Acceleration in XIL Programs 21

Obviously, much of the speed of an XIL program is determined by the speed of the individual functions that make up the API. However, XIL applications can also realize big performance improvements when the library is able to replace the execution of a sequence of API-level functions (or *atoms*) with the execution of an optimized function that performs the work of all the atoms in the sequence. This type of optimized function is called a *molecule*.

You do *not* call a molecule directly. Rather, the library itself recognizes sequences of atoms that can be replaced by a molecule and performs the replacement automatically. This type of replacement is made possible by the library's deferred-execution scheme.

What Is Deferred Execution?

Here's how deferred execution works. In general, when an atomic function that affects the state of an image or a compressed image sequence is called, the function is not executed immediately. Instead, information about the operation—such as the function called and the arguments to the function—is stored by the library. This information continues to be stored until the library must produce a particular destination image—for example, because that image is to be displayed.

Let's say that the library has stored the five atomic operations shown in Figure 21-1 and that by performing these operations, it could decompress an image from a Cell CIS and prepare that image for display on a monochrome display.

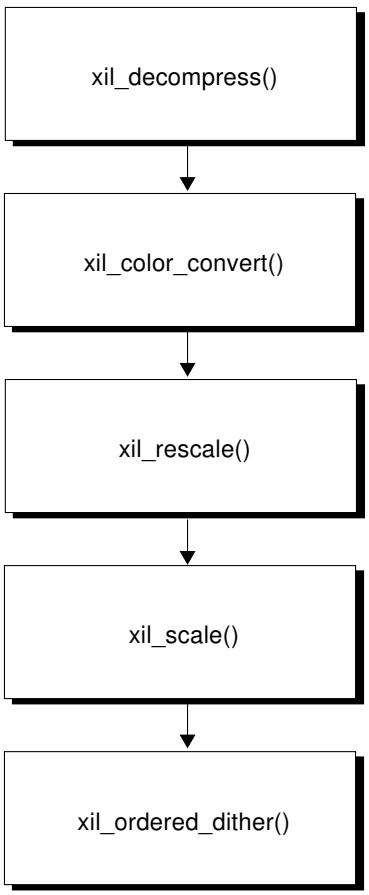

Figure 21-1 Stored Atomic Operations

When it must produce the destination of the ordered dither, instead of simply performing the five atomic functions, the library searches a list of molecules to see if all or part of this sequence can be replaced by a molecule. If the library finds a molecule that can perform this entire sequence of operations, the execution of the program will proceed as shown in Figure 21-2.

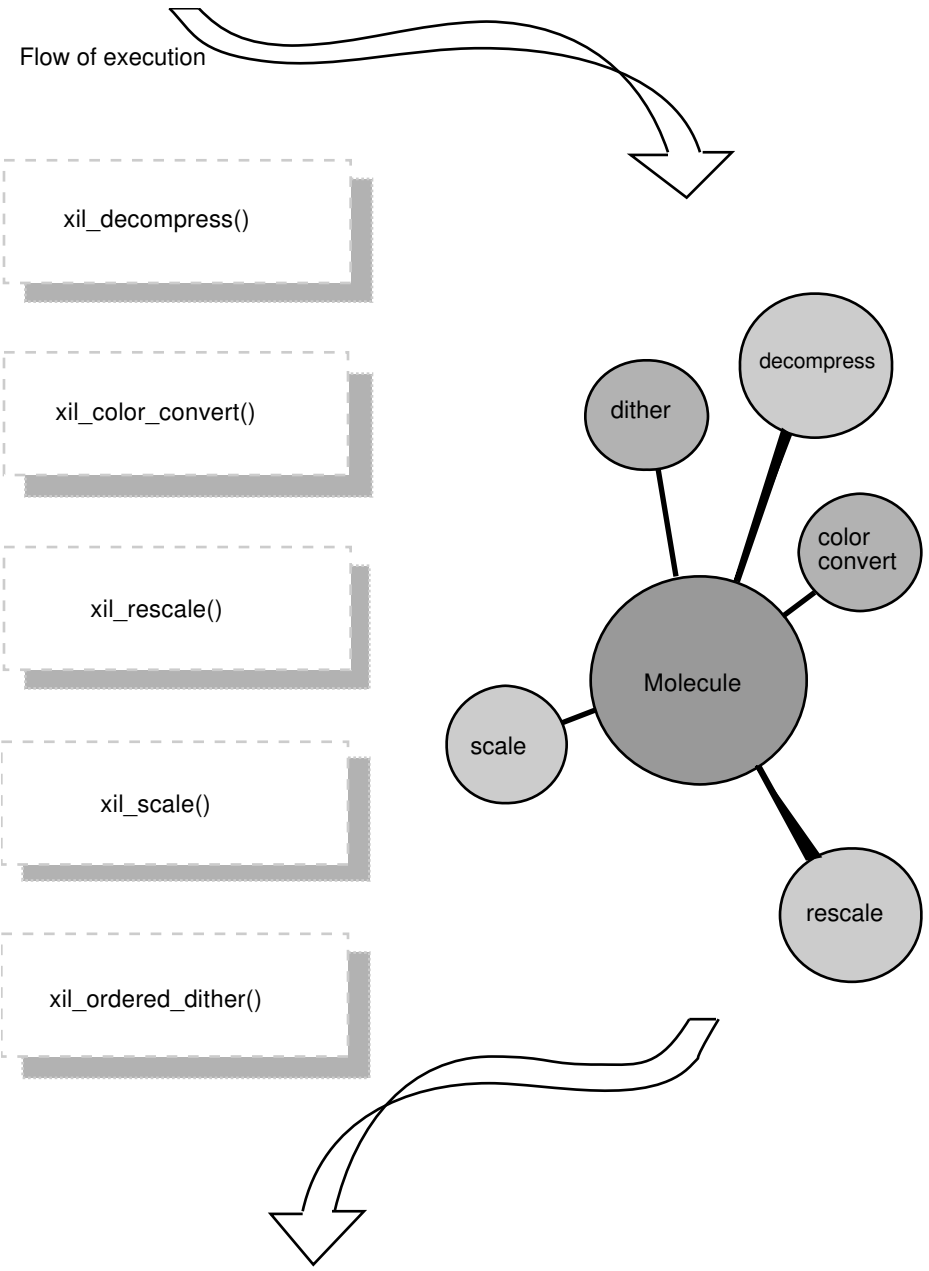

Figure 21-2 Replacing Atomic Functions with a Molecule

The molecule shown is executed and performs the jobs of all five atomic functions. None of the atomic functions is ever executed.

The ability to replace a series of atomic functions with a molecule like this can lead to dramatic increases in performance. Two reasons for this increase are that:

- A molecule may need to page an image into memory only once whereas the equivalent atomic functions would have paged the image into memory several times
- A molecule may not need to create temporary images that would be required by the equivalent atomic functions

XIL Molecules

The molecules available in the XIL library can be grouped into two categories:

- Molecules that involve decompressing images stored in a compressed image sequence. This is the largest category of molecules, and these molecules are available on all platforms that can run XIL applications.
- **SPARC:** Molecules that perform a common XIL operation and then display the results of the operation on a GX frame buffer. These molecules are available only on local GX frame-buffer screens.

This section first looks at some general rules you must follow to execute XIL molecules. (Particular molecules may require that you follow other rules as well.) The section then discusses the molecules that have been implemented in the categories mentioned in the preceding paragraph.

Rules for Executing Molecules

There are a few general rules you must follow to execute XIL molecules. These rules are listed below:

- The images that a molecule works with—source, intermediate, and destination—must have the same width and height. The principal exception to this rule occurs when a molecule performs a scaling operation. In that case, the destination image for the scale operation must have the same dimensions as the scaled source image.

- The images the molecule works with must have the same region of interest. For decompression molecules, this rule dictates that the molecule's destination image and any intermediate images have no region of interest because the images stored in the CIS cannot have a region of interest. In addition, if a molecule includes a scale operation, the images involved cannot have a region of interest. These images will not have a region of interest unless you have explicitly set their region-of-interest attributes.
- The images the molecule works with must have the same origin. For decompression molecules, this rule dictates that the molecule's destination image and any intermediate images have origins of 0.0, 0.0. This is true because the images stored in a CIS cannot have anything other than the default origin.

There are one or two exceptions to these rules among the decompression molecules. These exceptions are covered in the section "Video Decompression Molecules," where these molecules are defined.

Video Decompression Molecules

The library's video decompression molecules decompress an image from a compressed image sequence, process that image in some way, and write it to a destination, usually a display image. This destination may have a depth of 1 bit (a monochrome display), 8 bits (an indexed-color display), or 24 bits (a true-color display). To display video at the best possible speed, it is important that you use these molecules. They provide much better performance than the equivalent set of atomic functions.

Note – The molecules discussed in this section can be used to decompress and display images stored in JPEG baseline sequential, CellB, H.261, and MPEG-1 compressed image sequences. If you're working with a Cell bytestream, you should use the molecules discussed in the section "Cell Molecules" on page 288.

One-Bit Destination

The decompression molecule that displays an image on a 1-bit display works only with the JPEG decompressor and expects either a three-band YC_bC_r image or a one-band Y image as input. The molecule performs the following tasks:

- Decompresses an image from a CIS
- If necessary, converts a 3-band image to a 1-band image by extracting the luminance of the YC_bC_r image
- *Optionally* rescales the values in the Y image
- Dithers the 8-bit image to a 1-bit image
- *Optionally* zooms the dithered image by a factor of 2 in both the *x* and *y* dimensions

A requirement for this molecule is that the images in the CIS have a width and height that are multiples of the bitstream macroblock width and height. For example, single-band images must be divisible into 8-by-8 blocks, 4:2:2 images must be divisible into 16-by-8 blocks, and 4:1:1 images must be divisible into 16-by-16 blocks.

For this molecule to be called in the case where the source image is a YC_bC_r image, your application must include the code shown below.

```
XilColorspace ycc, y;
float scale_values[1], offset_values[1];
XilLookup cmap;
int mults[1] = {-1};
unsigned short dims[1] = 2;
XilKernel distribution;

scale_values[0] = 255.0 / (235.0 - 16.0);
offset_values[0] = -16.0 * scale_values[0];

/* The lookup table cmap must have two entries, and the values of
   those entries on the output side of the table must be 255 and
   0. One way to create this table is to use the call to
   xil_colorcube_create() shown in the example. You could also
   use xil_lookup_create(). */
cmap = xil_colorcube_create(state, XIL_BIT, XIL_BYTE, 1, 0,
    mults, dims);
distribution = xil_kernel_get_by_name(state,
    "floyd-steinberg");
ycc = xil_colorspace_get_by_name(state, "ycc601");
y = xil_colorspace_get_by_name(state, "y601");
xil_set_colorspace(imageYCC_24, ycc);
xil_set_colorspace(imageGRAY_8, y);

/* MOLECULE STARTS HERE */
xil_decompress(cis, imageYCC_24);
xil_color_convert(imageYCC_24, imageGRAY_8);

/* Rescale if the input image has CCIR 601 color space */
if (CCIR601)
    xil_rescale(imageGRAY_8, imageGRAY_8, scale, offset);

xil_error_diffusion(imageGRAY_8, imageMONO_1, cmap,
    distribution);

/* Execute if user has requested zoom */
if (ZOOM)
    xil_scale(imageMONO_1, imageMONO_1_zoom, "nearest", 2.0,
        2.0);
/* MOLECULE ENDS HERE */
```

Note – Arguments shown in boldface must be typed as shown for the molecule to execute correctly.

If the source image is a one-band image, the molecule should look like this.

```
/* MOLECULE STARTS HERE */
xil_decompress(cis, imageGRAY_8);

/* Rescale if the input image has CCIR 601 color space */
if (CCIR601)
    xil_rescale(imageGRAY_8, imageGRAY_8, scale, offset);
xil_error_diffusion(imageGRAY_8, imageMONO_1, cmap,
    distribution);

/* Execute if user has requested zoom */
if (ZOOM)
    xil_scale(imageMONO_1, imageMONO_1_zoom, "nearest", 2.0,
        2.0);
/* MOLECULE ENDS HERE */
```

Eight-Bit Destination

This molecule is optimized where the molecule's destination image is an 8-bit display image associated with a local GX frame buffer. However, the molecule will also accelerate the decompression and subsequent processing of images being written to other 8-bit destinations as well.

The molecule performs the following tasks:

- Decompresses an image from a CIS
- *Optionally* rescales the values in the three bands of the YC_bC_r image
- Dithers the 24-bit YC_bC_r image to an 8-bit pseudocolor image
- *Optionally* zooms the dithered image by a factor of 2 in both the *x* and *y* dimensions

Note – If you are using the CellB decompressor, the zoom operation (if requested) must precede the ordered dither.

There are certain restrictions on the use of this molecule. One is that the molecule can decompress images from a JPEG CIS only if those images are YC_bC_r images and the images are 4:1:1 interleaved in the bitstream. (CellB, H.261, and MPEG-1 CISs always contain YC_bC_r images.) In addition, in most cases, JPEG, H.261, and MPEG-1 CISs must contain images whose width and height are multiples of 16. Exceptions to this rule are noted later.

For this molecule to be executed, your application must include the code shown below.

```
float scale_values[3], offset_values[3];
XilLookup colorcube, colormap;
XilDitherMask dmask;

scale_values[0] = 255.0 / (235.0 - 16.0);
scale_values[1] = 255.0 / (240.0 - 16.0);
scale_values[2] = 255.0 / (240.0 - 16.0);
offset_values[0] = -16.0 * scale_values[0];
offset_values[1] = -16.0 * scale_values[1];
offset_values[2] = -16.0 * scale_values[2];
colorcube = xil_lookup_get_by_name(state, "cc855");
dmask = xil_dithermask_get_by_name(state, "dm443");

/* The lookup table yuv_to_rgb is matched with the colorcube
   cc855. If you use cc855 when dithering your images, you must
   install the 200 sets of RGB values in yuv_to_rgb in your
   application's X colormap before executing the molecule.
   Install the first set of RGB values at colorcell 54. */
colormap = xil_lookup_get_by_name(state, "yuv_to_rgb");

/* MOLECULE STARTS HERE */
xil_decompress(cis, imageYCC_24);

/* Perform this rescale if your image's color space is CCIR Rec.
   601 YCC. The rescale adjusts the range of the values in the
   three bands of the image to 0 to 255. The ordered dither
   operation that takes place later will produce the best results
   if the values are in this range. Note that YCC images produced
   by compressors that use the JFIF format have already been
   rescaled to 0 to 255 and, therefore, should not be rescaled
   here. */
if (CCIR601)
    xil_rescale(imageYCC_24, imageYCC_24, scale_values,
        offset_values);
```

```
if (ZOOM) {
    xil_ordered_dither(imageYCC_24, imageYCC_8, colorcube,
        dmask);
    xil_scale(imageYCC_8, zoom_displayimage, "nearest", 2.0,
        2.0);
}
else
    xil_ordered_dither(imageYCC_24, displayimage, colorcube,
        dmask);
/* MOLECULE ENDS HERE */
```

Exceptions to the Rules

Table 21-1 shows the exceptions to the general rules for executing this molecule. The exceptions have been classified on the basis of the decompressor to which they relate.

Note – Among these exceptions, the term *GX window* refers to a local GX frame-buffer screen.

Table 21-1 Exceptions to the General Decompression-Molecule Rules

Decompressor	Exceptions
CellB	If you want the molecule to zoom the image it decompresses, the call to `xil_scale()` that zooms the image must precede the call to `xil_ordered_dither()`.
JPEG	If the destination image is a display image associated with a GX window, the images in the CIS do not have to have a width and height that are multiples of 16. If the destination image is *not* a display image associated with a GX window, the molecule cannot perform the optional zoom. If the destination image is *not* a display image associated with a GX window, the destination cannot be a single-band child of a multiband parent image.
H.261	If the destination image is a display image associated with a GX window, the images in the CIS do not have to have a width and height that are multiples of 16. There is one exception to the general rules that a molecule's destination image must have the same width and height as the images in the CIS and that the destination image must have an origin of 0.0, 0.0. The molecule can execute even though the two rules mentioned above are not met, if the following conditions are satisfied: (1) the destination image is a display image associated with a window on a local GX frame buffer, (2) the coordinates of the destination image's origin are less than or equal to 0.0, and (3) all of the pixels in the display image are written to. This exception allows for the clipping of letterboxed images.
MPEG-1	If the destination image is a display image associated with a GX window, the images in the CIS do not have to have a width and height that are multiples of 16.

Twenty-Four-Bit Destination

The molecule performs the following tasks:

- Decompresses a YC_bC_r image from a CIS
- Converts the color space of the YC_bC_r image to RGB

Obviously, the images in a JPEG CIS must be YC_bC_r images before you can use this molecule. (CellB, H.261, and MPEG-1 CISs always contain YC_bC_r images.) In addition, the images in a JPEG CIS must be 4:1:1 interleaved in the bitstream. One other restriction is that JPEG, H.261, and MPEG-1 CISs must contain images whose width and height are multiples of 16.

For this molecule to be executed, your application must include the code shown below.

```
XilColorspace ycc, rgb;

ycc = xil_colorspace_get_by_name(state, "ycc601");
rgb = xil_colorspace_get_by_name(state, "rgb709");
xil_set_colorspace(imageYCC_24, ycc);
xil_set_colorspace(imageRGB_24, rgb);

/* MOLECULE STARTS HERE */
xil_decompress(cis, imageYCC_24);
xil_color_convert(imageYCC_24, imageRGB_24);
/* MOLECULE ENDS HERE */
```

The molecule can only convert images from `ycc601` to `rgb709` (as shown in the example).

CCITT Group 4 Decompression Molecule

Because it is common to transpose an image immediately after decompressing it from a CCITT compressed image sequence, there is a molecule for accelerating the transposition of images that were compressed with the Group 4 compressor. The molecule takes advantage of the fact that, when decompressing these images, an intermediate representation of the original image is generated, and it is faster to transpose this representation than it is to transpose the bits of the actual image. For images that were compressed with the Group 3 compressor, this intermediate representation is not generated and so the molecule does not execute.

The `xil_transpose()` call must immediately follow the image decompression; otherwise the intermediate representation is lost and the molecule cannot execute. In this case, the transposition is performed on the bit image.

The type of flip designated on the `xil_transpose()` call determines whether the destination size is checked as

```
src_x_size == dest_x_size && src_y_size == dest_y_size
```

or

```
src_x_size == dest_y_size && src_y_size == dest_x_size
```

One of the above expressions must be true for the molecule to be called. Table 21-2 shows the destination image size expected for the source image size (100, 140).

Table 21-2 Type of Flip Designated on the Call to `xil_transpose()`

Type of Flip	Expected Destination Size
XIL_FLIP_X_AXIS	(100, 140)
XIL_FLIP_Y_AXIS	(100, 140)
XIL_FLIP_X_180	(100, 140)
XIL_FLIP_MAIN_DIAGONAL	(140, 100)
XIL_FLIP_ANTIDIAGONAL	(140, 100)
XIL_FLIP_90	(140, 100)
XIL_FLIP_270	(140, 100)

21

Image-Filtering Molecule

The XIL library provides a molecule for accelerating image filtering when you use a 5-by-5 separable kernel. Assuming you observe the normal rules for executing molecules (see "Rules for Executing Molecules" on page 398), the molecule is executed when you filter either an `XIL_BYTE` or `XIL_SHORT` image and make two separate `xil_convolve()` calls, one call using a 1-by-5 kernel and the other call using a 5-by-1 kernel. It does not matter which kernel is used first; however, both `xil_convolve()` calls must use the `XIL_EDGE_ZERO_FILL` edge condition. There is no molecule for filtering `XIL_BIT` images.

The following code fragment shows an example of how your application code might execute this molecule.

```
XilImage src, dst;
XilKernel filter_1, filter_2;
XilSystemState state;
unsigned int width_1, height_1, keyx_1, keyy_1,
             width_2, height_2, keyx_2, keyy_2;
float data_1[] = {0.1117, 0.2365, 0.3036, 0.2365, 0.1117};
float data_2[] = {0.1117, 0.2365, 0.3036, 0.2365, 0.1117};

width_1 = height_2 = 1;
width_2 = height_1 = 5;
keyx_1 = keyx_2 = 2;
keyy_1 = key2_2 = 0;
filter_1 = xil_kernel_create(state, width_1, height_1, keyx_1,
    keyy_1, data_1);
filter_2 = xil_kernel_create(state, width_2, height_2, keyx_2,
    keyy_2, data_2);

/* MOLECULE STARTS HERE */
xil_convolve(src, dst, filter_1, XIL_EDGE_ZERO_FILL);
xil_convolve(src, dst, filter_2, XIL_EDGE_ZERO_FILL);
/* MOLECULE STARTS HERE */
```

SPARC *Molecules That Result in a Display*

Molecules that combine a common operation and a display are available only on local GX frame-buffer screens. Before you look over the list of molecules in this category, you should understand that operations that read from or write to a device image (including a display image) are treated specially by the XIL library.

Generally, calling an XIL function results in the library's storing one operation. However, operations that read from or write to a device image cause *two* operations to be stored. For example, you might call `xil_scale()` using a scanner as your source image in order to read an image from the scanner and zoom it. The two operations stored will be a *capture* operation, which writes its output to a temporary image, and a scale operation, which writes its output to the destination image named in the call to `xil_scale()`. Likewise, you might call `xil_scale()` using a display as your destination image in order to zoom the image and display it. In this case, the two operations stored will be a scale and a *display* operation.

Several molecules are available that replace two-operation sequences in which the second operation is a display to a GX frame buffer. These molecules are listed below:

- Copy (`xil_copy()`) an 8-bit image to a GX display image.
- Cast (`xil_cast()`) a 1-bit image to an 8-bit image and display the 8-bit image on a GX frame buffer.
- Pass a 1-bit image through a lookup table (`xil_lookup()`) to produce an 8-bit image and display the 8-bit image on a GX frame buffer.
- Pass an 8-bit image through a lookup table to produce a different 8-bit image and display the new image on a GX frame buffer.
- Rescale (`xil_rescale()`) an 8-bit image and display the result on a GX frame buffer.
- Set all the values in an 8-bit image to a constant (`xil_set_value()`) and display the result on a GX frame buffer.
- Translate (`xil_translate()`) an 8-bit image using nearest-neighbor interpolation, and display the result on a GX frame buffer.

21

Troubleshooting Molecules

Once you have coded your application with a replaceable sequence of atomic functions, you may want to make sure a molecule executes when your program runs. And if the molecule is not called, you will want to determine why it's not being called. This section discusses both how to determine whether a molecule is executing and what to check for if the molecule is not being called.

Determining Whether Molecules Are Executing

To determine whether a molecule you want to execute is actually executing, set the environment variable `XIL_DEBUG` as shown below before executing your program.

```
% setenv XIL_DEBUG show_action
```

Once you've done this, the XIL library will print a message to `stderr` each time an operation that affects the state of an XIL image or a compressed image sequence (CIS) is executed. By looking at these messages, you can determine whether the molecules your application is attempting to call are being executed.

For example, try setting `XIL_DEBUG` to `show_action` and then playing a JPEG movie using the example program `xilcis_example`. Instructions for running the example are presented in the section "Running the Movie Player" on page 231. As the program runs, you will see one or more pairs of messages—depending on the length of the movie—displayed on `stderr`.

```
XIL_ACTION[XilDeviceCompJpegMemory]:ordereddither8to8(rescale8(
decompress_Jpeg()))
XIL_ACTION[XilDeviceIOioxlib]:display_ioxlib()
```

The first message indicates that a molecule has been executed. This molecule performs a decompression, a rescale, and an ordered dither. The second message indicates that this molecule was followed by an atomic display operation; this message assumes the display operation uses function calls from `ioxlib`.

If you were now to edit this program so that it did something that prevented the molecule from being executed—like setting the origin of the display image to 1.0, 1.0—you would see the following messages:

```
XIL_ACTION[XilDeviceCompJpegMemory]:ordereddither8to8(rescale8(
decompress_Jpeg()))
XIL_ACTION:FAILED
XIL_ACTION[XilDeviceCompJpegMemory]:decompress_Jpeg()
XIL_ACTION[XilDeviceComputeMemory]:rescale8()
XIL_ACTION[XilDeviceComputeMemory]:ordereddither8to8()
XIL_ACTION[XilDeviceIOioxlib]:display_ioxlib()
```

As you can see, the molecule would be unable to execute, and the decompression, rescale, and ordered dither would have to be executed individually.

Besides the environment variable `XIL_DEBUG`, there are two XIL functions that enable you to control whether the messages discussed above are printed. One of these, `xil_state_set_show_action()` sets a system-state attribute called `SHOW_ACTION`. The setting of this attribute determines whether the library prints the messages, does not print the messages, or prints the messages only if `XIL_DEBUG` is set to `show_action`. The other function, `xil_state_get_show_action()`, reads the value of `SHOW_ACTION`. See the man pages for more information about these functions.

Determining Why a Molecule Is Not Executing

Basically, there are two reasons why a molecule might not execute. Either you didn't call the proper sequence of atomic functions with the proper arguments, or you did something that caused the library to flush some operations from storage before you finished building the molecule. These two situations are discussed in more detail below.

Not Calling a Correct Sequence of Functions

The most likely reason a molecule is not executing is that you haven't followed all the rules for calling the molecule. Remember, you must call the correct functions, in the proper order, sometimes with specific arguments. Any mistake can cause the molecule not be called or to fail.

For example, let's say you are decompressing a JPEG movie and displaying frames on an 8-bit frame buffer, and that you call the code shown below.

```
xil_decompress(cis, imageYCC);
xil_rescale(imageYCC, imageYCC, scale, offset);
xil_ordered_dither(imageYCC, image8, colorcube, dmask);
xil_scale(image8, zoom_displayimage, "bilinear", 2.0, 2.0);
```

If for no other reason, these four functions won't be replaced by a molecule because the call to `xil_scale()` uses an interpolation type of `bilinear`. For this molecule to execute, the interpolation type for the scale must be `nearest` (nearest neighbor). So, if you expect a molecule to execute and it doesn't, check first to make sure that you've met all the guidelines presented in the section "XIL Molecules" on page 398.

A more subtle problem can arise when the calls that make up a molecule are not consecutive calls in your program. For instance, consider the following code fragment taken from the Cell movie playback code discussed in "Playing Cell Movies" on page 241.

```
xil_decompress(cis, image24);

/* Look at the Lookup version number to see if it is time
   to reinstall the colormap. */
if (lu_version != xil_lookup_get_version( xil_cmap )) {
    cell_install_cmap(x_cmap, displayimage, xil_cmap, ilist);
    lu_version = xil_lookup_get_version( xil_cmap );
}
xil_nearest_color(image24, displayimage, xil_cmap);
```

This code is attempting to call a molecule that replaces the sequence

- `xil_decompress()`
- `xil_nearest_color()`

Your first thought may be that this code won't cause the molecule to be executed because the calls to `xil_decompress()` and `xil_nearest_color()` are not consecutive. Between these two calls, the program calls `xil_lookup_get_version()` at least once and may call a user-defined function `cell_install_cmap()`, which doesn't contain any XIL functions. Actually, this code will cause the molecule to be called because only the decompress and nearest-color operations are deferred and become part of the stored chain of operations that the library tries to replace with a molecule.

Calls to non-XIL functions, like `cell_install_cmap()`, of course are never deferred by the library. In addition, there is a class of XIL functions that are never deferred, and calls that read object attributes fall in this class. Therefore, the call to `xil_lookup_get_version()` does not affect the deferred-operation chain.

On the other hand, if you used the following code, the library would not execute a molecule.

```
xil_decompress(cis, image24);
xil_add_const(image24, constants, image24);
xil_nearest_color(image24, displayimage, xil_cmap);
```

The add-constant operation would be deferred (because it changes values in the image) and would prevent the library from finding an `xil_decompress()-xil_nearest_color()` sequence.

Flushing Operations Before a Molecule Is Complete

One other subtle problem to guard against is calling a function that causes the library to execute atomically functions that you intended to be the beginning of a molecule. For example, let's say that you want to execute a molecule that replaces four API-level functions. You call the first three atomic functions, and information about these deferred operations is stored. Then before you call the fourth function, you do something that causes the first three operations to be flushed from storage and executed.

What could you do to cause this to happen? There are a number of things, some obvious and some not so obvious.

At the obvious end of the spectrum, you could

- Turn deferred execution off using the function `xil_state_set_synchronize()`. Once deferred execution is turned off, of course, no operations are deferred.
- Call `xil_sync()` on an intermediate image. This call causes the library to produce that image immediately, which involves flushing all the stored operations needed to produce it.
- Use an intermediate image whose synchronization flag has been set using the function `xil_set_synchronize()`. The operations required to produce this image cannot be deferred.

You can also flush operations that you meant to be part of a molecule by

- Calling `xil_export()` to export an intermediate image. The operations needed to produce the intermediate image must be executed before the image can be exported.
- Calling a function that returns information about the values in an intermediate image—like `xil_extrema()`, `xil_histogram()`, or `xil_get_pixel()`. Such a call causes the operations that will produce the source image to be executed.

Side Effects of Executing Molecules

When a molecule executes, it may not create one or more temporary images that would have been created had the set of atomic functions replaced by the molecule been executed. This section briefly explains the possible side effects of this behavior and how you can deal with them.

As an example, consider the JPEG decompression code below.

```
XilImage imageYCC, displayimage;

xil_decompress(cis, imageYCC);
xil_rescale(imageYCC, imageYCC, scale, offset);
xil_ordered_dither(imageYCC, displayimage, colorcube, dmask);
```

If these functions are executed atomically, decompressed data is written to the 24-bit image `imageYCC`, the data in `imageYCC` is rescaled, and then the data in `imageYCC` is dithered to produce an 8-bit image. However, if a JPEG decompression molecule is executed, `imageYCC` is never actually used. This difference can lead to unexpected results later on.

Say that the decompression molecule executed, and `imageYCC` was never written to. If you later ask the library to destroy the CIS (without having first destroyed `imageYCC`), the library, before it destroys the CIS, will call `xil_decompress()` to decompress data from the CIS and write it to `imageYCC`. The library does this because it knows that it never wrote data to `imageYCC`, that the contents of `imageYCC` depend on the CIS, and that the image may be used again later in the program. If you don't plan to use the image later in your program, the call to `xil_decompress()` is unnecessary and hurts the performance of your program.

Note – The solution to this particular problem would be to destroy the 24-bit image before destroying the CIS or to use the function `xil_toss()` to toss the contents of the 24-bit image before destroying the CIS. When you toss an image, you tell the library that you don't care about the current contents of the image, but that you might write data to the image later.

You can check for unexpected side effects like the one described above by setting `XIL_DEBUG` to `show_action` and monitoring `stderr` to see which functions execute when you run your program.

21

XIL Functions That Relate to Deferred Execution

Several XIL functions have a direct effect on the way deferred execution works. These are listed in Table 21-3.

Table 21-3 Functions That Affect Deferred Execution

Function Name	What the Function Does
`xil_sync`	Forces the library to compute the value of an image when the operations that will produce the image have been deferred.
`xil_cis_sync`	Forces any outstanding call to `xil_compress()` to complete when it would otherwise have been deferred.
`xil_set_synchronize`	Set an image's synchronization attribute. When an image has this attribute set, operations on it cannot be deferred.
`xil_get_synchronize`	Determines whether an image's synchronization attribute is set.
`xil_state_set_synchronize`	Turns deferred execution on or off.
`xil_state_get_synchronize`	Determines whether deferred execution is being used or not.
`xil_toss`	Tells the library to "toss," or throw away, the contents of an image, but not to destroy the image.