MIT-CSAIL-TR-2007-055 November 30, 2007

Mapping Stream Programs into the Compressed Domain

William Thies, Steven Hall, and Saman Amarasinghe

Mapping Stream Programs into the Compressed Domain

William Thies Steven Hall Saman Amarasinghe

MIT Computer Science and Artificial Intelligence Laboratory

November 30, 2007

Abstract

Due to the high data rates involved in audio, video, and signal processing applications, it is imperative to compress the data to decrease the amount of storage used. Unfortunately, this implies that any program operating on the data needs to be wrapped by a decompression and re-compression stage. Re-compression can incur significant computational overhead, while decompression swamps the application with the original volume of data.

In this paper, we present a program transformation that greatly accelerates the processing of compressible data. Given a program that operates on uncompressed data, we output an equivalent program that operates directly on the compressed format. Our transformation applies to stream programs, a restricted but useful class of applications with regular communication and computation patterns. Our formulation is based on LZ77, a lossless compression algorithm that is utilized by ZIP and fully encapsulates common formats such as Apple Animation, Microsoft RLE, and Targa.

We implemented a simple subset of our techniques in the StreamIt compiler, which emits executable plugins for two popular video editing tools: MEncoder and Blender. For common operations such as color adjustment and video compositing, mapping into the compressed domain offers a speedup roughly proportional to the overall compression ratio. For our benchmark suite of 12 videos in Apple Animation format, speedups range from 1.1x to 471x, with a median of 15x.

1. Introduction

With the emergence of data-intensive applications such as digital film, medical imaging and geographic information systems, the performance of next-generation systems will often depend on their ability to process huge volumes of data. For example, each frame of a digital film requires approximately 2 megabytes, implying that a fully-edited 90-minute video demands about 300 gigabytes of data for the imagery alone [10]. Industrial Light and Magic reports that, in 2003, their processing pipeline output 13.7 million frames and their internal network processed 9 petabytes of data [4]. The U.S. Geological Survey had archived over 13 million frames of photographic data by the end of 2004, and estimates that 5 years is needed to digitize 8.6 million additional images [36]. In all of these situations, the data is highly compressed to reduce storage costs. At the same time, extensive post-processing is often required for adding captions, watermarking, resizing, compositing, adjusting colors, converting formats, and so on. As such processing logically operates on the uncompressed format, the usual practice is to decompress and re-compress the data whenever it needs to be modified.

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 [8, 21, 32, 38]. 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 have two limitations. First, they focus on lossy compression formats (e.g., JPEG, MPEG) rather than lossless compression formats, which are important for the applications cited previously. Second, they rely on specialized and ad-hoc techniques for translating individual operations into the compressed domain. For example, for DCT-based spatial compression formats (JPEG, Motion-JPEG), researchers have developed separate algorithms for resizing [12, 23], edge detection [28, 29], image segmentation [15], shearing and rotating inner blocks [30], and arbitrary linear combinations of pixels [33]. Techniques extending to DCT-based temporal compression (MPEG) include captioning [24], reversal [37], distortion detection [11], transcoding [1], and others [38]. For run-length encoded images, algorithms have been devised for efficient transpose and rotation [22, 31]. A compressed audio format has been invented that allows direct modification of pitch and playback speed [19]. While these techniques are powerful, they remain inaccessible to most application programmers because they demand intricate manipulation of the underlying compression format. It is difficult for non-experts to compose existing compressed-domain operations into a complete program, let alone translate a new and unique operation into the compressed domain.

This paper presents a technique for automatically mapping complete user-defined programs into the compressed domain. The technique applies to stream programs: a restricted but practical class of applications that perform regular processing over long data sequences. Stream programming captures the essential functionality needed by image, video, and signal processing applications while exposing the flow of data to the compiler. Our formulation is based on LZ77, a lossless compression algorithm utilized by ZIP, and naturally applies to formats such as Apple Animation, Microsoft RLE, and Targa (which are special cases of LZ77). Lossless compression is widely used in computer animation and digital video editing in order to avoid accumulating compression artifacts. By providing an automatic mapping into the compressed domain, our technique enables a large class of transformations to be customized by the user and directly applied to the compressed data.

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 a broad class of programs: those which input and output multiple data items at a time, and those which split, combine, and reorder the data in the stream.

We implemented a subset of our general technique in the StreamIt compiler, which generates plugins for two popular video editing tools: MEncoder and Blender. While our current implementation handles only certain streaming patterns, it is sufficient to express many practical transformations such as pixel adjustment (brightness, contrast, color inversion) and video compositing (overlays and mattes). Using a suite of 12 videos (screencasts, animations, stock footage) in Apple Animation format, computing directly on compressed data offers a speedup roughly proportional to the compression factor. For pixel transformations, speedups range from 2.5x to 471x, with a median of 17x; for video compositing, speedups range from 1.1x to 32x, with a median of 6.6x.

In the general case, compressed processing techniques may need to partially decompress the input data to support the behavior of certain programs. Even if no decompression is performed, the output may benefit from an additional re-compression step if new redundancy is introduced during the processing (for example, increasing image brightness can whiteout parts of the image). This effect turns out to be minor in the case of our experiments. For pixel transformations, output sizes are within 0.1% of input sizes and often (more than half the time) are within 5% of a full re-compression. For video compositing, output files maintain a sizable compression ratio of 8.8x (median) while full re-compression results in a ratio of 13x (median).

To summarize, this paper makes the following contributions:

- An algorithm for mapping an arbitrary stream program, written in the cyclo-static dataflow model, 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 analysis of popular lossless compression formats and the opportunities for direct processing on each (Section 4).
- An experimental evaluation in the StreamIt compiler, 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 relies on the cyclo-static dataflow representation for input programs and the LZ77 representation of compressed data. These are described in the next two sections.

2.1 Cyclo-Static Dataflow

In the cyclo-static dataflow model, a program is represented by a set of independent *actors* that communicate using FIFO data channels [5, 18]. Each actor has an independent program counter and address space; all communication is done using the data channels. Actors have one or more atomic execution steps that execute in a fixed pattern throughout the lifetime of the program. A key restriction of the cyclo-static dataflow model is that, for each execution of a given actor, the number of items produced and consumed on the data channels is known at compile time. This enables the compiler to perform static scheduling of the actors and to guarantee deadlock freedom [5, 18].

Cyclo-static dataflow is a natural fit for many multimedia and signal processing kernels, as such programs often have a regular structure with known communication patterns. As a programmer, one can use a high-level language such as StreamIt [35] to express a cyclo-static dataflow program. An example StreamIt program appears in Figure 1. 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 programs, and our analysis handles each one separately:

- **Filters** have a single input stream and a single output stream, and perform general-purpose computation. Filters perform the same function on every execution (there is no cyclic pattern of computations). 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[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();
}
```

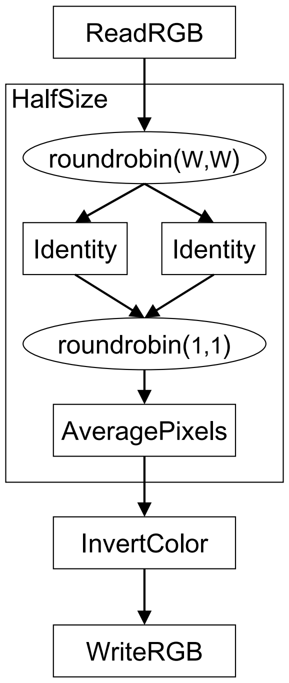


Figure 1. Example StreamIt program.

- **Splitters** have a single input stream and multiple output streams, and perform pre-defined computations. There are two types of splitters: *duplicate* splitters, which replicate their inputs to all of the target streams, and *roundrobin* splitters, which distribute the input items across the streams. Roundrobin splitters are parameterized: roundrobin(n_1, n_2) indicates that the first n_1 items are sent to the first output, and the next n_2 items are sent to the second output. Splitters execute in a fine-grained, cyclo-static fashion: regardless of the parameters, each execution step passes only a single item from the input stream to an output stream. In Figure 1, the splitter sends `WIDTH` pixels in each direction, distributing the lines of the image across alternate streams.
- **Joiners** have multiple input streams and a single output stream, and perform pre-defined computations. The only type of joiner is roundrobin, which acts analogously to a roundrobin splitter. In Figure 1, 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.

[1] Though StreamIt also allows filters to peek at the input stream (i.e., to perform a sliding window computation) and to maintain mutable state across executions, these features are uncommon in other stream languages and we do not support them in the current work.

(a) Uncompressed domain	(b) Compressed domain (LZ77)

stream := value*	stream := (value | repeat)*
	 repeat := $\langle$distance, count$\rangle$

Figure 2. Representation of data in the uncompressed and compressed domains.

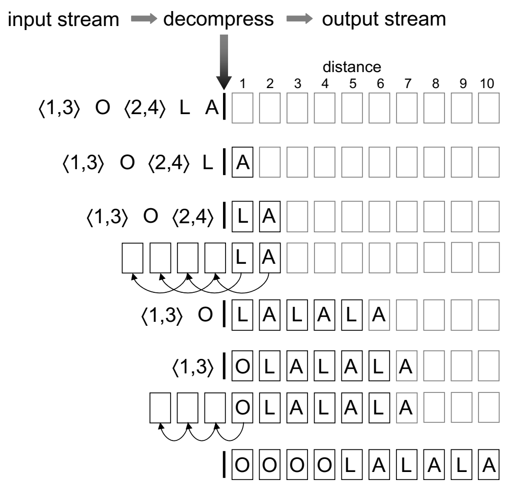


Figure 3. Example of LZ77 decompression.

2.2 LZ77 Compression

LZ77 is a lossless, dictionary-based compression algorithm. Named after its creators Lempel and Ziv, who published the algorithm in 1977 [40], LZ77 is asymptotically optimal [39] and forms the basis for many popular compression formats, including ZIP, GZIP and PNG. As described in Section 4, LZ77 also serves as a generalization of simpler encodings such as Apple Animation, Microsoft RLE, and Targa, allowing our transformations to naturally extend to these formats.

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* (see Figure 2). A value indicates a token that should be copied directly to output of the decoded stream. A repeat 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. The distances are bounded, which enables the decoder to operate with a fixed buffer size. 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 of LZ77 decoding is given in Figure 3.

3. Program Transformation

Our program transformation inputs a cyclo-static dataflow program and outputs an equivalent program in which all of the data channels use a compressed representation. One can think of this process as mapping from the uncompressed domain to the compressed domain (see Figure 2). 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).

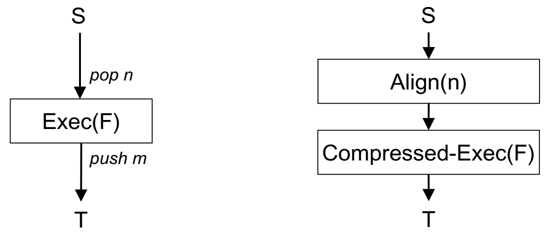
Variables		Constants	
S, T	Input, output streams	n_1, n_2	Actor’s pop rates
V	List of values (may be empty)	m	Actor’s push rate
$\langle d, c\rangle$	Repeat distance & count		

Functions (list $\times$ list $\rightarrow$ list)	
$\bullet$	List concatenation
F	Filter work function, outputs m-element list

Figure 4. Notations used in the semantic rules.

$$\frac{S \bullet V \rightarrow T \quad |V| = n}{S \rightarrow F(V) \bullet T} \qquad \boxed{\text{exec-uncompressed}}$$

Figure 5. Semantics of EXEC(F): execution of filter F in the uncompressed domain. Notations are defined in Figure 4.



(a) Uncompressed domain (b) Compressed domain

Figure 6. Overall execution of a filter F in the uncompressed and compressed domains.

For the sake of presentation, we use an operational semantics to express the execution under the uncompressed and compressed domains. Though some of the transition rules require dynamic data rates and thus fall outside the cyclo-static dataflow model, all of them have an efficient implementation in StreamIt. The rules use the notations given in Figure 4, and have the following form:

$$\frac{S \rightarrow T}{S' \rightarrow T'}$$

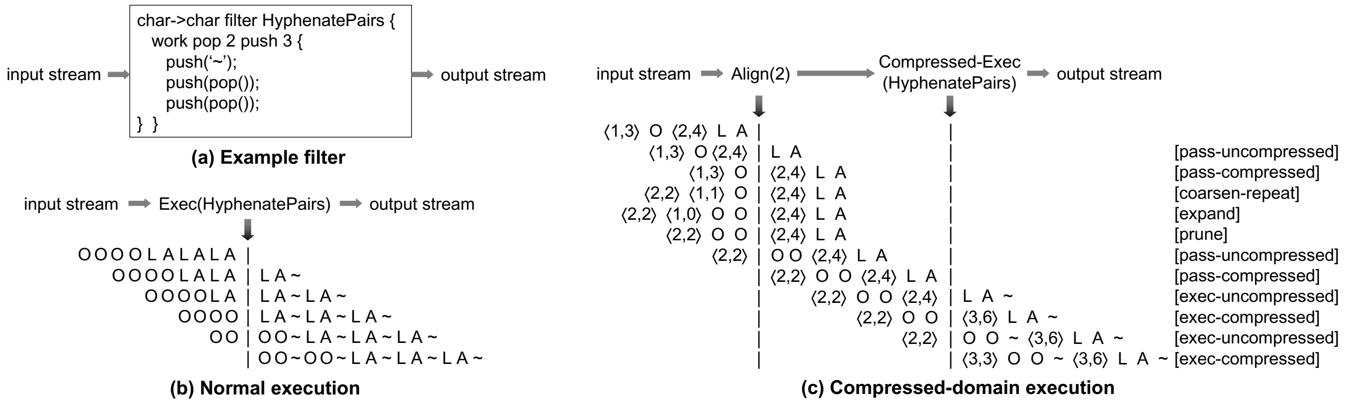
This rule reads: if the incoming stream has value S and the outgoing stream has value T, then after an execution step, the streams have values S' and T', respectively. As detailed in Figure 2, we represent streams as lists of tokens; inputs to the stream are added to the front of the list, while outputs from the stream are removed from the end of the list.

To describe the mapping into the compressed domain, we consider each StreamIt construct in turn: filters, splitters, and joiners.

3.1 Filters

Filter execution in the uncompressed domain is described by the rule in Figure 5. The rule expresses the simple fact that a filter inputs a list of n values from the end of its input stream; this list is denoted by V. The filter pushes its results, denoted by $F(V)$, onto the front of the output stream.

Filter execution in the compressed domain requires a two-stage transformation (see Figure 6 for an overview, and Figure 7 for a detailed example). First, the input stream is aligned to a granularity n that matches the input rate of the filter. This alignment guarantees that every token in the stream is either a sequence of n values, or a repeat in which both the distance and the count are multiples of n. The alignment stage (described in detail later) is a no-op for filters that pop only one item ($n = 1$).



```
char->char filter HyphenatePairs {
  work pop 2 push 3 {
    push('~');
    push(pop());
    push(pop());
} }
```

```
O O O O L A L A L A |
    O O O O L A L A | L A ~
        O O O O L A | L A ~ L A ~
            O O O O | L A ~ L A ~ L A ~
                O O | O O ~ L A ~ L A ~ L A ~
                    | O O ~ O O ~ L A ~ L A ~ L A ~
```

```
⟨1,3⟩ O ⟨2,4⟩ L A |                      |
    ⟨1,3⟩ O ⟨2,4⟩ | L A                  |                                [pass-uncompressed]
          ⟨1,3⟩ O | ⟨2,4⟩ L A            |                                [pass-compressed]
    ⟨2,2⟩ ⟨1,1⟩ O | ⟨2,4⟩ L A            |                                [coarsen-repeat]
  ⟨2,2⟩ ⟨1,0⟩ O O | ⟨2,4⟩ L A            |                                [expand]
        ⟨2,2⟩ O O | ⟨2,4⟩ L A            |                                [prune]
            ⟨2,2⟩ | O O ⟨2,4⟩ L A        |                                [pass-uncompressed]
                  | ⟨2,2⟩ O O ⟨2,4⟩ L A  |                                [pass-compressed]
                  |    ⟨2,2⟩ O O ⟨2,4⟩   | L A ~                          [exec-uncompressed]
                  |          ⟨2,2⟩ O O   | ⟨3,6⟩ L A ~                    [exec-compressed]
                  |              ⟨2,2⟩   | O O ~ ⟨3,6⟩ L A ~              [exec-uncompressed]
                  |                      | ⟨3,3⟩ O O ~ ⟨3,6⟩ L A ~        [exec-compressed]
```

Figure 7. Example execution of a filter in the uncompressed and compressed domains.

$$\frac{S \bullet V \to T \quad |V| = n}{S \to F(V) \bullet T} \qquad \boxed{\text{exec-uncompressed}}$$

$$\frac{S \bullet \langle d, c\rangle \to T \quad d\%n = 0 \quad c\%n = 0}{S \to \langle m\, d/n, m\, c/n\rangle \bullet T} \qquad \boxed{\text{exec-compressed}}$$

Figure 8. Semantics of COMPRESSED-EXEC(F): execution of filter F in the compressed domain. Notations are defined in Figure 4.

After the alignment stage comes the execution of the compressed filter, which appears in Figure 8. The `exec-uncompressed` rule deals with values on the input stream, and is identical to that in the uncompressed execution. The `exec-compressed` rule deals with repeats on the input stream and encapsulates the key idea of the paper. Because the inputs are repeating at the correct granularity, the repeat can be copied directly to the output of the filter without performing any new computation. The only change needed is to adjust the repeat distance and count to match the filter's output rate.

3.1.1 Stream Alignment

The alignment phase is needed for filters that pop more than one item from the input stream. Its goal is to align the execution boundaries of the filter with the repeat boundaries of the compressed data; this alignment is required for the compressed execution. Following alignment, each execution of a filter will input either n consecutive values, or a repeat token with a distance and count that are evenly divisible by n (where n represents the pop rate of the filter).

The alignment stage sometimes needs to partially decompress the data in the stream. Due to the sliding-window dictionary in LZ77, in general it is difficult to decode only a few items without decompressing others. Thus, our formulation assumes that a fully decompressed version of the stream is available; the transition rules access the decompressed data using the *decode* function, which returns the sequence of values represented by a repeat token at its current position in the stream. However, in practice, this decompression can be avoided whenever the repeat distance is the same as the window size, as this simply causes a value in the window to be overwritten by itself. This case is very common in several practical compression formats; for example, in Apple Animation, the vast majority of repeats reference the same pixel in the previous frame (which is also the window size), and thus most decompression is avoided. In run-length encoding, the repeat distance and the window size are always equal to one, so no decompression is needed. While general LZ77 does require a decompressed window to be maintained, our technique still offers significant benefits by

$$\frac{S \bullet V \to T \quad |V| = n}{S \to V \bullet T} \qquad \boxed{\text{pass-uncompressed}}$$

$$\frac{S \bullet \langle d, c\rangle \to T \quad d\%n = 0 \quad c \geq n}{S \bullet \langle d, c\%n\rangle \to \langle d, c - c\%n\rangle \bullet T} \qquad \boxed{\text{pass-compressed}}$$

$$S \bullet \langle d, c\rangle \bullet V \to T \qquad \boxed{\text{expand}}$$
$$\frac{c < n \;\vee\; 1 \leq |V| < n \;\vee\; (d\%n > 0 \;\wedge\; \neg(d < \mathrm{LCM}(d, n) < c))}{S \bullet \langle d, c-1\rangle \bullet \mathit{decode}(\langle d, 1\rangle) \bullet V \to T}$$

$$\frac{S \bullet \langle d, 0\rangle \bullet V \to T}{S \bullet V \to T} \qquad \boxed{\text{prune}}$$

$$\text{let } L = \mathrm{LCM}(d, n) \qquad \boxed{\text{coarsen-repeat}}$$
$$\frac{S \bullet \langle d, c\rangle \bullet V \to T \quad d\%n > 0 \quad d < L < c}{S \bullet \langle L, c - (L - d)\rangle \bullet \langle d, L - d\rangle \bullet V \to T}$$

Figure 9. Semantics of ALIGN(n): aligning data to a granularity of n. The *decode* function uncompresses a repeat token into a list of values; other notations are given in Figure 4.

computing on a smaller volume of data and avoiding the cost of re-compression. For general algorithms such as gzip, compression can be up to 10x slower than decompression [41].

The semantics of stream alignment are given in Figure 9. If the end of the input stream contains n values, then alignment is satisfied and the values are moved to the output stream (rule `pass-uncompressed`). Likewise, if the input contains a repeat in which the distance is a multiple of n and the count is at least n, then a number of aligned repeats are peeled from the input and moved to the output (rule `pass-compressed`). If the count is not a multiple of n, then part of the repeat is leftover and remains on the input stream.

There are some cases in which a repeat cannot be moved to the output stream, in which case the data needs to be partially decompressed (rule `expand`). This occurs if the repeat has a count less than n, if it occurs in the middle of an aligned stretch of n values, or if its distance is not a multiple of n (this last condition can sometimes be remedied by another rule, see below). The `expand` rule decodes only one value from an unaligned 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 by the `prune` rule.

The final rule, `coarsen-repeat`, preserves a specific kind of compression in the input stream. Consider that a filter pops two

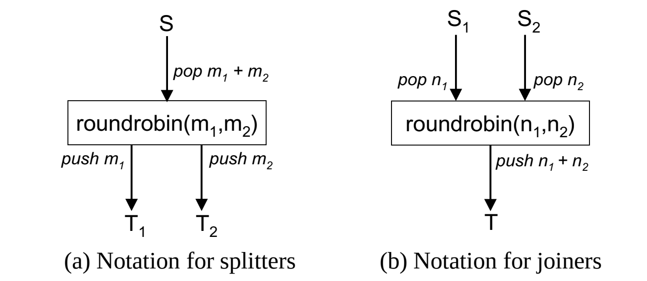


Figure 10. Notations used in the semantics for splitters and joiners. In addition, the variable *pos* indicates how many items have been written to (in the case of splitters) or read from (in the case of joiners) the active tape during the current execution cycle.

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 pop 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.

3.2 Splitters

It is necessary to consider splitters and joiners separately from general-purpose actors because of their pass-through semantics: the inputs are distributed to the outputs without performing any computation. Our translation to the compressed domain leverages this fact to preserve considerably more compression than would be possible if splitters and joiners were viewed as opaque computational nodes with multiple inputs and multiple outputs. Consequently, splitters and joiners should be employed by the programmer not only as a natural expression of parallelism, but as a powerful way of exposing the data reordering to the compiler.

As mentioned previously, splitters and joiners adopt a fine-grained cyclo-static execution model, in which each execution step transfers only one item from an input tape to an output tape. That is, a roundrobin(k_1, k_2) splitter or joiner has $k_1 + k_2$ distinct execution steps. We refer to every group of $k_1 + k_2$ steps as an *execution cycle*.

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. This captures all of the fundamental ideas; extension to additional streams is straightforward. Rewrite rules now take the following form:

$$\frac{S \rightarrow T_1; T_2}{S' \rightarrow T_1'; T_2'}$$

where T_1 and T_2 represent the output streams of the splitter (see Figure 10). In addition, we make two further simplifications:

- The rules assume that the next execution step of the splitter will write to T_1. The subscripts should be interpreted without loss of generality.

$$\frac{S \bullet V \rightarrow T_1; T_2 \quad |V| = 1}{S \rightarrow V \bullet T_1; T_2} \qquad \boxed{\text{pass-uncompressed}}$$

Figure 11. Semantics of SPLITTER: execution of a roundrobin splitter in the uncompressed domain.

$$\frac{S \bullet V \rightarrow T_1; T_2 \quad |V| = 1}{S \rightarrow V \bullet T_1; T_2} \qquad \boxed{\text{pass-uncompressed}}$$

let offset $= d\%(m_1 + m_2)$ $\boxed{\text{pass-compressed-long}}$
let $(L_1, L_2) =$ run_splitter(c)

$$\frac{S \bullet \langle d, c\rangle \rightarrow T_1; T_2 \quad \text{offset} = 0}{\begin{array}{l} S \rightarrow \langle d\, m_1/(m_1+m_2), L_1\rangle \bullet T_1; \\ \quad \langle d\, m_2/(m_1+m_2), L_2\rangle \bullet T_2 \end{array}}$$

let offset $= d\%(m_1 + m_2)$ $\boxed{\text{pass-compressed-short}}$
let offset' $=$ **if** offset $\leq$ *pos* **then** offset
else offset $- m_2$
let actual_repeat $= \min(c, \text{split\_potential}(d))$

$$\frac{S \bullet \langle d, c\rangle \rightarrow T_1; T_2 \quad \text{offset} > 0 \quad \text{split\_potential}(d) > 0}{\begin{array}{l} S \bullet \langle d, c - \text{actual\_repeat}\rangle \rightarrow \\ \langle m_1 * \text{floor}(d/(m_1+m_2)) + \text{offset'}, \text{actual\_repeat}\rangle \bullet T_1; T_2 \end{array}}$$

let offset $= d\%(m_1 + m_2)$ $\boxed{\text{expand}}$

$$\frac{S \bullet \langle d, c\rangle \rightarrow T_1; T_2 \quad \text{offset} > 0 \quad \text{split\_potential}(d) = 0}{S \bullet \langle d, c-1\rangle \bullet \textit{decode}(\langle d, 1\rangle) \rightarrow T_1; T_2}$$

$$\frac{S \bullet \langle d, 0\rangle \rightarrow T_1; T_2}{S \rightarrow T_1; T_2} \qquad \boxed{\text{prune}}$$

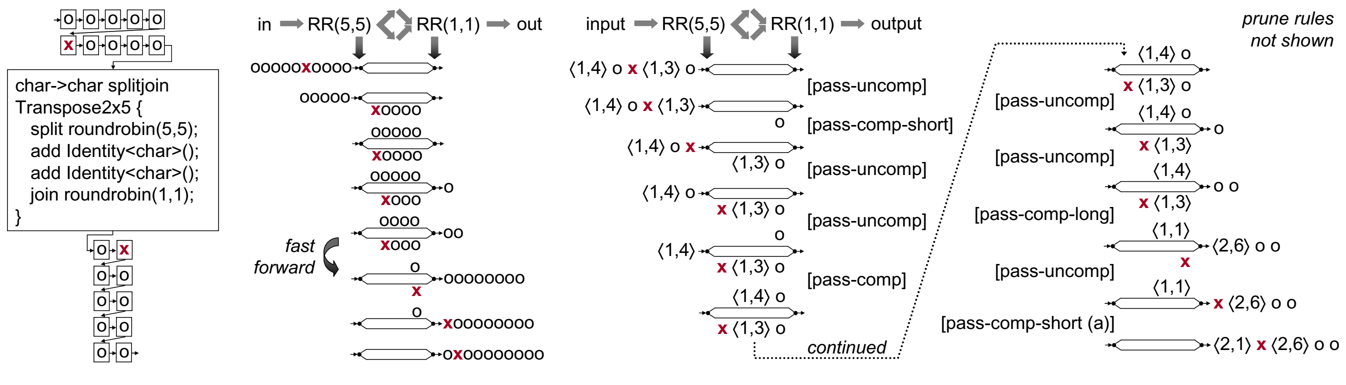
Figure 12. Semantics of COMPRESSED-SPLITTER: execution of a roundrobin splitter in the compressed domain.

- 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 (T_1) in the current execution cycle. For brevity, the rules do not maintain the value of *pos*, though it is straightforward to do so.

The semantics for splitter execution in the uncompressed domain appear in Figure 11. The `pass-uncompressed` rule simply passes a single value from the input tape to the current output tape, T_1. Note that the current position *pos* is implicitly incremented; once *pos* reaches m_1, it is reset to zero and the output tapes are switched (tape T_2 will be named T_1 on the next execution).

The compressed-domain semantics for splitters are given in Figure 12, and a detailed example appears in Figure 13. 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, expressed by the `pass-compressed-long` rule in Figure 12, 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 `run_splitter` in Figure 14).



(a) Example splitjoin **(b) Normal execution** **(c) Compressed-domain execution of splitter (left) and joiner (right)**

Figure 13. Example execution of splitters and joiners in the compressed domain. As illustrated by the input/output pairs in (a), the example performs a transpose of a 2x5 matrix. When the matrix is linearized (as in (b) and (c)), 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.

The second case, handled by the `pass-compressed-short` rule, 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_potential` (Figure 14) by comparing the repeat 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` rule) because the current repeat token references items that will be sent to the wrong output tape. The `prune` rule is also needed to clear empty repeats generated by `expand`.

Though we omit the details, it is also desirable to employ an analog of the `coarsen-repeat` rule (Figure 9) to preserve even more compression across a splitter. The intuition is that, by increasing certain repeat distances, the splitter's output tapes can become more independent (referencing themselves rather than each other). This enables a compressed rule to fire in place of an expansion step.

3.3 Joiners

Analogously to splitters, 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. Both of these cases are illustrated by example in Figure 13. Unlike splitters, there is never a need to decompress repeat tokens into values before passing through a joiner. Though the repeat length may shrink to one, it will remain a reference to a previous item rather than becoming a value itself.

In the uncompressed domain, joiners have the semantics given in Figure 15. The `pass-uncompressed` rule passes a single value from the current input tape (S_1) to the output tape. Analogously to splitters, the variable *pos* represents the number of items that have been read from the current input tape and is implicitly updated. Once *pos* reaches n_1, it is reset to zero and the input tapes are switched (tape S_2 will be named S_1 on the next execution).

The first and most powerful way to execute joiners in the compressed domain is to combine repeat tokens from both input streams

```
// 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.
run_splitter($c, pos$) returns (int, int) {
  // the number of complete splitter cycles, and the leftover
  total_cycles $= \text{floor}(c/(m_1 + m_2))$
  total_leftover $= c\%(m_1 + m_2)$

  // the last partial cycle may end in three regions:
  if total_leftover $\le m_1 - pos$ {
    // 1. in writing to the first output stream
    $L_1 =$ total_leftover
    $L_2 = 0$
  } else if total_leftover $\le m_1 - pos + m_2$ {
    // 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 $+ L_2)$ }

// Given a repeat token with distance $d$ that is input to a split-
// ter, returns the maximum count of a repeat token that could
// safely be emitted to the current output stream of the splitter. As-
// sumes that only a single repeat token can be emitted (i.e., the
// pass-compressed-long rule does not apply).
split_potential($d$) returns int {
  offset $= d\%(m_1 + m_2)$
  if offset $\le pos$ {
    // repeat for remainder of this execution cycle
    return $m_1 - pos$
  } else if offset $> m_2 + pos$ {
    // repeat until referenced data goes out of range
    return offset $- (m_2 + pos)$
  } else {
    // referenced data is on the other output stream
    return 0
  }
}
```

Figure 14. Helper functions for COMPRESSED-SPLITTER.

$$\frac{S_1 \bullet V; S_2 \rightarrow T \quad |V| = 1}{S_1; S_2 \rightarrow V \bullet T} \qquad \boxed{\text{pass-uncompressed}}$$

Figure 15. Semantics of JOINER: execution of a roundrobin joiner in the uncompressed domain.

$$\frac{S_1 \bullet V; S_2 \rightarrow T \quad |V| = 1}{S_1; S_2 \rightarrow V \bullet T} \qquad \boxed{\text{pass-uncompressed}}$$

$$\text{let } (L_1, L_2) = \text{repeat\_lengths}(c_1, c_2, pos) \qquad \boxed{\text{pass-compressed-long}}$$
$$\frac{\begin{array}{l} S_1 \bullet \langle d_1, c_1\rangle; \\ S_2 \bullet \langle d_2, c_2\rangle \rightarrow T \quad d_1\%n_1 = 0 \quad d_2\%n_2 = 0 \quad d_1/n_1 = d_2/n_2 \end{array}}{\begin{array}{l} S_1 \bullet \langle d_1, c_1 - L_1\rangle; \\ S_2 \bullet \langle d_2, c_2 - L_2\rangle \rightarrow \langle d_1(n_1+n_2)/n_1, L_1 + L_2\rangle \bullet T \end{array}}$$

$$\begin{array}{l} \text{let offset'} = \textbf{if } d\%n_1 \le pos \textbf{ then } pos \\ \qquad\qquad\quad \textbf{else } d\%n_1 + n_2 \\ \text{let } L = \min(c, \text{join\_potential}(d)) \end{array} \qquad \boxed{\text{pass-compressed-short (a)}}$$
$$\frac{S_1 \bullet \langle d, c\rangle; S_2 \bullet V \rightarrow T}{\begin{array}{l} S_1 \bullet \langle d, c - L\rangle; S_2 \bullet V \rightarrow \\ \langle (n_1 + n_2)\,\text{floor}(d/n_1) + \text{offset'}, L\rangle \bullet T \end{array}}$$

$$\begin{array}{l} \text{let offset'} = \textbf{if } d\%n_1 \le pos \textbf{ then } pos \\ \qquad\qquad\quad \textbf{else } d\%n_1 + n_2 \\ \text{let } L = \min(c, \text{join\_potential}(d)) \end{array} \qquad \boxed{\text{pass-compressed-short (b)}}$$
$$\frac{S_1 \bullet \langle d, c\rangle; S_2 \bullet \langle d_2, c_2\rangle \rightarrow T \quad d_2\%n_2 > 0}{\begin{array}{l} S_1 \bullet \langle d, c - L\rangle; S_2 \bullet \langle d_2, c_2\rangle \rightarrow \\ \langle (n_1 + n_2)\,\text{floor}(d/n_1) + \text{offset'}, L\rangle \bullet T \end{array}}$$

$$\frac{S_1 \bullet \langle d, 0\rangle; S_2 \rightarrow T}{S_1; S_2 \rightarrow T} \qquad \boxed{\text{prune}}$$

Figure 16. Semantics of COMPRESSED-JOINER.

(rule `pass-compressed-long` in Figure 16). 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 `repeat_lengths` routine (Figure 17) calculates the maximum repeat length depending on the current position of the joiner and the repeat lengths of the inputs.

The second mode of compressed joiner execution inputs only a single repeat token, extracting the maximum length that can safely move to the output. This rule is needed when the previous one does not apply: if the second stream ends in a value rather than a repeat (`pass-compressed-short (a)`) or the repeat distance has the wrong granularity (`pass-compressed-short (b)`). The `join_potential` routine (Figure 17) determines how much of the repeat can be moved to the output before the data referenced would have originated from a different input stream.

As in the case of splitters, further compression gains are possible by adding rules to coarsen the repeat distance or shift the distance to align with other streams. We omit the details here.

4. Supported File Formats

As LZ77 refers to a compression algorithm rather than a complete compression format, there are additional factors to consider in mapping computations to real-world image and video codecs. Some codecs are a subset of LZ77, utilizing only run-length encoding or a fixed window size; these are supported very efficiently by our technique. Others are a superset of LZ77, incorporating additional techniques such as delta coding or Huffman coding; these may incur additional processing overhead. In the following sections, we describe the practical considerations involved in targeting various compression formats with our technique. Formats are ordered by approximate goodness of the achievable mapping.

```
// Given that $c_1$ and $c_2$ 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.
repeat_lengths($c_1, c_2, pos$) returns (int, int) {
  // the number of complete joiner cycles, and the leftovers
  total_cycles = floor($c/(n_1 + n_2)$)
  leftover$_1$ = $c_1$ − total_cycles ∗ $n_1$
  leftover$_2$ = $c_2$ − total_cycles ∗ $n_2$

  // the last partial cycle may end in three regions:
  if leftover$_1 \le n_1$ − pos {
    // 1. in reading from the first input stream
    $L_1$ = leftover$_1$
    $L_2 = 0$
  } else if leftover$_2 \le n_2$ {
    // 2. in subsequent reading from the second input stream
    $L_1 = n_1$ − pos
    $L_2$ = leftover$_2$
  } else {
    // 3. in wrap-around reading from the first input stream
    $L_1$ = leftover$_1$
    $L_2 = n_2$
  }
  return ($n_1$ ∗ total_cycles + $L_1$, $n_2$ ∗ total_cycles + $L_2$) }

// Given a repeat token with distance $d$ on the current input
// stream of a joiner, returns the maximum count of a repeat
// token that could safely be emitted to the output stream. As-
// sumes that only a single repeat token is available (i.e., the
// pass-compressed-long rule does not apply).
join_potential($d$) returns int {
  offset = $d\%n_1$
  if offset ≤ pos {
    // repeat for remainder of this execution cycle
    return $n_1$ − pos
  } else {
    // repeat until referenced data goes out of range
    return offset − pos
  }
}
```

Figure 17. Helper functions for COMPRESSED-JOINER.

4.1 High-Efficiency Mappings

All of the formats in this category can be considered to be subsets of LZ77.

Apple Animation. The Apple Animation codec (which forms the basis for our experimental evaluation) is supported as part of the Quicktime MOV container format. It serves as an industry standard for exchanging computer animations and digital video content before they are rendered to lossy formats for final distribution [2, p. 106][16, p. 284] [20, p. 367][27, 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.

Flic Video. Flic Video files (FLI/FLC) were originally produced by Autodesk Animator and are still supported by many animation packages today. Their compression of frame data is almost identical to Apple Animation.

Microsoft RLE. Microsoft RLE compression can appear in both BMP images and AVI animations. Apart from bit-depth and formatting details, its capabilities are identical to Apple Animation; it can perform run-length encoding within a frame, and can skip over pixels to exploit inter-frame redundancy.

Targa. The Truevision Targa (TGA) format is a simple image format that is widely used to render frame sequences in the computer animation and video industries. The format includes an optional RLE compression stage, making it a good target for our technique.

PXY. The pxy format is a research-based image format designed to support efficient transpose and rotation of black-and-white images [31]. It consists of the series of (x, y) coordinates at which the image changes color during a horizontal scan. As this information can be converted to a run-length encoding, it can also be targetted by our technique.

4.2 Medium-Efficiency Mappings

While the formats in this category utilize an encoding that is compatible with LZ77, they incur extra overhead because the data is reorganized prior to the compression stage.

Planar RGB. The Planar RGB video format is supported by Apple Quicktime files. It utilizes run-length encoding for pixels within a frame, with partial support for expressing inter-frame repeats (only the end of lines can be skipped). The red, green, and blue planes are encoded separately in order to increase compression. For user transformations that need to process red, green, and blue values together, this introduces additional alignment overhead when applying our technique.

OpenEXR. OpenEXR is an emerging image format (backed by Industrial Light and Magic) for use in digital film. It offers several compression options, including run-length encoding, zip, and wavelet-based compression. However, in run-length encoding mode, the low and high bytes of the pixels are separated and encoded as separate run-length sequences; this enables pixels with variations in the low bytes to nonetheless benefit from compression of the high bytes. As most user transformations would utilize the entire bit-width of the pixel, our technique suffers additional alignment overhead in processing these files.

4.3 Low-Efficiency Mappings

The formats in this category are supersets of LZ77. While our technique could offer some gains in exploiting the LZ77 compression, it would have to undo any compression sitting on top of LZ77 and offers limited benefit for filters (as in PNG) applied underneath LZ77.

DEFLATE. DEFLATE is a general-purpose algorithm that provides all of the compression for popular formats such as ZIP and GZIP. The algorithm consists of a full LZ77 encoder followed by Huffman coding, which resizes the symbols in the stream to match their usage frequencies. In targeting ZIP or GZIP with our transformations, we would first have to undo the Huffman coding (unless the application simply reordered data, in which case the coding could remain intact). Though Huffman decoding is a lightweight lookup operation, it would also increase the memory footprint. In addition, as DEFLATE's LZ77 algorithm operates on individual bytes, there may be an exaggerated alignment cost if the application operates on a larger word size.

PNG. The PNG image format also relies on DEFLATE to compress the pixels in the image. However, before running DEFLATE, the pixels are usually filtered with a delta encoding; each pixel is replaced with the difference between its value and a predicted value, where the prediction is a linear combination of neighboring pixels. While program segments that compute a linear function [17] could perhaps be mapped to this compressed format, our current technique only applies if the delta encoding is turned off. Even in this scenario, there is a large amount of overhead due to the Huffman coding in DEFLATE.

5. Experimental Evaluation

To demonstrate the potential benefits of mapping into the compressed domain, we implemented a few 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 future work, these two building blocks are sufficient to express a number of useful programs and to characterize the performance of the technique.

Our evaluation focuses on applications in digital video editing. Given StreamIt source code that operates on pixels from each frame of a video, the StreamIt compiler maps the computation into the compressed domain and emits executable plugins for two popular video editing tools, MEncoder and Blender. The plugins are written for the Apple Animation format (see Section 4.1).

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.

The following sections provide more details on our video workloads, the evaluation of pixel transformations, and the evaluation of video compositing.

5.1 Video Workloads

Our evaluation utilizes a suite of 12 video workloads that are described in Table 1; some of the videos are also pictured in Figure 20. 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 4.1), which is widely used by video editing professionals [2, p. 106] [16, p. 284] [20, p. 367] [27, p. 280]. The Apple Animation format is also popular for capturing video from the screen or camera, as the encoder is relatively fast.

Our suite of videos is assembled from a variety of realistic and industry-standard sources. The first screencast is an online demo

	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.

of an authentication generator for rails [3]; the second is a PowerPoint presentation (including animations), captured using Camtasia Studio. As Internet content is often watermarked with a logo or advertisement, we include two animated logos in the "Internet video" category. These logos are taken from Digital Juice [9], a standard source for professional animations, and rendered to Apple Animation format using their software. The animated logos are rendered full-frame (with the logo in the corner) because compositing operations in our testbed (Blender) are done on equal-sized videos.

The computer animation clips are derived from Elephant's Dream, a short film with entirely open-source content [13]; our videos are rendered from source using Blender. Finally, the digital television content is also taken from a Digital Juice library [9]. The backgrounds represent high-resolution, rotating backdrops as might appear in the introduction to a program. The mattes are black-and-white animations that can be used to synthesize a smaller overlay (such as a frame or a "lower third", often used for text) from a full animated background (see Figure 20b for an example).

The videos exhibit a wide range of compression factors. The screencasts have very high compression ($\sim$400x-700x) because only a small part of the screen (e.g., a mouse, menu, or PowerPoint bullet) is changing on any given frame; the Apple Animation format compresses the inter-frame redundancy. The compression for `anim-scene1` is also in excess of 200x because motion is limited to a small animated character. The animated logos are the next most compressed ($\sim$50-70x), influenced largely by the constant blank region outside the logo. The computer animation content ($\sim$10-30x compression) has a high level of detail but benefits from both inter-frame and intra-frame redundancy, as some rendered regions have constant color. Next are the digital video mattes ($\sim$5-10x compression), which have fine-grained motion in some sections. Finally, the digital video backgrounds offer almost no compression gains (1.0-1.1x) under Apple Animation, as they have pervasive motion and detail across the entire frame.

The Apple Animation format supports various bit depths. All of our source videos use 32 bits per pixel, allocating a single byte for each of the red, green, blue, and alpha channels.

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 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. For the timing measurements, we execute each program five times and report the median user time.

5.2.2 Results

Detailed results for the pixel transformations appear in Table 2. Figure 18 illustrates the speedups, which range from 2.5x to 471x. As illustrated in Figure 19, the speedups are closely correlated with the compression factor in the original video. For the highly-compressed screencasts and `anim-scene1`, speedups range from 58x to 471x. For the medium-compression computer animations (including the animated logos), speedups range from 11x to 46x. And for the low-compression digital television content, speedups range from 2.5x to 8.9x.

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 19. 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 19.

		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.

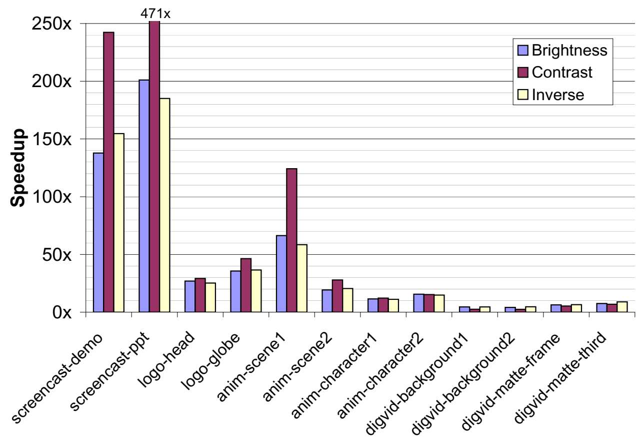


Figure 18. Speedup on pixel transformations.

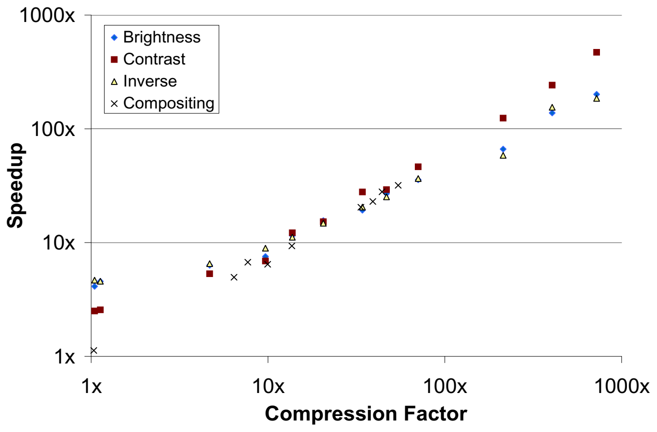


Figure 19. Speedup vs. compression factor for all transformations.

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[2], performing pixel transformations directly on compressed data will never increase the size of the file. This is illustrated in the middle columns of Table 18, 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).

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 will 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).

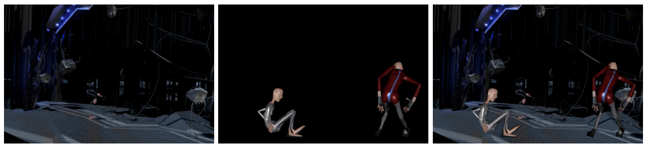
5.3 Video Compositing

In video compositing, two videos are combined using a specific function to derive each output pixel from a pair of input pixels (see Figure 20). 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.

[2] 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.

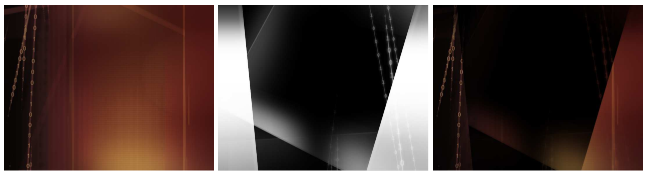
				COMPRESSION FACTOR		
	VIDEO COMPOSITE	EFFECT	SPEEDUP	Compute on Compressed Data	Uncompress, Compute, Re-Compress	Ratio
Internet Video	screencast-demo + logo-head	alpha-under	20.46x	34	52	1.55
	screencast-demo + logo-globe	alpha-under	27.96x	44	61	1.39
	screencast-ppt + logo-head	alpha-under	22.99x	39	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.



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 20. Examples of video compositing operations.

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. Blender has logged 1.8 million downloads in the last year [7] and was used in the production of Spiderman 2 [6]. Like MEncoder, Blender relies on the FFMPEG library for video coding, so we utilize the same Apple Animation decoder/encoder as in the pixel transformations.

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, unrolled loops, and the following comment: "this complex optimalisation is because the 'skybuf' can be crossed in". 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.

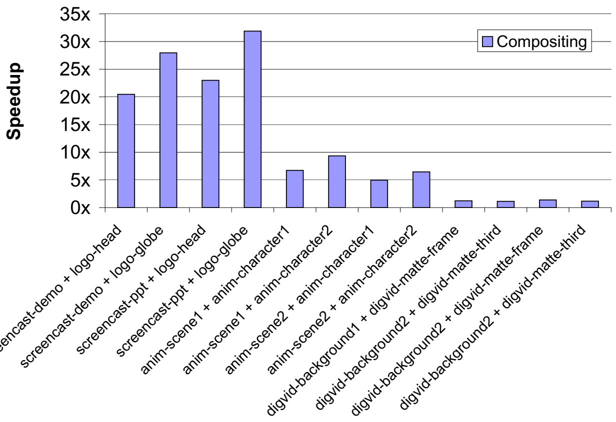


Figure 21. Speedup on composite transformations.

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 dimensions of the frame.

5.3.2 Results

Full results for the compositing operations appear in Table 3. Figure 21 illustrates the speedups, which range from 1.1x to 32x. As in the case of the pixel transformations, the speedups are closely correlated with the compression factor of the resulting videos, a relationship depicted in Figure 19. The highly-compressed screencasts enjoy the largest speedups (20x-32x), the computer animations have intermediate speedups (5x-9x), while the digital television content has negligible speedups (1.1x-1.4x). Overall, the speedups on video compositing (median = 6.6x) are lower than the pixel transformations (median = 17x); this is because the compression achieved on composite videos is roughly proportional to the minimum compression across the two input files.

As for 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 a large amount of redundancy (black regions), thereby enabling the re-compression stage to shrink the file by 1.8x to 5.4x over the compressed processing technique.

Even if the composite transformation does not introduce any new redundancy in the video, the compressed processing technique may increase file sizes by ignoring a specific kind of redundancy in the inputs. Suppose that in the first frame, both inputs are 100% black, while in the second frame, one input is 100% black and the other is 100% white. If the inputs are averaged, the second frame of output will be 100% gray and can be run-length encoded within the frame. However, because the inputs have different kinds of redundancy on the second frame (one is inter-frame, the other is intra-frame), the technique is unable to detect the intra-frame redundancy in the output and will instead produce N distinct pixels (all of them gray). We believe that this effect is small in practice, though we have yet to quantify its impact in relation to the new redundancy introduced by a transformation. Future work will explore alternate data structures for the compressed processing technique that may be able to preserve this redundancy with low overhead.

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 ability to map an arbitrary stream program, rather than a single predefined operation, into the compressed domain; and second, in its focus on lossless compression formats.

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, 11, 12, 15, 23, 24, 29, 28, 30, 33, 37]. 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. Also, we are unaware of any previous methodology for translating a generic program to operate on compressed data; previous efforts have mapped each algorithm in a manual and ad-hoc way.

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 [31]; the memory behavior of the technique was later improved by Misra et al. [22]. As described in Section 4, 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 13, our technique can also preserve a certain amount of compression during a transpose, though we may achieve lesser 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 [14] while deterministic strategies exist for LZ78 and LZW [25, 26]. These solutions are specialized to searching text; they do not apply to our transformations, and our technique does not apply to theirs.

In the realm of programming languages, Swartz and Smith present RIVL, a Resolution Independent Video Language [34]. The language is used to describe a sequence of image transformations; this allows the compiler to analyze the sequence and, via lazy evaluation, to eliminate any operations that do not effect the final output. Such a technique is complementary to ours and could also be implemented using StreamIt as the source language.

7. Conclusions

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.

While we formulated our transformation in terms of the cyclo-static dataflow model, the techniques can be applied within other functional and general-purpose languages so long as the right information is available and certain constraints are satisfied. The transformation relies on a regular pattern of data access; we use a streaming abstraction, but structured iteration over arrays could also suffice. We rely on static data rates in actors, which could also be expressed as functions with a fixed number of arguments and return values. Actors (functions) must be pure, without side effects or unresolvable dependences on potentially mutable data. While these properties are intrinsic to a language such as StreamIt, they also come naturally in most functional languages and may be adaptable to general-purpose languages in the form of a runtime library with a restricted API.

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 significant practical value.

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] Authentication generator demo. Online screencast. `http://penso.info/auth_generator`.

[4] S. Benza. Interview transcript. Computer Graphics Society. `http://forums.cgsociety.org/showthread.php?s=&threadid=115293`.

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

[6] Blender. Wikipedia, The Free Encyclopedia, November 2006.

[7] Blender.org: website statistics. Blender Foundation, 2006.

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

[9] Digital Juice, Editor's Toolkit 4: High Tech Tools, 2006.

[10] G. Dolbier and V. Megler. Building an animation and special effects studio from the ground up. IBM Report, 2005.

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

[12] 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.

[13] Elephant's Dream. `http://orange.blender.org/`.

[14] M. Farach and M. Thorup. String matching in lempelziv compressed strings. *Algorithmica*, 20, 1998.

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

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

[17] A. Lamb, W. Thies, and S. Amarasinghe. Linear analysis and optimization of stream programs. In *PLDI*, 2003.

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

[19] S. Levine. *Audio representations for data compression and compressed domain processing*. PhD thesis, Stanford University, 1998.

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

[21] 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.

[22] 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.

[23] 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.

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

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

[26] G. Navarro and J. Tarhio. Lzgrep: a boyermoore string matching tool for zivlempel compressed text. *Soft. Pract. Exper.*, 35, 2005.

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

[28] 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.

[29] B. Shen and I. Sethi. Direct feature extraction from compressed images. *Proc. SPIE Storage & Retrieval for Image and Video Databases IV*, 2670, 1996.

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

[31] 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.

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

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

[34] J. Swartz and B. Smith. RIVL: A resolution independent video language. *Proceedings of the Tcl/TK Workshop*, 1995.

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

[36] U.S. Geological Survey. Annual report of data sales, distribution, and archiving, 2004.

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

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

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

[40] J. Ziv and A. Lembel. A universal algorithm for sequential data compression. *IEEE Transaction on Information Theory*, 23(3), 1997.

[41] 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.