Executive Summary

CriticalBlue, a pioneer of flexible, automated embedded system design solutions, recognizes that running existing software applications on 2nd generation Intel® Core™ processor-based devices can result in an immediate performance boost, even before taking advantage of advanced features such as Intel® Advanced Vector EXtensions (Intel® AVX).

However, things do not always go according to plan, especially when moving multi-threaded applications from single-core or dual-core systems onto the new processors with as many as eight logical cores.

This paper examines how Prism* — a multicore embedded software design environment developed by CriticalBlue — can be used to evaluate the correctness and scalability of the open-source h.264 video encoder x264. Working closely with Intel, CriticalBlue has enhanced Prism with awareness of 2nd Generation Intel Core processor data caches, pipeline and Intel® Hyper-Threading Technology.1

Using Prism*, CriticalBlue analyzed cache performance over a range of 2nd generation Intel® Core™ processors. This analysis showed that for the large data sets involved in HD video encode, the availability of large Last Level Cache (LLC) in 2nd generation Intel® Core™ i7 processors can have a significant impact on application performance.
Table of Contents

H.264 Encoding ................................................ 2
x264 Analysis With Prism ........................................ 2
Verifying Correct Behavior on Multicore Devices ................. 4
  Race between instances of x264_threadpool_thread .......... 4
  Race between x264_threadpool_thread and main .............. 5
  Race between x264_lookahead_thread and main ............... 5
Evaluating Scalability Across Targets Based on 2nd Generation Intel® Core™ Processors ............... 6
Conclusion .................................................... 8

H.264 Encoding

The H.264 video coding standard is an evolution of previous MPEG standards and follows the same basic algorithm. Each input frame is split into 16x16 pixel macroblocks (MBs) for processing. The compression occurs in either intra-frame or inter-frame mode. MBs for intra-frames are formed from samples taken directly from the reference frame, which is the encoded, decoded and reconstructed form of the input frame. In inter-frame mode, the MB is constructed from motion compensation data and as many as five reference frames, which could be from before or after the current frame in the video sequence.

Another main design goal for H.264 is to reduce the complexity of the decoder, which means that the encoder must do a great deal of additional work. In fact, the encoder contains the entire decoder chain so that it can evaluate the decode performance of a given encoding of a video frame. Furthermore, the use of as many as five reference frames requires a substantial frame buffer in the implementation.

This additional workload means that a large amount of data must be processed when dealing with HD video streams. Because the resolution of each frame can be as high as 1920x1080 pixels, this requires almost 3 MB of memory for each frame being encoded or stored in the reference frame buffer. Throughput is another issue; it takes more than 2.5 billion instructions for x264 to process each frame. For a real-time encode of 30 frames per second, this implies running at more than 75,000 MIPS while moving data into the processor at 90 MB/s.

The high computational requirement of HD video encoding has many consequences for analysis tools. Capture of memory access and instruction execution patterns must be carefully limited to prevent excessively large traces while still retaining enough useful data for meaningful analysis.

x264 Analysis With Prism*

The first step in any analysis with Prism is to capture a representative trace of the application, in this case x264 encoding of an HD video file, as it runs on the target platform. The target platform can be based on a recent Intel® Core™ microarchitecture (not necessarily a 2nd generation Intel® Core™ processor) since Prism provides the ability to model the impact of running the trace on the new architecture, as will be discussed later in this paper.

The trace contains information about the dynamic behavior of x264, including the execution path through the code, the memory locations accessed and which Pthread API calls are made. The trace is then copied into Prism running on a host machine for processing.

Capture of the trace is achieved with the prismtrace utility, which is run on the target and instruments x264 as it executes. During instrumentation, prismtrace controls and monitors each instruction executed by x264. If an instruction does something of interest to Prism, it is logged in the trace file and execution continues until completed.
Once loaded into Prism, the execution profile x264 encoder is displayed as in the screenshot in Figure 1, where the histogram shows the active threads of execution and the function call tree for each thread.

After the initial execution of the main and x264_lookahead_thread threads, the actual encoding is handled by the threads running the x264_threadpool_thread function. The Call Count column in the Functions view (see Figure 2) shows that 13 of these threads are in existence. Expanding the call tree from the x264_threadpool_thread shows the 30 calls to the x264_slices_write function, which performs the actual encoding of each of the 30 frames that ran during tracing.

The rest of the functions in the tree at this level clearly relate to the various steps in the H.264 encode algorithm discussed previously.

Back in the histogram view, the most important information is contained in the Data Races row. The spikes of activity here indicate that data races have been detected in the trace at these points that require further investigation.
Verifying Correct Behavior on Multicore Devices

Data races are caused by two or more threads accessing the same memory location, with at least one access being a write and without any synchronization. These are a common source of bugs in multi-threaded programs. Often, data races will not cause problems on devices with a low number of cores, but as the number of cores increases, the number of possible thread orderings also increases, making it more likely that a data race will become an issue.

In Prism, all detected data races are in a single view, which shows the pairs of threads, functions and source lines that access shared memory locations. Double-clicking on the source-to-source dependencies displays the corresponding source code locations, allowing the user to quickly investigate the causes of the data races.

In the x264 trace, Prism shows data races between the main and both the x264_threadpool_thread and the x264_lookahead_thread, as well as between the x264_threadpool_thread threads themselves, as shown in Figure 3.

Race between instances of x264_threadpool_thread

Brief investigation using the source code lookup from the data race view shows that the inter x264_threadpool_thread data races appear to be harmless since they run through the debugging output. Thus, they should not cause a problem in production, but they could cause confusion for anyone trying to make sense of the debug output.

The screenshot in Figure 4 shows the relevant sections of the code. The race is due to the accesses to the i_lines_completed member of the frame_t data structure.

Race between x264_threadpool_thread and main

The race between the x264_threadpool_thread and main occurs during initialization of the encoder. There is no synchronization between x264_threadpool_thread threads, as they each call the x264_encoder_thread_init function as part of their initialization, and the main thread as it calls x264_thread_sync_context, which is inlined into the x264_encoder_encode function. The screen shot in Figure 5 shows the locations of the source lines causing the races.

The problem is due to the update of the params member of the x264_t structure in x264_thread_sync_context (which is called at the beginning of each frame encode) while it is being read by the per thread initialization routine. Fortunately, there is a large amount of computation between the thread initialization and the start of the first frame encode, so this race is very unlikely to become an issue in reality. But it could easily be resolved by a mutex protected counter, which main blocks on until all of the threads have signaled their initialization.

Race between x264_lookahead_thread and main

This race is very clear from code inspection, but it is an interesting example of how Pthreads programming can quickly become full of knots. The screen shot in Figure 6 shows the relevant code sections.

Figure 5. Complex dependency causing a data race highlighted by Prism.

Figure 6. Prism displays relevant code sections in data race.
At first glance, the dynamic between the two sections of code is a classic Pthread pattern. The main thread in function `x264_encoder_encode` signals the `x264_lookahead_thread` thread that there is no more work to do by setting the `b_exit_thread` to 1, indicating that it should terminate. The communication uses a condition variable and mutex for synchronization, but on the `x264_lookahead_thread` side, the shared variable is not protected by the same mutex. A few lines down into the loop, the screenshot in Figure 6 shows another while loop testing `b_exit_thread`, this time within the mutex lock and synchronized on the condition variable.

This was likely the intended target for the signal from main, but at some point the code has possibly been refactored with the external while loop introducing the data race. As in the previous cases, it is unsurprising that this race has not been causing bugs since the large amount of other synchronization within the loop body is likely to enforce the correct ordering most of the time. However, it is difficult to be sure, so fixing this race is important in maintaining the robustness of the implementation.

### Evaluating Scalability Across Targets Based on 2nd Generation Intel® Core™ Processors

This section of the white paper looks at how well the threading strategy used in x264 works across two examples of the 2nd generation Intel Core microarchitecture.

x264 automatically adjusts the number of threads to take advantage of the number of available cores. Fortunately, one can override the number of threads from the command line, which — when combined with Prism’s ability to model various Intel® architectures — allows an evaluation of x264 running on a range of platforms from a single target.

This evaluation looks first at the performance on a 2nd generation Intel® Core™ i5-2510E processor that has 3 MB of Last Level Cache (LLC) and two cores with Intel® Hyper-Threading Technology, which, to the operating system, looks like a four-core processor. This is simply selected in the architecture view in Prism and applied to the schedule. The result is shown in Figure 7.

The core schedule displayed in Figure 7 shows that the threading is working well, with each core scheduled with threads for more than 80 percent of the time. However, the utilization of each core is quite low, suggesting that there is a problem with the cache or pipeline performance of the implementation.

Switching to the core view makes it immediately obvious that the cache performance of the application is not good. This is not surprising given the large amount of data involved when processing several frames in parallel. The cache performance view is shown in Figure 8 (shown on the next page).
The majority of the misses are due to the `x264_pixel_sad_x3_16x16_sse2` function, which is an assembler routine that implements an optimized sum of absolute differences algorithm. The high miss rate is directly due to the huge bandwidth requirement caused by having several cores performing similar processing in parallel. The high ratio of misses (16 percent) at the shared LLC for this function means that the routine is often waiting on accesses to main memory.

The second part of this evaluation looks at how the performance scales to a higher-end device. The 2nd generation Intel® Core™ i7-2710E processor has 6 MB of LLC and presents eight logical cores to the operating system, which allows for double the number of threads. Again, the appropriate architecture is selected and applied to the schedule for the trace containing the additional threads.

The schedule view in Figure 9 shows that the performance impact of going to the eight-core device is approximately a 2.1x speedup. This seems impossible since the maximum ideal speedup for doubling the cores is 2x, but looking at the slightly improved utilization figures for the core indicates that other architectural differences are in play.

Switching to the cache performance view (Figure 10, shown on the next page) makes it clear that the improvement of the cache performance is due entirely to the fact that the 2nd generation Intel Core i7-2710E processor-based device has twice as much LLC as the 2nd generation Intel Core i5-2510E processor-based device.
The miss count is still high for the `x264_pixel_sad_x3_16x16_sse2` function, which has only dropped from 177K to 145K, but stall cycles have dropped dramatically (5.3 M cycles down to 2 M cycles) since all of the miss reduction has been on the LLC, minimizing the number of main memory accesses. Many of the other functions have benefited from this kind of performance increase also.

Conclusion

`x264` is already well respected for both the quality and the performance of its encoding on existing Intel architecture platforms. While this analysis by CriticalBlue has uncovered a few minor threading issues, these can be easily remedied with negligible impact on overall performance.

The scalability approach of dynamically altering the number of threads based on core availability provides a good workload to keep the entire processor utilized while keeping the thread management overhead to a minimum.

Analysis of the cache performance over a range of 2nd generation Intel Core processors has shown that for the large data sets involved in HD video encode, the availability of large LLC in 2nd generation Intel Core i7 processor-based devices can have a real impact on performance.

For more information on 2nd generation Intel® Core™ processors visit www.intel.com/p/en_US/embedded/hwsw/hardware/core.

For additional information on CriticalBlue, please visit Intel’s Web site at www.criticalblue.com.

To learn more about Intel’s ecosystem with 160+ members and 1,000+ solutions, visit the Intel® Embedded Alliance Web site at www.intel.com/design/network/ica/index.htm.