$ **then**
        decode $\langle d, c\rangle$ into $\langle d, c - 1\rangle, V$ on input

      / prune */*
      else */* c = 0 */*
        remove $\langle d, c\rangle$ from input
  }
}

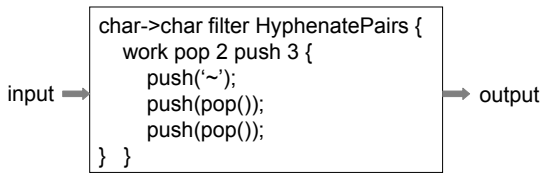
Figure 3: Translation of filters into the compressed domain. We use $\%$ to denote a modulo operation.

- **Joiners** interleave multiple input streams into a single output stream. Items are interleaved in a roundrobin pattern according to a set of *weights*; for example, weights of (n_1, n_2) indicate that the first n_1 items are drawn from the first input stream, and the next n_2 items are drawn from the second input stream. In Figure 2, the joiner reads one pixel at a time from each input stream, serving to interleave the pixels from neighboring lines. Once the pixels are interleaved, each group of 4 pixels is averaged together in order to decrease the picture width by two.
- **Splitters** have a single input stream and multiple output streams. They may perform either roundrobin distribution, or duplicate distribution. In Figure 2, the splitter sends `WIDTH` pixels in each direction, distributing the lines of the image across alternate streams.

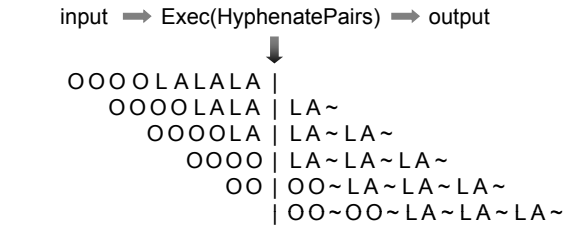
3. MAPPING TO COMPRESSED DOMAIN

Our technique allows any cyclo-static dataflow program to operate directly on LZ77-compressed data. Rather than modifying the code within the actors, our transformation treats actors as black boxes and wraps them in a new execution layer. The transformation attempts to preserve as much compression as possible without ever performing an explicit re-compression step. While there exist cases in which the output data will not be as compressed as possible, under certain conditions the output is guaranteed to be fully compressed (relative to the compression of the input). We quantify this issue later.

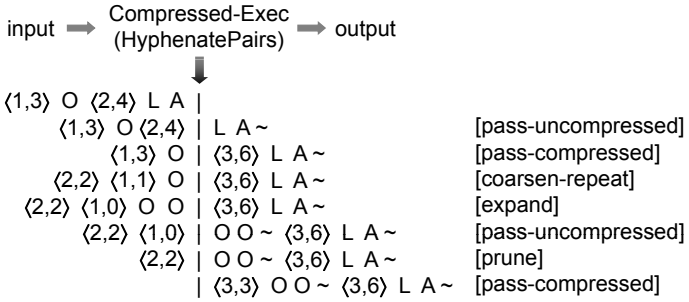
To describe the mapping into the compressed domain, we consider each StreamIt construct in turn. An alternate formulation (in terms of an operational semantics) is available elsewhere [25].



(a) Example filter



(b) Normal execution



(c) Compressed-domain execution

Figure 4: Example execution of a filter in the uncompressed and compressed domains.

3.1 Filters

The procedure for translating a filter into the compressed domain is given in Figure 3, and an example appears in Figure 4. The behavior of the compressed-domain filter can be considered in two pieces. The first piece consists of the simple case (annotated *pass-uncompressed* in the code) in which the upcoming inputs to the filter are uncompressed values. In this case, the original filter is called with those inputs, transforming n input items to m output items. The rest of the code deals with repeat tokens, attempting to translate them across the filter with the minimum decompression needed.

The **key idea of the paper** is encapsulated in the *pass-compressed* case in Figure 3. This case specifies how to translate a repeat token directly from a filter's input tape to a filter's output tape without invoking the filter's computation. This translation is possible whenever the repeat distance d is a multiple of the filter's input rate n. In other words, the repeat is aligned with the execution boundaries of the actor, so invoking the actor would produce the same results as before. In transferring the repeat token to the output tape, two adjustments are made: 1) the distance and count are scaled by a factor of m/n, since the repeat now refers to the m outputs of the filter rather than the n inputs, and 2) if the count is not an even multiple of the input rate, then some leftover items ($c\%n$, where $\%$ represents the modulo operation) are left on the input tape.

In cases where the repeat distance does not match the granularity of the actor, the distance can sometimes be adjusted to allow compressed-domain processing. The *coarsen-repeat* logic in Figure 3 represents such an adjustment. Consider that a filter inputs

two items at a time, but encounters a long repeat with distance three and count 100. That is, the input stream contains a regular pattern of values with periodicity three. Though consecutive executions of the filter are aligned at different offsets in this pattern, every third filter execution (spanning six values) falls at the same alignment. In general, a repeat with distance d can be exploited by a filter with input rate n by expanding the distance to LCM(d, n). In order to perform this expansion, the count must be greater than the distance, as otherwise the repeat references old data that may have no periodicity. Also, the stream needs to be padded with LCM $- d$ values before the coarsened repeat can begin; this padding takes the form of a shorter repeat using the original distance.

A second way to adjust a repeat token for compressed-domain processing is by changing its count rather than its distance (case *expand* in Figure 3). This case applies if a repeat has a count less than n, if it is unaligned with the boundaries of an actor's execution, or if its distance is not a multiple of n (and cannot be coarsened appropriately). The expand logic decodes a single value from a repeat token, thereby decreasing its count by one; the rest of the repeat may become aligned later. If the count of a repeat reaches zero, it is eliminated in the *prune* case.

Note that the *expand* logic requires partial decompression of the data stream. In order to perform this decompression, it may be necessary to maintain an auxiliary data structure–separate from the filter's input stream–that holds a complete window of decompressed data. This auxiliary structure is needed because the sliding-window dictionary of LZ77 makes it difficult to decode one element without decoding others. However, even if the stream is fully decompressed in parallel with the main computation, our technique retains many benefits because the filters still operate on the compressed stream; the volume of data processed is reduced, and the cost of re-compression is averted. For general algorithms such as gzip, compression can be up to 10x slower than decompression [30].

3.2 Splitters

Duplicate splitters are trivial to transform to the compressed domain, as all input tokens (both values and repeats) are copied directly to the output streams. For roundrobin splitters, the central concern is that a repeat token can only be transferred to a given output tape if the items referenced are also on that tape. If the items referenced by the repeat token were distributed to another tape, then the repeat must be decompressed.

The rest of this section focuses on roundrobin splitters. To simplify the presentation, we consider a splitter with only two output streams, distributing m_1 and m_2 items to each respective stream. This case captures all of the fundamental ideas; extension to additional streams is straightforward. In addition, we use the following notations:

- As mentioned previously, splitters adopt a fine-grained cyclostatic execution model, in which each execution step transfers only one item from the input tape. That is, a roundrobin(m_1, m_2) splitter has $m_1 + m_2$ distinct execution steps. We refer to every group of $m_1 + m_2$ steps as an *execution cycle*.
- The pseudocode for our algorithm assumes, without loss of generality, that the next execution step of the splitter will write to the first output stream (output1).
- We use *pos* to denote the number of items (in terms of the uncompressed domain) that have already been written to the current output stream (output1) in the current execution cycle. For brevity, the pseudocode does not maintain the value of *pos*, though it is straightforward to do so.

Execute a roundrobin splitter in the compressed domain, given that it outputs m_1 items to output1 and m_2 items to output2 on each execution cycle.

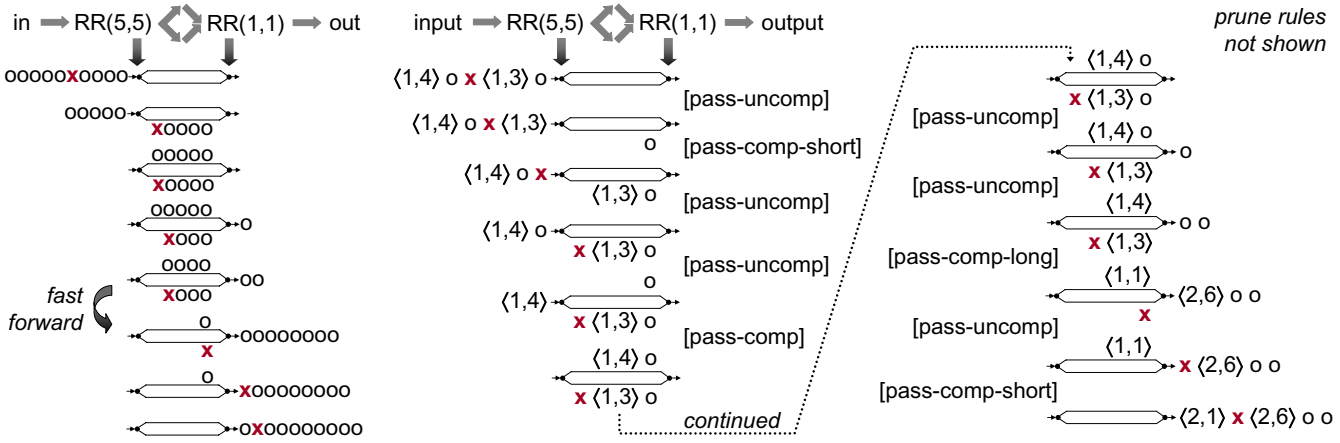
```
EXECUTE-COMPRESSED-SPLITTER (int m1, int m2) {
  while true {
    /* pass-uncompressed */
    if input endswith value then
      transfer value from input to output1

    else
      let ⟨d, c⟩ = end of input
      let offset = d%(m1 + m2)

      /* pass-compressed-long */
      if offset = 0 then
        let (L1, L2) = SPLIT-TO-BOTH-STREAMS(c)
        pop ⟨d, c⟩ from input
        push ⟨d m1/(m1 + m2), L1⟩ to output1
        push ⟨d m2/(m1 + m2), L2⟩ to output2

      /* pass-compressed-short */
      else if SPLIT-TO-ONE-STREAM(d, c) > 0 then
        let offset' = if offset ≤ pos then offset else offset − m2
        let L = SPLIT-TO-ONE-STREAM(d, c)
        replace ⟨d, c⟩ with ⟨d, c − L⟩ on input
        push ⟨m1 floor(d/(m1 + m2)) + offset', L⟩ to output1

      /* expand */
      else /* SPLIT-TO-ONE-STREAM(d, c) = 0 */
        decode ⟨d, c⟩ into ⟨d, c − 1⟩, V on input

      /* prune */
      if input endswith ⟨d, 0⟩ then
        pop ⟨d, 0⟩ from input
  }
}
```

Figure 5: Translation of splitters into the compressed domain.

The procedure for executing a roundrobin splitter in the compressed domain appears in Figure 5, while an example appears in Figure 6. As mentioned previously, a repeat token can be transferred to an output tape so long as the items referenced also appear on that tape. However, the repeat may need to be fragmented (into several repeats of a lesser count), depending on the repeat distance. There are two cases to consider.

The first case, called *pass-compressed-long* in Figure 5, distributes an entire repeat token to both output tapes without any fragmentation. This is only possible when the repeat can be cleanly separated into two independent sequences, one offset by m_1 and the next offset by m_2. In other words, the repeat distance must be a multiple of $m_1 + m_2$. In this case, the repeat token is moved to the output streams. The repeat distance is scaled down to match the weight of each stream, and the count is divided according to the current position of the splitter (a simple but tedious calculation implemented by SPLIT-TO-BOTH-STREAMS in Figure 7).

The second case, called *pass-compressed-short*, is when the repeat distance is mis-aligned with the splitter's execution cycle, and thus the repeat (if it is long enough) eventually references items that are distributed to a different output tape. Nonetheless, part of the repeat may be eligible to pass through, so long as the items referenced refer to the current output tape. This judgment is performed by SPLIT-TO-ONE-STREAM (Figure 9) by comparing the repeat



(a) Normal execution

(b) Compressed-domain execution of splitter (left) and joiner (right)

Figure 6: Example execution of splitters and joiners in the compressed domain. As illustrated by the input/output pairs in Figure 8, the example performs a transpose of a 2x5 matrix. When the matrix is linearized as shown here, the input stream traverses the elements row-wise while the output stream traverses column-wise. Due to redundancy in the matrix, this reordering can be done largely in the compressed domain.

Given that c items are available on input stream of a splitter, returns the number of items that can be written to each output stream before the input is exhausted. Assumes that the splitter is currently writing to the first output stream, to which pos items have previously been written in the current execution cycle.

SPLIT-TO-BOTH-STREAMS (int c) returns (int, int) {
  // the number of complete splitter cycles, and the leftover
  let total_cycles = floor(min($c_1/n_1, c_2/n_2$))
  let total_leftover = $c\%(m_1 + m_2)$

  // the last partial cycle may end in three regions:
  if total_leftover $\leq m_1 - pos$ **then**
    // 1. in writing to the first output stream
    L_1 = total_leftover
    $L_2 = 0$
  else if total_leftover $\leq m_1 - pos + m_2$ **then**
    // 2. in subsequent writing to the second output stream
    $L_1 = m_1 - pos$
    L_2 = total_leftover $- m_1 - pos$
  else
    // 3. in wrap-around writing to the first output stream
    L_1 = total_leftover $- m_2$
    $L_2 = m_2$

  return $(m_1 *$ total_cycles $+ L_1, m_2 *$ total_cycles $+ L_2)$
}

Figure 7: The SPLIT-TO-BOTH-STREAMS function is called during compressed splitter execution. In the case where an input token can be split across both output streams, it calculates the maximum numbers of items that can be written to the outputs before the input is exhausted.

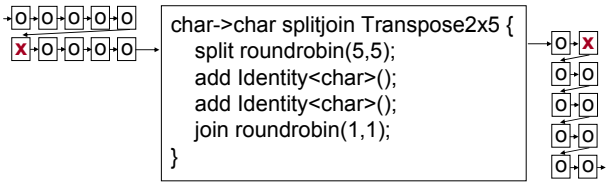


Figure 8: Example splitter and joiner in StreamIt, which combine to form a Transpose. Translation to the compressed domain is illustrated in Figure 6.

Given a repeat token with distance d and count c that is input to a splitter, and that is not possible to divide across both output streams of the splitter, returns the maximum count of a repeat token that could safely be emitted to the current output stream of the splitter.

SPLIT-TO-ONE-STREAM (int d, int c) returns int {
  let offset $= d\%(m_1 + m_2)$
  if offset $\leq pos$ **then**
    // repeat for remainder of this execution cycle
    return $\min(c, m_1 - pos)$
  else if offset $> m_2 + pos$ **then**
    // repeat until referenced data goes out of range
    return $\min(c,$ offset $- (m_2 + pos))$
  else
    // referenced data is on the other output stream
    return 0
}

Figure 9: The SPLIT-TO-ONE-STREAM function is called during compressed splitter execution. In the case where an input token cannot be split across both output streams, it calculates the maximum number of items that can be passed to a single output stream.

distance to the current position in the output stream. If one or more of the repeated values are in range, the valid segment of the repeat (of length *actual_repeat*) is moved to the output tape. As before, the repeat distance needs to be scaled according to the weights of the splitter, and an extra offset is needed if the repeat distance wraps around to reference the end of a previous cycle.

If neither of the above transfers apply, then the input stream needs to be partially decompressed (according to the *expand* case) because the current repeat token references items that will be sent to the wrong output tape. The *prune* case is also needed to clear empty repeats generated by *expand*.

Execute a roundrobin joiner in the compressed domain, given that it inputs n_1 items from input1 and n_2 items from input2 on each execution cycle.

```
EXECUTE-COMPRESSED-JOINER (int n1, int n2) {
  while true {
    /* pass-uncompressed */
    if input1 endswith value then
      transfer value from input1 to output

    /* pass-compressed-long */
    else if input1 endswith ⟨d1, c1⟩ and d1%n1 = 0
        and input2 endswith ⟨d2, c2⟩ and d2%n2 = 0
        and d1/n1 = d2/n2 then
      let (L1, L2) = JOIN-FROM-BOTH-STREAMS(c1, c2)
      replace ⟨d1, c1⟩ with ⟨d1, c1 − L1⟩ on input1
      replace ⟨d1, c2⟩ with ⟨d2, c2 − L2⟩ on input2
      push ⟨d1(n1 + n2)/n1, L1 + L2⟩ to output

    /* pass-compressed-short */
    else /* input1 endswith ⟨d, c⟩ and c > 0 */
      let offset = if d%n1 ≤ pos then pos else d%n1 + n2
      let L = JOIN-FROM-ONE-STREAM(d, c))
      replace ⟨d, c⟩ with ⟨d, c − L⟩ on input1
      push ⟨(n1 + n2) floor(d/n1) + offset, L⟩ to output

    /* prune */
    if input1 endswith ⟨d, 0⟩ then
      pop ⟨d, 0⟩ from input1
    if input2 endswith ⟨d, 0⟩ then
      pop ⟨d, 0⟩ from input2
  }
}
```

Figure 10: Translation of joiners into the compressed domain.

3.3 Joiners

The procedure for executing a joiner in the compressed domain appears in Figure 10, while an example appears in Figure 6. Our formulation for joiners uses analogous notations as our formulation for splitters:

- A roundrobin(n_1, n_2) joiner has n_1+n_2 execution steps. We refer to every group of $n_1 + n_2$ steps as an *execution cycle*.
- We assume, without loss of generality, that the next execution step of the joiner will read from the first input stream (input1).
- We use *pos* to denote the number of items (in terms of the uncompressed domain) that have already been read from the current input stream (input1) in the current execution cycle.

There are two ways to pass repeat tokens through a joiner. If the input streams contain compatible repeat tokens, then they can be combined into a long repeat that spans multiple execution cycles; otherwise, a shorter repeat is extracted from only one of the streams.

The first and most powerful way to execute joiners in the compressed domain is to combine repeat tokens from both input streams (case *pass-compressed-long* in Figure 10). For this to be possible, both repeat distances must be the same multiple of their respective joiner weight (n_1 or n_2); the combined token has a repeat distance that is a multiple of $n_1 + n_2$. The JOIN-FROM-BOTH-STREAMS routine (detailed in Figure 11) calculates the maximum repeat length depending on the current position of the joiner and the repeat lengths of the inputs.

Given that c_1 and c_2 compressed items are available on the first and second input streams of a joiner, returns the number of items that can be read from each input before one of them is exhausted. Assumes that the joiner is currently reading from the first input stream, from which pos items have previously been consumed in the current execution cycle.

```
JOIN-FROM-BOTH-STREAMS (int c1, int c2) returns (int, int) {
  // the number of complete joiner cycles, and the leftovers
  let total_cycles = floor(c/(n1 + n2))
  let leftover1 = c1 − total_cycles ∗ n1
  let leftover2 = c2 − total_cycles ∗ n2

  // the last partial cycle may end in three regions:
  if leftover1 ≤ n1 − pos then
    // 1. in reading from the first input stream
    L1 = leftover1
    L2 = 0
  else if leftover2 ≤ n2 then
    // 2. in subsequent reading from the second input stream
    L1 = n1 − pos
    L2 = leftover2
  else
    // 3. in wrap-around reading from the first input stream
    L1 = leftover1
    L2 = n2

  return (n1 ∗ total_cycles + L1, n2 ∗ total_cycles + L2)
}
```

Figure 11: The JOIN-FROM-BOTH-STREAMS function is called during compressed joiner execution. In the case where the input tokens to the joiner have compatible repeat distances, it calculates the maximum repeat lengths that can be passed to the output.

Given a repeat token with distance d and count c on the current input stream of a joiner, and that cannot be combined with a token on the other input of the joiner, returns the maximum count of a repeat token that could safely be emitted to the output stream.

```
JOIN-FROM-ONE-STREAM (int d, int c) returns int {
  let offset = d%n1
  if offset ≤ pos then
    // repeat for remainder of this execution cycle
    return min(c, n1 − pos)
  else
    // repeat until referenced data goes out of range
    return min(c, offset − pos)
}
```

Figure 12: The JOIN-FROM-ONE-STREAM function is called during compressed joiner execution. In the case where the input tokens to the joiner have incompatible repeat distances, it calculates the maximum length of the current token that may be passable to the output.

The second mode of compressed joiner execution (*pass-compressed-short*) inputs only a single repeat token, extracting the maximum length that can safely move to the output. The JOIN-FROM-ONE-STREAM routine (detailed in Figure 12) determines how much of the repeat can be moved to the output before the data referenced would have originated from a different input stream.

	VIDEO	DESCRIPTION	SOURCE	DIMENSIONS	FRAMES	SIZE (MB)	COMPRESSION FACTOR
Internet Video	screencast-demo	Online demo of an authentication generator	Software website	691 x 518	10621	38	404.8
	screencast-ppt	Powerpoint presentation screencast	Self-made	691 x 518	13200	26	722.1
	logo-head	Animated logo of a small rotating head	Digital Juice	691 x 518	10800	330	46.8
	logo-globe	Animated logo of a small rotating globe	Digital Juice	691 x 518	10800	219	70.7
Computer Animation	anim-scene1	Rendered indoor scene	Elephant's Dream	720 x 480	1616	10	213.8
	anim-scene2	Rendered outdoor scene	Elephant's Dream	720 x 480	1616	65	34.2
	anim-character1	Rendered toy character	Elephant's Dream	720 x 480	1600	161	13.7
	anim-character2	Rendered human characters	Elephant's Dream	720 x 480	1600	108	20.6
Digital Television	digvid-background1	Full-screen background with lateral animation	Digital Juice	720 x 576	300	441	1.1
	digvid-background2	Full-screen background with spiral animation	Digital Juice	720 x 576	300	476	1.0
	digvid-matte-frame	Animated matte for creating new frame overlays	Digital Juice	720 x 576	300	106	4.7
	digvid-matte-third	Animated matte for creating new lower-third overlays	Digital Juice	720 x 576	300	51	9.7

Table 1: Characteristics of the video workloads.

4. IMPLEMENTATION

As an initial demonstration of the potential benefits of mapping into the compressed domain, we implemented a core subset of our transformations as part of the StreamIt compiler. Our current implementation supports two computational patterns: 1) transforming each individual element of a stream (via a pop-1, push-1 filter), and 2) combining the elements of two streams (via a roundrobin(1,1) joiner and a pop-2, push-1 filter). The program can contain any number of filters that perform arbitrary computations, so long as the I/O rates match these patterns. While we look forward to performing a broader implementation in the future, these two building blocks are sufficient to express a number of useful programs and to characterize the performance of the technique.

Our implementation supports videos compressed with the Apple Animation codec. Supported as part of the Quicktime `.mov` format, Apple Animation serves as an industry standard for exchanging lossless computer animations and digital video content before they are rendered to lossy formats for final distribution [2, p. 106][9, p. 284] [11, p. 367][18, p. 280]. The Animation codec represents a restricted form of LZ77 in which repeat distances are limited to two values: a full frame or a single pixel. A repeat across frames indicates that a stretch of pixels did not change from one frame to the next, while a repeat across pixels indicates that a stretch of pixels has the same color within a frame.

To build a practical toolchain for video editing, we configured StreamIt to output plugins for two popular video editing tools: MEncoder and Blender. In the final workflow, the programmer authors StreamIt source code to operate on pixels from each frame of a video; the StreamIt compiler maps the computation into the compressed domain; and the compiler emits executable plugins for the tools, which can be used as part of a normal editing process.

5. EXPERIMENTAL EVALUATION

Our evaluation focuses on applications in digital video editing. Our benchmarks fall into two categories: 1) pixel transformations, such as brightness, contrast, and color inversion, which adjust pixels within a single video, and 2) video compositing, in which one video is combined with another as an overlay or mask.

The main results of our evaluation are:

- Operating directly on compressed data offers a speedup roughly proportional to the compression factor in the resulting video.
- For pixel transformations, speedups range from 2.5x to 471x, with a median of 17x. Output sizes are within 0.1% of input sizes and about 5% larger (median) than a full re-compression.
- For video compositing, speedups range from 1.1x to 32x, with a median of 6.6x. Output files retain a sizable compression ratio (1.0x to 44x) and are about 52% larger (median) than a full re-compression.

5.1 Video Workloads

Our evaluation utilizes a suite of 12 video workloads that are described in Table 1 and detailed further in an extended report [24]; some of the videos are also pictured in Figure 14. The suite represents three common usage scenarios for lossless video formats: Internet screencasts, computer animation, and digital television production. While videos in each area are often rendered to a lossy format for final distribution, lossless codecs are preferred during the editing process to avoid accumulating compression artifacts. All of our source videos are in the Apple Animation format (described in Section 4), which is widely used by video editing professionals [2, p. 106] [9, p. 284] [11, p. 367] [18, p. 280]. The Apple Animation format is also popular for capturing video from the screen or camera, as the encoder is relatively fast.

5.2 Pixel Transformations

The pixel transformations adjust the color of each pixel in a uniform way. We evaluated three transformations:

- Brightness adjustment, which increases each RGB value by a value of 20 (saturating at 255).
- Contrast adjustment, which moves each RGB value away from the center (128) by a factor of 1.2 (saturating at 0 and 255).
- Color inversion, which subtracts each RGB value from 255 (useful for improving the readability of screencasts or for reversing the effect of video mattes).

We implemented each transformation as a single StreamIt filter that transforms one pixel to another. Because the filter has a pop rate of one, it does not incur any alignment overhead.

5.2.1 Setup

The pixel transformations were compiled into plugins for MEncoder, a popular command-line tool (bundled with MPlayer) for video decoding, encoding, and filtering. MEncoder relies on the FFMPEG library to decode the Apple Animation format; as FFMPEG lacked an encoder for this format, the authors implemented one. Additionally, as MEncoder lacks an interface for toggling only brightness or contrast, the baseline configuration was implemented by the authors.

The baseline configuration performs decompression, pixel transformations, then re-compression. Because the main video frame is updated incrementally by the decoder, the pixel transformations are unable to modify the frame in place (otherwise pixels present across

		SPEEDUP			OUTPUT SIZE / INPUT SIZE (Compute on Compressed Data)			OUPUT SIZE / INPUT SIZE (Uncompress, Compute, Re-Compress)		
	VIDEO	Brightness	Contrast	Inverse	Brightness	Contrast	Inverse	Brightness	Contrast	Inverse
Internet Video	screencast-demo	137.8x	242.3x	154.7x	1.00	1.00	1.00	0.90	0.90	1.00
	screencast-ppt	201.1x	470.6x	185.1x	1.00	1.00	1.00	0.75	0.74	1.00
	logo-head	27.0x	29.2x	25.2x	1.00	1.00	1.00	0.87	0.86	1.00
	logo-globe	35.7x	46.4x	36.6x	1.00	1.00	1.00	1.00	0.64	1.00
Computer Animation	anim-scene1	66.4x	124.3x	58.5x	1.00	0.98	1.00	0.99	0.92	1.00
	anim-scene2	19.3x	27.9x	20.5x	1.00	1.00	1.00	0.99	0.85	1.00
	anim-character1	11.5x	12.2x	11.2x	1.00	1.00	1.00	0.96	0.90	1.00
	anim-character2	15.6x	15.3x	14.8x	1.00	1.00	1.00	0.95	0.88	1.00
Digital Television	digvid-background1	4.6x	2.6x	4.6x	1.00	1.00	1.00	1.00	0.88	1.00
	digvid-background2	4.1x	2.5x	4.7x	1.00	1.00	1.00	0.92	0.91	1.00
	digvid-matte-frame	6.3x	5.3x	6.5x	1.00	1.00	1.00	0.98	0.64	1.00
	digvid-matte-third	7.5x	6.9x	8.9x	1.00	1.00	1.00	0.83	0.35	1.00

Table 2: Results for pixel transformations.

frames would be transformed multiple times). Thus, the baseline transformation writes to a separate location in memory. The optimized configuration performs pixel transformations directly on the compressed data, avoiding data expansion implied by decompression and multiple frame buffers, before copying the data to the output file.

Our evaluation platform is a dual-processor Intel Xeon (2.2 GHz) with 2 GB of RAM. As all of our applications are single-threaded, the second processor is not utilized.

5.2.2 *Results*

Detailed results for the pixel transformations appear in Table 2. As illustrated in Figure 13, the speedups range from 2.5x to 471x and are closely correlated with the compression factor in the original video.

There are two distinct reasons for the speedups observed. First, by avoiding the decompression stage, computing on compressed data reduces the volume of data that needs to be stored, manipulated, and transformed. This savings is directly related to the compression factor and is responsible for the upwards slope of the graph in Figure 13. Second, computing on compressed data eliminates the algorithmic complexity of re-compression. For the Apple Animation format, the cost of compressing a given frame does not increase with the compression factor (if anything, it decreases as fewer pixels need a fine-grained encoding). Thus, the baseline devotes roughly constant runtime to re-compressing each video, which explains the positive intercept in the graph of Figure 13.

The impact of re-compression is especially evident in the digital television examples. Despite a compression factor of 1.0 on `digvid-background2`, our technique offers a 4.7x speedup on color inversion. Application profiling confirms that 73% of the baseline runtime is spent in the encoder; as this stage is absent from the optimized version, it accounts for $1/(1-0.73) = 3.7$x of the speedup. The remaining speedup in this case is due to the extra frame buffer (and associated memory operations) in the decompression stage of the baseline configuration.

Another important aspect of the results is the size of the output files produced. Apart from the first frame of a video[1], performing pixel transformations directly on compressed data will never increase the size of the file. This is illustrated in the middle columns of Table 2, in which the output sizes are mostly equal to the input sizes (up to 2 decimal places). The only exception is contrast adjustment on `anim-scene1`, in which the output is 2% smaller than the input due to variations in the first frame; for the same reason, some cases experience a 0.1% increase in size (not visisble in the table).

[1] In the Apple Animation format, the first frame is encoded as if the previous frame was black. Thus, adjusting the color of black pixels in the first frame may increase the size of the file, as it removes inter-frame redundancy.

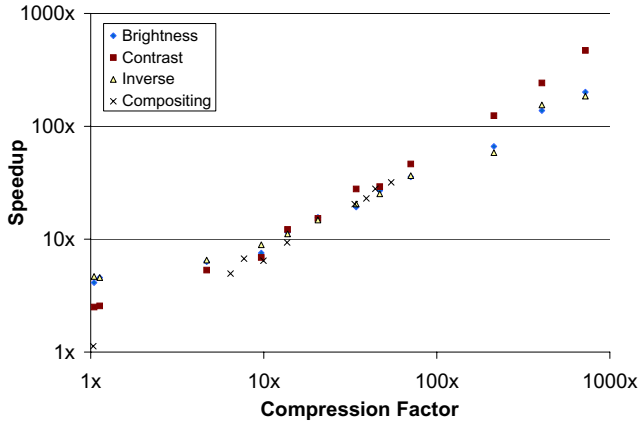


Figure 13: Speedup vs. compression factor for all transformations.

Though computing on compressed data has virtually no effect on the file size, there are some cases in which the pixel transformation increases the redundancy in the video and an additional re-compression step could compress the output even further than the original input. This potential benefit is illustrated in the last three columns of Table 2, which track the output size of the baseline configuration (including a re-compression stage) versus the original input. For the inverse transformation, no additional compression is possible because inverse is a 1-to-1 transform: two pixels have equal values in the output file if and only if they have equal values in the input file. However, the brightness and contrast transformations may map distinct input values to the same output value, due to the saturating arithmetic. In such cases, the re-compression stage can shrink the file to as low as 0.75x (brightness) and 0.35x (contrast) its original size. These are extreme cases in which many pixels are close to the saturating point; the median re-compression (across brightness and contrast) is only 10%.

To achieve the minimal file size whenever possible, future work could explore integrating a lightweight re-compression stage into the compressed processing technique. Because most of the compression is already in place, it should be possible to improve the compression ratio without running the full encoder (e.g., run-length encoded regions can be extended without being rediscovered).

	VIDEO COMPOSITE	EFFECT	SPEEDUP	COMPRESSION FACTOR		
				Compute on Compressed Data	Uncompress, Compute, Re-Compress	Ratio
Internet Video	screencast-demo + logo-head	alpha-under	20.46x	34	52	1.55
	screencast-demo + logo-globe	alpha-under	27.96x	44	61	1.39
	screencast-ppt + logo-head	alpha-under	22.99x	39	54	1.38
	screencast-ppt + logo-globe	alpha-under	31.88x	55	64	1.18
Computer Animation	anim-scene1 + anim-character1	alpha-under	6.72x	7.7	12	1.57
	anim-scene1 + anim-character2	alpha-under	9.35x	14	19	1.39
	anim-scene2 + anim-character1	alpha-under	4.96x	6.4	10	1.49
	anim-scene2 + anim-character2	alpha-under	6.45x	10	13	1.32
Digital Television	digvid-background1 + digvid-matte-frame	mul	1.23x	1.0	2.2	2.28
	digvid-background2 + digvid-matte-third	mul	1.13x	1.0	5.6	5.42
	digvid-background2 + digvid-matte-frame	mul	1.38x	1.0	1.8	1.84
	digvid-background2 + digvid-matte-third	mul	1.16x	1.0	4.8	4.91

Table 3: Results for composite transformations.

5.3 Video Compositing

In video compositing, two videos are combined using a specific function to derive each output pixel from a pair of input pixels (see Figure 14). In the case of subtitling, animated logos, and computer graphics, an alpha-under transformation is common; it overlays one video on top of another using the transparency information in the alpha channel. In applying an animated matte, the videos are combined with a multiply operation, thereby masking the output according to the brightness of the matte. For our experiments, we generated composites using each foreground/background pair within a given application area, yielding a total of 12 composites.

In StreamIt, we implemented each compositing operation as a roundrobin(1,1) joiner (to interleave the streams) followed by a filter (to combine the pixel values). The intuition of the compressed-domain execution is that if both streams have the same kind of repeat (inter-frame or intra-frame), then the repeat is copied directly to the output. If they have different kinds of repeats, or if one stream is uncompressed, then both streams are uncompressed.

5.3.1 Setup

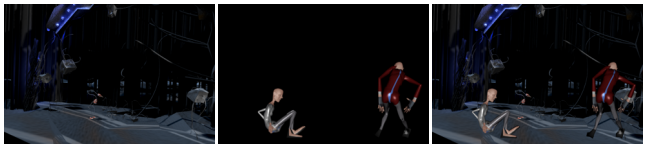
The compositing operations were compiled into plugins for Blender, a popular tool for modeling, rendering, and post-processing 3-D animations. As Blender already includes support for video compositing, we use its implementation as our baseline. The compositing operations have already been hand-tuned for performance; the implementation of alpha-under includes multiple shortcuts and unrolled loops. We further improved the baseline performance by patching other parts of the Blender source base, which were designed around 3-D rendering and are more general than needed for video editing. We removed two redundant vertical flips for each frame, two redundant BGRA-RGBA conversions, and redundant memory allocation/deallocation for each frame.

Our optimized configuration operates in the compressed domain. Outside of the auto-generated plugin, we patched three frame-copy operations in the Blender source code to copy only the compressed frame data rather than the full frame dimensions.

5.3.2 Results

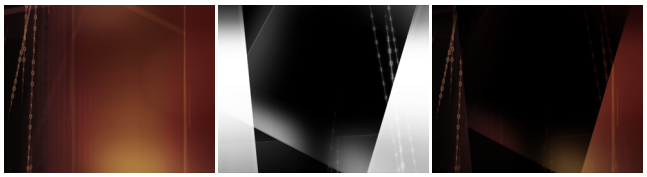
Full results for the compositing operations appear in Table 3; speedups range from 1.1x to 32x. As for the pixel transformations, the speedups are closely correlated with the compression factor of the resulting videos, a relationship depicted in Figure 13.

As in the case of the pixel transformations, the composite videos produced by the compressed processing technique would sometimes benefit from an additional re-compression stage. The last three columns in Table 3 quantify this benefit by comparing the compression factors achieved by compressed processing and normal processing (including a re-compression step). For screencasts and computer animations, compressed processing preserves a sizable compression factor (7.7x-44x), though the full re-compression can further reduce file sizes by 1.2x to 1.6x. For digital television, the matting operations introduce considerable redundancy (black regions), thereby enabling the re-compression stage to shrink the file by 1.8x to 5.4x over the compressed processing technique.



anim-scene1 + anim-character2 = video composite

(a) Computer animation composite (alpha-under)



digvid-background1 + digvid-matte-frame = video composite

(b) Digital television composite (multiply)

Figure 14: Examples of video compositing operations.

6. RELATED WORK

Several other researchers have pursued the idea of operating directly on compressed data formats. The novelty of our work is two-fold: first, in its focus on lossless compression formats, and second, in its ability to map a flexible stream program, rather than a single predefined operation, into the compressed domain.

Most of the previous work on mapping algorithms into the compressed domain has focused on formats such as JPEG that utilize a Discrete Cosine Transform (DCT) to achieve spatial compression [1, 5, 6, 8, 14, 15, 19, 20, 23, 27]. This task requires a different analysis, with particular attention given to details such as the blocked decomposition of the image, quantization of DCT coefficients, zig-zag ordering, and so-on. Because there is also a run-length encoding stage in JPEG, our current technique might find some application there; however, it appears that techniques designed for JPEG have limited application to formats such as LZ77.

There has been some interest in performing compressed processing on lossless encodings of black-and-white images. Shoji presents the pxy format for performing transpose and other affine operations [21]; the memory behavior of the technique was later improved by Misra et al. [13]. The pxy format lists the (x, y) coordinate pairs at which a black-and-white image changes color during a horizontal scan. As illustrated in Figure 6, our technique can also preserve a certain amount of compression during a transpose, though we may achieve less compression than the pxy format due to our one-dimensional view of the data.

Researchers have also considered the problem of pattern matching on compressed text. A randomized algorithm has been developed for LZ77 [7] while deterministic strategies exist for LZ78 and LZW [16, 17]. These solutions are specialized to searching text; they do not apply to our transformations, and our technique does not apply to theirs.

7. CONCLUSIONS AND FUTURE WORK

In order to accelerate operations on compressible data, this paper presents a general technique for translating stream programs into the compressed domain. Given a natural program that operates on uncompressed data, our transformation outputs a program that directly operates on the compressed data format. We support lossless compression formats based on LZ77. In the general case, the transformed program may need to partially decompress the data to perform the computation, though this decompression is minimized throughout the process and significant compression ratios are preserved without resorting to an explicit re-compression step.

We implemented some of our transformations in the StreamIt compiler and demonstrated excellent speedups. Across a suite of 12 videos in Apple Animation format, computing directly on compressed data offers a speedup roughly proportional to the compression ratio. For pixel transformations (brightness, contrast, inverse) speedups range from 2.5x to 471x, with a median of 17x; for video compositing operations (overlays and mattes) speedups range from 1.1x to 32x, with a median of 6.6x. While previous researchers have used special-purpose compressed processing techniques to obtain speedups on lossy, DCT-based codecs, we are unaware of a comparable demonstration for lossless video compression. As digital films and animated features have embraced lossless formats for the editing process, the speedups obtained may have practical value.

In the future, it would be desirable to extend our technique to support codecs other than Apple Animation. We would expect to achieve comparable performance on Flic Video and Microsoft RLE, which are very similar to Apple Animation. Targa images (sometimes used to store each video frame) utilize run-length encoding, which can be exploited by our technique. Our technique could also offer gains for the Planar RGB and OpenEXR formats; however, they re-arrange data (by color for Planar RGB, by high/low bytes in OpenEXR) that would prevent general transformations from achieving comparable speedups. General-purpose formats such as ZIP and GZIP are based on LZ77, but also perform Huffman coding which would have to be undone prior to the application of our technique. Unfortunately our technique would not apply directly to most PNG images, which include a delta encoding prior to LZ77 compression. Further details on extending our technique to other formats are available in our extended report [24].

8. ACKNOWLEDGMENTS

We thank Viktor Kuncak for helpful feedback. This work was funded in part by the National Science Foundation (CNS-0305453, ACI-0325297) and the Gigascale Systems Research Center.

9. REFERENCES

[1] S. Acharya and B. Smith. Compressed domain transcoding of MPEG. *Int. Conf. on Multimedia Computing and Systems*, 1998.

[2] About digital video editing. Adobe online education materials, 2006. http://www.adobe.com/education/pdf/cib/pre65_cib/pre65_cib02.pdf.

[3] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static data flow. In *ICASSP*, 1995.

[4] S. Chang. Compressed-domain techniques for image/video indexing and manipulation. *Conference on Image Processing*, 1995.

[5] C. Dorai, N. Ratha, and R. Bolle. Detecting dynamic behavior in compressed fingerprint videos: distortion. *CVPR*, 2, 2000.

[6] R. Dugad and N. Ahuja. A fast scheme for image size change in the compressed domain. *IEEE Trans. on Circuits and Systems for Video Technology*, 11(4), 2001.

[7] M. Farach and M. Thorup. String matching in lempel-ziv compressed strings. *Algorithmica*, 20, 1998.

[8] G. Feng and J. Jiang. Image segmentation in compressed domain. *Journal of Electronic Imaging*, 12(3), 2003.

[9] R. Harrington, R. Max, and M. Geduld. *After Effects on the Spot: Time-Saving Tips and Shortcuts from the Pros*. Focal Press, 2004.

[10] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, 1987.

[11] B. Long and S. Schenk. *Digital Filmmaking Handbook*. Charles River Media, 2002.

[12] M. Mandal, F. Idris, and S. Panchanathan. A critical evaluation of image and video indexing techniques in the compressed domain. *Image and Vision Computing*, 17(7), 1999.

[13] V. Misra, J. Arias, and A. Chhabra. A memory efficient method for fast transposing run-length encoded images. *Int. Conf. on Document Analysis and Recognition*, 1999.

[14] J. Mukherjee and S. Mitra. Image resizing in the compressed domain using subband DCT. *IEEE Trans. on Circuits and Systems for Video Technology*, 12(7), 2002.

[15] J. Nang, O. Kwon, and S. Hong. Caption processing for MPEG video in MC-DCT compressed domain. *ACM Multimedia*, 2000.

[16] G. Navarro. Regular expression searching on compressed text. *Journal of Discrete Algorithms*, 1, 2003.

[17] G. Navarro and J. Tarhio. LZgrep: a Boyer-Moore string matching tool for Ziv-Lempel compressed text. *Soft. Pract. Exper.*, 35, 2005.

[18] D. Pogue. *iMovie 3 & IDVD: The Missing Manual*. O'Reilly, 2003.

[19] B. Shen and I. Sethi. Convolution-based edge detection for image/video in block DCT domain. *Journal of Visual Communication and Image Representation*, 7(4), 1996.

[20] B. Shen and I. Sethi. Block-based manipulations on transform-compressed images and videos. *Multim. Sys.*, 6(2), 1998.

[21] K. Shoji. An algorithm for affine transformation of binary images stored in pxy tables by run format. *Systems and computers in Japan*, 26(7), 1995.

[22] B. Smith. A survey of compressed domain processing techniques. Cornell University, 1995.

[23] B. Smith and L. Rowe. Compressed domain processing of JPEG-encoded images. *Real-Time Imaging*, 2(2), 1996.

[24] W. Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, 2009.

[25] W. Thies, S. Hall, and S. Amarasinghe. Mapping stream programs into the compressed domain. Technical Report MIT-CSAIL-TR-2007-055, MIT, 2007. http://hdl.handle.net/1721.1/39651.

[26] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Int. Conf. on Compiler Construction*, 2002.

[27] B. Vasudev. Compressed-domain reverse play of MPEG video streams. *SPIE Conf. on Multimedia Systems and Applications*, 1998.

[28] S. Wee, B. Shen, and J. Apostolopoulos. Compressed-domain video processing. *HP Labs Technical Report, HPL-2002*, 282, 2002.

[29] A. Wyner and J. Ziv. The sliding-window Lempel-Ziv algorithm is asymptotically optimal. *Proceedings of the IEEE*, 82(6), 1994.

[30] N. Ziviani, E. Silva de Moura, G. Navarro, and R. Baeza-Yates. Compression: a key for next-generation text retrieval systems. *Computer*, 33(11), 2000.