HALCON
a product of MVTec

HALCON Memory Management

HALCON 24.11 *Progress-Steady*

MVTec Software GmbH

Building Vision for Business

This technical note describes HALCON's memory management

More information about HALCON can be found at: http://www.halcon.com

About This Technical Note

This technical note describes HALCON’s memory management.

Contents

Chapter 1

Introduction

Memory management ensures that applications always have fast and sufficient access to memory. In this technical note, we focus solely on main memory (or RAM) within the memory hierarchy. Managing main memory is crucial for an application's memory usage and speed.

From a software perspective, HALCON's memory management can also be represented as a hierarchy:

- The operating system (OS) manages memory in pages, which are made available to a process such as HALCON. For x86_64 these are 4KB or 2MB in size.
- In the process's runtime environment, a memory allocator is provided, enabling software to request and release memory from the system. Under Linux, HALCON uses malloc/free from the C runtime library by default, mimalloc can also be set using `set_system`. Under Windows, since version 22.11, HALCON uses either Microsoft Research's mimalloc allocator or the Win32 default-heap. For more information, see the respective Microsoft online documentation. Importantly, note that these allocators typically maintain their own caches, beyond HALCON's control. This can lead to HALCON releasing all memory, but from the system's point of view, the process may still occupy memory, which is actually being held within the allocator's cache.
- HALCON itself uses the memory allocator of the system, and places the caches on top.

In computer science, caching typically involves storing frequently accessed data for future use. However, HALCON's caches operate differently by retaining unused memory. This is advantageous because allocating new memory from the operating system is time-consuming. By caching memory that has been freed instead of returning it to the OS, HALCON optimizes performance and accelerates application speed.

Chapter 2

HALCON Caches

We generally do not recommend modifying HALCON's own cache settings unless necessary. For practical advice regarding when, and how to modify HALCON's caching behavior, and how to deal with memory related issues see chapter 4 on page 14.

HALCON uses three memory caches, which are discussed in the following sections.

2.1 Image Cache

Images typically require a lot of memory and allocating large memory blocks is slow. Therefore, HALCON does not release image memory back to the operating system if an image is freed. Instead, the image is cached for it to be used later. The upper limit for the image cache can be set via `set_system ('image_cache_capacity',<limit>)`. Choosing a higher limit will tend to speed up an application, but at the same time usually increases total memory consumption. To switch off the image cache, set the limit to *0*.

2.2 Global Memory Cache

Global memory refers to memory which is visible beyond a HALCON operator, and is typically used for output values of operators, such as tuples or regions. See Extension Package Programmer's Manual, section 3.2 on page 41 for more information on global memory.

The global memory cache is in *'exclusive'* mode by default. In this mode, each thread has its own global memory cache. If a memory block of 1024 bytes or less is requested by the thread, it will first look in its cache for a suitable block. If no block is found in the cache, the memory is requested from the runtime's memory allocator. Larger blocks are requested from the runtime's memory allocator directly. If a memory block of 1024 bytes or less is freed by the same thread that allocated it, it will be cached in the thread's global memory cache. Larger blocks or blocks that were allocated by a different thread than the one freeing it are returned to the runtime's memory allocator directly. This mechanism avoids thread synchronization during memory allocation, improving performance.

After setting the global cache mode to *'idle'*, any global memory that gets freed is released to the system immediately instead, and any global memory allocated is taken directly from the system. Any memory blocks already held in the cache are not freed. To do that, you need to explicitly call `set_system('global_mem_cache', 'cleanup')`.

2.3 Temporary Memory Cache

Temporary memory is only in use during the execution of an operator. As soon as the operator is finished, the used temporary memory will be freed. An example for this are intermediate images which are created during the execution of an operator but are not available as an output parameter. See Extension Package Programmer's Manual, section 3.2 on page 41 for more information on temporary memory.

If the temporary memory cache is switched on, this memory will be cached so that it can be reused during the next operator call. The main conceptional difference to the global memory cache is that the temporary memory cache works by allocating so-called *superblocks* which are managed in a stack. This makes the temporary memory particularly fast. The basic idea of a superblock is to allocate more memory than is actually needed for the current memory request. This provides a good chance that no new memory needs to be allocated for the next memory request. As a result of this, and because temporary memory is allocated and de-allocated much more frequently than global memory, the temporary memory cache usually has a much higher impact on performance, but also on HALCON's memory consumption than the global memory cache.

Some of the available settings provide thread specific variants *'tsp_'*. See the operator reference of `set_system` for general information on setting parameters with and without *'tsp_'*. Figure 2.1 shows the approximate performance of the different temporary memory cache modes with respect to speed and memory consumption. The exact behavior will depend on several factors such as the chosen parameters, the used hardware, the operating system, and the memory usage pattern of the application.

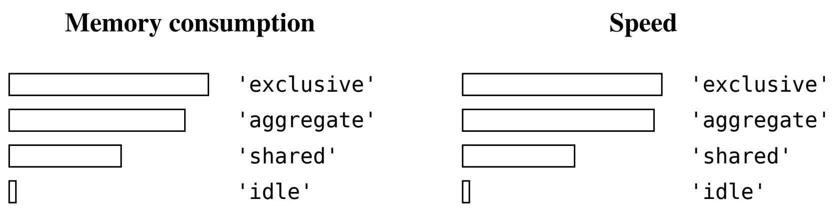

Figure 2.1: Performance of different temporary memory cache modes.

'exclusive'

In *'exclusive'* mode, each thread has an exclusive memory cache that can be used without needing to synchronize with any other threads, which saves time. The threads used for AOP (automatic operator parallelization) also have their own caches. AOP threads are assigned to tasks as they are ready, which means that work can be shared differently between them in each operator run. Over time, each AOP threads' cache will therefore grow to the maximum size required by any one AOP thread. This can lead to high memory consumption in rare cases.

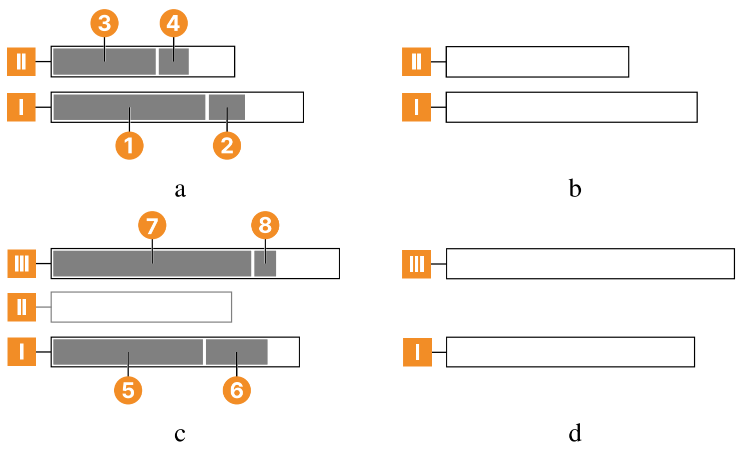

Figure 2.2: Superblocks in *'exclusive'* mode.

Figure 2.2 illustrates how the superblocks work in *'exclusive'* mode. For the first memory request ① of the first HALCON operator executed by a thread, a superblock I is allocated which is larger than the current request (figure 2.2 a). Per default, HALCON will use a heuristic based on the current image size to determine a sensible block size. The user can also set the minimum and maximum sizes of a superblock explicitly via `set_system` with the parameters *'alloctmp_min_blocksize'*, and *'alloctmp_max_blocksize'*. For most applications, however, it is not necessary to set either of these two parameters. For the second memory request ②, HALCON first checks whether there is enough free memory left in the current superblock and if there is, no new memory needs to be allocated. If a new memory request ③ cannot be fulfilled with the remaining space of the first superblock, a second superblock II is allocated in the same way as the first one. For subsequent memory requests, for example ④, HALCON will start checking the highest superblock which is already in use II. Therefore, the unused memory at the end of the lower superblock I is wasted. This wastage is not a memory leak as it can be reused later if the memory usage pattern allows it. This procedure will continue until the operator is finished.

At the end of the operator, all the temporary memory is freed and the superblocks are cached (figure 2.2 b). Since temporary memory is managed in a stack, it can only be freed in reverse order.

When the next operator is executed and memory is needed ,⑤ and ⑥, HALCON starts by checking the lowest superblock I first and will use it if enough memory is available (figure 2.2 c). If a new memory request ⑦ cannot be fulfilled with the remaining space of the first superblock, HALCON will check the next higher block II. If this block cannot fulfill the memory request either, HALCON will check the next higher block. If no higher block is available, HALCON will create one III, and release all lower blocks II that are unused back to the operating system. This release avoids wasting more and more memory in situations where individual memory request keep growing larger .This could happen, for example, if image sizes grow over time. Again, for subsequent memory requests ⑧, HALCON will start checking the highest superblock which is already in use.

At the end of the operator, memory will be freed and cached (figure 2.2 d).

’shared’

All temporary memory is cached globally in the temporary memory reservoir. This global reservoir is not to be confused with the global memory cache. A size limit for the reservoir can be set with the parameter *’temporary_mem_reservoir_size’*.

As with the *’exclusive’* mode, when a HALCON operator needs temporary memory, it will first check the (local) superblock(s) belonging to the thread from which the operator is called. In contrast to the *’exclusive’* mode, if the current memory request cannot be fulfilled this way, HALCON will check the reservoir for a suitable superblock. The superblocks in the reservoir are sorted by size and HALCON will choose the smallest one which is large enough according to the heuristic mentioned in the *’exclusive’* section. If HALCON can’t find such a superblock in the reservoir, it will create a new one. This new superblock will belong to the current thread until the operator finishes. When the operator is finished, the temporary memory belonging to its thread will be freed and the superblocks will be cached in the reservoir. If the reservoir is switched off via the parameter *’temporary_mem_reservoir’*, the superblocks will be returned to the operating system instead.

If there is more than one thread, including AOP threads, HALCON must ensure that only one thread can access the reservoir at a time. This leads to a synchronization overhead, which usually renders the *’shared’* mode slower than the *’exclusive’* mode. However, the total memory consumption is usually lower.

’aggregate’

The *’aggregate’* mode uses both caching options: local and reservoir.

As with the *’shared’* mode, when an operator needs memory, it will first check the local superblock(s) and then the superblocks in the reservoir. In contrast to the *’shared’* mode, when the operator finishes, HALCON will not cache all the used superblocks in the reservoir. Instead, temporary memory blocks that are larger than the threshold set with the *’alloctmp_max_blocksize’* parameter are cached in the global memory reservoir, while all smaller blocks are aggregated into a single block that is cached locally for each thread. If the global memory reservoir is disabled, the large blocks are freed instead. The aggregated block will be sized according to the temporary memory usage the thread has seen so far, but if set it will not be larger than *’alloctmp_max_blocksize’* (if set) or smaller than *’alloctmp_min_blocksize’*. Due to the aggregation of superblocks, this mode can avoid some of the memory wastage that the *’exclusive’* mode produces. At the same time, it can reach a similar speed because the aggregated superblocks in each thread can grow to a size that is big enough to fulfill most or all future temporary memory requests. Therefore, this mode presents a good balance between memory usage and speed.

Explicitly setting *’alloctmp_max_blocksize’* can be advantageous in this mode, as it can prevent the local caches from growing too large while still allowing occasional large memory requests to be satisfied from the reservoir.

’idle’

With significant impact on performance, the temporary memory cache can be switched off by setting it to *’idle’*. When the cache is switched off, HALCON will request new memory from the operating system every time an operator needs a piece of memory.

Keep in mind that turning the cache off does not reduce the amount of temporary memory HALCON needs to process an operator, it only reduces the amount of temporary memory HALCON will cache.

Switching between cache modes

Setting the cache mode via the thread specific variant `set_system('tsp_temporary_mem_cache', <mode>)` will only change the cache mode of the thread from which it is called.

Calling the global variant without *'tsp_'* will have the following effect on the active threads of a multi-threaded program:

- The cache mode of the current user thread (the one from where `set_system` was called) is changed.
- The new setting is applied to all user threads which are started afterwards.
- The new setting is applied to all AOP threads once they process the next operator in the current thread.

This has several important consequences:

- It is not possible to immediately change the cache mode of all threads with a single call of `set_system`.
- If the mode is changed in the middle of a multithreaded program from any one of the user threads, including the main thread, it will at least temporarily lead to a mixture of cache modes.
- To ensure that the entire program runs with the same cache setting, it is best to set the cache mode at the very beginning of your program.

As an example, switching off the temporary memory cache leads to:

- The cache of the current user thread, from where `set_system` was called, is deleted.
- User threads which are started afterwards will not cache temporary memory.
- AOP threads will delete their cache, and stop caching once they process the next operator in the current thread.

This implies that it is usually not possible to release the complete temporary memory cache back to the operating system immediately. It is also not possible to achieve this with a single operator call in a multithreaded program.

Procedure handles

A thread-specific, local, temporary memory cache is not directly associated with a particular thread, but with a proc-handle. Each thread, including AOP threads, has its own proc-handle. If a thread is deleted, the proc-handle is not deleted, but cached, so that it can be re-used by another thread. This ensures that the temporary memory cache associated with the terminated thread is not lost but kept with the proc-handle and can thus be re-used which improves performance.

This mechanism is particularly important for understanding the caching behavior when threads are created and deleted repeatedly. If a global cache setting is changed, e. g. the mode, via the variant without *'tsp_'*, it will be applied to the local caches associated with currently cached proc-handles, and to proc-handles which are cached afterwards. When the mode is globally changed to *'idle'* from one thread and another thread is deleted afterwards, this thread's proc-handle will be cached and the *'idle'* setting will be applied to this cache, which means the superblocks belonging to this thread will be released. If a new thread is created, it will receive one of the cached proc-handles and will therefore start with a temporary memory cache set to *'idle'*.

It is not possible to switch off the proc-handle cache or to release proc-handles.

Chapter 3

Image Data

HALCON images are composed of one or multiple channels, each consisting of a pixel matrix of the same dimensions and pixel type. Each channel refers to a memory buffer depending on height, width and the pixel type of the image. There is also a domain associated with each image, defining the the region of the image where valid pixel values are assumed.

Generally HALCON *only* creates a new image when pixel data or image size are modified. In this case, memory is allocated from the OS or reused from the HALCON image cache, see section 2.1 on page 8. Operators, such as `compose3` or `copy_obj`, do not modify pixel data and HALCON will not allocate new memory or write any pixel data, to improve performance. In HALCON, several images can share the same gray value matrix. This is also true for any domain (e.g., `reduce_domain`) or channel operators (e.g., `access_channel`), where the actual image data remains untouched. `reduce_domain`, for example, creates a new HALCON image that references a different domain but the same gray value matrices. If you want to reduce the size of the image, you must use the `crop_domain`, `crop_domain_rel`, `crop_part`, or `crop_rectangle1` operator. To enforce a deep copy, use the `copy_image` operator. In the C++ and .NET language interfaces, an additional `Clone` method for HALCON classes is available, which also performs a deep copy.

When a filter, cropping or other transformations, which modify pixel data or image size are applied, HALCON will create a new image. In-place operations are not supported, even if the input and output variables share the same name. A few exceptions to this rule are mentioned in the Quick Guide, section 2.1 on page 10. In-place operations are not always possible or reasonable to implement. For typical filters, for example, if the first (filtered) pixel was calculated and written into the original image, the original value of the first pixel would not be available anymore for the calculation of neighboring pixels. These kinds of complications would likely render the in-place operation slower. Since new pixel data has to be written anyway and HALCON's caching minimizes memory requests to the operating system, the advantage in terms of memory consumption would be small.

Internally, the HALCON library uses reference counters to keep track of the number of images that reference the same gray value matrix. Once the reference counter drops to zero, the memory used by the corresponding gray value matrix is automatically freed (and added to the image cache per default).

Example 1 - Read and filter an image

When reading image data from disk, new memory is allocated for the gray value matrix. The image `Image` points to this memory location and the reference counter is set to 1. The memory address can be queried via `get_image_pointer1`:

```
read_image (Image, 'fabrik')
get_image_pointer1 (Image, Pointer1, Type, Width, Height)
```

When a filter operation such as `mean_image` is applied, HALCON allocates new memory and writes the filtered image data to this new address. If the same variable is used for input and output, then the old image in it is replaced with the new image that points to the new address with an initial reference count of 1. This can be checked with another call of `get_image_pointer1`. The old memory location's reference counter drops to 0 and the destructor is called, freeing the old image memory:

```
mean_image (Image, Image, 3, 3)
get_image_pointer1 (Image, Pointer2, Type, Width, Height)
```

Example 2 - compose3 and overpaint_region or set_grayval

compose3 will not perform a deep copy of the image data. When used in the following way, the three channels of MultiChannelImage will therefore point to the same gray value matrix:

```
read_image (GrayImage, 'fabrik')
compose3 (GrayImage, GrayImage, GrayImage, MultiChannelImage)
```

If this is combined with one of the in-place operators, the result may be unexpected. In this case, over-paint_region first paints the first color into the gray value matrix, then the second color and lastly the third color (of the specified RGB values). Since all three channels refer to the same gray value matrix, only the last gray value (255) is seen in MultiChannelImage:

```
gen_rectangle1 (ROI_0, 100, 200, 200, 400)
overpaint_region (MultiChannelImage, ROI_0, [0,100,255], 'fill')
```

To have three independent channels in MultiChannelImage, copy_image can be used to produce deep copies:

```
read_image (GrayImage, 'fabrik')
copy_image (GrayImage, Red)
copy_image (GrayImage, Green)
copy_image (GrayImage, Blue)
compose3 (Red, Green, Blue, MultiChannelImage)
gen_rectangle1 (ROI_0, 100, 200, 200, 400)
overpaint_region (MultiChannelImage, ROI_0, [0,100,255], 'fill')
```

Chapter 4

Handling Suspected Memory Leaks in HALCON

A memory leak is defined as a piece of main memory, which is not being released and is either no longer used or no longer accessible. In many cases, an increase in memory consumption occurs due to memory caching, either by HALCON or by the operating system.

This increase will usually stabilize after a finite number of runs. Unless there is an actual memory related problem, such as an out-of-memory error, we recommend simply accepting this behavior. After all, the memory caching is there to speed up the application. Any investigations of memory behavior and any changes to HALCON's memory management settings should ideally be carried out on the final production setup with the final application and the actual production images. This is because there are many hardware and software factors, including image content, which can affect memory behavior.

If there is an actual memory related problem, follow these steps:

1. Switch off HALCON's memory caching. Enter this code at the very beginning of your application to switch off the caches:

   ```
   set_system ('global_mem_cache', 'idle')
   set_system ('temporary_mem_cache', 'idle')
   set_system ('image_cache_capacity', 0)
   ```

   If this resolves the memory problem and the decreased runtime is still acceptable, keep the caches switched off.

2. If resolving the memory issue in step 1 causes the application to become slow, consider restricting the amount of memory HALCON caches. There are various options available, such as:

   (a) Switch off one or two caches instead of all.

   (b) The temporary memory cache typically exerts the most significant influence on memory consumption and speed. A better balance between memory and speed can potentially be achieved by using the *'aggregate'* mode and specifying an application-dependent limit for the size of the superblocks:

   ```
   set_system ('temporary_mem_cache', 'aggregate')
   set_system('alloctmp_max_blocksize', <limit_in_bytes>)
   ```

   The size of the temporary memory reservoir can be limited via the parameter *'temporary_mem_reservoir_size*. The temporary memory cache can also be put into *'shared'* mode. Please see section section 2.3 on page 8 for more information on the temporary memory cache modes.

   (c) Another way to limit the memory consumption of the temporary memory cache is by reducing the number of threads. For AOP threads, this can easily be done via `set_system('thread_num', <number_of_AOP_threads>)`. More information on parallelization can be found in the technical note Parallel Programming or in the Programmer's Guide, section 2.2 on page 16.

(d) In some situations, for example when using a threadpool as in .NET Tasks, the memory issue can be mitigated by switching off the temporary memory cache at the end of each task, and switching it on at the beginning of each task.

3. Switch off mimalloc (under Windows). mimalloc tends to cache memory more aggressively than the Win32 default heap allocator. Therefore, switching to the default allocator can help resolve memory related problems in some cases:

```
set_system('memory_allocator', 'system')
```

4. When working with .NET, please refer to Programmer's Guide, section 11.1.3.3 on page 72. Memory consumption can potentially accumulate due to the behavior of the garbage collector.

5. Think about how to run the code suspected of leaking memory:

   (a) Run the operator or piece of code in a loop. If there is a leak, memory consumption should rise continuously, without saturating. How many runs are necessary to decide whether it is a leak and not a caching effect depends on the amount of memory leaked. For small amounts of leaked memory, it might be necessary to run the loop for a long time.

   (b) In general, using HDevelop to track memory use is problematic, as HDevelop constantly allocates and frees memory in response to user input. To minimize measurement noise as much as possible, export the HDevelop script as C or C++ code and run the exported code (see HDevelopUser's Guide, section 10.2 on page 307 for more information on the export of HDevelop programs). Alternatively, reduce the memory influence by HDevelop by switching off the updates at the beginning of the script:

   ```
   dev_update_off ()
   dev_update_time ('off')
   ```

6. Think about which tool to use for tracking memory consumption.
   The Windows task manager is not well suited for finding suspected memory leaks in HALCON (or any other process), unless it is an obvious, continuous leak. As explained in chapter 1 on page 7, memory already released by HALCON may still be allocated to HALCON from the operating system's and the task manager's point of view.
   There are many diagnostic tools to track memory consumption. To ensure that the results are interpreted correctly, it is always important to understand what the tool measures exactly.

Chapter 5

Frequently Asked Questions

Why does the memory consumption of my HALCON application keep climbing?

It is most likely caused by memory caching. Please refer to chapter 4 on page 14.

What should I do if I suspect there is a memory leak?

It is most likely caused by memory caching. Please refer to chapter 4 on page 14.

Why does the memory consumption not (immediately) decrease after I delete a HALCON object?

It is most likely caused by memory caching. Please refer to chapter 4 on page 14.

What is the purpose of memory caching?

Please refer to chapter 1 on page 7.

How do HALCON's caches work?

Please refer to chapter 2 on page 8.

Can all the cached memory be released immediately?

For the memory cached by HALCON, this depends on the type of cache.

- Image cache: `set_system('image_cache_capacity', 0)`
- Global memory cache: `set_system('global_mem_cache', 'cleanup')`
- Temporary memory cache: Please refer to the respective section on switching the cache mode in section 2.3 on page 8.
  Even when the total amount of memory cached by HALCON is 0, the memory allocator may still cache memory (please refer to chapter 1 on page 7).

Is there a way to predict or limit the total amount of memory HALCON uses?

Not generally, because HALCON uses dynamic memory allocation. The amount of memory HALCON needs depends on several factors such as the operators used, the chosen parameters, the number of threads, the image size, and the image content. An example for the dependence on image content is shape-based matching, where the amount of memory depends on the number of candidates found in the image.
HALCON's memory consumption can be reduced, for example by lowering the number of (AOP and user) threads, by reducing image size or image domain and by limiting the amount of cached memory.
Please refer to chapter 4 on page 14.

Is there a way to reduce the amount of memory HALCON caches?

Yes, please refer to chapter 4 on page 14.

Does HALCON cache data?

No, HALCON only caches unused memory.

Why are the first few operator calls or program runs (in a loop) usually slower?

In the very beginning, there is no cached memory to speed up the program. The caches will grow over time, increasingly speeding up operator execution, until they reach a plateau. If necessary, the user can run the relevant piece of code a few times using offline image data to 'warm up' the caches and achieve better processing times at the start of the live image acquisition.

Does HALCON allocate new memory for every output image?

Please refer to chapter 3 on page 12.

Are in-place operations supported for images? What happens if input and output image share the same name?

Please refer to chapter 3 on page 12.