Performance Analysis of Short-Term Memory in a State-of-the-Art H.264 Video Encoder

MASTERARBEIT

zur Erlangung des Mastergrades
an der Naturwissenschaftlichen Fakultät der
Paris-Lodron-Universität Salzburg

eingereicht von

Dipl.-Ing. (FH) Andreas Unterweger

Gutachter

Univ.-Prof. Dr.-Ing. Christoph Kirsch

Fachbereich

Computerwissenschaften

Salzburg, Oktober 2011

Performance Analysis of Short-Term Memory in a State-of-the-Art H.264 Video Encoder

MASTER'S THESIS

to obtain the Master's degree
at the Faculty of Natural Sciences of the
University of Salzburg

submitted by

Dipl.-Ing. (FH) Andreas Unterweger

Academic supervisor

Univ.-Prof. Dr.-Ing. Christoph Kirsch

Department of

Computer Sciences

Salzburg, October 2011

Acknowledgments

"A retentive memory may be a good thing, but the ability to forget is the true token of greatness." – Elbert Hubbard

I want to thank Professor Christoph Kirsch for supervising this thesis and all the time he spent originating new ideas to further improve the benchmark results. I also want to thank his Ph.D. student Andreas Haas and his Masters student Martin Aigner for their help concerning short-term-memory-related questions and implementation details. Finally, I want to thank all members of my family for their patience regarding my seldom visits due to the preparation of this thesis, most of all my mother who unfortunately did not live to see its completion.

Abstract

This thesis describes the process of porting *x264*, an H.264-compliant state-of-the-art video encoder, to short-term memory, a recent paradigm in memory management, in order to show the ease of use of the latter for heap management in complex applications. Different ways to apply the new paradigm are described and compared to one another and the original implementation. By evaluating each implementation variant's run time behavior, advantages and disadvantages of short-term memory management over classical memory management are discussed, revealing that the ease of use of those implementation variants which work for all tested parameter combinations comes at the expense of higher peak memory consumption (less than 57% overhead for the default multi-threaded configuration) with a negligible impact on execution time. Furthermore, it is pointed out that a reduction of this overhead is possible through a special implementation variant which only works for some parameter combinations, but reduces the aforementioned overhead to less than 19%. Nonetheless, the adequacy and efficiency of certain *x264* implementation variants for single-threaded configurations is shown.

Keywords: H.264, x264, Short-term memory, Memory management

Contents

List of Figures

List of Tables

Introduction

Since the development of the paradigm of short-term memory, numerous benchmarks in various programming languages have been carried out in order to demonstrate its performance and ease of use. However, most of these benchmarks deal with programs having only a few hundred or thousand lines of code. Therefore, they lack to reflect the effort necessary to port a complex application to short-term memory and to draw a conclusion on the ease of use of the paradigm in such an application compared to "classical" memory management using *malloc* and *free*.

In order to overcome these limitations, this thesis describes the transformation of *x264*, an H.264 compliant open source video encoder whose architecture is more complex than those of the applications previously ported, to short-term memory. Hence, it is a case study analyzing the simplicity-efficiency-trade-off of short-term memory management in comparison to "classical" memory management in a complex application, aiming at showing that significant short-term-memory-based simplifications of *x264* come at the expense of higher memory consumption, while keeping the change in execution time negligible.

Therefore, this thesis gives an overview of the paradigm of short-term memory, followed by an overview of *x264*'s architecture, including implementation details related to memory management. Due to the complexity of the H.264 standard, an overview of the latter's main concepts and definitions relevant for memory management as well as *x264*'s implementation of them is given. Thereafter, the port of *x264* to short-term memory is discussed, explaining multiple possible approaches. Finally, the performance of the ported versions is compared to the original, unmodified version, considering execution time, memory consumption and the number of changed lines of code, concluding with an analysis on the trade-off between ease of use and efficiency.

This thesis contributes an analysis of *x264*'s architecture and custom memory management as well as multiple approaches to port *x264* to short-term memory. Additionally, the design of various benchmarks measuring memory consumption and execution time among different *x264* implementation variants is provided.

1. Short-Term Memory Management

Short-Term Memory (STM) management is a new memory management approach developed by Christoph Kirsch and his group[1]. As opposed to "classical" Memory Management (MM) which is based on manual *malloc* and *free* calls[2], STM management relies on a limited lifetime of each allocated object, thus requiring no explicit *free* call. This chapter gives an overview of STM management based on [1] and [3] by describing its concepts and comparing it to "classical" MM. This includes an overview of the available C implementation which is used for the benchmarks described herein.

1.1. The concept of Short-Term Memory

In the STM model, objects are allocated for a limited time span, specified by an expiration extension, which implicitly defines an expiration date. This expiration extension can be prolonged arbitrarily using so-called *refresh* calls describing a new expiration extension. The time span specified by these extensions is thereby always relative to the current time, measured in so-called *ticks*, the concept of which is described below. After its expiration, the allocated object ceases to exist from the programmer's point of view, allowing the memory manager to reuse the allocated memory, thereby abolishing the need for an explicit *free* call.

1.1.1. The notion of time

In the STM model, time advances through explicit *tick* calls, concealing a time representation in integer *ticks*. Expiration extensions are represented in *ticks* relative to the current time. Figure 1.1 illustrates this by depicting two allocations with pre-defined expiration extensions of six and two, respectively. In the case of the first object which is allocated after the first *tick* with an expiration extension of six, this yields a lifetime of six *ticks* plus the time between the object's allocation and the first *tick* thereafter. Therefore, the first

object expires at $t = 8$, i.e. at the eighth *tick*. In general, the expiration date of an object is $t_{current} + e + 1$ where $t_{current}$ is the current time in integer *ticks*, e is the expiration extension and the offset of one accounts for the time difference between the object's allocation and the first *tick* thereafter.

As this offset of one makes the expiration extension being considered relative to the next

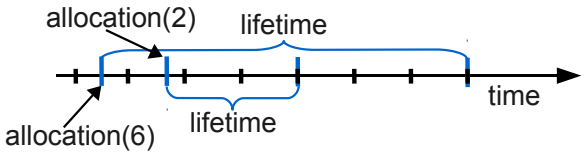

Figure 1.1.: The representation of time and expiration extensions in *ticks* and their influence on lifetime with time advancing constantly[1]

tick after allocation from the programmer's point of view, the shortest expiration extension possible, i.e. zero, yields expiration at the next *tick*. Even if the actual lifetime of an object is shorter than one *tick*, it expires at integer *ticks* only as both, current time and expiration extensions, are integers. This also applies to objects which are known to live for e.g. 4 *ticks* and one instruction, requiring an expiration extension of 5.

Note that the advance of time is crucial for the actual lifetime, i.e. "real world" time, of objects. If the actual time between *tick* calls is short, time in *ticks* advances fast, causing objects to expire faster. By contrast, if there are no *tick* calls at all, time in *ticks* does not advance, preventing all objects from expiring. Note that the notion of time does not rely on equidistant *ticks* relative to "real world" time, thus allowing for changes in the speed of object expiration during program execution.

1.1.2. Lifetime extension through *refreshing*

As the lifetime of objects and therefore their expiration extension is not known in advance in general, the STM model allows prolonging the initial life time through *refreshing* operations. Similar to the expiration extension specified at the time of allocation, the new expiration extension specified by the *refreshing* operation defines the lifetime of the object relative to the point of the *refreshing* operation. Note, however, that the lifetime cannot be shortened by *refreshing*. If the new expiration extension shortened the object's lifetime, the old extension is kept.

Figure 1.2 depicts an example which illustrates the impact of *refreshing*. The object allo-

cated after the first *tick* has an initial lifetime of two *ticks* due to its expiration extension of one. However, as it is *refreshed* shortly before the end of its initial lifetime at the third *tick*, its lifetime is extended up to the end of the sixth *tick* due to the expiration extension of three. Similarly, it is extended up to the eighth *tick* shortly before the end of its preliminary lifetime at the sixth *tick*. As no more *refreshing* operations are performed thereafter, the object expires at the eighth *tick*.

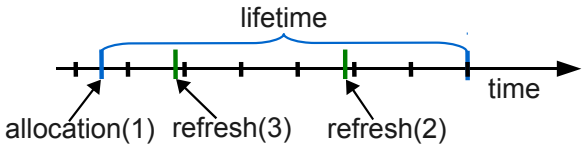

Figure 1.2.: Lifetime prolongation through *refreshing*[1]

1.1.3. The notion of time in multiple threads

When more than one thread performs MM operations using the STM model described above, a more sophisticated notion of time is necessary. Each thread involved in MM *ticks* separately, representing so-called *local* time. Thus, if an object is *refreshed* by n threads, it is logically associated with n expiration extensions – one for each thread. However, it is possible to over-approximate the object's lifetime per thread by one single expiration extension using the concept of *global* time described hereafter.

Global time advances synchronously with the slowest-*ticking* thread as shown in figure 1.3. This requires one *local tick* counter (also referred to as clock) per thread as well as one *global tick* counter. Although *global* time over-approximates the objects' lifetimes and therefore increase memory consumption due to possible later expiration, it avoids the need to store one expiration date per thread for each object, reducing management overhead and complexity.

Nonetheless, in order to prevent *global* time from standing still when a single thread blocks and hence does not *tick* anymore, the complete STM model requires the reintroduction of multiple expiration dates and further clocks, referred to as *thread-global* clocks. As no blocking of threads occurs within the scope of this thesis, *thread-global* time and its implementation details are not described herein, but can be found in [1, 3].

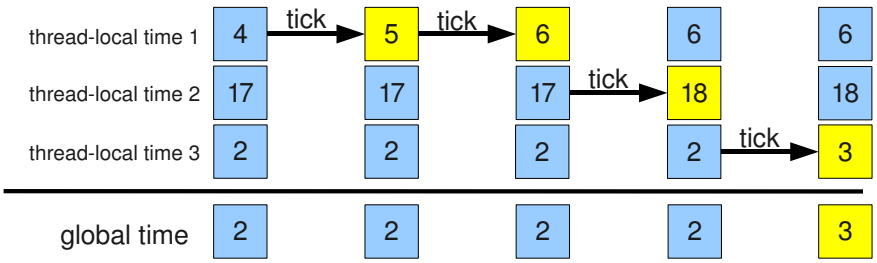

Figure 1.3.: *Global* time derivation. Adopted from [1]

1.1.4. Comparison to "classical" Memory Management

As the STM model is a different approach to MM than the "classical", i.e. the *malloc*- and *free*-based, approach, this subsection provides a comparison of the two. Objects in both paradigms can be in one of three states at any point in time: unallocated (i.e. not yet allocated), allocated and deallocated (or expired). Figure 1.4 shows these states and their transitions in terms of explicit function calls.
Both, the "classical" and the STM approach, allocate memory using *malloc*, yielding a

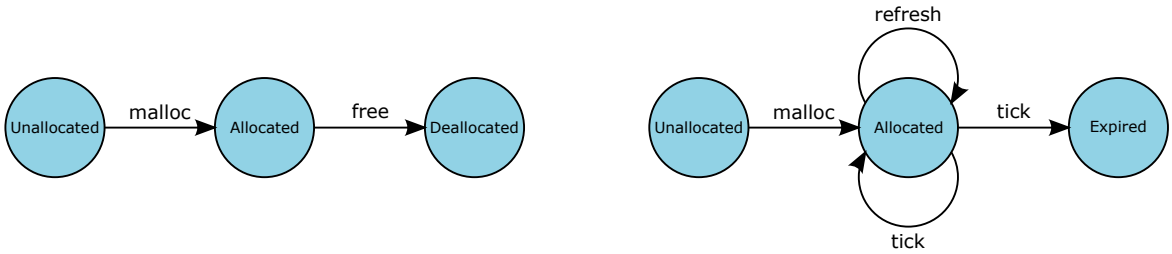

Figure 1.4.: Object allocation states and transitions in the "classical" (left) and the STM approach (right)

transition from the *unallocated* to the *allocated* state. Using the "classical" paradigm, the *allocated* state is not left until an explicit *free* call is issued, yielding the deallocation of the object with a final transition to the *deallocated* state. By contrast, in the STM model, a *tick* call leads to expiration if the lifetime of the allocated object ends in the current time period as described in subsection 1.1.1. This transition to the *expired* state may be preceded by one or more *tick* and/or *refresh* calls. The preceding *tick* calls don't affect the object's allocation state as long as the calls don't advance time so far that the object's expiration date is exceeded. In the case of *refreshing*, the expiration date is postponed, prolonging the object's lifetime as described in subsection 1.1.2.

Besides the differences in the state transitions between allocation and deallocation/expiration in the two approaches, the lifetime of objects is controlled differently. The "classical" paradigm allocates objects for a theoretically infinite amount of time, requiring the programmer to explicitly free them when they are not needed anymore, whereas this is done implicitly in the STM approach as the objects automatically expire if they reach the end of their lifetime. If the lifetime of an object is too short, however, this leads to a dangling pointer, as known from "classical" MM when objects are *freed* prematurely.
In multi-threaded environments, a programmer using the "classical" approach has to synchronize the *free* calls additionally so that there is neither a double free, nor premature deallocation in any of the threads involved. The STM paradigm does not require concurrent reasoning as objects expire if they expire for the last thread. However, the programmer must *refresh* all objects used by a thread with a sufficiently large expiration extension and place appropriate *tick* calls so that *global* time advances.
If time does not advance, no objects expire, yielding increased memory consumption and potentially out-of-memory errors. The same issue can occur in programs which use "classical" MM when a sufficiently large number of memory leaks is introduced through missing *free* calls. As objects require explicit *freeing*, the lack thereof cannot be compensated by other operations, contrary to the STM approach where reconstituting time advance through *ticking* can prevent an out-of-memory error.

1.2. C implementation

At the time of writing, implementations of the STM model for MM are available in the C, Java and Go programming languages. As *x264* is mostly written in C (see chapter 3), this section only gives an overview of C the implementation which is provided in form of a dynamic library relying on the *ptmalloc2* allocator of *glibc*[4]. By replacing the original allocation Application Programming Interface (API), i.e. the *malloc* and *free* functions, and including additional *refresh* and *tick* functions, it provides an MM interface for STM management called Self-collecting mutators (SCM).
SCM provide a conservative approximation of the STM model with no additional threads required for MM, thus being self-collecting. As multiple, potentially blocking threads have to be considered, the concept of *thread-global* time mentioned in subsection 1.1.3 is required, implying multiple expiration extensions per object as described in subsection 1.2.1.

1.2.1. Expiration extension management

Expiration extensions in the C implementation are stored in form of descriptor lists. A descriptor is a word containing the address of an object, i.e., it represents a pointer. Multiple descriptors belonging to objects with the same expiration date are grouped into a descriptor list. For each possible expiration date (limited by a compile time constant), there is one descriptor list. Storing expiration extensions from 0 to n requires $n + 1$ lists when using *local* time and a larger number of lists when using *thread-global* time as described in [1]. Note that the actual expiration date does not need to be stored, as it can be decoded implicitly from the list number.

For increased runtime performance, the library implements descriptor lists as singly-linked lists of so-called descriptor pages (instead of descriptors), depicted in figure 1.5. Each descriptor page is of constant size (defined at compile time), consisting of a pointer to the next page, a field storing the number of descriptors in the page and the descriptors themselves. Together with a list head and tail pointer, called *first* and *last*, respectively, this allows for the execution of all operations in constant time as described below.

At run time, two operations are required to manage expiration extensions: *insert* and

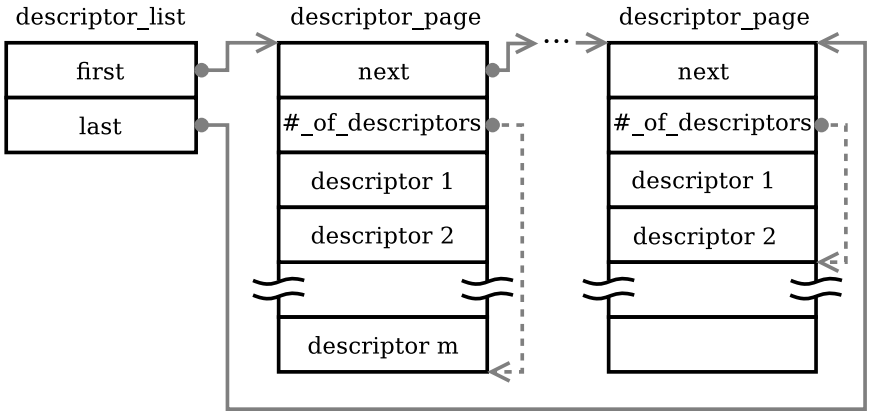

Figure 1.5.: Structure of descriptor lists and the contained descriptor pages[1]

move-expired. Given a descriptor and an expiration extension, the *insert* operation first determines the list number the descriptor is placed in by determining the difference i between the current time and the expiration extension, with i being within the range $[0; n]$. The descriptor is then stored in the i$^{\text{th}}$ descriptor list as follows: if the last page is not full, it is stored there by using the tail pointer of the list to get to said page and the number of descriptors in the descriptor page as offset to get to the final position within the page. If

the last page is full, a new page is allocated (using the underlying allocator) and the descriptor placed therein. Finally, the tail pointer of the list is adapted accordingly. All these operations can be performed in constant time as described.

The *move-expired* operation removes the i[th] descriptor list containing expired descriptors and replaces it with a new list consisting of one empty page. In order to improve the runtime performance, removed pages are not deallocated, but stored in a thread-local buffer called *expired-descriptor list* so that they can be reused when fetching a new page during the *move-expired* operation or appending a page during the *insert* operation. At each *tick* call, one descriptor of the *expired-descriptor list* is removed, thereby *freeing* the underlying object.

For each thread involved in MM, two sets of lists are stored: one for *local* time with $n + 1$ lists and one for *thread-global* time (which is equal to *global* time if no thread blocks[1]). *Insert* operations with an expiration extension of e are placed in list number $(i + e)$ mod $(n + 1)$ for the *local* set of lists, and similarly for the *thread-global* set of lists. Evaluating expiration dates requires a comparison to *local* and *thread-global* time, respectively. In the i[th] *local* time unit, objects with an expiration date of i are stored in list number i mod $(n + 1)$, expiring with the next *local tick* incrementing i, thereby issuing a *move-expired* operation on list number $(i - 1)$ mod $(n + 1)$ which is equal to $(i + n)$ mod $(n + 1)$. This applies similarly to *thread-global* expiration dates.

In order to distinguish between *local* and *thread-global* expiration extensions, the SCM API provides two *refresh* and two *tick* functions, respectively. *scm_refresh* prolongs the lifetime of an object using a *local* expiration extension, whereas *scm_global_refresh* uses a *thread_global* one. Both use the corresponding sets of lists in order to manage the expiration extensions. The *scm_tick* function advances thread-local time and advances both *thread-global* and *global* time if necessary, additionally performing one *move-expired* operation on the *local* set of lists. Similiarly, *scm_global_tick* advances *thread-global* time and performs the described actions on the *thread-global* set of lists. All described functions perform operations in constant time, thus being of constant time complexity themselves.

In addition to the sets of descriptor lists, each STM-managed object contains one word of header information which counts the number of descriptors stored for this object. This entails a space overhead of the size of one word per object, but it is crucial in order to determine which objects have expired for all threads and can therefore be *freed*.

All descriptor counters are initialized to zero after allocation. *Refresh* operations atom-

ically increment the descriptor counter of the affected object, whereas *remove-expired* operations decrement them atomically for all removed objects. If the counter reaches zero again, the corresponding object is *freed* using the underlying (de)allocator so that the memory can be reused. In order to guarantee constant time complexity for these operations, they are performed on one descriptor at a time, i.e. within a *malloc* or *refresh* call. Explicit *free* calls are also possible, but only perform an actual *free* operation if the descriptor counter is zero which allows for compatibility to "classical" MM as described in subsection 1.2.2.

1.2.2. Compatibility to "classical" Memory Management

The described SCM implementation cannot only be used for applications designed for the use of STM, but can also be linked against applications designed for the use with "classical" MM as the SCM API represents a subset of the "classical" MM API. From a programmer's point of view, SCM behave like "classical" MM as long as no *refresh* operations are issued. As this keeps the descriptor counter of each object at zero at any point in time, *free* calls always deallocate the specified object like in "classical" MM. However, the overhead of the descriptor counter for each object and the sets of empty descriptor lists remain, although this overhead can be minimized by setting the number of expiration dates n e.g. to zero at (library) compile time.
The extended API of the SCM implementation in comparison to the "classical" MM API is shown in table 1.1. Despite the allocation and deallocation functions which behave similarly in both approaches, the SCM implementation provides functions for the lifetime management of objects, whereas the "classical" MM API approach does not. Note that the SCM API also supports so-called finalizers which can be registered and assigned to objects. Finalizers are executed before *freeing* objects using the underlying allocator, allowing releasing resources which are not automatically *freed* if an object expires, making them eventually unreachable without finalizers. For the sake of readability, the finalizer functions are not included in table 1.1. The same applies to the thread management functions *scm_unregister_thread*, *scm_block_thread* and *scm_resume_thread* as they are not required for this thesis. Thread registration is performed implicitly by the library at the first *refresh* or *tick* call, thus requiring no separate function call.

MM functionality	"Classical" MM function call	SCM function call
Allocation	$malloc$	$malloc$
Deallocation	$free$	$free$
Reallocation	$realloc$	$realloc^*$
Lifetime extension (*local*)	–	$scm\_refresh$
Lifetime extension (*global*)	–	$scm\_global\_refresh$
Time advance (*local*)	–	scm_tick
Time advance (*global*)	–	scm_global_tick

* Implemented as a combination of *free* and *malloc* calls as described above

Table 1.1.: API comparison of "classical" MM and the SCM implementation. *Local* denotes the current thread, *global* means all threads.

1.3. Related work

The idea of STM was influenced by the concept of cyclic allocation[5] as stated in [3]. This approach uses buffers of fixed size for each allocation site, serving allocation requests by cyclically reusing buffer elements. Given a buffer size of $n - 1$, the nth allocation reuses the memory used to serve the first allocation request at this allocation site, thereby implicitly invalidating the data stored therein previously, omitting the need for explicit *free* calls. This corresponds to an STM-like scenario where each allocation at a given allocation site issues a *tick* of a per-allocation-site clock and a *refresh* of the newly allocated memory with an expiration extension of $n - 1$ which cannot be changed. By contrast, STM allows *refreshing* at any point in time and requires no reasoning concerning fixed buffer sizes.
Similarly, region-based MM[6] can be seen as a special case of STM in which each region, containing groups of allocated objects which can be deallocated jointly, maintains a separate clock with all objects' expiration dates set to its current time. Time advances on an explicit *free-region* call or by other means described in detail in [3, 6], thereby *freeing* all objects contained in the affected region. Placing an object in multiple regions allows for multiple expiration dates, similar to objects which are *refreshed* by multiple threads in the STM model. As described for cyclic allocation above, no explicit *refreshing* is possible.
This is also true for stack allocation from an STM point of view. Stack allocation can be seen as a special case of STM with constant expiration extensions equal to the lifetime of a stack frame with time advancing on each stack frame deallocation, i.e. a *return* from a function call[2]. More similarities of STM to concepts such as barrier synchronization (as time advance of the *global* clock), garbage collection with reference counting (compared

to the reference counting performed in the SCM implementation) and priority queues (as the set of descriptor lists in the SCM implementation with expiration extensions as priorities) are described in detail in [3].

2. The H.264 standard from an MM point of view

H.264/Advanced Video Coding (AVC) is a joint standard developed by the Moving Picture Experts Group (MPEG) and the Video Coding Experts Group (VCEG), also published as Part 10 of MPEG-4[7]. Its main focus is the efficient coding of moving pictures through temporal and spatial redundancy exploitation[8]. As *x264* compresses videos complying with the H.264 standard[9], this chapter gives a brief overview of its basic concepts and describes those parts in detail which are relevant for memory management, giving concrete examples in terms of the minimum memory consumption of an H.264 compliant video encoder.

2.1. Typical encoder architecture

The H.264 standard only describes syntactical and semantical elements relevant for video decoding, enabling different and therefore competing encoder implementations[8]. Nonetheless, most H.264 encoders share a similar basic architecture[10] depicted in figure 2.1. The input video signal – a sequence of pictures, represented as YCbCr signal with separate luminance and chrominance components (see subsection 2.2.1) – is split into so called macroblocks of $16 \cdot 16$ luminance samples which are further partitioned into blocks of $4 \cdot 4$ samples. Each of these $4 \cdot 4$ sized blocks is then transformed, quantized, entropy coded and written to the output bit stream.
In order to make use of temporal and spatial redundancies, encoded blocks are stored in form of decoded, i.e. inverse quantized and transformed, samples as available at the decoder side. Spatial correlation is exploited by subtracting extrapolated neighboring blocks within the same picture from the currently coded block prior transformation which is referred to as intra prediction. Exploiting temporal correlation requires the storage of the

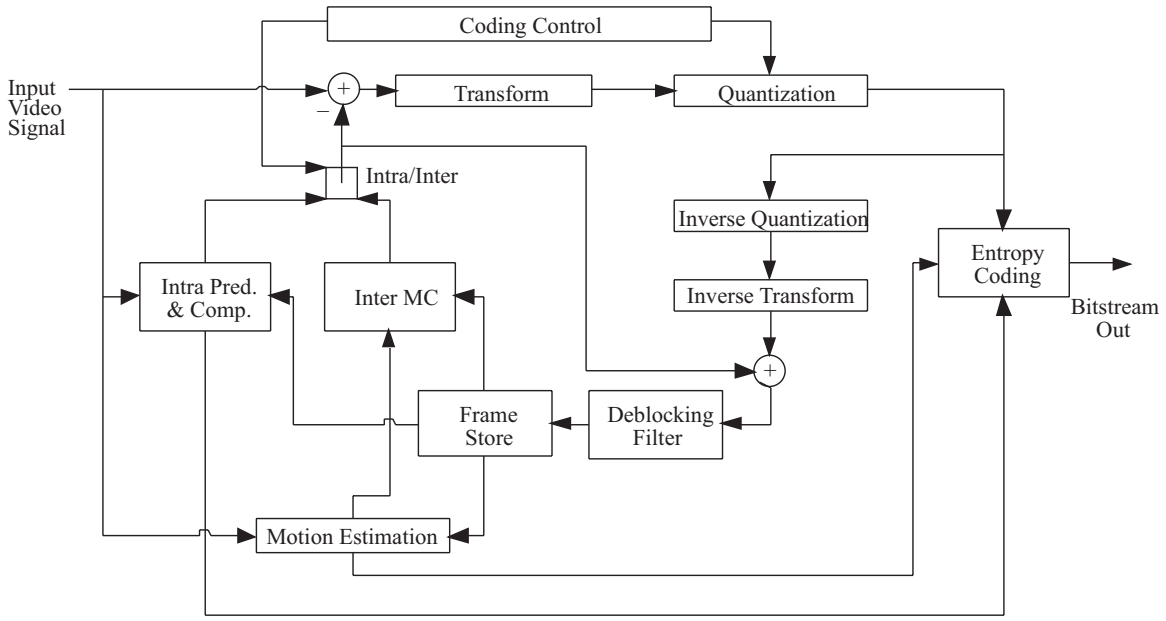

Figure 2.1.: Typical H.264 encoder architecture[10]

decoded blocks of previously coded pictures (also called frames in this context) which is achieved through a standardized frame store in which coded blocks are retained after being processed by a deblocking filter to reduce artifacts. Encoding a block using previously coded blocks from previous pictures consists of a motion estimation step which searches for a coded block which is similar to the one being coded and a motion compensation (abbreviated MC) phase which subtracts the found block from the current block. This process is also referred to as inter prediction.

The encoder's control unit (also called coding control) may choose between intra and inter prediction and adjust the degree of quantization, i.e. the strength of lossy compression. By doing so, the encoder is able to achieve a user-defined quality or bit rate goal, e.g. a limitation of the output bit stream to 1000 kbits/s. As both intra and inter prediction require previously coded blocks, the encoder may choose to exclude some of them from searching in order to speed up encoding to meet a user-defined time goal or omit these blocks completely in order to reduce the encoder's memory consumption.

2.2. MM-relevant encoder parts

The following section refers to both standardized and not standardized typical encoder parts described above by explaining which of them are relevant for memory management. The video coding aspects of these parts are described in detail in [7] and [8] whereas some of the details are omitted here for the sake of readability and the focus on memory management.

2.2.1. Input conversion

The H.264 standard handles pictures in the YCbCr color space, i.e. in form of a luminance plane (Y) and two chrominance planes (Cb and Cr), consisting of samples with n_{bits} bits size each with n_{bits} being a value between 8 as default for most broadcasting applications and 14 for high fidelity studio applications.[7]. Each plane is processed separately, although the chrominance planes may be present in a smaller resolution than the luminance planes, i.e. in subsampled form. Figure 2.2 illustrates the subsampling forms supported by the H.264 standard. With 4:4:4 subsampling, there is one chrominance sample for every luminance sample (rightmost figure), with 4:2:2 one for every two luminance samples and with 4:2:0 subsampling one for every four luminance samples (leftmost figure). Details about the J:a:b notation can be found in [11]. Herein, a value denoted by ρ_{chroma} will be used to specify the number of chrominance samples per luminance sample, i.e. e.g. $\frac{1}{4}$ for 4:2:0 subsampling.
If the input video is not present in form of uncompressed luminance and chrominance

Figure 2.2.: Chrominance subsampling (from left to right): 4:2:0, 4:2:2, 4:4:4. Crosses denote luminance samples, circles chrominance samples. Adopted from [7]

planes, input conversion is necessary. Details concerning conversions from other color spaces and picture representation forms can be found in [12]. Resizing, denoising and

similar operations may also be applied as part of the encoder's picture preprocessing, although this is out of the scope of this thesis. However, it has to be noted that some of these operations, most of all resizing, require additional memory for intermediate results.
After the conversion and preprocessing, the encoder has to decide upon the types of the input pictures[1]. The H.264 standard distinguishes between three different types – I (for intra), P (for predicted) and B (for bidirectionally predicted) – and a special form of I pictures called Instantaneous Decoder Refresh (IDR) pictures[7]. Although the exact differences between these types exceed the scope of this thesis, one aspect of B pictures is explained as it is relevant for the memory consumption of an encoder.
B pictures use both past and future picture references to predict the currently coded picture in order to improve compression[12]. As references to future pictures are not possible when processing the input video in a linear manner, the input order is changed in a predefined pattern so that future references are made possible. The H.264 standard allows the coding order to differ arbitrarily from the display, i.e. original, order, thus theoretically enabling arbitrary reordering in the encoder. Nevertheless, there are two common ways of reordering pictures[12, 13] as depicted in figure 2.3. Both skip the $n_B = 3$ (n_B may vary) B pictures to be and encode picture number 4 first, requiring the encoder to cache the B pictures to be. While successive reordering then encodes the 3 B pictures in their original order, each of them able to use multiple past and one future reference (picture number 4), hierarchical reordering changes the order of the B pictures too, allowing them to use the B pictures of the previous hierarchy level as references.

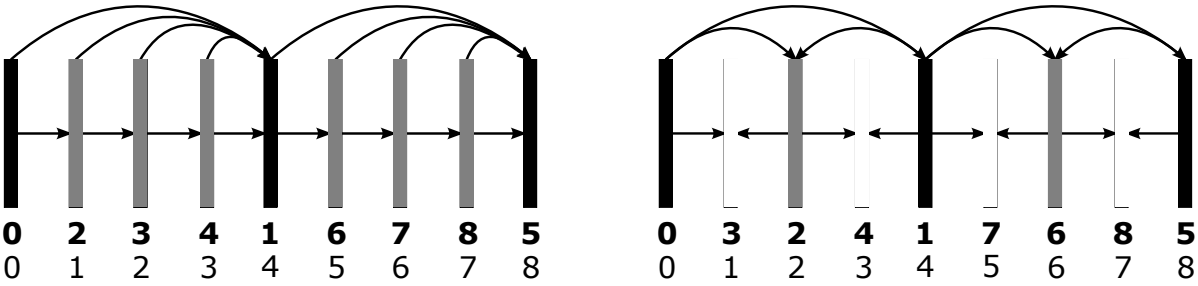

Figure 2.3.: Picture reordering for 3 bidirectionally predicted pictures (left: successive, right: hierarchical). Bold numbers denote coding order, regular numbers denote display order. Colors depict the hierarchy level if applicable (gray for first level, white for second level). Adopted from [12] and [13]

[1] Actually, this decision is made on parts of a picture, so-called slices. For the sake of simplicity, it is assumed that all pictures are split into one slice only which is possible

2.2.2. Macroblock splitting

As a picture is split into macroblocks of $16 \cdot 16$ luminance samples and the corresponding number of chrominance samples for encoding, it is necessary that both the width w_P and the height h_P of this picture are a multiple of 16. If they are not, the H.264 standard specifies padding through extrapolation at the borders to ensure that both width and height are a multiple of 16[7]. For the sake of simplicity and due the fact that the resolution of picture formats commonly used in video coding (like Common Interchange Format (CIF), Standard Definition (SD) etc.) satisfy this restriction, no padding is required, yielding no additional memory consumption. However, if input is used which does not satisfy the conditions described, it is necessary to consider that the encoder may need to allocate an additional amount of memory of the size of the padded picture for every stored input picture at any time instant.
In addition, access to samples and blocks may need to be byte-, word-, double-word- or quad-word-aligned on some platforms in order to be able to perform special instructions or to access single samples effectively[2]. For $n_{bits} = 8$ bits per sample, all samples are byte-aligned and each luminance and chrominance block is at least quadword-aligned for 4:2:0, 4:2:2 and 4:4:4 chrominance subsampling (the 4:2:0 subsampled chrominance blocks of a macroblock contain $16 \cdot 16 \cdot \frac{1}{4} = 64$ samples each which equal 2^8 bytes). If $n_{bits} > 8$, it is necessary to use 16 bits or 2 bytes per sample in order to ensure byte-aligned samples. The luminance and chrominance blocks of a macroblock will be at least quadword-aligned as for $n_{bits} = 8$.

2.2.3. Transform and quantization

As H.264 uses an integer transform which is an approximated Discrete Cosine Transform (DCT), optimized to be calculated by addition and shifting operations only[12]. The transform size can either be $4 \cdot 4$ or $8 \cdot 8$ luma samples for each macroblock, depending on the so-called profile used. The quantization of a transformed sample X_k is defined as the nearest integer to the following value according to [14]: $\frac{X_k \cdot c}{Q_{step} \cdot W_k}$ with W_k being a value of the predefined matrix W (which has either $4 \cdot 4$ or $8 \cdot 8$ entries, depending on the transform size used and differs for intra and inter coded blocks) to visually weigh transformed samples frequency-dependently and Q_{step} which defines the quantization step size and can be selected by the encoder within a defined range of 52 values, commonly represented as

Quantization Parameter (QP) between 0 and 51 from which Q_{step} can be calculated. c is a scaling factor which is defined as a power of 2 and can therefore be implemented using shift operations.
The inverse quantization process issued during picture reconstruction calculates the inverse quantized transformed samples based on the quantized values Y_k as $Y_k \cdot \frac{Q_{step} \cdot W_k}{c}$. The Inverse transform of these values yields the reconstructed samples which form the reconstructed, i.e. decoded, picture. Both quantization and inverse quantization can also be carried out using lookup tables for a limited number, i.e. 6, of combinations of Q_{step} and W as specified in detail in [7] and [14] by using periodicity properties. This allows for a less memory consuming implementation compared to an implementation with lookup tables for all possible combinations of Q_{step} and W.
After transform and quantization, the calculated values are zig-zag scanned as defined in [7] to increase the possibility of consecutive zeros which improves the compression efficiency in the subsequent entropy coding step. As the zig-zag scan can be implemented through reordering, is does not necessarily require any additional memory besides a temporary variable on the stack[15].

2.2.4. Entropy coding

The H.264 standard supports both Context Adaptive Variable Length Coding (CAVLC) and Context-Based Adaptive Binary Arithmetic Coding (CABAC) for entropy coding of transformed and quantized blocks after zig-zag scanning. Whereas CAVLC encodes a set of parameters of the current block, i.e. the number of non-zero coefficients, the pattern of trailing ones, the values of non-zero coefficients, the number of zeros embedded in non-zero coefficients and the location of the latter[14], CABAC models probabilities and probability transitions to increase the compression efficiency of the subsequent arithmetic coding step[16]. A detailed description of CAVLC and CABAC exceeds the scope of this thesis but may be found in [12] and [16].
CAVLC encodes the parameters described above by predefined bit patterns which can be implemented largely by lookup tables as described in [14]. Additional signaling of non-zero coefficient signs and sizes is necessary and can be partly implemented by lookup tables, followed by bitwise writing of coefficient delta values into the output stream. In contrast, CABAC requires saving separate context models for different coding modes, coefficient value ranges etc. to adapt to the probabilities of binary zeros and ones coded

and their transition probabilities in order to predict parts of the bits to be encoded, yielding higher compression ratios for the subsequent arithmetic coding step. The context models which store the probabilities are updated after each symbol coded in the respective context, thereby gradually adapting to the input signal. Depending on the number of modes and other parameters, a number of context models, including their initialization values as specified in the H.264 standard[7] need to be stored. The subsequent arithmetic coding step which is designed for low computational complexity and memory requirements is described in detail in [16].

All entropy coded coefficients of a picture (regardless of the entropy coding algorithm used) are grouped to macroblocks or partitions thereof and prefixed with coding mode information in form of macroblock headers and encapsulated in so-called Network Abstraction Layer (NAL) units whose format is described in detail in [8] and [17]. A start code which needs to be escaped when it appears within an NAL unit and a header-like set of bits indicate the type and content of an NAL unit if it is written to a file according to Appendix B of the H.264 standard. Although NAL units encapsulated in certain network protocols do not require these start codes, it is assumed that start codes are necessary as the benchmarks described in the subsequent chapters focus on file instead of network output.

2.2.5. Frame buffers

The Decoded Picture Buffer (DPB) is represented in form of two reference lists, L0 and L1[14], whose union is the DPB. While I pictures don't use references at all, P pictures exclusively use references in list 0. B pictures make use of references in both lists, IDR pictures clean the whole DPB. When using only P pictures, L0 acts like a First In First Out (FIFO) queue as shown in table 2.1. Assuming that 10 pictures with numbers[2] 0 to 9 are already encoded and picture number 10 is an IDR picture, followed by 9 P pictures with a maximum DPB/L0 size of 5, both lists develop as follows: after the IDR picture purges the DPB, the P pictures are added to it, discarding picture number 10 when picture number 15 is added due to the limited size of the DPB. Note that L0 is sorted in descending

[2] For the sake of clarity, the pictures are numbered consecutively, starting from 0. Actually, the pictures' numbers, i.e. their display order in form of the so-called Picture Order Count (POC), as well as their representations when referenced for reordering or removal in the list, i.e. the coding order or a value referred to as *frame_num*, are derived using different distances and wrapped around a defined maximum value. Details about this can be found in [7] and [14]

	L0 index				
Picture	**0**	**1**	**2**	**3**	**4**
...	...				
Picture 10 (IDR)					
Picture 11 (P)	10				
Picture 12 (P)	11	10			
Picture 13 (P)	12	11	10		
Picture 14 (P)	13	12	11	10	
Picture 15 (P)	14	13	12	11	10
Picture 16 (P)	15	14	13	12	11
Picture 17 (P)	16	15	14	13	12
Picture 18 (P)	17	16	15	14	13
Picture 19 (P)	18	17	16	15	14
Picture 20 (IDR)					
Picture 21 (P)	20				
Picture 22 (P)	21	20			
...	...				

Table 2.1.: L0 development for a scenario with 9 P pictures following an IDR picture and a DPB size of 5. Adopted from [14]

order of the pictures' display order to allow the representation of the list indices of pictures nearer to the one being currently encoded with fewer bits.

For B pictures, L0 is ordered differently: all pictures with a display order smaller than the display order of the one being currently coded are sorted in descending order, followed by those with a larger display order sorted in ascending order. In contrast, L1 is ordered the other way around: all pictures with a display order larger than the display order of the one being currently coded are sorted in ascending order, followed by those with a smaller display order sorted in descending order. Table 2.2 illustrates this using an example with an IDR picture and 9 preceding pictures, followed by a P picture, 5 successive B pictures (see subsection 2.2.1), a P picture, 5 B pictures etc. The picture numbers start at 0, the DPB size is 4.

After the IDR picture purges the DPB, picture number 16 is encoded before encoding the B pictures in between (see subsection 2.2.1). As IDR pictures don't use any references and P pictures only use L0, L1 is depicted empty for both of them. For the first B picture with picture number 11, two references are available which are ordered according to the description above. The same applies to the subsequent B pictures with picture numbers 12 and 13, respectively. However, the next B picture with picture number 14 requires the

Picture	L0 index 0	1	2	3	L1 index 0	1	2	3
...	...				...			
Picture 10 (IDR)								
Picture 16 (P)	10							
Picture 11 (B)	10	16			16	10		
Picture 12 (B)	11	10	16		16	11	10	
Picture 13 (B)	12	11	10	16	16	12	11	10
Picture 14 (B)	13	12	11	16	16	13	12	11
Picture 15 (B)	14	13	12	11	13*	14*	12	11
Picture 22 (P)	15	14	13	12				
Picture 17 (B)	15	14	13	22	22	15	14	13
Picture 18 (B)	17	15	14	22	22	17	15	14
...	...				...			

* If L0 and L1 are equal, the first two elements are switched temporarily[7]

Table 2.2.: L0 and L1 development for a scenario with 5 B pictures following each P picture and a DPB size of 4. Adopted from [14]

P picture with picture number 10 to be removed from the DPB due to its limited size. The same applies to the first B picture with picture number 11 when the last B picture with picture number 15 is being encoded. As in the example with P pictures only, L0 and therefore the DPB exhibits a FIFO-like behavior. The next P picture with picture number 22 does not use L1, thus discarding the oldest picture with picture number 11 from the DPB.

In addition to the default ordering and removal process defined in the H.264 standard, the encoder can issue a Memory Management Control Operation (MMCO) in between coded pictures in the bit stream to mark pictures in the DPB as unused for reference, thereby removing them. It is also possible to reorder one or more references in either list (L0 or L1) temporarily to improve compression efficiency by assigning smaller indices to references which are used more frequently, thus requiring fewer bits to be represented.

Pictures may also be marked as long-term references for either list so that they stay in it until they are removed (either by an explicit MMCO or an IDR picture), effectively reducing the size of the affected list for short-term, i.e. regular, references by 1. Details about long-term references can be found in [7] and [14]. As the encoder can choose to keep pictures for an arbitrary amount of time in the DPB, the lifetime of any picture is theoretically unbounded.

2.2.6. Motion estimation and compensation

Inter prediction combines motion estimation and motion compensation, where the former is not standardized, but the latter is[8]. Motion estimation tries to find a match for the current block in previously coded pictures stored in the DPB within a defined search area, i.e. usually a rectangular region of samples around the current block as depicted in figure 2.4[12]. Motion compensation subsequently subtracts the samples of the current block from those of the found block and continues with transform, quantization etc. In order to improve compression, the goal for motion estimation is to find a block with a minimal total difference[3] to the current block, although the encoder may choose to use different criteria to determine the best match. It may be necessary to store the minimum total difference of the current search and cache the corresponding difference block so that it does not have to be recalculated, although the encoder is not obliged to do so.
Both search range and pattern are subject to time and complexity constraints honored by

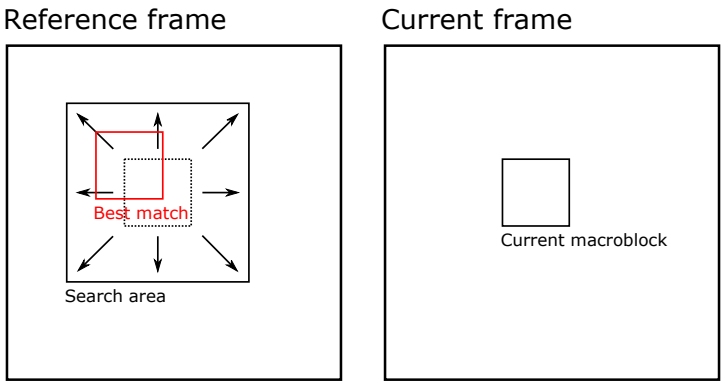

Figure 2.4.: Motion estimation in one previously coded reference frame. Adopted from [18]

the encoder with search range limits being implicitly defined by the maximum so-called motion vector length which may exceed the picture size and requires extrapolation[7]. The encoder may choose to avoid extrapolation and limit its search range differently as long as it stays within the limits defined by the H.264 standard. As motion estimation accuracy is not limited to samples, but quarter samples derived through interpolation from half samples, the encoder is free to decide upon the accuracy of motion estimation, influencing the complexity of the search. It is also possible to further split the macroblock into $16 \cdot 8$, $8 \cdot 16$ or $8 \cdot 8$ sized blocks and carry out a separate search on each of them.

[3] The sum of squared differences (SSD) and the mean squared error (MSE) are also commonly used[14]

2.2.7. Deblocking filter

The adaptive loop deblocking filter is applied to the whole picture after encoding[14]. As the DPB stores deblocked pictures which are used for inter prediction, the picture can be deblocked before being stored in the DPB. Together with intra prediction which operates on the currently reconstructed picture, i.e. the current one before deblocking, it is possible to handle all prediction cases by keeping a not yet deblocked reconstructed version of the picture currently being encoded which is deblocked after encoding and stored in the DPB. This requires additional memory of the size of one uncompressed input picture.

2.2.8. Encoder control

When compressing the input video, an H.264 compatible video encoder introduces a distortion, i.e. an error caused by quantization, which cannot be restored through inverse quantization[12]. Depending on its implementation and the bit rate or quality objectives, the encoder has to be aware of the degree of the distortion on the one hand and of the number of bits spent on the other hand in order to adapt the QP of every picture or macroblock accordingly.

The latter is commonly referred to as rate control which stores the number of bits per picture, macroblock group or macroblock (depending on the granularity) and adapts the QP used for the next macroblock(s) accordingly[14]. A prominent algorithm for rate control is *TMN5* which has been initially developed for MPEG-2 Video and takes the bits used for the last picture and the last macroblock into consideration, calculating a new QP based on the last macroblock's QP and the target bit rate. It is explained in detail, together with more sophisticated rate control algorithms for H.264, in [14].

In order to measure distortion in terms of difference, multiple approaches are available as explained in subsection 2.2.6. It is also possible to measure distortion in terms of visual distortion as approximated by Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity (SSIM) as explained in [17]. As choosing the encoding strategy for a macroblock yielding the smallest distortion disregarding its size may lead to suboptimal results[14], the computationally expensive approach of Rate-Distortion Optimization (RDO) may be implemented to overcome this issue. RDO consists of testing all possible encoding modes and measuring both each final size in bits and the corresponding distortion, selecting the mode which yields the lowest rate-distortion costs, i.e. the trade-off between rate and

distortion, through Lagrange optimization (for details see [17]). This approach is computationally more expensive as it requires each macroblock mode to be entropy coded in order to record its encoded number of bits, whereas distortion can be measured directly after inverse transform without the need for entropy coding.

2.3. Memory requirements

As explained in the previous section, a typical H.264 compliant encoder consists of multiple parts which are partially predetermined by the H.264 standard. This section gives an estimation of the memory consumption induced by each of these components using both general formulas and practical examples.

2.3.1. Input and decoded pictures

Both pictures from the video input and decoded pictures are stored uncompressed as they are accessed frequently[12]. One picture of width w_P and height h_P with ρ_{chroma} chrominance samples per luminance sample of n_{bits} bits each requires a total of $n_{bits} \cdot w_P \cdot h_P \cdot (1 + 2 \cdot \rho_{chroma})$ bits of memory for all luminance and chrominance samples (both Cb and Cr), assuming a whole number of luminance and chrominance samples. Using the default of 8 bits per sample and 4:2:0 chroma subsampling, an image of $720 \cdot 576$ samples (representing the default resolution of a video stream on a Digital Versatile Disc (DVD)) requires $8 \cdot 720 \cdot 576 \cdot \left(1 + 2 \cdot \frac{1}{4}\right)$ bits. This equals 622080 bytes or 607.5 KB. As shown in subsection 2.2.2, sample and block alignment don't need to be considered as $n_{bits} = 8$.

2.3.2. B pictures

Regardless of the exact reordering structure between B pictures (successive or hierarchical), an encoder using a predefined number n_B of B pictures has to skip n_B pictures after encoding an I or P picture in order to encode the next P picture, followed by the B pictures in between as explained in subsection 2.2.1. This requires the encoder to cache a maximum of n_B pictures before being able to proceed with the encoding of subsequent P pictures. With an input picture of size s_P in bytes, this yields a total extra memory consumption of $n_B \cdot s_P$ bytes for the required input caching. Using the DVD picture input

example from subsection 2.3.1 and $n_B = 3$ as default value in *x264*[19] this yields a maximum extra memory consumption of $3 \cdot 622080 = 1866240$ bytes or ≈ 1.8 MB.
B pictures usually improve compression efficiency, thus reducing the size of the output bit stream at constant quality[12]. While increasing encoder and decoder complexity and therefore encoding and decoding time, they do not require any additional memory besides the described input picture cache. As B pictures increase the number of possibilities how a single block is coded, RDO may however require some extra bytes to store the calculated costs for the possibilities actually tested if RDO is used (see subsection 2.3.8).

2.3.3. Transform and quantization

As described in section 2.2.3, the transform of a block with a transform block size of s_T requires $16 \cdot s_T^2 \cdot (1 + 2 \cdot \rho_{chroma})$ bits of memory for all luminance and chrominance samples as intermediate values are specified as being 16 bits large. Quantization can be implemented with or without lookup tables, influencing the required memory. If it is implemented without lookup tables, only the matrix W which has a size of $16 \cdot s_T^2$ requires additional memory as it contains $s_T \cdot s_T$ values. If lookup tables are used, $6 \cdot s_T^2 \cdot 16$ bits are required due to the 6 Q_{step} values and all possible values of W with an intermediate value size of 16 bits. In both cases, the transformed values can be overwritten by their corresponding quantized values as the intermediate transformed values are not required for any further operations, thus omitting the need to account for additional memory.
Using the DVD picture input example from subsection 2.3.1, a transform size of $8 \cdot 8$ and the lookup table approach described above, this yields a total of

$$16 \cdot 8^2 \cdot \left(1 + 2 \cdot \frac{1}{4}\right) + 6 \cdot 8^2 \cdot 16 = 1536 + 6144 = 7680$$

bits or 7.5 KB of additional memory for transform and quantization during encoding.

2.3.4. DPB

The size of the DPB is limited by the H.264 standard and may be explicitly signaled through syntax elements specifying the maximum L0 size in pictures in the bit stream[8]. The maximum size allowed by the standard depends on the picture size, but cannot exceed the size of 16 pictures. The actual DPB size also depends on the frequency of IDR pictures and changes depending on the picture type and position as well as on the reordering and

removal operations the encoder performs. Therefore, the exact DPB size cannot be determined before encoding (unless the encoder uses a constant prediction structure and handles the DPB reordering and removal operations independently of its input), requiring both encoder and decoder to allocate a number of s_{DPB} bytes depending on the maximum DPB size.

s_{DPB} depends on the so-called level specified in Annex A of the H.264 standard [7]. For the default level of *x264*, i.e. 4.1[19], it is specified as 32768 macroblocks. As one macroblock contains $16 \cdot 16$ luminance samples and $2 \cdot 16 \cdot 16\rho_{chroma}$ chrominance samples, s_{DPB} equals $32768 \cdot n_{bits} \cdot (1 + 2 \cdot \rho_{chroma}) \cdot (16 \cdot 16)$ bits or $1048576 \cdot n_{bits} \cdot (1 + 2 \cdot \rho_{chroma})$ bytes. As the number of pictures in the DPB is 16, this limits the DPB size depending on the picture size (with width w_P and height h_P) to

$$min\left(16 \cdot \frac{n_{bits}}{8} \cdot w_P \cdot h_P \cdot (1 + 2 \cdot \rho_{chroma}), 1048576 \cdot n_{bits} \cdot (1 + 2 \cdot \rho_{chroma})\right)$$

or

$$min\left(2 \cdot n_{bits} \cdot w_P \cdot h_P \cdot (1 + 2 \cdot \rho_{chroma}), 1048576 \cdot n_{bits} \cdot (1 + 2 \cdot \rho_{chroma})\right)$$

bytes.

Using the DVD input example of subsection 2.3.1, this yields a DPB size of

$$min\left(2 \cdot 8 \cdot 720 \cdot 576 \cdot \left(1 + 2 \cdot \frac{1}{4}\right), 1048576 \cdot 8 \cdot \left(1 + 2 \cdot \frac{1}{4}\right)\right) = 9953280$$

bytes or 9720 KB. As one picture requires 607.5 KB of memory, this limits the number of pictures in the DPB to $\lfloor \frac{9720}{607.5} \rfloor = 16$. As both L0 and L1 are only differently ordered representations of the DPB or parts thereof, they don't require any additional memory besides a maximum of the size of 16 pointers each which are referring to the corresponding DPB pictures.

2.3.5. Subsample interpolation

During motion estimation, the encoder may either search for matching blocks at sample positions or at half- and/or quarter-sample positions through subsample interpolation[8]. If the encoder chooses to search at sample positions only, no extra memory is required. Otherwise, it may be required to store the interpolated samples temporarily to improve performance when interpolated samples are accessed multiple times. As the motion es-

timation algorithm is not obliged to search every possible positions and fast algorithms which only search selected samples or subsamples are proposed in the literature (e.g. [20]), calculating the interpolated subsamples for the whole picture or the search region may not be feasible.
If the interpolation is performed on the fly for every subsample of each search position, no extra memory is required. Otherwise, searching with half-sample accuracy requires double the memory for at least the size of a macroblock (or the size of the search region when the interpolated samples are stored for reuse; the search region is not bounded by the picture size but the H.264 standard[7] and the respective encoder limits). Searching with quarter-sample accuracy requires double the memory of the half-sample accuracy search. As the extra memory required is highly implementation dependent, no concrete example is given at this point.

2.3.6. Entropy coding

As a H.264 compliant encoder may choose between CAVLC and CABAC entropy coding, its memory consumption changes accordingly, including the number of supported modes etc. implemented when using CABAC as described in subsection 2.2.4. The current version of the H.264 standard defines 1024 context models for CABAC, including their respective initialization values for the numbers of zeros and ones with 8 bit accuracy each, i.e. 16 bits per context model. As an encoder is not obliged to use CABAC in the first place, but may use CAVLC instead, the need to store context models is no longer applicable in this case, thus reducing its memory consumption. CAVLC does not require additional memory besides variables residing on the stack which temporarily store the number of non-zero coefficients etc. as described in subsection 2.2.4
Both CAVLC and CABAC may require buffers for one or more bytes in order to be able to write single bits to the output stream. As entropy coded blocks are not byte aligned for the sake of compression efficiency, storing previously coded bits may be necessary in order to append the currently coded ones at arbitrary bit positions. Buffering output in form of NAL buffers in general removes this requirement of extra bytes and is explained in subsection 2.3.7. As the extra memory required is highly implementation and platform dependent, no concrete example is given at this point.

2.3.7. Encoded pictures and buffers

The size of an encoded, i.e. entropy coded, macroblock depends on the QP used for the quantization of its transformed samples and its signal characteristics[8]. Although the QP has a significant impact on the size of the encoded macroblock, its signal characteristics may yield an encoded macroblock which is larger than the original, uncompressed macroblock for low QP. The H.264 standard allows to work around this issue by introducing a so-called I_PCM mode which bypasses the encoding process and writes the uncompressed macroblock directly into the bit stream[7]. However, the encoder is not forced to use this mode, theoretically allowing each encoded macroblock to be bigger than its uncompressed original.

As the encoder has to write the bit stream of encoded macroblocks and meta information to a file or a device eventually, it may implement buffers to temporarily hold multiple encoded macroblocks or even a whole encoded picture before writing them. Buffers of this kind may be heuristically dimensioned according to a macroblock's QP or dynamically reallocated if necessary. As certain sequences appearing within an NAL unit in the bit stream need to be escaped (in order to not be misinterpreted as NAL unit start codes, see Annex B of the H.264 standard[7]), it is possible in these cases to buffer a whole encoded picture or NAL unit in order to perform the escaping before writing. Theoretically, saving the last 2 bytes suffices as the start codes are 3 bytes long. As described above, the maximum size of such a buffer is not limited by the size of its uncompressed macroblocks for low QP. As the memory required is highly implementation, QP and input dependent, no concrete example is given at this point.

2.3.8. Rate control and RDO

If the encoder implements a rate control algorithm as described in subsection 2.2.8, is has to store at least the number of bits or bytes of the last coded macroblock, macroblock group or picture. For more sophisticated rate control algorithms, it is necessary to additionally store previous QP values and statistics in order to account for them when calculating the QP for the next macroblock. Although the encoder is not obliged to implement any form of rate control, it is assumed that it does as it is otherwise not possible to specify a target bit rate[14]. As the number of bits spent depends on the input, the QP and the rate control granularity, it is implementation dependent. As macroblock granularity is the finest

granularity possible in H.264 (the QP can only be changed per macroblock), the size in bits of one encoded macroblock determines the number of bits used for storing this size. As explained in 2.3.7, the size of one encoded macroblock is not bounded by the size of one uncompressed macroblock. Assuming the use of the I_PCM mode, the encoded size is bounded by $16 \cdot 16 \cdot n_{bits} \cdot (1+2 \cdot \rho_{chroma})$ bits, thus requiring $log_2(16 \cdot 16 \cdot n_{bits} \cdot (1+2 \cdot \rho_{chroma}))$ bits to store this size.
In order to calculate the mode with the smallest distortion, it is also necessary to save the minimum distortion of the previously tested modes and the corresponding mode number in order to determine the overall minimum distortion of all modes. As the number of modes in P and B pictures depends on the motion search range and accuracy as well as the number of reference frames[4], the number of possible modes is limited by the encoder parameters, but requires at least one byte, albeit located on the stack. The same applies to the minimum distortion, although it may require more memory as the maximum distortion exceeds n_{bits} bits when measuring distortion by squaring values.
When using RDO, additional memory may be required to store the bit stream for the mode with the lowest costs of all modes tested yet. This allows for a reduction of encoding time as the macroblock does not have to be coded again with the optimal mode found, but requires additional memory for one encoded macroblock as explained in subsection 2.3.7. As the encoder is not obliged to use RDO at all, this does not necessarily require any additional memory. As the memory required is highly implementation dependent, no concrete example is given at this point.

2.3.9. Total memory consumption estimation

A single-threaded H.264 encoder operating on input pictures of width w_P and height h_P with n_{bits} bits sample size and chrominance subsampling ρ_{chroma} using n_B B pictures and a transform size of s_T luminance samples requires at least the following amount of memory in bytes as explained in the previous subsections:

- $s_P = \frac{n_{bits}}{8} \cdot w_P \cdot h_P \cdot (1 + 2 \cdot \rho_{chroma})$ for the input picture currently being processed
- s_P for the picture currently being reconstructed, i.e. decoded
- $n_B \cdot s_P$ for the B picture buffer

[4] As intra modes may also be used in P and B pictures, there is an additional bias of the size of the number of intra modes

- $2 \cdot s_T^2 \cdot (1 + 2 \cdot \rho_{chroma})$ for transform and quantization without lookup tables
- $min\,(2 \cdot n_{bits} \cdot w_P \cdot h_P \cdot (1 + 2 \cdot \rho_{chroma}), max_{DPB}(l) \cdot n_{bits} \cdot (1 + 2 \cdot \rho_{chroma}))$ for the DPB where $max_{DPB}(l)$ is a value defined in Annex A of the H.264 standard dependent on the so-called level l
- $\lfloor 4 \cdot 2 \cdot \frac{s_{DPB}}{s_P} \rfloor$ for pointers in L0 and L1 to the DPB, assuming a 32-bit architecture and therefore pointers of 4 bytes size
- 0 for subsample interpolation
- 2 for NAL unit start code detection without buffers
- $\frac{log_2(16 \cdot 16 \cdot n_{bits} \cdot (1+2 \cdot \rho_{chroma}))}{8}$ for (simple) rate control (rounded to whole bytes)
- 0 for CAVLC entropy coding

In total, this yields the following minimum amount of memory for the DVD example described in the previous subsections and a minimum transform size of $s_T = 4$: $622080 + 622080 + 48 + 1866240 + 9953280 + 128 + 0 + 2 + 2 + 0 = 13063860$ bytes or ≈ 12.5 MB. As can be seen, the uncompressed input and decoded pictures in the DPB account for the greatest part of the overall minimum memory consumption. For larger input picture resolutions, e.g. High Definition (HD) resolutions of $1920 \cdot 1080$ samples, this yields a yet higher total memory consumption.

2.4. Related work

A detailed quantitative complexity and performance evaluation of the reference H.264 encoder is given in [21] which is based on a draft version of the H.264 standard and includes code parts which were omitted during the standardization procedure. The results shown therein are implementation specific and mainly aim at quantifying the additional encoding time and memory necessary to achieve a gain in compression. It also includes a decoder analysis whereas this thesis focuses on encoding only.
[22] gives a general overview of the H.264 standard and describes the parts of the final standard (as of 2004) in terms of processing speed and memory access frequency using different parameters. Similar to the overview papers [8], [10] and [13], the rate-distortion performance is analyzed for different scenarios and compared to previous video coding standards like MPEG-2 Video and MPEG-4 Part 2.

3. *x264* – an H.264-compliant video encoder

x264 is a video encoder which converts sequences of input pictures from a number of different input formats into compressed, H.264-compliant bit streams. This chapter gives an overview of *x264*'s architecture, mainly focusing on data flow and code parts relevant for MM, thus allowing explaining the modifications performed for STM in chapter 4. The description herein is based on the code of version *r1724* (committed on September 19, 2010) from the official *x264* repository[1] and the included, text-file-based overview of design choices related to multi-threaded encoding[23] as well as [9].

3.1. General encoder architecture

x264 is written in C, specifically a subset of C99[24], including various platform-specific optimizations written in the corresponding assembly languages. The latter are processed by *yasm*[2] and linked to the compiled C object files when creating the *x264* executable which is a command line application that allows specifying parameters (such as target bit rate, input and output file) for encoding. As this thesis focuses on MM-relevant aspects, an architectural overview of *x264* is provided, albeit limited to the minimum amount necessary to explain the modifications in chapter 4. At the time of this writing (April 2011), the encoder architecture of the current *x264* version *r1924* does not differ from the one described herein based on version *r1724*.

[1] Available at `http://git.videolan.org/gitweb.cgi?p=x264.git;a=shortlog`; revision *r1724* can be downloaded from `http://git.videolan.org/gitweb.cgi?p=x264.git;a=snapshot;h=b02df7b3b3b8616078851aab65d77ca435e2ff93;sf=tgz`

[2] `http://www.tortall.net/projects/yasm/`

3.1.1. Architectural overview

As *x264* processes video data separated into single pictures, the conversion of the latter by a tool chain forms a coarse-grain model of *x264*'s architecture, split into single tool chain items for each task. Figure 3.1 depicts the tool chain items, together with the main code files they appear in. Note that the file names relate to the position of the calls to the functions executing the tool chain element's task(s), thus hiding the complexity behind it as well as the number of files involved.
As depicted in figure 3.1, a compressed input file containing compressed picture data is

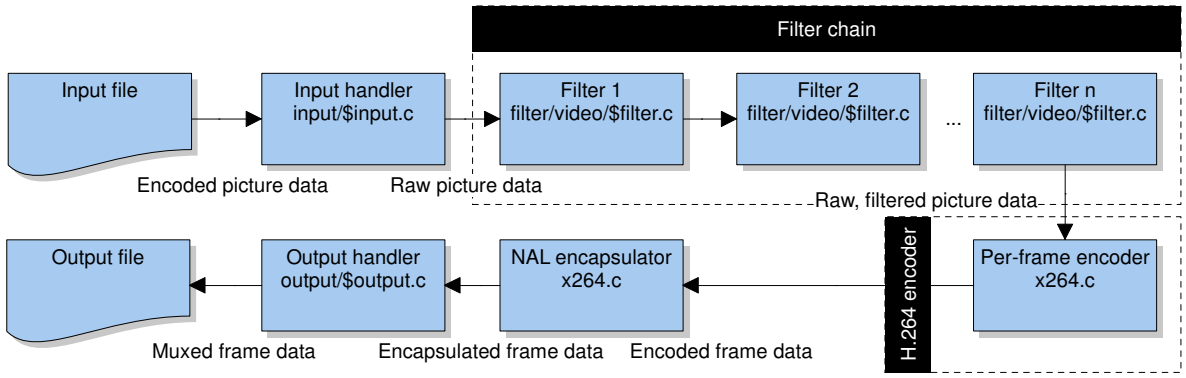

Figure 3.1.: Architectural overview of *x264*, depicting the processing chain from input and output and the main code file names corresponding to the tool chain items

opened by the corresponding input handler (various formats are supported as described in detail in [19]), decompressing the input picture by picture and handing the uncompressed picture data to a filter chain. Note that uncompressed input files are also supported. In this case, the input handler does not perform any operation but passing through the raw picture data picture by picture. This is assumed to be the default case herein as it does not require any additional libraries and introduces neither space overhead for decompression nor the need for dependency-related decompression buffers.
The optional filter chain allows preprocessing of the input pictures, e.g. cropping, resizing, noise filtering and more[19], all of which are optional. The output of filter $n-1$ is used as the input of filter n, thus forming a chain of filters. If m filters are applied, m intermediate, uncompressed pictures are required as filtering is not performed in-place. Furthermore, the picture size may change between two such representations, e.g. when a resizing filter is used. In a filter chain with m filters, the first one uses the decompressed input data from the input handler as input, whereas the m$^{\text{th}}$ filter's output is handed to the actual encoder.

x264's main task, i.e. the encoding of the filtered, uncompressed picture data into an H.264-compliant bit stream, is performed by the encoder which consists of several parts which are described below. Its output consists of single, encoded frames which correspond to the encoded input pictures, although their order may differ as described in subsection 2.3.2. Note that *x264* strictly distinguishes between pictures before and after encoding in terms of terminology – pictures denote pictures before encoding, whereas frames denote encoded pictures. As the output is written to a file, the frames have to be encapsulated into NAL units and preceded by meta-information like frame size, bit depth etc. Due to the fact that NAL unit escaping is enabled through a buffer which increases its size dynamically during runtime, it accounts for a total quasi-constant amount of allocated memory throughout program execution as described in subsection 2.3.7.
Finally, the encapsulated frame data is multiplexed (muxed for short) with audio and/or other data by the output handler to embed the NAL units into a specified output file format. Multiple such formats are available (for details see [19]), although it is also possible to pass through the NAL units without multiplexing which is the default case and results in a bit stream compliant to H.264 Annex B[7]. This requires no additional memory which necessitates inspecting the encoder in more detail.

3.1.2. The single-threaded encoder

x264's encoder can be separated into a so-called lookahead and the actual H.264-compliant encoder. The filtered, uncompressed input pictures are synchronously put into the lookahead for frame type decision, whereas the encoder subsequently compresses them one by one using an encoder structure similar to the H.264-typical one described in section 2.1. However, this synchronous approach which only requires one thread is only performed in the single-threaded mode, i.e. if *x264* is either executed on a single-core machine or the number of threads is deliberately limited to one through command line parameters[19].
The purpose of the lookahead is to reorder pictures based on the specified (or determined) B picture hierarchy and to determine frame types, detect scene cuts (which are also relevant for frame type decisions) and to perform a preliminary, so-called pre-motion-estimation on the input frames with reduced spatial resolution to estimate a temporary QP for each frame in order to preliminarily update the rate control algorithm to achieve the specified bit rate and/or quality goals. For scene cut detection and reordering B pictures

with a high number of hierarchy levels, the lookahead requires a buffer which allows it to "look into the future" to adapt frame types and QPs accordingly. Due to this, said buffer has to be filled up to a predefined minimum before encoding. While encoding, this leads to a delay between the pictures put into the lookahead and the ones obtained from it. This delay is variable due to B picture reordering as explained in subsection 2.3.2.
The actual encoder's architecture is similar to the one described in section 2.1, the differences not being described here as they are irrelevant in terms of MM. Although the output frames comply with the H.264 standard, *x264* only implements a subset of H.264's so-called coding tools. Although a detailed list of supported coding tools and restrictions can be found in [19], two restrictions are highlighted separately as they affect *x264*'s memory consumption directly. One the one hand, only 4:2:0 subsampled uncompressed picture data is supported, limiting ρ_{chroma} to $\frac{1}{4}$, and on the other hand, the maximum bit depth supported[3] is 16 with a default value of $n_{bits} = 8$. All other restrictions are not considered throughout the rest of this thesis as they don't have significant impact on the total memory consumption as described in subsection 2.3.9.

3.1.3. Multi-threaded encoding

Besides single-threaded encoding, *x264* is capable of two different multi-threaded encoding modes. On the one hand, each frame can be split into n slices which are encoded separately and merged after encoding, while on the other hand n frames can be processed simultaneously by n threads when obeying certain restrictions. Although both multi-threaded modes can be used (enabled or disabled by the command line parameters *–sliced-threads* and *–threads* respectively[19]), the latter of the two is enabled by default with $n = \lfloor 1.5 \cdot n_{CPU} \rfloor$ threads where n_{CPU} is the number of logical Central Processing Unit (CPU) cores available in the system which is detected during *x264*'s initialization. This default case is assumed in all descriptions related to multi-threaded encoding herein. As a frame depends on other frames to perform prediction (apart from IDR and I pictures), approaches for parallelization are not obvious. *x264* uses $n+2$ threads in total, separated into n encoding threads, one main thread for input and output and one lookahead thread as described below. Each encoding thread m of *x264* represents the encoder at the point

[3] Note that this refers to the internal bit depth used for coding which is not equal to the input bit depth. The latter is always $n_{bits} = 8$, although more recent versions of *x264* also support 16 bit input (see `http://git.videolan.org/gitweb.cgi?p=x264.git;a=commitdiff;h=284b3149b88f08c6ca324de05a92882c37fb1a44`)

in time when frame $x+m$ is being encoded where $0 \leq m \leq n-1$ and x denotes the frame number which the first thread is working on. In order to represent the different encoder states, each thread possesses its own reference lists which are generated by the main thread based on the reference list of the thread handling the previous frame.
The reference lists contain pointers to the actual frames, thereby neither requiring additional memory besides the pointers' space requirements nor copying of reference frames between threads. All threads access the same physical frames, although a number of them are not completely encoded as they are processed by other encoding threads. In order to avoid access to uncompleted regions of a frame during motion estimation, each frame contains a counter for the number of completed, i.e. encoded and decoded, lines, together with a mutex for synchronization[2]. Prior performing motion estimation for a predefined range on another frame, the encoding thread checks the number of completed lines of this frame and waits until all required lines are available. If the motion estimation range is small relative to the frame size (which it is for default settings and e.g. DVD input), this requires seldom waiting, thereby allowing for highly efficient parallelization[23].
Figure 3.2 gives an overview of the threads involved in multi-threaded encoding, assuming N encoding threads, which totals to $N + 2$ threads due to the main and the lookahead thread. The main thread reads pictures from the input file and puts them into the separate lookahead thread (which performs the same tasks as the lookahead described in subsection 3.1.2) using a synchronized lookahead input queue. As *x264*'s lookahead requires a minimum number of pictures in this queue and performs reordering on them if B pictures are used (as explained in subsection 2.3.2), a variable input delay (depending on the picture's position in the B picture hierarchy) is introduced between the main thread and the lookahead thread, denoted as d'.
Similarly, both the frame type decision including pre-motion-estimation for rate control purposes and the synchronized output queue introduce an additional delay, denoted as d. Therefore, the main thread retrieves frame i from the lookahead output queue after passing picture $i + d + d'$ to its input queue. The retrieved frame is then passed to a new encoding thread[4] which processes and deblocks it. Meanwhile, the main thread waits for the oldest encoding thread to finish, retrieving the encoded frame $i - N + 1$ and writing it to the output file. It then prepares the next input picture to be encoded for the lookahead step, continuing the encoding process.

[4] Actually, a thread pool is used which allows reusing the threads, improving performance due to a reduced number of creations and destructions of threads. However, this detail is omitted for the sake of readability

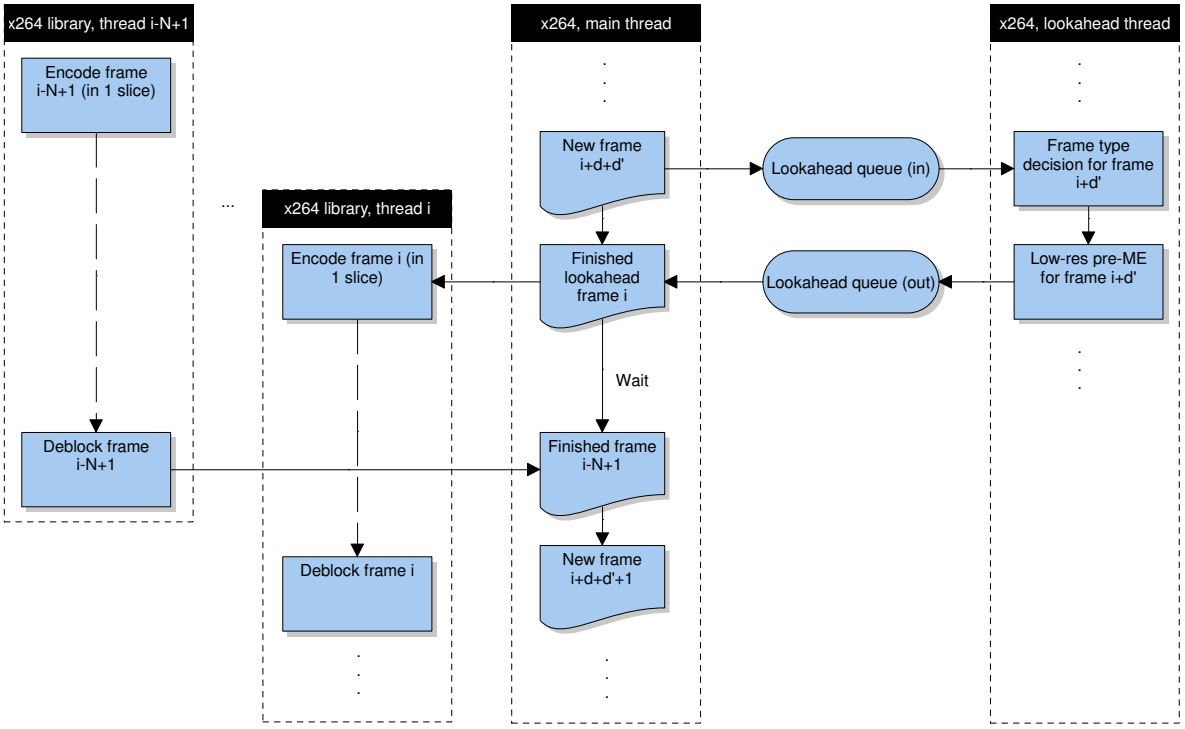

Figure 3.2.: Multi-threaded encoding in *x264* using $N+2$ threads, including lookahead and main thread. N denotes the number of encoding threads, d and d' denote the lookahead input and output delay, respectively

Note that the main thread's behavior slightly differs for the first and the last frames as the lookahead input queue has to reach its minimum fullness before encoding can start in the former case and the lookahead thread is no longer required as soon as the last frame is retrieved from its output queue in the latter case. These details are omitted for the sake of clarity. As can be seen, the multi-threaded encoding approach, together with the separate lookahead thread, introduces a significant delay which depends on N, d and d'. With N encoding threads, the frame lifetime relevant for MM is prolonged due to the frame's temporary residence in the lookahead buffers and not one, but N different DPBs simultaneously which have to be taken into consideration.

3.2. MM-related aspects

x264 relies on "classical" MM using *malloc* and *free* calls. Furthermore, it uses *memalign* to ensure aligned data structures for assembly language operations which require this and implements custom solutions for the latter if *malloc.h* (of *glibc*) is not available on the

target platform. This section gives an overview of *x264*'s main data structures and their MM, based on *x264*'s code. Details are omitted if they do not affect the explanations of the STM transformation in chapter 4.

3.2.1. Data structures

The main data structure in *x264* is a helper structure called *x264_t* (defined in *common/-common.h*), an instance of which is created during program initialization and passed to all functions which rely on its contents. Its main purpose is to store the encoding progress and related statistics (like the number of encoded frames and the cumulated PSNR) as well as to provide pointers to the lookahead and the frames currently encoded by the corresponding encoding threads, including the settings specified by the user using a *x264_param_t* structure. As there is only one instance of *x264_t* during the execution of *x264* which is destroyed when encoding is complete, it is of minor importance for the MM aspects relevant for this thesis and therefore not explained in more detail.
Two more important data structures are *x264_picture_t* (defined in *x264.h*) and *x264_-frame_t* (defined in *common/frame.h*) which allow storing pictures and frames, respectively. Pictures only contain their uncompressed data, represented by an *x264_image_t* structure which is not explained in detail, and meta information like time stamps and user-defined parameters which apply only to this picture and differ from the globally defined parameters. Pictures are allocated and freed by the the input handler, the filters and the main thread, but only account for a small part of the allocated memory in the default case when no filters are used.
Conversely, frames are used so frequently that a number of helper functions is available for their allocation, deallocation and management. *common/frame.c* defines the functions *x264_frame_new* and *x264_frame_delete* to simplify the allocation process which requires several allocations of structure elements depending on the frame type (see below). MM-relevant frame management functions are described below. Besides meta information like the frame number (in coding and display order), the frame stores uncompressed and compressed picture data, depending on the frame type, as well as its corresponding statistics. Additionally, each frame contains a mutex for motion estimation synchronization as explained in subsection 3.1.3.
Frames can be either so-called *fdec* or *fenc* frames (distinguished by a flag within the structure), which represent decoded, i.e. reconstructed, and encoded frames, respec-

tively. *fdec* frames are allocated by the main thread and made available to the encoding threads to store the data they are currently reconstructing as by-product of the encoding process. After being completely decoded by the encoding thread, the *fdec* frame is put into the reference list by the main thread, where it resides until it is no longer required. The uncompressed, decoded picture data is stored in the structure, together with meta information like the frame number (in coding and display order).
By contrast, *fenc* frames store the original picture data of the picture to be encoded, which is copied into the structure by the main thread before putting the frame into the lookahead after allocation. The lookahead performs its operations on the *fenc* frame and sets its frame type, preliminary QP and other parameters as required before the main thread passes the analyzed frame to the corresponding encoding thread. Additionally, each encoding thread's *fenc* frame stores the H.264-compliant encoded data which is encapsulated into NAL units by the main thread after encoding.
Besides the distinction between *fenc* and *fdec* frames, *x264*'s frame type decision in the lookahead adds another frame type dimension which relates to the H.264 picture types described in 2.2.1. Apart from the types described therein (IDR, I, P and B), *x264* distinguishes between B pictures which are kept as references and those which are not. The former are referred to as *B_REF* type frames, whereas the latter are named *B* frames. Whether B pictures are stored in the reference list (thus making them *B_REF* type frames) depends on the lookahead's frame type decision (e.g. when using hierarchical B pictures, pictures in the topmost hierarchy level are not used as references).
In addition to the frame types described, *x264* allows the use of so-called weighted P and B frames, the concept of which is described in detail in [8]. From an MM point of view, weighted frames are copies of other frames which are inserted into the reference list temporarily to allow encoding fades more efficiently. *x264* does not actually copy the frame structure, but creates a new structure containing a pointer to the original one, setting a flag so that such duplicate "dummy" frames can be distinguished from regular frames. For weighted frame management, *x264* provides allocation and deallocation functions which are based on regular frame allocation as described in subsection 3.2.2.
All frame-related MM operations take place in the main thread which also sets up the reference lists for the encoding threads. Depending on the size of the reference list and the user-specified parameters (like the DPB size and the number of frames used as references during motion estimation), *x264*'s main thread removes and reorders frames according to

the H.264 standard[5]. Furthermore, it uses MMCO commands to remove B frames from the reference lists which are not used as references (see above). *MMCO* commands are applied before building the reference list for the next frame to be encoded and can also be issued by a change in parameters (e.g. reduced DPB size).
In total, the required instances of all structures described above account for most of the memory consumption in *x264* during program execution as determined by a runtime memory usage analysis. As described in subsection 2.3.9, uncompressed picture data in the reference lists forms a significant part of the total amount of memory required. Additionally, n encoding threads require n *fdec* frames which also contain uncompressed picture data. This leads to the conclusion that the impact of all frame structure instances is significant, making frame management and lifetime an important aspect to consider for the transformation of *x264* to STM in chapter 4. As *x264* adds another abstraction layer for the MM of frames, referred to as pool allocation herein, this requires further inspection before describing the transformation.

3.2.2. Implemented frame pools

Due to the frequent allocation and use of frames, *x264* provides operations which facilitate their handling as explained in subsection 3.2.1. However, the frame allocation and deallocation functions *x264_frame_malloc* and *x264_frame_delete* mentioned therein are not called from the main thread directly, but wrapped by functions which allow reusing deallocated frames. This is referred to as pool allocation herein as *x264* maintains a number of allocated memory pools in the form of linked lists where it stores different types of deallocated frames which can be purged and returned on allocation requests, thus reducing the number of *malloc* and *free* calls, thereby reducing fragmentation for large picture sizes.
In total, *x264* maintains three pools: *unused[0]* for *fenc* frames, *unused[1]* for *fdec* frames and *blank_unused* for weighted P frames which are always *fenc* type frames as they are "dummies" which are only used for prediction in the reference list of the encoding thread they are used in. All three pools are located in a sub-structure of the global *x264_t* structure named *frames*. They are exclusively accessed by helper functions which wrap allocation and deallocation.

[5]Note that, during the writing of this thesis, a change in *x264*'s code has been made which reorders the frames in the reference lists subject to their distance from the frame to be encoded, which differs from the H.264 standard for "3-D", i.e. frame-packed, encoding (see `http://git.videolan.org/gitweb.cgi?p=x264.git;a=commitdiff;h=950de546c70874e174cc54f098551d05475caee9`)

Said helper functions are *x264_frame_pop_unused* and *x264_frame_push_unused* (both located in *common/frame.c*) which are used for allocating and deallocating regular, i.e. non-weighted, frames, respectively. A parameter distinguishes whether *fenc*- or *fdec*-type frames are (de)allocated. If the corresponding pool contains unused frames of the required type, it is purged, i.e. overwritten with zeros, and returned. If it does not, *x264_frame_new* is called, allocating a new frame and returning it. Weighted frames are allocated and deallocated by the functions *x264_frame_pop_blank_unused* and *x264_frame_push_blank_unused* (also located in *common/frame.c*), respectively. In contrast to the corresponding pool functions for regular frames, they only allocate and initialize an *fenc* type frame structure, marking it as duplicate of an existing, given frame.

As frames can appear in the reference lists of multiple threads, depending on the lists' development, an additional counter in the *x264_frame_t* structure is required to keep track of the number of references pointing to a frame. This is a simple form of reference counting which is performed by the pool (de)allocation functions and the main thread when synchronizing the reference lists. The pool allocation functions initialize the counter to one on allocation, while the main thread atomically recalculates the counters of all frames of the oldest and the newest thread's reference lists before handing a new frame to the newest encoding thread (compare subsection 3.1.3). Removing frames from the reference lists in the main thread can therefore use the pool deallocation functions, thereby atomically decrementing the counter. When decrementing yields a counter which is equal to zero, *x264_frame_push_unused* returns the frame to the pool for reuse.

As depicted in figure 3.3, the use of pool allocation prolongs a frame's lifetime as it is not deallocated when it is no longer required by any thread. By contrast, its reuse yields a quasi infinite lifetime which is only bounded by the number of input pictures determining the end of the program execution. Each time the frame is reused, it moves through the lookahead buffers (of which there are several, the details of which are omitted here), the so-called *current* buffer storing frames which are ready to encode and multiple threads' reference lists during encoding until it is finally placed in the frame-type-dependent *unused* buffer for reuse.

It has to be noted that the frame pool is deallocated at the end of the complete encoding process, i.e. before the termination of *x264*. As no further reuse is necessary at this point, frames do not need to be stored, but can be deallocated using *x264_frame_delete* and *free*. Similarly, no reuse is possible at the beginning of the encoding process as there are no unneeded frames available from previous allocations. In order to decrease the delay of

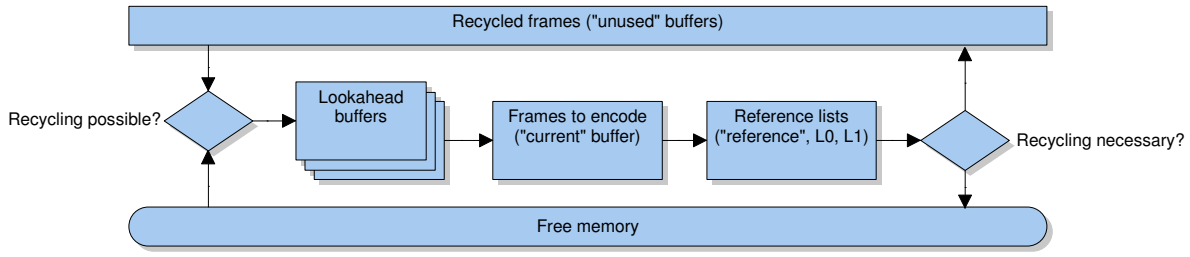

Figure 3.3.: Frame life cycle with pool allocation in *x264*

the first allocation calls via pool allocation before encoding, all three pools allocate a predefined number of frames of the corresponding types during program initialization, thus being able to provide allocated frames from the pool instead of issuing time-consuming allocation calls.

3.2.3. Encoder parameters

As the H.264 standard only specifies the decoding process and leaves a number of encoding decisions (like the coding modes of each macroblock and its partitions' reference frames used for motion compensation) to the encoder, *x264* offers various command line parameters which allow controlling these decisions. Besides H.264-specific parameters, the user has to specify a rate control mode and the desired rate (including rate restrictions) and/or quality. *x264* supports the specification of a constant QP (using the *–qp* parameter) as well as a target bit rate in kbits/s (using the *–bitrate* parameter), including the possibility to specify a minimum and maximum bit rate. Further rate control modes are available and described in [19]. The rate control mode and target bit rate influence the size of the encoded frames as well as the NAL output buffer size as described in subsection 2.3.8.
Besides the rate control configuration, *x264* requires the specification of input and output files (the former being the last, unnamed command line parameter and the latter being specified by the *–output* parameter), the format of which is determined by their file extensions. When using uncompressed input files in the YUV format, *x264* requires additional information about the picture size as the YUV format does not include header information by default[11]. The picture size can either be encoded in the input file name as described in [19] or specified explicitly using the *–input-res* parameter.
Besides the obligatory parameters described above, *x264* allows excluding modes (e.g. using the *–partitions* parameter to control the partitions and subpartitions of each mac-

Parameter	Description	Default value
–threads	Number of encoding threads	$\lfloor 1.5 \cdot n_{CPU} \rfloor$
–ref	DPB size	3
–weightp	Weighted prediction on (> 0) or off (0)	2
–no-weightb	No weighted prediction for B pictures	Not set
–bframes	Number of consecutive B pictures	3
–b-pyramid	Hierarchical (> 0) or successive B pictures (0)	0
–badapt	B picture decision: static (0) or input-based (> 0)	1
–sync-lookahead	Number of pictures used in the lookahead thread	*–bframes*$+1$
–scenecut	Threshold for I and IDR picture placement	40
–noscenecut	Disables scene cut detection	Not set
–minkeyint	Minimum IDR picture distance	See [19]
–keyint	Maximum IDR picture distance	250

Table 3.1.: Overview of *x264*'s MM-relevant encoder parameters based on [19]. n_{CPU} denotes the number of logical CPUs detected during initialization as described in subsection 3.1.3

roblock to be tested during encoding) which affects the encoding speed, as does e.g. limiting the range and accuracy of motion estimation using the *–me-range*, *–me* and *–subme* parameters. To facilitate the parameter specification, so-called presets (configured by the *–preset* parameter) can be used to specify a default parameter configuration to achieve a certain speed (implying quality degradations as explained in [17]), ranging from *ultrafast* to *veryslow*[6] with a default of *medium*.

Furthermore, the *–profile* and *–level* parameters allow limiting the coding tools and parameter values (like the DPB size) to the H.264 profile and level limits as described in subsection 2.3.9. A summary of further parameters which are relevant for memory management considerations is given in table 3.1. Note that this list focuses on MM-relevant parameters as described in section 2.2, including parameters which influence the picture types (decision). A complete parameter list can be found in [19].

[6] Actually, the slowest preset is called *placebo*, but its use is discouraged as it does not effect quality noticeably compared to the second-slowest preset *very-slow*[19]

3.3. Related work

Besides the references used in this chapter for the documentation of *x264*'s parameters and its multi-threaded architecture[19, 23], no literature describing *x264*'s general architecture is currently available. [9] is a partly outdated description of the rate control and motion estimation as well as the mode and frame type decision mechanisms, including an explanation of an algorithm commonly referred to as *Trellis quantization*[14] implemented in *x264*. Additionally, it contains benchmarks comparing a 2006-dated version of *x264* to the H.264 reference software available at `http://iphome.hhi.de/suehring/tml/`.
In addition, *x264*'s source code files come with a folder named "doc" which contains [23], parts of [9] and information about assumptions made in the C code regarding stack alignment and compiler-specific handling of negative numbers and sign-extending for shift operations which are both not guaranteed by the C99 standard[24]. Furthermore, it includes documentation with regards to code maintenance and a document describing the semantics of a so-called Supplemental Enhancement Information (SEI) message embedded in the output stream used for signaling the subsampling positions of the chrominance samples and other picture-related information (summarized as Video Usability Information (VUI), see Annex H of the H.264 standard[8]). The rest of all known documentation is limited to the comments within the code of *x264* itself.

4. Transformation of *x264*'s MM to STM

This chapter describes the modifications performed in *x264* in order to replace its "classical" MM with the SCM library version 0.4.1[1] described in section 1.2. As frames account for the major part of the total amount of memory consumed during run time, as described in subsection 3.2.1, only allocations and deallocations of them are modified to use STM with expiration extensions. All other allocated objects remain managed "classically", albeit through SCM's compatible interface as described in subsection 1.2.2.

4.1. Preparations for using SCM

The aforementioned compatibility to "classical" MM allows the C implementation of the SCM library to be linked against *x264* without code modifications if STM is not to be used. In order to prepare the usage of SCM in *x264*, the *Makefile*[2] has to be adapted to dynamically link the SCM library (abbreviated *libscm* from here on). Dynamic linking[26] has the advantage of loading *libscm* during run time which makes changes of e.g. the debugging level and recompilation of the library possible without the need to recompile *x264*. *x264*'s *Makefile* is designed for the use with the GNU Compiler Collection (GCC)[27], albeit generated by a custom *configure* script.
As *x264* uses a *configure* script which allows enabling and disabling support for features (e.g. additional input and format support on certain platforms) before compilation, no direct modification of the *Makefile* is necessary, but the *configure* script is adapted to make the use of *libscm* an option, too. In order to make this possible, support for a parameter *–enable-scm* has been added to the script, defining the necessary linking options[3] and *HAVE_SCM* for the preprocessor before compilation in order to be able to distinguish between pure "classical" MM and the use of *libscm* in *x264*'s code.

[1] Available at `http://tiptoe.cs.uni-salzburg.at/short-term-memory/`
[2] The syntax and semantics of *Makefiles* for *GNU make* is described in [25]
[3] *malloc*, *realloc* and *free* calls have to be handled by *libscm*, enabled through *-Wl,–wrap=malloc* etc.

In addition, two more options have been implemented for the *configure* script: on the one hand, *–scm-mode* allows choosing different *refreshing* approaches as described below, while, on the other hand, *–disable-pool* disables *x264*'s frame pool allocation mechanism, i.e. frames are allocated and deallocated directly. Experiments which have been performed during the making of this thesis with SCM and *x264*'s frame pool revealed that the vast majority of allocated frames have a lifetime which is equal to the total program run time, making the use of STM with explicit expiration extensions futile. Hence, the *–disable-pool* option has been introduced and used for all SCM-enabled configurations throughout the rest of this thesis. Implementing the use of *x264*'s pool as an option for the *configure* script allows re-enabling it for future experiments if desired. As all the aforementioned modifications in the *configure* script are solely made for convenience, they are not taken into consideration when calculating the total number of modified lines of code.

4.2. Changes for aligned allocation

As *x264* supports multiple platforms with different alignment behavior of *malloc* (or *memalign* if available), *x264_malloc* in *common/common.c* ensures 16-byte alignment if not provided by default on the platform *x264* is compiled for. Platform detection is performed in the *configure* script, defining corresponding preprocessor flags (e.g. *ARCH_X86_64* for 64-bit *x86*[28] platforms). Allocations in *x264_malloc* on platforms which cannot guarantee 16-byte alignment are extended in the following way by *x264* itself in order to do so nonetheless: instead of allocating n bytes, another 15 bytes plus the size required for an additional pointer (s_p) are added to the size to be allocated. $n' = n + 15 + s_p$ bytes of memory allow offsetting the start of the n used bytes by such a number of bytes that the first byte of the n ones used resides at an address which is a multiple of 16 with at least s_p bytes of space between the first allocated byte and the first byte used, as depicted in figure 4.1.

Thus, *x264_malloc* offsets the allocated pointer by said number of bytes before returning to ensure 16-byte alignment after allocating n' bytes. In order to regain access to the memory address originally returned by *malloc*, the s_p bytes before the address returned by *x264_malloc* are used to store a pointer to said address (i.e. a pointer to a pointer). When freeing a pointer allocated by *x264_malloc* in this way, the original pointer can be restored and passed to the *free* function of the underlying allocator in order to perform

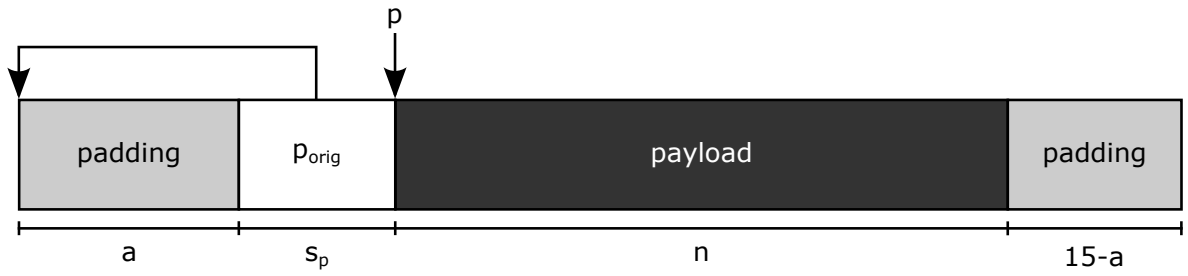

Figure 4.1.: *x264*'s padding for enforcing 16-byte alignment for the pointer p referring to n bytes of payload (dark gray). The originally allocated pointer p_{orig} is stored before the payload (white), the total of 15 padding bytes (light gray) are unused. a is chosen so that p's address modulo 16 is 0.

deallocation. The following listing illustrates the described algorithm for $n =$ `i_size` and $s_p =$ `sizeof(void **)`, i.e. the size of a pointer to a pointer, adopted from the original *x264* code of *x264_malloc* for platforms which do not support aligned allocation (platforms which support alignment don't require such treatment):

```
1     uint8_t *align_buf = NULL;
2     uint8_t *buf = malloc( i_size + 15 + sizeof(void **) );
3     if( buf )
4     {
5       align_buf = buf + 15 + sizeof(void **);
6       align_buf -= (intptr_t) align_buf & 15;
7       *( (void **) ( align_buf - sizeof(void **) ) ) = buf;
8     }
9     return align_buf;
```

libscm cannot guarantee alignment on every platform as the allocator it is based upon (compare section 1.2) cannot either[4]. Thus, if *x264* is configured with *–enable-scm* and compiled, *x264_malloc* has to force alignment regardless of the platform using the *HAVE_SCM* preprocessor flag for distinction as described above, resulting in no effective change in the total number of lines of code. This also affects the *x264_free* function as it always has to retrieve the pointer originally returned by *malloc* when using *libscm* functions. Accessing this pointer (referred to as p_{orig} in figure 4.1) is possible by following the pointer stored in the `sizeof(void **)` bytes before the pointer p returned by *x264_malloc*, as shown in the following listing taken from *x264*'s *x264_free* function for platforms which do not support aligned allocation:

```
1     free( *( ( ( void **) p ) - 1 ) );
```

Accessing the original pointer in this way is also necessary for other operations which have been introduced to perform STM operations, e.g. *refreshing*. As *libscm* is unaware of the aligned addresses used in *x264*, its functions require the use of the original addresses and the original pointers, respectively. Due to the need to *refresh* allocated objects and set finalizers, *common/common.c* and the corresponding header file *common/-common.h* are extended by the functions *x264_scm_refresh*, *x264_scm_global_refresh* and *x264_set_finalizer* as well as their respective prototypes, the former of which sharing the signatures of their pendants in *libscm*, but passing the original pointers instead of the aligned ones to the *libscm* functions as explained above. In total, they account for 15 lines of code, disregarding the preprocessor directives.

4.3. Changes in frame (de)allocation functions

As the MM of frames in *x264* is modified to use STM with explicit expiration extensions, various changes in the frame allocation and deallocation functions described in subsections 3.2.1 and 3.2.2 are necessary. Frames allocated by *x264_frame_new* are not explicitly freed, but expire when using STM, rendering *x264_frame_delete* obsolete, saving 42 lines of code. Therefore, *common/frame.c* and the corresponding header file are modified so that *x264_frame_delete* is only available if *HAVE_SCM* is not true, i.e. SCM are disabled using preprocessor directives[24] which are not counted as changes in the number of lines of code throughout the rest of this thesis.
Although *x264_frame_delete* is no longer needed, expired frames require their mutex objects to be freed explicitly as they are not managed by the memory allocator, but the Portable Operating System Interface for Unix (POSIX) thread library specified in [29] (section 2.9.3 "Thread Mutexes"). Freeing the mutex of a frame without having a *x264_frame_-delete* function requires the use of finalizers as described in subsection 1.2.2. Hence, on every frame *refresh* in *x264_frame_refresh* as described below, a finalizer which frees the mutex object of the created frame is registered, if *HAVE_SCM* is defined. As every frame created by *x264_frame_new* requires the same treatment, a single, file-local finalizer function is defined in *common/frame.c* and the corresponding header file, resulting in additional 9 lines of code. Whenever a frame expires, this function is called, freeing the

mutex and its underlying resources.

Despite the removal of *x264_frame_delete* when *HAVE_SCM* is defined, the use of STM entails the definition of a new function, called *x264_frame_refresh*, for convenience. As the *x264_frame_t* structure contains a significant number of fields pointing to the heap which are allocated in *x264_frame_new*, *refreshing* each field separately is error-prone. Thus, *x264_frame_refresh refreshes* all fields pointing to allocated STM with the same expiration extension, given by a parameter to *x264_frame_refresh*, using *x264_scm_refresh*. For *global refreshing*, the function *x264_global_refresh* provides the corresponding functionality, relying on *x264_scm_global_refresh*. Both functions, for *global* and *local refreshing*, respectively, account for additional 102 lines of code.

As all frames in *x264* are created and freed by the pool allocation functions *x264_frame_pop_unused* and *x264_frame_push_unused*, respectively, circumventing pool allocation when the *–disable-pool* option is selected as described in section 4.1 requires modifying the two aforementioned functions. Alternatively, all occurrences of the function calls in the code can be replaced by calls to *x264_frame_new* and *x264_frame_delete*, although this makes the code hard to read if both configurations, with and without pool allocation, are distinguished through preprocessor directives only. Thus, the behavior of *x264_frame_push_unused* and *x264_frame_pop_unused* is modified so that frames are allocated and freed directly if no pool allocation is desired, resulting in one additional line of code for each function. The same applies to *x264_frame_push_blank_unused* and *x264_frame_pop_blank_unused*.

If pool allocation is disabled, the *unused* and *blank_unused* arrays resembling the actual pools in the *x264_t* structure are removed by preprocessor directives as well, effectively saving two lines of code. However, the reference counting mechanism and the corresponding field in the *x264_frame_t* structure representing the number of references to this frame as described in subsection 3.2.2 must remain in order to avoid premature freeing of frames used by multiple threads. If *–disable-pool* is not specified when running *configure* before compiling, the code behaves like the original *x264* code with frame pool allocation and reference counting.

4.4. Frame lifetime management

In order to demonstrate different *refreshing* approaches, multiple variants of modifications have been implemented. The following subsections list and explain the required code changes for each modification, eventually showing the total required effort in terms of changed lines of code. The names of the modifications are chosen so that they are self-explanatory, although numbers have been assigned to them in the *configure* script for easier configuration (see section 4.1 for details concerning the *–scm-mode* parameter).

4.4.1. Continuous vs. single *refreshing*

The default approach for transforming an application to use STM is to find the periodic action with the largest period in the code and place a *tick* call at its end, preceded by *refresh* operations for all objects which are required in the next period and those which are no longer[3]. In the case of *x264*, this periodic action is the encoding of one frame. Therefore, a *tick* call is placed at the end of the *x264_encoder_frame_end* in *encoder/encoder.c*. This approach is used for two modifications, referred to as single *refreshing* and continuous *refreshing* in the main thread (using *local refreshing* as only the latter performs MM operations).

Continuous *refreshing* uses an expiration extension of 0 for *fenc* and all weighted frames in L0 as they are no longer required after the end of this function, and 1 for *fdec* and all frames in the DPB as they are required as references when encoding the next frame. This approach uses simple expiration extensions, thereby keeping those objects alive which are required until the next *tick* (to *refresh* them there again is necessary) and discarding those with an expiration extension of 0 which are no longer required.

Conversely, single *refreshing* uses a more complex expiration extension but *refreshes* objects only once during their lifetime, i.e. frames in the DPB are not *refreshed* as they are "old" *fdec*s. In this approach, *fenc* and all weighted frames in L0 are *refreshed* with an expiration extension of 0, as are all non-reference *fdec* B frames (which are not stored in the DPB), while *fdec*s of other frame types use an extension of $s_{DPB} \cdot (n_B + 1) + 1$ where s_{DPB} denotes the size of the DPB and n_B denotes the number of B frames.

The described formula for *fdec* expiration extensions can be split into two parts – the addition of 1 which is required as *fdec* is reused at the beginning of the next iteration to set up the environment for the next coding thread and the multiplication which describes the

Object	Exp. ext. (continuous)	Exp. ext. (single)
fenc	0*	0*
fdec	1	0 or $s_{DPB} \cdot (n_B + 1) + 1$
Weighted frames in L0	0	0
DPB frames	1	–

* *Refreshed* with 0 in the next iteration if equal to the last non-B frame in the lookahead

Table 4.1.: Expiration extensions (abbreviated "exp. ext.") for continuous and single *refreshing*

subsequent DPB lifetime of *fdec*. This lifetime can be derived as follows: if the size of the DPB is 0, *fdec* will not be stored in the DPB, therefore yielding a DPB lifetime of 0. With a non-zero DPB size and no B frames, the DPB exposes a FIFO-queue-like behavior, yielding a lifetime of s_{DPB} which equals $s_{DPB} \cdot (n_B + 1)$ for $n_B = 0$. For each increase in the number of B frames, the DPB lifetime increases linearly due to the intermediate encoding of non-reference B frames using the same DPB contents as the preceding and following IDR, I or P frames. Note that this yields a small over-approximation of the expiration extension for large n_B and small DPB sizes.

For the sake of simplicity, the lookahead thread is not involved in the MM and *refreshing* operations in both modifications, although it is necessary to consider that the lookahead thread stores a reference to the last non-B frame which it accesses. Therefore, *refreshing* of *fenc* as described above is only possible if it is not the last non-B frame of the lookahead. If it is, it has to be *refreshed* in the next iteration (or the one after it and so on) so that it does not expire before the lookahead thread replaces its last non-B frame. In order to avoid race conditions, a mutex is used to synchronize access to the last non-B frame of the lookahead (when comparing and *refreshing* and *fenc* in the main thread and replacing it in the lookahead thread).

A summary of all *refreshing* operations and their expiration extensions is given in table 4.1, while table 4.2 lists all required code changes in terms of Lines of Code (LOC). As described in sections 4.2 and 4.3, all modifications share $42+2$ deleted and $9+102+15+4\cdot1$ additional lines of code which totals to 174 changed lines of code.

4.4.2. *Refreshing* in coding threads

Although the straight-forward approach for *refreshing* and *ticking* only requires said operations in the main thread, it is also possible to relocate them to the coding threads (i.e.

Change	File	LOC (cont.)	LOC (single)
Common	See 4.2 and 4.3	174	174
Mutex management	*encoder/lookahead.c*	5	5
*Refreshing fenc**	*encoder/encoder.c*	18	18
Refreshing fdec	*encoder/encoder.c*	1	1
Refreshing weighted frames	*encoder/encoder.c*	3	3
Refreshing the DPB	*encoder/encoder.c*	2	0
Local tick	*encoder/encoder.c*	1	1
Sum	–	**204**	**202**

* Includes synchronization and *refreshing* in the next iteration if required

Table 4.2.: Code changes required for continuous (abbreviated "cont.") and single *refreshing*

to the end of *x264_slices_write*) without involving the main thread in *refreshing* or *ticking*. Both, continuous and single *refreshing*, can be implemented this way using similar expiration extensions to the ones described in subsection 4.4.1. However, *global refreshing* and *ticking* are used as multiple threads are involved in MM operations, sharing objects with different lifetimes for each thread.

The expiration extensions of the *refresh* operations for the frames in the DPB remain the same as for continuous *refreshing*. However, *fenc*, *fdec* and the weighted frames in L0 require an expiration extension of 1 as they are accessed by the main thread when writing *fenc* to the output file and preparing the reference lists for next coding thread. As *x264* uses a thread pool and the main thread waits for the oldest thread to finish before setting up the next coding thread, it is guaranteed that an expiration extension of 1 suffices to keep *fenc*, *fdec* and all weighted frames in L0 alive until they are not required anymore by the main thread or *refreshed* again by one of the coding threads, respectively.

Similarly, the expiration extensions of the single *refresh* implementation variants are adapted. Additionally, the formula for non-B *fdec* frames requires a major modification. As each coding thread represents the coding process at a given time instant, n coding threads represent n steps of the coding process simultaneously. Therefore, the DPB lifetime of each non-B *fdec* frame is reduced by the factor n and becomes $\lceil \frac{s_{DPB} \cdot (n_B+1)}{n} \rceil$. The use of the ceiling function in the formula is due to the fact that the nominator may not be divisible by n without remainder. Implemented in integer arithmetic, this avoids expensive floating point operations, thereby over-approximating the frame lifetime.

Note that both modifications, continuous and single *refreshing*, require a mutex for *fenc*

Object	Exp. ext. (continuous)	Exp. ext. (single)
fenc	1*	1*
fdec	1	1 or $\lceil \frac{s_{DPB} \cdot (n_B+1)}{n} \rceil + 2$
Weighted frames in L0	1	1
DPB frames	1	–

* *Refreshed* with 1 in the next iteration if equal to the last non-B frame in the lookahead

Table 4.3.: Expiration extensions (abbreviated "exp. ext.") for continuous and single *refreshing* in the coding threads

synchronization with the lookahead as explained in subsection 4.4.1. As only the expiration extensions differ, the number of changed of lines of code is the same as for continuous and single *refreshing* in the main thread. A summary of all *refreshing* operations and their expiration extensions in the coding threads is given in table 4.3.

4.4.3. *Local refreshing* in coding threads

In order to demonstrate that *local refreshing* in the coding threads is also possible and allows more fine-grain control over expiration extensions, one variant referred to as continuous *local refreshing* is implemented. Derived from continuous *global refreshing* in coding threads as described in subsection 4.4.2, the number of lines of code which need to be changed in total remain the same, although the expiration extension and the type of operations (*local* instead of *global*) differ.
While *fenc* and all weighted frames in L0 can be *refreshed* with 1 as they are only required by the main thread, *fdec* and all frames in the DPB require an extra expiration extension of 1 in order to give the other coding threads a chance to *refresh* the frames if needed. As *global refreshing* adds this expiration extension by default for all objects, *local refreshing* in the coding threads has the advantage of keeping the expiration extensions of *fenc* and all weighted frames in L0 lower. A summary of the expiration extensions for *local refreshing* in coding threads is given in table 4.4.

4.4.4. *Refreshing* in the lookahead thread

In order to avoid the use of a mutex for *refreshing fenc* in the implementation variants described above, another variant has been implemented which involves the lookahead thread in MM operations. It is loosely based on the continuous *refresh* approach described

Object	Expiration extension
fenc	1*
fdec	2
Weighted frames in L0	1
DPB frames	2

* *Refreshed* with 1 in the next iteration if equal to the last non-B frame in the lookahead

Table 4.4.: Expiration extensions for *local* continuous *refreshing* in the coding threads

above, although the main thread does not *tick* when a frame is encoded, but at the point where its reference list is set up completely, i.e. at the end of *x264_reference_build_list* in *encoder/encoder.c*. This also allows using different expiration extensions.
As the coding threads are not involved in MM, *fenc*, *fdec*, all weighted frames in L0 and all frames in the DPB are *refreshed* with the number of coding threads. This is due to the fact that the main thread waits for the oldest thread to finish before it starts a new one, thus limiting the lifetime of each frame before its next *refresh* or expiration. In addition to a *local tick*, all frames in the *current* buffer (which stores all frames which are ready to encode) are *refreshed* with an expiration extension of 1 as the main thread *ticks* only once before setting up the reference list for a new frame to be encoded, thereby *refreshing* all frames in the *current* again. The *refresh* operations in the lookahead thread, however, require a more detailed view of the different lookahead buffers in order to be able to explain the expiration extensions used.
Figure 4.2 depicts these buffers for both single- and multi-threaded configurations, i.e. without and with a lookahead thread, respectively. In the former case, frames are put directly into the *next* buffer where they are reordered and analyzed on the next *lookahead_get_frames* call, at the end of which they are moved to the *ofbuf* and finally the *current* buffer. *x264* always moves $n_B + 1$ frames from the *ofbuf* to the *current* buffer, including at least one non-B frame which leads to an update of the latter in the lookahead for future decisions. Hence, all moved frames are *refreshed* with n_B+2 where n_B denotes the number of B frames, ensuring that a reference to the last non-B frame is kept alive for at least the next $n_B + 1$ *ticks* of the main thread as *lookahead_get_frames* cannot move any frames to the *current* buffer if there are not at least $n_B + 1$ in the *next* buffer.
If there is a separate lookahead thread, the main thread puts the frames whose type is yet to be decided into the *ifbuf* lookahead buffer. In the lookahead thread, all frames in the *ifbuf* and *next* buffer are *refreshed* with $n_B + 2$ as in the case without a lookahead

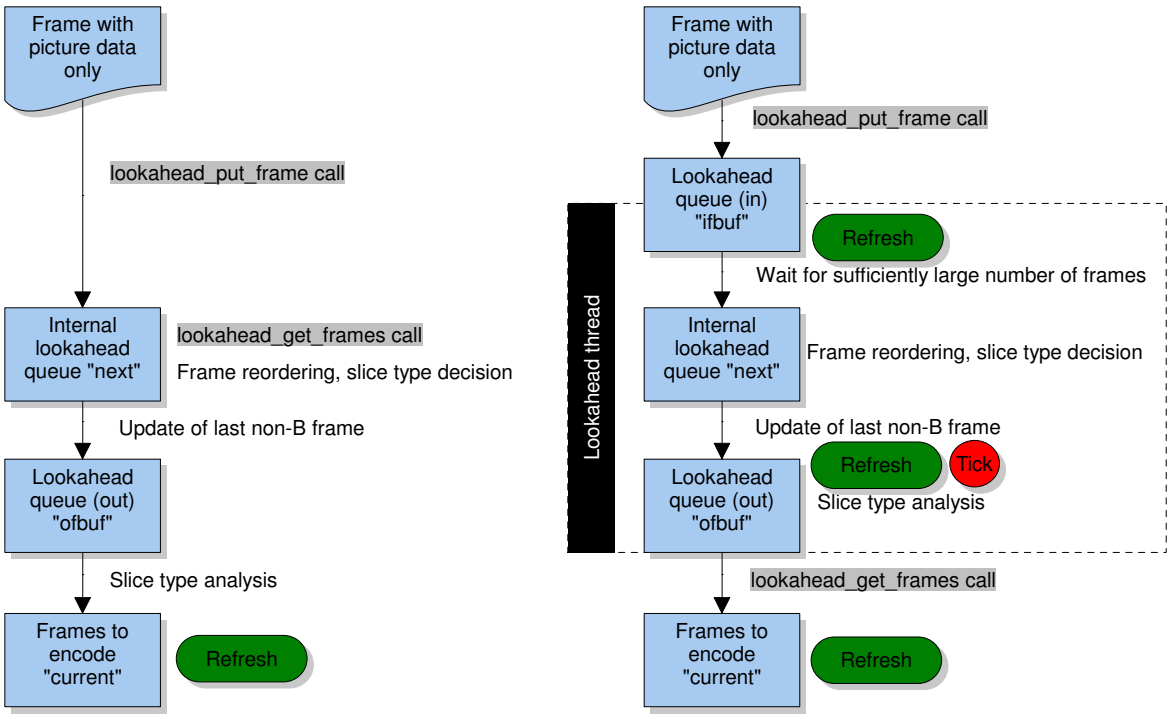

Figure 4.2.: *x264*'s lookahead buffers with (right) and without (left) a lookahead thread. Gray labels denote function calls from the main thread

thread described above. The frame reordering and slice type decision is then performed directly on the frames in the *next* buffer, followed by an update of the last non-B frame. Before *ticking*, all frames in *ofbuf* are *refreshed* with $n_B + 2$ once more. In order to let both the main and the lookahead thread *tick* at approximately the same speed, i.e. at the rate of processed frames, the lookahead thread *ticks* $n_B + 1$ times for each of its $n_B + 1$ analyzed frames after moving them to the *ofbuf* buffer from which they are fetched by the main thread using the *lookahead_get_frames* call. The latter moves the analyzed $n_B + 1$ frames to the *current* buffer and *refreshes* them with the number of coding threads plus one to ensure that they are kept alive until they can be *refreshed* by the main thread when setting up the reference lists as described above.

Table 4.5 summarizes all expiration extensions used in this implementation variant, involving both, the main and the lookahead thread, in MM operations while table 4.6 lists the required code changes in terms of lines of code. Note that it is possible to combine the existing code with additional *refreshing* in the coding threads. However, this remains future work due to the number and complexity of the described implementation variants.

Object	Expiration extension
All frames in *ifbuf*	$n_B + 2$ (with lookahead thread)
All frames in *next*	$n_B + 2$ (with lookahead thread)
All frames in *ofbuf*	$n_B + 2$ (with lookahead thread)
Frames moved from *ofbuf* to *current*	$n_B + 2$ or n
All frames in *current*	1
fenc	n
fdec	n
Weighted frames in L0	n
DPB frames	n

Table 4.5.: Expiration extensions for continuous *refreshing* including the lookahead thread. n denotes the number of coding threads, n_B the number of B frames. Lookahead- and main-thread-related objects are separated by a horizontal line

Change	File	LOC
Common	See 4.2 and 4.3	174
Refreshing all frames in *ifbuf*	*encoder/lookahead.c*	2
Refreshing all frames in *next*	*encoder/lookahead.c*	2
Refreshing all frames in *ofbuf*	*encoder/lookahead.c*	3
Local tick (lookahead)	*encoder/lookahead.c*	3
Refreshing frames moved from *ofbuf* to *current*	*encoder/lookahead.c*	1
Refreshing fenc	*encoder/encoder.c*	1
Refreshing fdec	*encoder/encoder.c*	1
Refreshing weighted frames	*encoder/encoder.c*	3
Refreshing DPB	*encoder/encoder.c*	2
Local tick (main)	*encoder/encoder.c*	1
Sum	–	**193**

Table 4.6.: Code changes required for continuous *refreshing* including the lookahead thread. Lookahead- and main-thread-related changes are separated by a horizontal line

Component	Component name
CPU	4x AMD Opteron Processor 8425 HE (6 cores each, 2.1 GHz)
RAM	48 GB
Operating system	Custom based on Linux kernel 2.6.39-rc5
C compiler	gcc 4.4.3
*yasm**	0.8.0.2194

* Required for *x264* compilation in order to use assembly code optimizations

Table 4.7.: Machine configuration used for validation

4.5. Validation of the applied changes

In order to validate all implemented variants of the described code changes, a validation script has been developed. As the changes focus on frames and their lifetime, all parameters influencing the former significantly are considered in this script, i.e. the size of the DPB and the number of B frames. Additionally, the number of threads is considered to evaluate both single-threaded and multi-threaded use cases on the one hand, and to reveal possible race conditions for large numbers of threads on the other hand.
Using selected values for the number of threads (see below) and all possible combinations of values of the number of DPB sizes and B frames, the different modified versions of *x264* are executed sequentially using one parameter combination after the other. As the parameters influence the coding process and therefore the output files, the latter cannot be compared to a reference output, entailing successful program execution combined with manually evaluated control samples as a validation criterion.
In order to reveal possible race conditions, a machine with a high number of physical CPU cores is used to run the validation script. Table 4.7 lists the relevant hardware configuration of said machine which offers 24 CPU cores and a significantly large amount of Random Access Memory (RAM) to encode the test sequence *foreman*[4] which is commonly used in video coding for performance tests[12, 13, 21].
Due to the high number of CPU cores and the need to include single-threaded configurations, 1, 4, 16 and 64 threads are used for testing. Whereas the latter value is purely included to reveal race conditions, the remaining two – 4 and 16 – are chosen for different reasons. While 16 represents the maximum DPB size, the value of 4 enables multi-threaded processing without using any of the values used in the subsequent benchmarks,

[4] `http://www.cipr.rpi.edu/resource/sequences/sif.html` (4:2:0-subsampled YUV sequence)

Component	Component name
CPU	2x Intel Xeon E5530 (4 cores each, 2.4 GHz)
RAM	8 GB
Operating system	Yellow Dog Linux (kernel 2.6.18-164.ydl.3)
C compiler	gcc 4.1.2 20080704
yasm	1.1.0.2352

Table 4.8.: Machine configuration used for cross-validation

thus showing that a small number of threads also allows for stable operation.
Aside from the number of threads, both, the DPB size and the number of B frames are varied between their minimum and maximum values of 0 and 16 with a step size of 1, yielding a total number of $17 \cdot 17 \cdot 4 = 1156$ parameter combinations which are used for validating each variant of the described code changes. At the time of this writing (June 2011), all of the latter have been validated successfully, including a cross-validation on a different machine (whose configuration is listed in table 4.8) for the purpose of reproducibility.

4.6. Related work

MM-related modifications of H.264 encoders in general are quasi non-existent in the literature which is most likely due to the fact that the main aim of encoder modifications is either an improvement in speed or video quality[12]. The former goal is driven by the complexity of the H.264 standard as described in [21, 22], yielding efforts to reduce said complexity without impacting quality (e.g. [30, 31]), while the latter goal allows for eventual bit rate savings without perceptual quality loss, enabling a reduction of costs by a reduction of transmission bandwidth[14]. Although the number of H.264-related quality improvement suggestions throughout the literature is large, most of them use the H.264 reference software (e.g. [32]) or other, proprietary encoders (e.g. [17]) for performance evaluation.
Modifications of *x264* in terms of the described goals are also available, such as [33] which changes *x264*'s RDO algorithms to use SSIM as distortion metric instead of PSNR, albeit limited to I picture coding only. [34] modifies *x264*'s parameters in order to reduce encoding time without significantly impacting rate-distortion performance. A similar approach described in [35] reduces the RDO complexity when encoding videos depicting sign language. More proposed modifications of *x264* are available throughout literature (see [9] for the ones officially introduced into *x264*), although none of them are MM-related.

5. Performance analysis

In order to evaluate the performance of the described code changes using *libscm*, both time- and space-related (i.e. memory-consumption-related) benchmarks are carried out. This chapter gives an overview of the benchmark setup and the benchmarking-related code changes in both *libscm* and *x264*. Subsequently, different benchmarks are shown and compared to one another, concluding with final remarks on the observed differences.

5.1. Benchmark setup

The benchmarks described herein are separated into time- and space-related measurements. This is due to the fact that the measurement of memory consumption entails output to a file or *stdout* which could change the execution time, thus requiring separate configurations with no output for time-related benchmarks. The implementation details concerning this distinction between time- and space-related benchmarks can be found in section 5.2.

Despite the changes required for benchmarking, the following changes are made to the default configuration of *libscm* version 0.4.1. First, the maximum expiration extension is increased to 300 which is the number of frames of the input file used for benchmarking (see below). This is due to the single *refresh* approach which requires expiration extensions of up to $16 \cdot (16+1)+1 = 273$ when using a DPB size of 16 and 16 B frames. As no approach uses expiration extensions which are higher than the number of frames in the input file, a maximum expiration extension of 300 ensures that no frame expires too early due to the expiration extension limit.

In addition, the descriptor page size is set to 32. As most STM-allocated objects in the benchmarks are frames being decoded or in the DPB, i.e. *fdec* type frames, whose creation via *x264_frame_new* additionally allocates $12 + (2 + n_B)^2$ of the *x264_frame_t*'s fields as STM objects, this makes it more likely for a descriptor page to contain a frame

and its fields which all expire at the same time. For the default setting of $n_B = 3$, this yields a total of 38 allocations for each frame, including the *x264_frame_t* frame itself, making 32 as the next-lowest power of two a more use-case-optimized choice than the default value of 4096.

Finally, *libscm* is configured to perform eager collection. In contrast to lazy collection which collects one expired descriptor page per *tick* as described in subsection 1.2.1, eager collection collects all expired descriptors, requiring more time for collection, but simultaneously reducing the number of expired descriptor pages causing management overhead. In the benchmarks described in all subsequent sections besides the ones dealing with lazy collection, eager collection is used.

Similarly, all benchmarks use the same input files and *x264* command line parameters as described below, unless noted otherwise. As input file, the *foreman* sequence mentioned in section 4.5 is used. Constant Rate Factor (CRF) mode is chosen as rate control method with a CRF value of 16 as it is the default method used by *x264*. The choice of 16 is arbitrary (although valid, of course) and does not influence the memory consumption of *x264* significantly as described in subsection 2.3.8. As the H.264-compliant output file is not required for the benchmarks, its path is set to */dev/null* in order to minimize its impact on the benchmarks. All other settings remain at their default values, except for the number of threads which is described subsequently for each benchmark.

In order to minimize measurement errors and inaccuracies, all benchmarks are executed as follows. For the purpose of cache warming, each execution of *x264* for the purpose of space-related benchmarking is preceded by three "blank" runs which are not taken into account for the actual measurements. This also ensures that the lookahead thread (in multi-threaded configurations) is not Input/Output (I/O)-bound and reads faster than the coding threads can process the frames to be encoded, thus keeping memory consumption quasi equal among multiple executions.

As opposed to space-related benchmarks, time-related benchmarks are executed five times for each configuration after three "blank" runs in order to compensate for the measurement inaccuracies of the *time* utility[36] which is used to measure *x264*'s execution time. The sequence of three "blank" and five actual runs is repeated four times for each configuration, yielding 20 time measurements in total for each configuration which allows a conclusion about the actual range of execution time(s). All benchmarks, i.e., both, time- and space-related, are executed on the machine whose configuration is listed in table 4.7, using the benchmark scripts described in appendix B.

5.2. Code changes required for benchmarking

In order to be able to distinguish between time- and space-related benchmarks, a parameter named *–enable-benchmark* is introduced in the *configure* script which is by default assumed to be absent, i.e. disabled. In this state, no further code changes besides the ones described in chapter 4 take effect, thus allowing benchmarking the execution time of the final *x264* executable using an external tool. If, however, the *–enable-benchmark* parameter is enabled, a function call to *print_memory_consumption* (see below) is added at the end of the *x264_encoder_frame_end* function in *encoder/encoder.c* to write the current memory consumption to *stdout* for the purpose of evaluation. Although this changes the program's execution time, this has no effect on the execution time benchmarks as the latter are run with *–enable-benchmark* disabled.

As explained above, time-related benchmarks don't require additional changes, but space-related benchmarks print the memory consumption after a frame has been processed completely for the sake of comparability among different implementation variants. Although *libscm* allows printing the memory consumption on each allocation, deallocation, *refresh* and *tick* operation using a flag in its *Makefile*, this is not suitable for the desired benchmarks due to the high amount of allocations and *refresh* calls. Therefore, the code of *libscm* is modified so that the actual *print_memory_consumption* function can be called directly from *x264* through linking. In addition, all other calls of *print_memory_consumption* from within *libscm* itself are removed to reduce the output to the desired minimum.

Note that this change entails the creation of two different versions of *libscm* – one without the described changes which is linked to *x264* executables compiled with *–enable-benchmark* disabled, and one including the described changes which is linked to the *x264* executables used for space-benchmarking. For convenience, the *configure* script is adapted to link the appropriate variant of *libscm* automatically, depending on the presence of the *–enable-benchmark* parameter. In order to do so, it relies on two compiled versions of *libscm* – the unmodified version with the original name, and the modified version named *libscm_x264_scm_bench*. Both can be created automatically by the scripts described in annex B.

In order to measure the memory consumption of the original *x264* executable and the implementation variant without pool allocation, the *print_memory_consumption* function of *libscm* cannot be used as linking *libscm* to the corresponding executables would falsify the benchmark results. For these scenarios, a modified version of a wrapper library originally

developed by Martin Aigner and Andreas Haas is linked to the *x264* executables instead of *libscm* (including the required changes in the *configure* script). The purpose of this library is to wrap all MM calls (i.e. *malloc*, *free*, *calloc*, *realloc* and *memalign*) like *libscm* and to provide an atomic "allocated memory" counter which tracks the amount of currently allocated memory based on the allocation and deallocation requests. Additionally, a function called *print_memory_consumption* is provided which prints the counter in the same format as *libscm*'s *print_memory_consumption* function, enabling the output of the current memory consumption.
Due to inconsistencies during the benchmarks relating to the memory consumption output of *mallinfo*[1] for the purpose of comparison, an additional line of code at the beginning of *x264*'s *main* function in *x264.c* has been introduced in order to prevent the underlying allocator from using *mmap* to serve allocation requests. This is done by setting the *M_MMAP_THRESHOLD* to 0 using the *mallopt* function[2]. Note that this does not change the memory consumption of *x264* as this is independent of the actual source of free memory provided by the allocator. Like all other changes described in this section, the number of lines of code which need to be added or modified are not taken into consideration for the total number of changed lines of code as they are for the purpose of benchmarking only and not required for regular operation.

5.3. Space-related results

Measuring the memory consumption of the different implementation variants allows drawing conclusions on their efficiency and establishing a relation between the aforementioned efficiency and the complexity of the code changes for each of them. The following section presents the memory-consumption-related benchmark results whose setup is described in section 5.1. Note that this section gives an overview of the results, whereas a detailed list of all results can be found in the log files on the attached Compact Disc (CD) as described in appendix B.

[1] `http://www.gnu.org/s/hello/manual/libc/Statistics-of-Malloc.html`
[2] `http://www.gnu.org/s/hello/manual/libc/Malloc-Tunable-Parameters.html`

Name	Abbreviation	Implementation variant
original	orig.	Unmodified version of *x264*
no pool	nopool	*x264* without pool allocation
continuous refresh	cont. refr.	Coding threads *refresh locally*
continuous refresh global	cont. refr. global	Coding threads *refresh globally*
continuous refresh main	cont. refr. main	Main thread *refreshes locally*
full continuous refresh	full cont. refr.	Main & lookahead thread *refresh locally*
single refresh	single refr.	Coding threads *refresh locally* once
single refresh main	single refr. main	Main thread *refreshes globally* once

Table 5.1.: List of names and abbreviations of the implementation variants described in section 4.4

5.3.1. Overview of all implementation variants

Henceforth, the implementation variants are given the names and abbreviations listed in table 5.1 for the sake of readability. Figure 5.1 gives an overview of each implementation variant's net memory consumption over time with one thread. Note that the time axis measures *ticks*, i.e. finished frames, which are associated with actual time in a non-linear way. In all implementation variants, including the unmodified *x264*, memory consumption increases quasi linearly at least until the fifth *tick* which is due to the filling of the lookahead buffers described in subsection 3.1.2. Similarly, memory consumption decreases quasi linearly from the 258th *tick* onwards as the buffers become continuously emptier as the end of the input file is reached. This is true for all implementation variants but the original *x264* whose memory consumption increases monotonically. This is due to the frame pool implementation's strategy not to return additionally allocated frames to the underlying allocator, but to keep them for possible future use. Increases in the original *x264*'s net memory consumption therefore only occur if the frame pool cannot satisfy the current allocation request.

These increases can also be observed in the "no pool" implementation variant at the same points in time. However, the memory consumption of this implementation variant also decreases due to frame deallocations, the reasons of which are twofold: on the one hand, a DPB flush forced by an IDR picture, like e.g. at the 250th *tick*, deallocates all frames in the DPB simultaneously, while on the other hand, the lookahead's decision to temporarily decrease the actual number of B pictures (up to the maximum value specified) shortens the lifetime of the other B frames in the DPB, thus lowering the memory consumption.

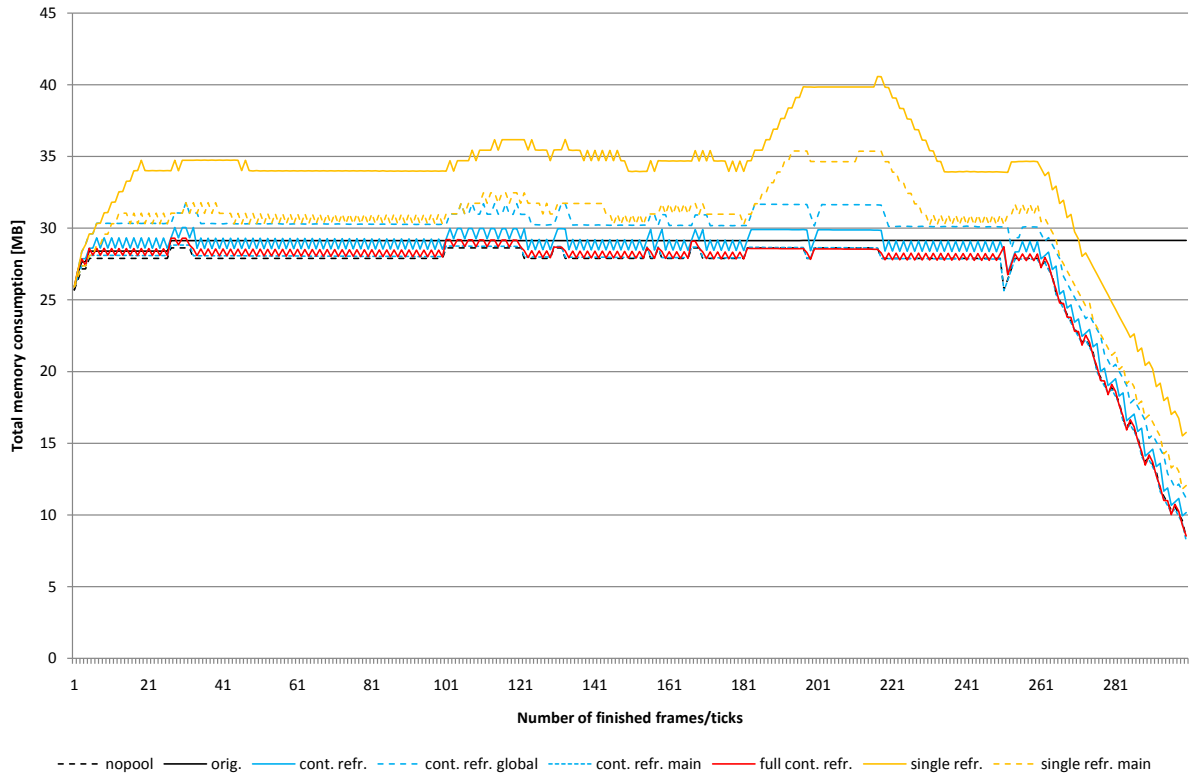

Figure 5.1.: Net memory consumption for all implementation variants with one thread

Conversely, an increase in the actual number of B pictures is reflected by an increase in memory consumption as can be seen e.g. at the 100th *tick*.
This also applies to the "full continuous refresh" and the "continuous refresh main" implementation variants which are nearly congruent with the "no pool" implementation variant. The same applies to the "continuous refresh" implementation variant in terms of the memory consumption curve's shape, albeit offset by about 1.3 MB on average. This is due to the longer expiration extensions for *fenc* and all weighted frames in L0 compared to the "continuous refresh main" implementation variant. As to be expected, the "continuous refresh global" implementation variant exhibits a yet higher memory consumption than the "continuous refresh" implementation variant as *global refreshing* effectively makes the expiration extensions one unit, i.e. *tick*, longer than *local refreshing*. This also explains the visible shift of increases and decreases in memory consumption by one *tick* compared to the "continuous refresh" implementation variant.
While all approaches involving *continuous refreshing* but the one using *global refreshing* are comparable to the "no pool" implementation variant, those approaches which use *single refreshing* exhibit a significantly higher net memory consumption. This is due to

the fact that the expiration extensions are over-approximated for most frames as they are based on the maximum number of allowed B pictures rather than the number of actual ones and the lookahead's frame type decision algorithm tends to choose a lower number of B pictures than the maximum number allowed. As the expiration extension cannot consider future decisions of the frame type decision algorithm, it always has to assume the maximum number of B pictures allowed.

With the P picture life times being over-approximated, too, the series of P pictures between the 181st and the 216th *tick* (which is only "interrupted" by one single B picture in its middle) yields a visible increase in memory consumption up to the point where the P pictures allocated before the start of this period expire. The subsequent decrease in memory consumption is due to the constant use of B pictures after the 126th *tick*. This decrease is shifted in the "single global refresh" implementation variant which, similar to the "continuous global refresh" implementation variant, exhibits delayed deallocations by one *tick* due to the additional expiration extension addition of one.

Changing the number of threads from one to 24, i.e. the number of physical CPU cores of the machine the benchmarks are performed on (see section 5.1), the difference between the *continuous refreshing* approaches and the *single refreshing* approaches remains, as shown in figure 5.2. In contrast to the results with one thread, the difference between the "no pool" implementation variant and those implementation variants which *refresh* in the main thread is significantly larger. This is due to the fact that each thread's *fdec* and all frames in its DPB are *refreshed* with an expiration extension of one, including the ones which are not required anymore, thereby extending their lifetime up to at least the next *tick*. As this applies to all threads' frames, this over-approximation of the frame lifetimes occurs continuously, thus adding up to a memory consumption offset which is proportional to the number of threads. Due to the high number of threads, this offset is greater than or equal to 50% at all times.

Those implementation variants which *refresh* frames in the coding threads, exhibit yet a higher memory consumption. This is due to the fact that the lifetimes of the frames to be *refreshed* are over-approximated to a higher extent as they are not only proportional to the number of threads (from the main thread's point of view), but both, *fenc* and *fdec*, have longer expiration extensions than in the aforementioned implementation variants as they have to stay alive until the main thread has written them to the output file or inserted them into another thread's DPB, respectively. Similar to the benchmark results with one thread described above, all implementation variants which use *global refreshing* show an

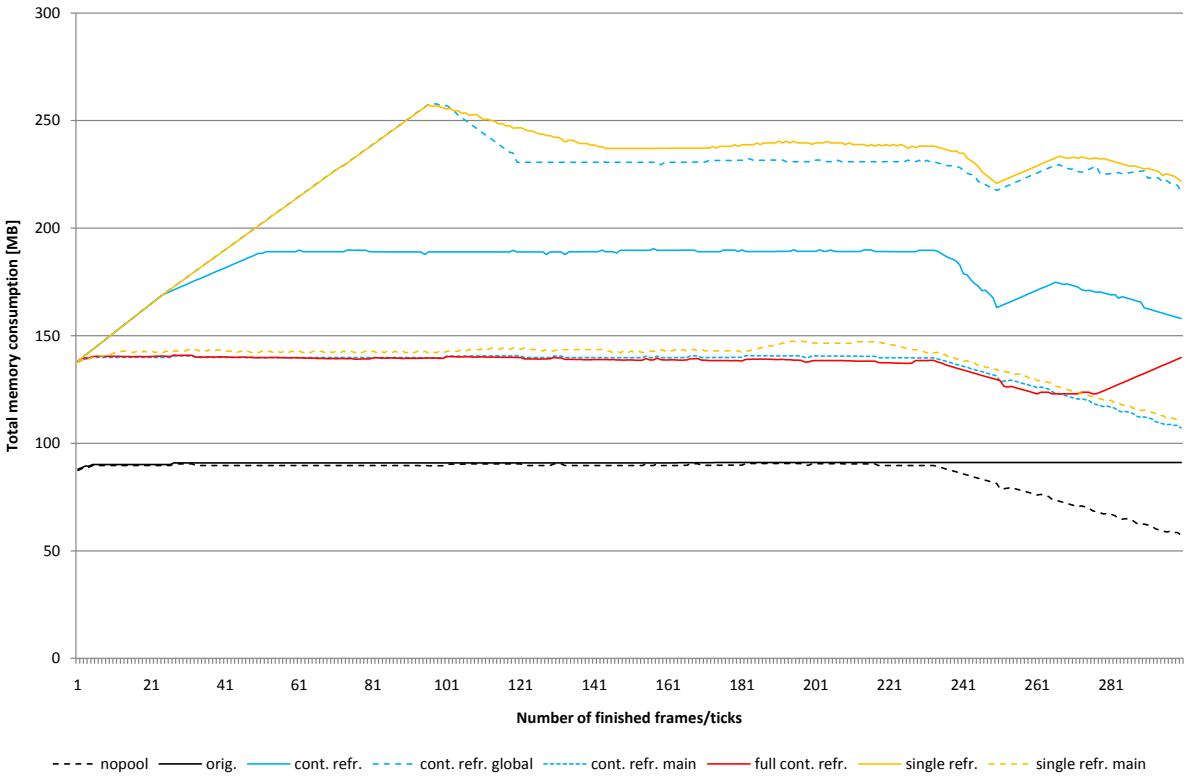

Figure 5.2.: Net memory consumption for all implementation variants with 24 threads

additional offset and the deallocation-related shift, both of which are clearly visible in figure 5.2.

In conclusion, the benchmark results with 24 threads show that no implementation variant is as efficient as the unmodified *x264*, but the "full continuous refresh" implementation variant exhibits the smallest difference to it in terms of net memory consumption. Although the "continuous refresh main" implementation variant which uses simpler expiration extensions and a smaller total number of *refresh* operations shows a comparable performance in the benchmarks with one thread, the "full continuous refresh" implementation variant is chosen to represent the "best" of all SCM-based options for showing the best performance in the benchmark with 24 threads. The fact that the default configuration of *x264* uses multiple threads underlines the importance of this aspect, making the "full continuous refresh" implementation variant a suitable representative to analyze the influence of coding parameters and management overhead in the subsequent subsections.

5.3.2. Influence of parameters

As the "full continuous refresh" implementation variant caused the lowest memory consumption of all implementation variants, the influence of the two parameters which influence the total memory consumption of *x264* most, i.e. *–ref* and *–bframes*, is analyzed. Figure 5.3 shows the maximum net memory consumption with one thread for all possible values of the two parameters, indicating a linear increase of memory consumption for the *–ref* parameter and an exponential increase for the *–bframes* parameter. The plateaus for high values of *–bframes* and low values of *–ref* originate from the fact that the DPB size limits the actual number of B pictures that can be used, which is checked by *x264* and adapted by it automatically.

The unmodified *x264*'s net memory consumption (not depicted) shows a similar depen-

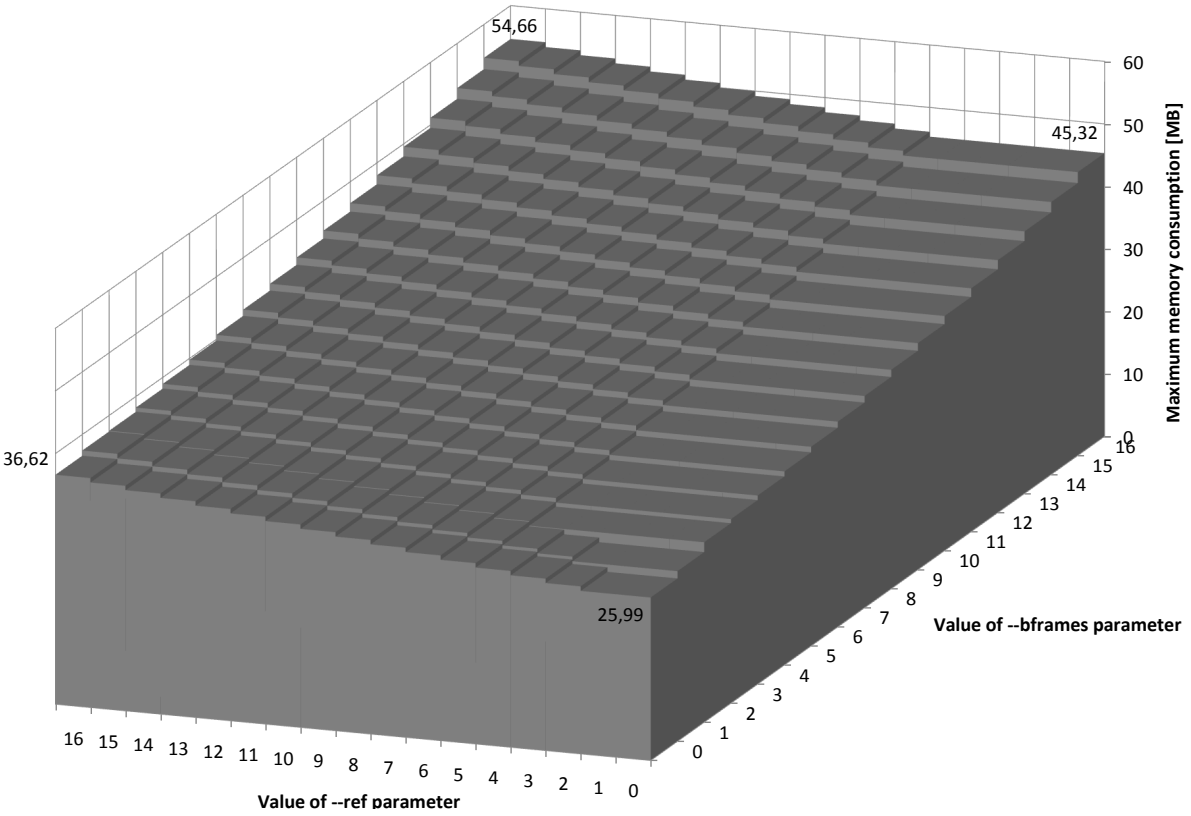

Figure 5.3.: Influence of *x264*'s parameters *–ref* and *–bframes* on the maximum net memory consumption with one thread in the "full continuous refresh" implementation variant

dency on the two parameters, although its offset is different. With 25.15 MB for a DPB size of 0 and no B pictures, a difference of about 840 KB, i.e. less than 3.5% of the net

memory consumption of the unmodified *x264*, marks the point of the minimum difference between the two implementation variants. For a DPB size of 16, the memory consumption becomes 35.78 MB which also yields a total difference of 840 KB.
In contrast, a DPB size of 0 and 16 B pictures leads to a memory consumption of 44 MB and thus a total difference of 1.32 MB, yielding a steeper slope for the B-frame-dependent memory consumption. Note that this represents a relative difference of 3% which indicates that a higher DPB size makes the "full continuous refresh" implementation variant more efficient, considering that a DPB size of 0 yields a relative overhead of less than 3.5%. For the maximum DPB size and number of B pictures, the unmodified *x264* consumes 53.15 MB of memory, while the "full continuous refresh" implementation variant requires 54.66. The total difference of 1.51 MB yields a relative difference which is smaller than 3%, indicating that, overall, higher values of *–ref* and *–bframes* make the "full continuous refresh" implementation variant slightly more efficient in terms of relative net memory consumption overhead.

5.3.3. Lazy vs. eager collection

As the decision to use eager collection in all benchmarks influences the results of the latter, the effect of switching to lazy collection for the default configurations with one and 24 threads is analyzed in order to assess the difference between the two forms of collection. Note that this section focuses on memory consumption only, whereas an analysis of the influence of lazy collection on execution time can be found in subsection 5.4.3.
Figure 5.4 shows the net memory consumption of all implementation variants using lazy collection with one thread. Its similarity to figure 5.1 is clearly visible, although the absolute and relative differences in memory consumption are highly dependent on the implementation variant. While the unmodified *x264* and the "no pool" implementation variant show no difference at all due to the fact that they do not use *libscm*, all other implementation variants are offset and shifted compared to the "no pool" implementation variant due to the later deallocation of expired objects. This is most clearly visible for the "continuous refresh main" implementation variant whose memory consumption is about 1.333 MB higher in average compared to the "no pool" implementation variant, but otherwise congruent with the latter when using eager collection.
The "full continuous refresh" implementation variant shows a comparable performance difference, albeit higher (1.918 MB), thereby making the "single refresh main" implementa-

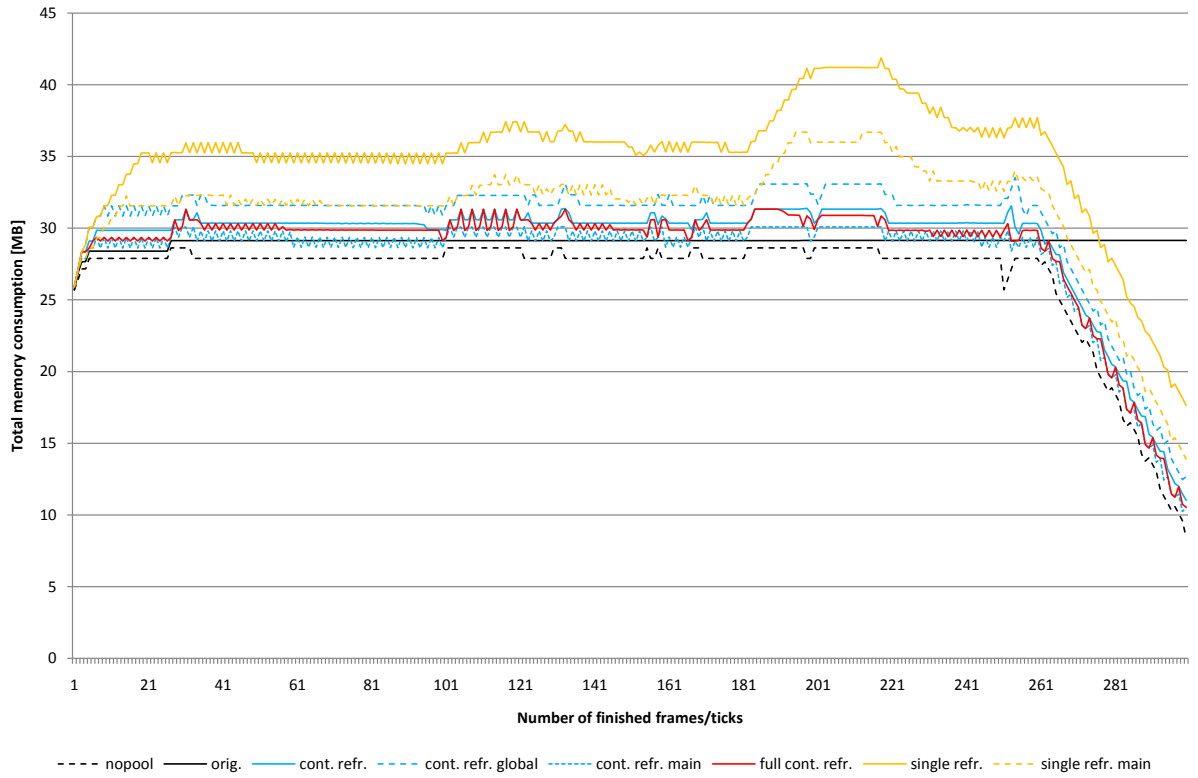

Figure 5.4.: Net memory consumption for all implementation variants using lazy collection with one thread

tion variant the best in terms of memory consumption with one thread using lazy collection. This is due to the higher number of *refreshes* of the "full continuous refresh" implementation variant which increases the number of expired descriptor pages to be collected, thus increasing memory consumption. Similarly, the "continuous refresh" and the "continuous refresh global" implementation variants show a yet higher offset in addition to a stronger shift which is due to the use of longer expiration extensions. Likewise, those implementation variants which use *single refreshing* exhibit a yet stronger shift and offset which is 4.705 MB for the "continuous refresh main" implementation variant and 8.304 MB for the "continuous refresh" implementation variant, respectively.

With 24 threads, lazy collection shows similar effects, as illustrated in figure 5.2. When compared to eager collection with 24 threads (see figure 5.2), the shift is yet significantly stronger than it is with one thread. This is due to the increased number of descriptors for those implementation variants which *refresh* in the coding threads and is most clearly visible when comparing the points at which the memory consumption stabilizes after the initial increase due to the continuous deallocation of expired objects. While this takes 95

ticks in the "single refresh" implementation variant when using eager collection (see figure 5.2), it takes 119 *ticks* when using lazy collection and requires more than 33 MB of additional memory.

Although these numbers suggest that lazy collection should always be avoided in favor

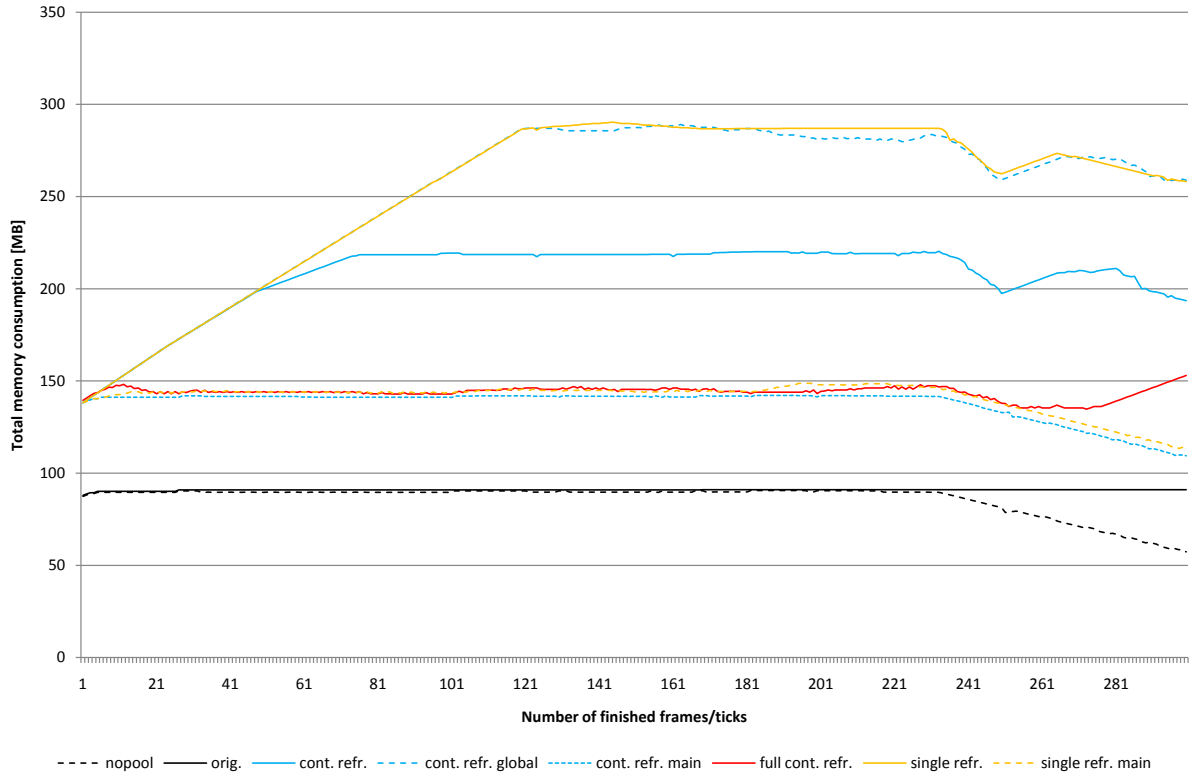

Figure 5.5.: Net memory consumption for all implementation variants using lazy collection with 24 threads

of eager collection, one has to consider possible overheads in execution time due to the additional time required to collect all expired descriptor pages. This aspect is discussed in subsection 5.4.3, taking into consideration that different implementation variants entail a different number of expired descriptor pages to collect. Additionally, fragmentation may increase when using eager instead of lazy collection, the reasons and consequences of which are discussed in the subsequent subsection.

5.3.4. Management overhead and fragmentation

In order to assess the management overhead introduced by *libscm*, various configurations are analyzed, the first of which being the unmodified *x264* linked to *libscm* with its compat-

ibility with "classical" MM. Due to *x264*'s frame pool, the number of allocated bytes and, therefore, the overhead monitored by the library itself, is quasi constant. With one thread, it totals to a maximum of 108.064 KB which accounts for less than 4% of the total memory consumption. With 24 threads, the overhead increases to 219.376 KB or less than 2.5% which differs from the one-thread configuration. The reason for this difference is the fact that more frames are allocated at any point in time as there are more threads concurrently processing them, each with their own DPB, which extends the overall lifetime of all frames and thus increases the number of allocated frames.

A different configuration to be analyzed in terms of management overhead is the implementation variant with the lowest memory consumption, i.e. the "full continuous refresh" implementation variant as shown in subsection 5.3.1. Using eager collection with one thread, the maximum management overhead is 147.672 KB which is higher than the management overhead of the unmodified *x264* linked to *libscm*. The reason for this increase lies in the use of multiple different expiration extensions as opposed to the configuration compatible with "classical" MM which does not use explicit expiration extensions at all. Lazy collection yields an increase to 155.224 KB which is yet higher, but can be explained by the slightly increased memory consumption due to the increased number of expired descriptor pages which are not collected.

With 24 threads, the management overhead significantly increases in the "full continuous refresh" implementation variant for both, eager and lazy collection. Despite the fact that the memory consumption and the number of descriptor pages are higher, two threads, i.e. the main and the lookahead thread, are involved in MM, thus practically doubling the number of descriptors. This explains the maximum (rounded) management overhead of 1.093 MB for eager and 1.929 MB for lazy collection. Although the absolute values are significantly higher than they are with one thread, they only account for less than 0.8% and 1.3% of the maximum total memory consumption, respectively. This shows that, on the one hand, lazy collection increases the management overhead due to the number of uncollected expired descriptor pages, while, on the other hand, the management overhead's portion of the total memory consumption decreases relatively to the total memory consumption when the latter increases.

Finally, the difference between net and total memory consumption, i.e. the degree of memory fragmentation, is analyzed. While *libscm* keeps track of the net memory consumption, the gross value must be obtained from the underlying allocator which returns the total amount of system memory allocated by itself in the *uordblks* field of the *mallinfo*

Implementation variant	Max. fragmentation (eager)	Max. fragmentation (lazy)
original	0.10%	0.10%
no pool	0.10%	0.10%
continuous refresh	2.80%	0.12%
continuous refresh global	2.61%	0.13%
continuous refresh main	3.70%	0.11%
full continuous refresh	5.15%	0.13%
single refresh	0.87%	0.12%
single refresh main	1.11%	0.11%

Table 5.2.: Maximum memory fragmentation (rounded to two decimal places of the percent values) for all implementation variants with one thread

structure (see section 5.2). As this field is not thread-safe, table 5.2 only lists the maximum fragmentation, i.e. the relative difference between the net and the gross memory consumption, of all implementation variants with one thread.
As can be seen, the use of eager collection leads to a significant increase in fragmentation, while the increase is minimal when using lazy collection. This is due to the fact that less descriptor pages can be reused in the former case, requiring the continuous allocation of new descriptor pages over time. As descriptor pages are relatively small in comparison to the allocated frames and their fields, fragmentation increases over time.
Besides the difference between eager and lazy collection, the maximum fragmentation between different implementation variants diverges. In general, a high total number of *refreshes* yields higher fragmentation, as can be seen in all implementation variants using continuous *refreshing* when compared to those implementation variants which use single *refreshing*. This is due to the fact that multiple *refreshes* add descriptors when using SCM (see section 1.2), thereby increasing the number of pages to be collected over time which leads to higher fragmentation as described above. Thus, the "full continuous refresh" implementation variant exhibits the highest maximum fragmentation.

5.3.5. Reducing memory consumption in multi-threaded scenarios

In order to show that the net memory consumption in multi-threaded scenarios can be reduced further when using STM, an additional implementation variant is provided. It is based on the "continuous refresh main" implementation variant (see subsection 4.4.1) and uses the same *refresh* and *tick* positions and expiration extensions, therefore behaving ex-

actly the same as the "continuous refresh main" implementation variant in single-threaded scenarios. In multi-threaded scenarios it attemps to additionally *refresh* those *fdec* frames in *x264_reference_update* in *encoder/encoder.c* with 0 which are allocated before the first frame is returned from the lookahead thread. These frames get unreachable before being *refreshed* by the main thread later, thus increasing net memory consumption.

As the moment in which the first lookahead-processed frame becomes available depends on the speed of the lookahead thread, this approach over-approximates the number of *fdec* frames to be *refreshed*. More precisely, using ascending coding thread numbers for the sake of illustration, it *refreshes* the *fdec* frame created for thread i with 0 when setting up the reference lists for thread $i + d$, where n is the number of coding threads. d is the smallest number so that both, thread $(i - n)$'s and thread $(i + d - n)$'s *fdec*s, have not been kept in the DPB. Thread numbers which are smaller than zero denote preallocated frames for each thread.

This approach has been developed empirically and fails the validation irreproducibly. This is due to the fact that, in some configurations, thread i's *fdec* is actually used by one of the coding threads after being *refreshed* with 0. As soon as the first thread finishes its frame, the main thread *ticks*, making thread i's *fdec* a dangling pointer due to expiration, leading to an error when thread i attempts to reconstruct its frame. Although n and d influence the probability of this happening, no consistently failing configuration could be found. Therefore, this implementation variant is named "*unstable* continuous refresh main", abbreviated "unstable cont. refr. main".

However, the default parameter values described in section 5.1 allow for stable and reproducible results. As the single-threaded results are exactly the same as the ones for the "continuous refresh main" implementation variant, only the net memory consumption in a multi-threaded scenario with 24 threads depicted in figure 5.6 is described. It is obvious that an overhead compared to the original *x264* implementation still exists, albeit smaller than the overhead of the "continuous refresh main" implementation variant. While the latter exhibits a peak memory consumption overhead of more than 54%, the unstable approach reduces this overhead to less than 19%.

This shows that the reduction of memory consumption in multi-threaded scenarios with STM is possible in principle. As the "unstable continuous refresh main" implementation variant does not work for all parameter combinations and the subset of working parameter combinations cannot be determined exactly, this implementation variant is not used for further testing. Due to this and the implementation variant's similarity to the "continuous

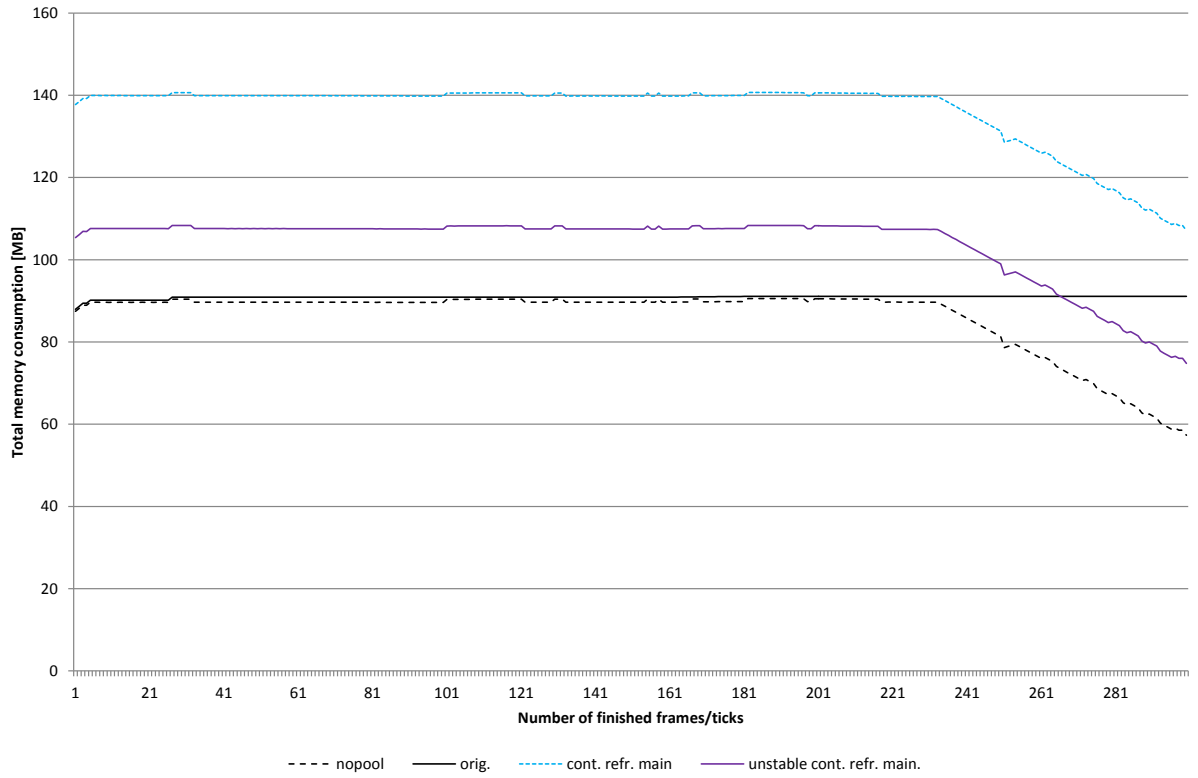

Figure 5.6.: Net memory consumption for the "unstable continuous refresh" and reference implementation variants with 24 threads

refresh main" implementation variant, no overview of the expiration extensions and the number of changed lines of code is given. Note that, however, the code of all implementation variants is located on the attached CD as described in appendix B.

5.4. Time-related results

Similar to the memory-consumption-related results described in section 5.3, this section presents the results of the execution time measurements whose setup is described in section 5.1. Again, the location of all detailed results is denoted in appendix B.

5.4.1. Overview of all implementation variants

A comparison of the execution time of all implementation variants with one thread is depicted in figure 5.7. The median execution time of the unmodified *x264* is 9.515 seconds, with the maximum time measured being 50 ms larger than the median and the minimum

time being 18 ms smaller. Relating the minimum and maximum to the median value shows that both deviate by less than 0.53%. This also applies to all other implementation variants apart from "continuous refresh main" and "single refresh main" whose minimum and maximum values don't deviate by more than 0.56 and 0.62%, respectively. Repeated measurements showed that the deviations are almost constant, i.e. they fluctuate behind the second decimal place after the decimal point of the deviations in per cent.
The median execution time of the unmodified *x264* is notably larger than the execution

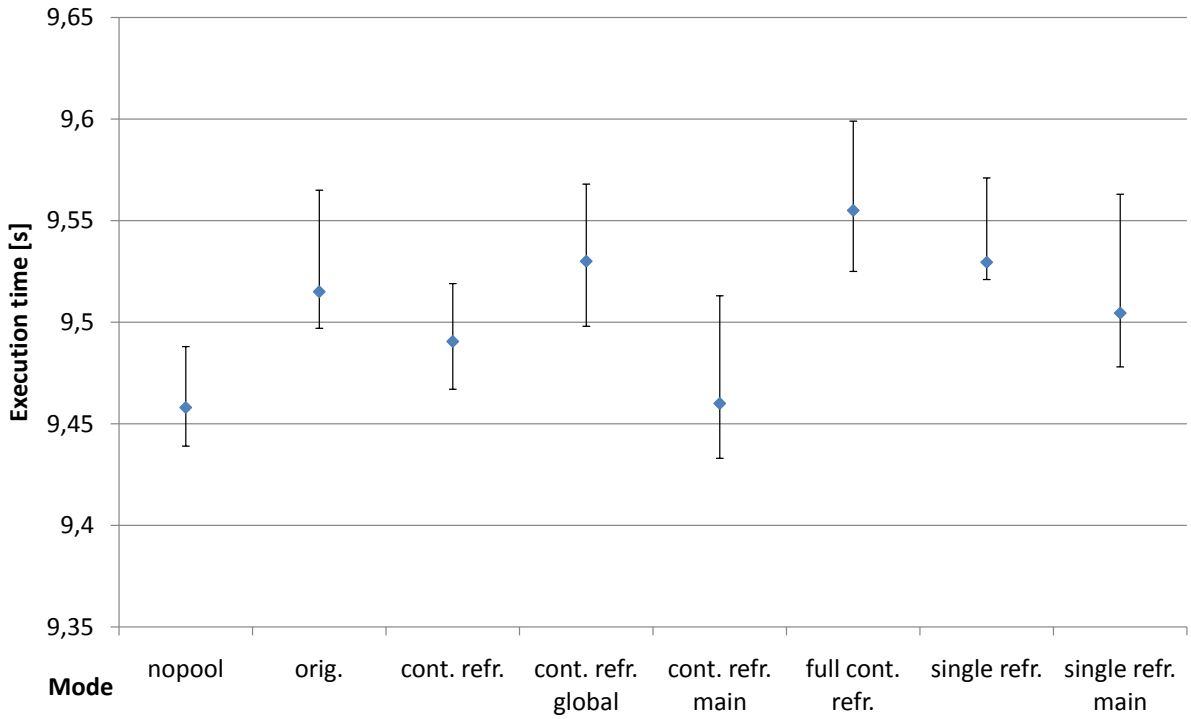

Figure 5.7.: Execution time of all implementation variants with one thread. The rhombs denote the median execution time of each implementation variant, the horizontal bars indicate the minimum and maximum execution time, respectively

time of the "no pool" configuration which is due to the overhead of pool management which entails iterating through lists of frames on every push or pop operation, the duration of the iteration being proportional to the size of the lists. The implementation variant "continuous refresh global" exhibits quasi the same range of execution times as the unmodified *x264*, although its median is more "centered", i.e. the minimum and and maximum measurements have nearly the same distance to the median. Similarly, the implementation variants "single refresh" and "single refresh main" can be interpreted as having the same or a lower median execution time.

The "continuous refreshing" and "continuous refreshing main" implementation variants exhibit execution time ranges which lie between the ones of the unmodified *x264* and the "no pool" implementation variant, allowing for the conclusion that no implementation variant is slower than the unmodified *x264* but the "full continuous refresh" implementation variant. As the latter extensively *refreshes* frames in multiple locations in both the lookahead and the main thread, this is to be expected. Nonetheless, the maximum execution time is only 52 ms larger than the maximum execution time of the unmodified *x264*, representing a relative difference of 0.55%. From a total execution time point of view, the median values of the two implementation variants differ by less than 40 ms or 0.45%.
Note that all SCM-based implementation variants are based on the "no pool" implementation variant, exhibiting their MM-related overhead relative to the total execution time of the former. While the "continuous refresh main" implementation variant introduces only a minimal overhead, the "continuous refresh" implementation variant shows that the additional MM operations in the coding threads increase the median execution time visibly. Global *refreshing* yet increases the execution time due to thread-local and thread-global time management as can be seen in the "continuous refresh global" variant. The median execution time of the "single refresh" and "single refresh main" implementation variants is similar to the one of the "continuous refresh" variant which indicates that longer expiration extensions are as expensive as multiple *refreshes* in this scenario.
In contrast to most of the measurements with one thread, the benchmarks with 24 threads, i.e. the number of physical CPU cores of the machine the benchmark is executed on (see section 5.1), show the influence of MM in multiple threads. As depicted in figure 5.8, all implementation variants in which only the main thread performs MM operations, i.e. the "no pool", the "original", the "continuous refresh main" and the "single refresh main" implementation variants, show a quasi indistinguishable execution time. As there is a separate lookahead thread which processes the frames faster than the coding threads do when using 24 threads, most pool management operations don't influence the execution time anymore, thus making the execution time of the aforementioned implementation variants indistinguishable. Like in the benchmark results with one thread described above, the "continuous refresh" and the "single refresh" implementation variants have a similar execution time range and median. This, again, shows that the single *refreshing* approach with longer expiration extensions yields about the same execution time as the continuous *refreshing* approach with shorter expiration extensions, but a higher number of *refresh* operations. The "continuous refresh global" implementation variant is yet more expensive

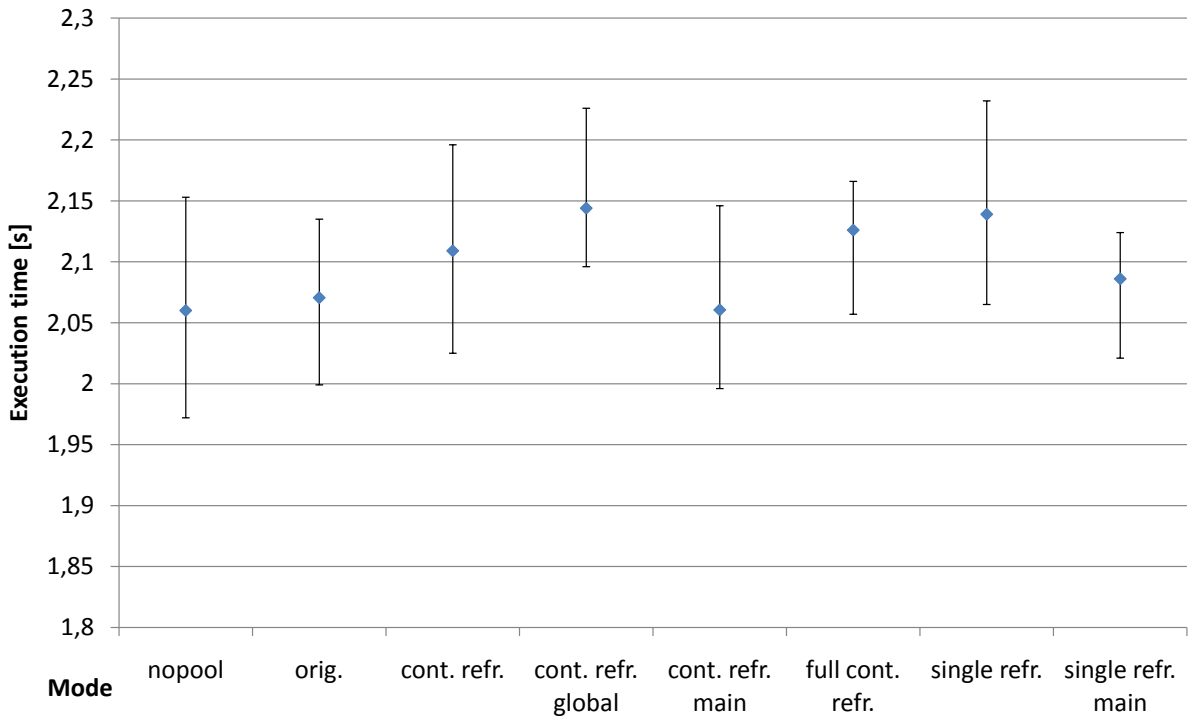

Figure 5.8.: Execution time of all implementation variants with 24 threads. The rhombs denote the median execution time of each implementation variant, the horizontal bars indicate the minimum and maximum execution time, respectively

in terms of execution time, with the reasons being the same as in the benchmark with one thread described above. It is notable, however, that the "full continuous refresh" implementation variant is not the most expensive one when using 24 threads, as only two of them – the main and the lookahead thread – perform MM operations as opposed to those implementation variants in which all 24 coding threads and the main thread are involved in MM.

5.4.2. Influence of parameters

As in section 5.3.2, the influence of the parameters *–ref* and *–bframes* on the median execution time of the implementation variant with the lowest net memory consumption, i.e. the "full continuous refresh" implementation variant, is analyzed. The absolute and relative measurement errors for the execution time benchmarks in this implementation variant are described in subsection 5.4.1.

Figure 5.9 depicts the median execution time with one thread case for all possible parameter values of *–ref* and *–bframes*. It is clearly visible that the number of B pictures

changes the execution time in a barely notable way (less than 24 ms for every increase in the number of B pictures), except for the case where the number of B pictures is 0. This value yields a significant increase in *x264*'s execution time compared to the configurations where the number of B pictures is not 0. Nonetheless, this behavior does not require a more detailed inspection as the unmodified *x264* (results not depicted) exhibits the same behavior, thus showing that this phenomenon is unrelated to the MM implementation and therefore exceeds the scope of this thesis. The same applies to the influence of the *–ref* parameter which is quasi linear, taking the values of 1 and 2 for *–ref* into consideration which deviate from this tendency.

The difference between the unmodified *x264* and the "full continuous refresh" implemen-

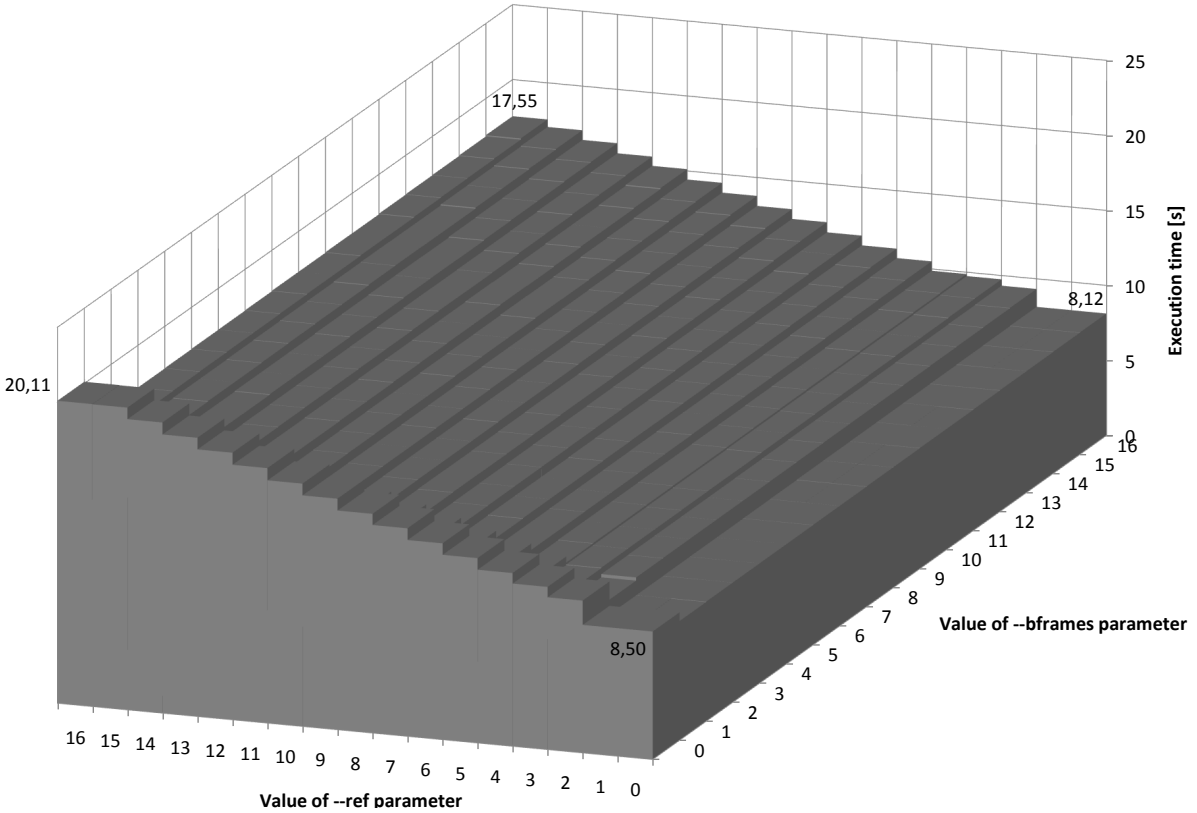

Figure 5.9.: Influence of *x264*'s parameters *–ref* and *–bframes* on the median execution time with one thread in the "full continuous refresh" implementation variant

tation variant in terms of execution time is within the measurement error for a DPB size of 0 and 0 B pictures, but increases with both parameters. While the unmodified *x264* shows an increase of less than 2.4 ms for every increase in the number of B pictures, the "full continuous refresh" implementation variant does so with an increase which is 10 times as

large (see above). This is due to the increased number of frames to be reordered and *refreshed* in the lookahead thread which depends on the number of B pictures in a linear way.
Although the DPB size influences the number of *refresh* operations, this is not reflected in the measurements. An increase in the value of *–ref* has the same consequences for both, the unmodified *x264* and the "full continuous refresh" implementation variant, taking the effect of B pictures on the execution time into consideration which is dominant. The largest difference between the execution time of both implementation variants can be measured with a DPB size of 16 and 16 B pictures. While it takes 17.13 seconds to execute the unmodified *x264*, it takes 17.55 seconds to do so for the "full continuous refresh" implementation variant, yielding a relative difference of 2.5%.

5.4.3. Lazy vs. eager collection

When using lazy instead of eager collection, in general, memory consumption decreases at the expense of a higher execution time[3]. However, as shown in table 5.3, lazy collection does not have a measurable effect on *x264*'s execution time in most implementation variants – neither with one, nor with 24 threads. The unmodified *x264* and the "no pool" implementation variant show a deviation of up to 8 ms with one thread, although they should not be affected at all by a change in *libscm* since the two implementation variants do not use *libscm*. Considering that this entails a measurement error which is at least as high as the aforementioned deviation, all speedups which are within the range $[-8; 8]$ are more likely to represent noise than actual speedup values. Repeated execution of the benchmarks supports this assumption.
Thus, lazy collection has no measurable effect on execution time compared to eager collection in this *x264* benchmark for all implementation variants but the "continuous refresh main", the "single refresh" and the "single refresh main" implementation variants. As the latter three show complementary results although the number of *refreshes* and their respective expiration extensions are equal in both implementation variants with one thread, the obtained values are most likely to be measurement errors, too, thus allowing concluding that no implementation variant exhibits a measurable difference in execution time between lazy and eager collection with one thread.
With 24 threads, the deviation of the "no pool" implementation variant and the unmodified *x264* increases, although the total execution time decreases, entailing that the relative

Implementation variant	Speedup [ms] one thread	Speedup [ms] 24 threads
original	-8	-24
no pool	0	-19
continuous refresh	-2	-15
continuous refresh global	-4	-1
continuous refresh main	14	-14
full continuous refresh	-8	28
single refresh	10	8
single refresh main	-15	-28

Table 5.3.: Absolute speedup in execution time (in ms, rounded to full ms) for all implementation variants through lazy collection. Base values are the respective execution times using eager collection

measurement error is significantly higher than it is with one thread. Using the same arguments as above, no implementation variants but the "full continuous refresh" and the "single refresh main" implementation variants exhibit a measurable difference in execution time between lazy and eager collection. As the MM-related execution time overhead does not change significantly between the eager and the lazy version of the "full continuous refresh" implementation variant as shown in subsection 5.4.4, it is concluded that this also applies to the "single refresh main" implementation variant, thus entailing that no implementation variant exhibits a significant difference in execution time between lazy and eager collection in this *x264* benchmark.

5.4.4. MM-related execution time overhead

As the STM paradigm introduces two new functions for MM, i.e. *refreshing* and *ticking*, the overhead introduced by the latter is analyzed. Using the best SCM-based implementation variant in terms of memory consumption, i.e. the "full continuous refresh" implementation variant, both, lazy and eager collection are included in the analysis, each of them with one and 24 threads. Details on the benchmark setup and the assumptions made can be found in appendix A.

Figure 5.10 (left) depicts the portions of the total execution time spent for *refreshing*, *ticking* and the rest using eager collection with one thread. While the MM-unrelated functions are dominant, the STM-related functions account for less than 0.2% of the total execution time. As the number of expired objects is higher before *tick* calls than it is before *refresh*

calls in this implementation variant (see subsection 4.4.4), a higher percentage of the execution time is spent in *tick* calls for expired descriptor page cleanup than in *refresh* calls. Conversely, the amount of time spent in *refresh* calls when using lazy collection with one

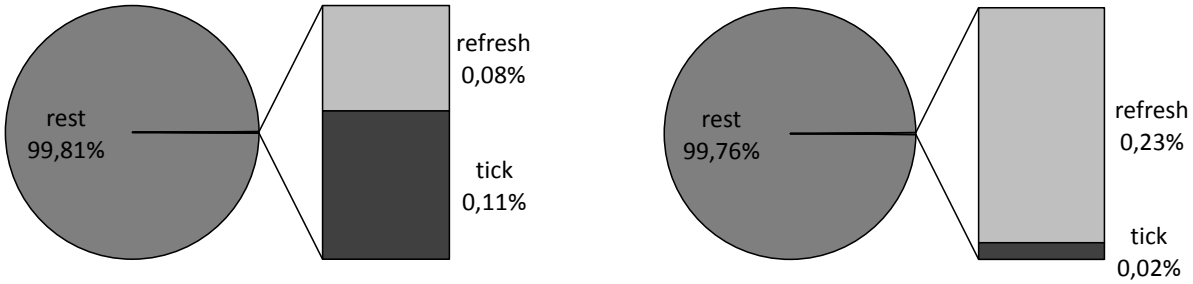

Figure 5.10.: Distribution of execution time to MM-related and MM-unrelated functions in the "full continuous refresh" implementation variant with one thread and eager (left) and lazy collection (right), respectively

thread (see figure 5.10, right) is 0.23% as opposed to 0.02% for the *tick* call portion which is due to the fact that there are more *refresh* calls which possibly collect one expired descriptor page than there are *tick* calls. The ratio between the two portions approximately reflects the ratio of total *refresh* to *tick* calls throughout program execution, not considering the different instructions processed in each function apart from collecting expired descriptor pages which accounts for a significant portion of their execution time.
Using 24 threads (not depicted), lazy collection also exhibit this behavior, i.e. the amount of time spent in *refresh* calls is significantly higher than the time spent in *tick* calls. With the latter being smaller than 0.01% of the total execution time and a *refresh* execution time portion of 1.07%, this also indicates that the time spent in MM-related functions is significantly higher than it is with one thread, the reason for which is the increased number of descriptors due to the increased number of threads involved in MM. Nonetheless, all other parts of the program account for 98.92% of the total execution time.
In contrast to eager collection with one thread, eager collection with 24 threads (not depicted) decreases the execution portion of *tick* calls significantly. This is due to the fact that more threads are involved in *refreshing* which is intensified by additional *refresh* calls in this implementation variant which are only called when there is more than one thread (see subsection 4.4.4). With 1.02% and 0.07% of the execution time spent in *refresh* and *tick* calls, respectively, eager collection with 24 threads is more similar to lazy collection with 24 threads in terms of execution time distribution than eager collection with one thread is to lazy collection with one thread.

5.5. Related work

[3] describes STM, SCM and their implementation in various programming languages, including benchmarks of different applications, some of which are based on preliminary results of the work described in this thesis. Besides the benchmarks for other applications, sections 2.1 and 4.1 in [3] briefly describe an *x264*-based benchmark with multiple STM implementation variants similar to the one described herein, including *local* and *global refreshing* as well as *refreshing* in *x264*'s coding threads. It should be noted, however, that, although the results are similar, they are not exactly the same as the results described in this thesis, as thorough validation revealed race conditions in the preliminary code whose correction yielded adapted expiration extensions and therefore slightly different benchmark results.

As mentioned in section 4.6, no other comparable modifications of *x264* exist which involve the change of *x264*'s MM. Thus, no benchmarks are available to compare the results of this chapter to. However, as *x264* already comes with a frame pool which greatly reduces the use of the underlying allocator for frames, thus representing a simple form of MM, the difference of the benchmark results with and without this frame pool are discussed in this chapter.

Conclusion

Porting *x264* to short-term memory is challenging and highly non-trivial. Despite the required knowledge of the H.264 standard, a deep understanding of a number of implementation-specific details is essential. However, it is possible to abstract some internal processes to an extent which is sufficient to determine *refresh* and *tick* positions for short-term memory management and to derive the necessary expiration extensions. The latter can be over-approximated for the sake of simplification as the short-term memory model does not rely on upper bounds for object lifetimes to ensure correct deallocation, pointing out the ease of use of short-term memory in a complex application.

Due to the aforementioned ease of use, all implementation variants described in this thesis only require about 200 lines of code, i.e. less than 0.5% of *x264*'s total number of lines of code, to be changed and additionally make *x264*'s reference counting mechanism for frames obsolete, while impacting execution time negligibly. However, only two implementation variants perform slightly, i.e. on average by less than 1%, better than the original *malloc*- and *free*-based implementation in terms of memory consumption when using one thread. A simplification of *x264*'s memory management, referred to as "no pool" implementation variant, which is unrelated to short-term memory, does so too.

When using multiple threads, those implementation variants which work for all tested parameter combinations exhibit a peak memory consumption which is at least 56% higher than the original *x264* implementation's. This overhead is due to the over-approximated expiration extensions which target simplicity and ease of use. Implementation variants using more sophisticated models of *x264*'s heap object life times and performing better in terms of memory consumption are possible as demonstrated by a special implementation variant which exhibits an overhead of less than 19%, but only works for some parameter combinations. However, the comparison of the different approaches described herein shows that using more sophisticated *refreshing* strategies and/or expiration extensions does not necessarily decrease memory consumption.

In conclusion, the proposed short-term-memory-based implementation variants exhibit

both strengths (ease of use) and weaknesses (increased memory consumption) compared to "classical" memory management. As *x264* is already highly optimized, it is notable that two approaches can be provided as a result of this thesis whose net memory consumption is lower than the original *x264*'s when using one thread.

Future work

The main focus of possible future work lies on the performance improvement of the stable implementation variants when using multiple threads as described above, as the default settings in *x264* enable multi-threaded encoding whenever the underlying hardware and/or operating system allows it. Additionally, other parts of *x264* may be transformed to use short-term memory, too, although the overall decrease in memory consumption is expected to be low or non-existent due to the fact that most of the objects which have not been already transformed to short-term memory objects are allocated only once and used throughout the whole program execution. Furthermore, the influence of other parameters besides the ones already tested herein, including mainly those which influence the frame type decision mechanisms of *x264*, on memory consumption and execution time may be analyzed in order to further optimize the implementation variants for specific parameter combinations.

Regarding future work for the short-term memory implementation, it is possible to introduce intermediate levels of collection between lazy and eager collection. This would allow for more fine-grain control over the number of collected expired descriptor pages per memory management operation, enabling the programmer to choose a number of collected pages which does not effect fragmentation significantly, but improves the net memory consumption. Although this would introduce yet another degree of freedom to be configured, it would allow for a more precise adjustment of the time/space trade-off in general and therefore improve the benchmark results of short-term-memory-managed applications with an allocation behavior similar to that of *x264*.

Bibliography

[1] Aigner, M., Haas, A., Kirsch, C. M., and Sokolova, A. Short-term Memory for Self-collecting Mutators – Revised Version. Technical report, Salzburg University, Department of Computer Sciences, October 2010.

[2] Tanenbaum, A. S. *Modern Operating Systems*. Prentice-Hall, second edition, 2001.

[3] Aigner, M., Haas, A., Kirsch, C. M., Lippautz, M., Sokolova, A., Stroka, S., and Unterweger, A. Short-term Memory for Self-collecting Mutators. In *International Symposium on Memory Management 2011 (ISMM 2011)*, San Jose, USA, June 2011.

[4] Free Software Foundation. GNU C Library. `http://www.gnu.org/s/libc/` (retrieved March 10, 2011), 2011.

[5] Nguyen, H. H. and Rinard, M. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proceedings of the International Symposium on Memory Management 2007*, 2007.

[6] Gay, D. and Aiken, A. Memory management with explicit regions. In *Proceedings of the 1998 Conference on Programming Language Design and Implementation*, 1998.

[7] ITU-T. Recommendation ITU-T H.264 – Advanced video coding for generic audiovisual services. Technical report, International Telecommunication Union, March 2010.

[8] Wiegand, T., Sullivan, G. J., Bjøntegaard, G., and Luthra, A. Overview of the H.264/AVC Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003.

[9] Merritt, L. et al. X264: A High Performance H.264/AVC Encoder. `http://neuron2.net/library/avc/overview_x264_v8_5.pdf` (retrieved December 28, 2010), November 2006.

[10] Özbek, N. and Tunali, T. A Survey on the H.264/AVC Standard. *Turk J Elec Engin*, 13(3):287–302, 2005.

[11] Kerr, D. A. Chrominance Subsampling in Digital Images. `http://dougkerr.net/pumpkin/articles/Subsampling.pdf` (retrieved December 28, 2010), December 2009.

[12] Richardson, I. E. G. *H.264 and MPEG-4 Video Compression: Video Coding for Next-Generation Multimedia*. John Wiley & Sons, 2003.

[13] Schwarz, H., Marpe, D., and Wiegand, T. Overview of the Scalable Video Coding Extension of the H.264/AVC Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, September 2007.

[14] Lee, J. and Kalva, H. *The VC-1 and H.264 Video Compression Standard for Broadband Video Services*. Springer Science+Business Media, LLC, 2008.

[15] Warren, H. S. *Hacker's Delight*. Addison-Wesley, 2010.

[16] Marpe, D., Schwarz, H., and Wiegand, T. Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard. *IEEE Transactions on Circuits and Systems*, 13(7):620–636, July 2003.

[17] Unterweger, A. Wahrnehmungsgestützte Steuerungskriterien für Modi- und Quantisierungsparameter-Entscheidungen zur subjektiven Qualitätsverbesserung von H.264-kodierten Videosignalen in Echtzeit. Diploma thesis, Salzburg University of Applied Sciences, June 2008.

[18] Chau, P. M. Final Project: H.264 Encoder Lite. `http://ece-classweb.ucsd.edu/fall05/ece111/Assignments/Final/final.html` (retrieved January 3, 2011), 2005.

[19] Jurkiewicz, A. et al. X264 Settings. `http://mewiki.project357.com/wiki/X264_Settings` (retrieved December 28, 2010), December 2010.

[20] Zhu, S. and Ma, K. A New Diamond Search Algorithm for Fast Block-Matching. *IEEE Transactions on Image Processing*, 9(2):287–290, February 2000.

[21] Saponara, S., Blanch, C., Denolf, K., and Bormans, J. The JVT Advanced Video Coding Standard Complexity and Performance Analysis on a Tool-By-Tool Basis. In *Packet Video Workshop, Nantes, France*, April 2003.

[22] Ostermann, J., Bormans, J., List, P., Marpe, D., Narroschke, M., Pereira, F., Stockhammer, T., and Wedi, T. Video Coding with H.264/AVC: Tools, Performance and Complexity. *IEEE Circuits and Systems Magazine*, 4(1):7–28, August 2004.

[23] Merritt, L. Unnamed (doc/threads.txt). Part of *x264*'s *doc* folder, December 2006.

[24] ISO and IEC. Programming Lanuages – C (ISO/IEC 9899:1999). Technical report, International Standardization Organization, June 2005.

[25] Free Software Foundation. GNU 'make'. `http://www.gnu.org/software/make/manual/make.html` (retrieved April 30, 2011), 2011.

[26] Wirth, N. *Compiler Construction*. Addison-Wesley, 1996.

[27] Free Software Foundation. GNU Compiler Collection. `http://gcc.gnu.org/` (retrieved April 30, 2011), 2011.

[28] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. `http://www.intel.com/Assets/PDF/manual/253665.pdf` (retrieved April 30, 2011), April 2011.

[29] ISO, IEC and IEEE. Information technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 7 (ISO/IEC/IEEE 9945:2009). Technical report, International Standardization Organization, 2009.

[30] Chang, C. and Pan, C. and Chen, H. Fast Mode Decision for P-Frames in H.264. In *Proceedings of the 24th Picture Coding Symposium*, pages 10–21, San Francisco, USA, 2004.

[31] Fan, Z. and Xudong, Z. Fast macroblock mode decision in H.264. In *Proceedings of the 7th International Conference on Signal Processing*, volume 2, pages 1179–1182, 2004.

[32] Chun, S. S. and Kim, J. and Sull, S. Intra Prediction Mode Selection for Flicker Reduction in H.264/AVC. *IEEE Transactions on Consumer Electronics*, 52(4):1303–1310, 2006.

[33] Mai, Z. and Yang, C. and Xie, S. Improved Best Prediction Mode(s) Selection Methods Based on Structural Similarity In H.264 I-Frame Encoder. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 3, pages 2673–2678, 2005.

[34] Vanam, R., Riskin, E. A. and Ladner, R. E. H.264/MPEG-4 AVC Encoder Parameter Selection Algorithms for Complexity Distortion Tradeoff. *Data Compression Conference*, pages 372–381, 2009.

[35] Ciaramello, F. M. and Hemami, S. S. Complexity constrained rate-distortion optimization of sign language video using an objective intelligibility metric . In *IEEE Western New York Image Processing Workshop 2007*, Rochester, USA, 2007.

[36] IEEE and The Open Group. The Open Group Base Specifications, Issue 7 (IEEE 1003.1-2008). Technical report, International Standardization Organization, 2008.

[37] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z. `http://www.intel.com/Assets/PDF/manual/253667.pdf` (retrieved July 23, 2011), May 2011.

[38] ISO and IEC. Information technology – Document description and processing languages – Office Open XML File Formats – Part 1: Fundamentals and Markup Language Reference (ISO/IEC 29500-1:2008). Technical report, International Standardization Organization, 2008.

[39] Ramey, C. The GNU Bourne-Again SHell. `http://tiswww.case.edu/php/chet/bash/bashtop.html` (retrieved July 2, 2011), 2011.

[40] ISO. Document management – Portable document format – Part 1: PDF 1.7 (ISO 32000-1:2008). Technical report, International Standardization Organization, July 2008.

Abbreviations

API Application Programming Interface

AVC Advanced Video Coding

CABAC Context-Based Adaptive Binary Arithmetic Coding

CAVLC Context Adaptive Variable Length Coding

CD Compact Disc

CIF Common Interchange Format

CRF Constant Rate Factor

CPU Central Processing Unit

DCT Discrete Cosine Transform

DPB Decoded Picture Buffer

DVD Digital Versatile Disc

IDR Instantaneous Decoder Refresh

FIFO First In First Out

HD High Definition

I/O Input/Output

LOC Lines of Code

MM Memory Management

MMCO Memory Management Control Operation

MPEG Moving Picture Experts Group

NAL Network Abstraction Layer

OOXML Office Open XML

PDF Portable Document Format

POSIX Portable Operating System Interface for Unix

QP Quantization Parameter

PSNR Peak Signal-to-Noise Ratio

RAM Random Access Memory

RDO Rate-Distortion Optimization

SD Standard Definition

SEI Supplemental Enhancement Information

SCM Self-collecting mutators

SSIM Structural Similarity

STM Short-Term Memory

VCEG Video Coding Experts Group

VUI Video Usability Information

XML Extensible Markup Language

A. Time overhead benchmark setup

In order to measure the portion of the execution time spent inside *refreshes* and *ticks* in *libscm* during benchmarking, the following test setup is used: based on the benchmark setup for the measurement of memory consumption (see section 5.1), the "full continuous refresh" implementation variant is linked against a version of *libscm* which is compiled with *ENABLE_MICROBENCHMARKS* enabled. This way, the number of CPU cycles spent inside each of the following functions is written to *stdout* after each function call: *scm_refresh*, *scm_global_refresh*, *scm_tick* and *scm_global_tick*.
It is assumed that each function call takes the same number CPU cycles to finish when given an identical configuration and internal variable states (see below). Thus, the number of repetitive executions is the same as in all space-related benchmarks, i.e. one actual run, preceded by three "blank" runs, even though cycles, and therefore a form of time, is measured. For the purpose of evaluation, the output of the actual run is filtered for measured CPU cycles and grouped accumulatively by *refresh* and *tick* calls.
Internally, *libscm* uses *RDTSC* instructions[37] to measure the number of CPU cycles spent between the beginning and end of each affected function. Although the number of CPU cycles is proportional to the speed of the CPU core, the "blank" runs assure that the CPU cores are operating at 100% of their frequency. CPU cycles spent in other applications or kernel mode (e.g. for context switches) are not considered in this calculation, but tolerated as it is assumed that the probability of a context switch during the execution of one of the aforementioned functions is the same as it is in all other functions of *x264*.
The total execution time the number of CPU cycles are compared to is the median execution time of the corresponding implementation variant obtained through time-related benchmarks (see subsection 5.4.1), assuming that the aforementioned functions' execution time does not change significantly between multiple runs. As *ENABLE_MICROBENCHMARKS* writes several MB of CPU cycle measurements to *stdout*, the execution time of the whole application is significantly increased due to the increased I/O activity and therefore not adequate to compare the time spent inside *refreshes* and *ticks* to.

B. Contents of the attached CD

The attached CD contains the source code of *libscm* and *x264*, including the described changes as well as the test scripts used for benchmarking and the acquired result sets. The folder structure is as follows:

- *libscm* – contains the source code of *libscm*, including the changes described in section 5.2
- *libwrapper* – contains the source code of the wrapper library for space benchmarking described in section 5.2
- *x264new* – contains the source code of *x264*, including the changes described in chapter 4
- *Benchmark results* – contains visualizations of the benchmark results (both time- and space-related) in Office Open XML (OOXML)-compatible files[38], including the original log files for reference

The scripts used for benchmarking are located in the root folder and require *bash*[39] in order to be executed. Note that some of them take multiple hours or even days to execute. They can be used as follows:

- *prepare* – creates two versions of *libscm* (one for regular operation and one for benchmarking, as described in section 5.2) and all SCM and non-SCM *x264* implementation variants for regular operation and benchmarking. Always execute this script before executing any of the others and note that *x264* requires a number of prerequisites, like a *yasm* installation (see *configure* script error messages), prior compilation
- *test_scm* – validates all SCM-based *x264* implementation variants as described in section 4.5

- *test_scm_quick* – performs a quick validation of all SCM-based *x264* implementation variants similar to the *test_scm* script, but with a significantly reduced number of parameter combinations in order to consume less time in total
- *measure_time* – executes the time-related benchmarks for all *x264* implementation variants described in subsection 5.4.1. To include or exclude certain implementation variants or to change the number of passes, the corresponding parameters in the script can be modified (this also applies to all subsequent scripts)
- *measure_median_time* – executes the time-related benchmarks for all possible parameter combinations of *–ref* and *–bframes* as described in subsection 5.4.2
- *measure_memory* – executes the space-related benchmarks for all *x264* implementation variants described in subsection 5.3.1
- *measure_max_memory* – executes the space-related benchmarks for all possible parameter combinations of *–ref* and *–bframes* as described in subsection 5.3.2

In addition, the following files are located in the root folder for the purpose of documentation:

- *thesis.pdf* – this thesis in the Portable Document Format (PDF)[40]
- *Locations and expiration extensions of all SCM modes.xlsx* – a list of all code changes in *x264* related to STM, including the changes' locations and expiration extensions for *refreshes* as OOXML-compatible file[38]
- *foreman-352x288.yuv* – the video sequence used for testing