ANALYSIS OF THE THREAD ASSIGNMENT BEHAVIOUR OF PARALLEL PROGRAMS ON CHIP MULTIPROCESSORS

Michael Bogner [a], Ematinger Markus [b], Franz Wiesinger[c]

[a, b, c] University of Applied Sciences Upper Austria,
Hardware/Software Design & Embedded Systems Design

[a] michael.bogner@fh-hagenberg.at,
[b]markus.ematinger@fh-hagenberg.at,
[c]franz.wiesinger@fh-hagenberg.at

ABSTRACT
Nowadays, a chip multiprocessor following the x86-architecture combines at least four dedicated cores. To take benefit of this computing power, the application has to use multiple threads. To provide the parallel behavior for all processes and threads, the allocation has to change frequently. Depending on the situation, this allocation can differ and is worth to be analyzed.
The application runs a configurable amount of hard-working threads. By interfering with the core-thread-allocation several different scenarios have been tested. By recording this thread-core-allocation and the execution time, it is possible to compare the different scenarios.
The paper shows that Microsoft Windows 7 handles the thread-core-allocation in a lot of situations quite well. But the exclusion of core zero provides a performance increase. This is only useful if the number of threads is lower than the number of available processor cores. This situation also shows an additional interesting incident. Windows tries to balance the load over all processor cores very frequently.

1. INTRODUCTION
1.1. Operating System
The most actual devices use an abstraction layer between hardware and applications. This layer is called operating system (Tanenbaum, Moderne Betriebssysteme, 2003). It provides a runtime environment for the applications and handles the hardware resources like the main processor and the memory (Tanenbaum, Computerarchitektur, 2006). An operating system runs many different applications at the same time. This means that the operating system has to share the processor time between the applications. To provide a decent allocation the operating system has to switch between running application in small time slices.
An application is a process. And a process can contain one or more threads. Threads and processes are basically in one of three different states. These three states are “Running”, ”Blocked” and “Ready”. When a process or a thread gets created, it starts in the Ready state. The scheduler selects one of all available Ready states by using a special algorithm. When the thread gets selected by the scheduler it gets switched into the Running state. At this state the thread executes his code. There are many situations, where the thread has to wait for other resources. A resource can for example be a File IO or Events. The thread gets switched to the Blocked state and has to wait until the resource is available. When the resource gets available, the thread gest switched to the Ready State. The scheduler can select this thread and continue its execution. All threads are scheduled by the operating systems scheduler. Simple applications only run their single main thread. This means if the program has to wait for a resource the whole application has to block and wait for this. The operating system runs the scheduler and switches to another application. This happens if the application runs only one thread like in Figure 1 at (a). This behavior can slow down the application. To continue the execution of the applications, the applications process has to contain multiple threads. These threads can continue the execution while another thread is blocking. This multiple threads are shown in Figure 1 at (b).
But multiple threads have also another advantage. If the machine has more the one processor core, the operating system is able to schedule multiple threads of a process to different cores. This can provide a performance increase, by executing different code on different processor cores at the same time. But this also means that the application has to hold the data of the threads consistent. In a multithreaded application many threads work together to finish faster, but depending on the application the threads are not allowed to modify the same data as another thread. This can be avoided by making the access exclusive for a single thread. This can be achieved by different synchronization mechanisms like critical sections, mutexes, semaphores, events and similar techniques. This functionality is provided by the operating systems API (application programming interface). The difference between these mechanisms is that some of them are working across the borders of a process and others work only in a single process. Depending on the operating systems, there are huge performance differences.

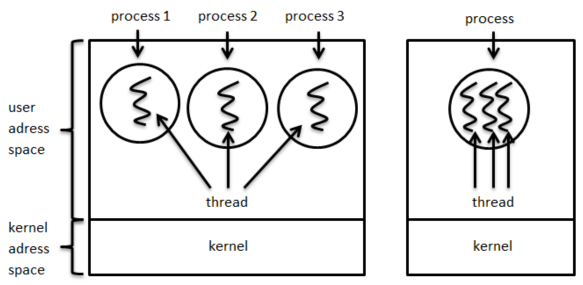


Figure 1: Threads and Process (Tanenbaum, Moderne Betriebssysteme, 2003)

1.2. Multicore processors

Before the rise of the on chip multicore processors a machine had to use multiple single core processors on a single motherboard to gain real advantage of multithreaded applications. These systems had to share the data by using special bus. It was not possible to use the on chip cache for data sharing.

Current multicore processors (David Harris, 2007) combine multiple cores in one single chip. Figure 2 shows the architecture (David A. Patterson, 2007) of an Intel Core i7 processor (Intel, 2011). The figure shows that the processor combines four dedicated processor cores on a single chip (Becchi, 2006) (Sondag, 2009). The chip uses the shared L3 cache for data sharing between the cores. The most actual multicore processors are comparable, like the AMD Phenom (AMD, 2009) and the Phenom II (AMD, 2011). Only the Intel Core 2 Quad (Intel, 2011) uses a different design, because it combines two dual core chips in one processor case. For data exchange between the two dual cores the processor has to use the main memory. Using the main memory for data exchange on a Core 2 Quad processor causes another performance loss, because the memory controller of this processor was place on a second chip on the Motherboard. This chip is called the Northbridge. For data sharing the data has to be sent form one core to the memory controller by using a bus and the same way back. This is slower than using the cache on actual quad core processors.

These multicore designs provide that the operating system is now able to run applications in parallel. It is also possible to share the data by using cache and the main memory. This data exchange is very quick. But the number of cores is not as high as the number of runnable applications. To get an increase in performance the application has to use multiple threads (Akhter S., 2006). The operating system can assign these threads to different processor cores and gain an increase in performance. Multithreaded programming does not gain performance automatically. This technique has to be used intelligent. If one thread marks the cached data of one core invalid by writing on them, this thread has to load the data from the main memory. This synchronization causes a decrease in performance. Creating threads and switching the cores decreases the performance. The threads should use the cores of a multicore processor, but should also minimize the needed overhead for synchronizations.

Depending on how these threads and processes are scheduled the performance can rise or fall. The scheduling can be restricted by setting the thread affinity. This is used to determine weak spots at Windows 7.

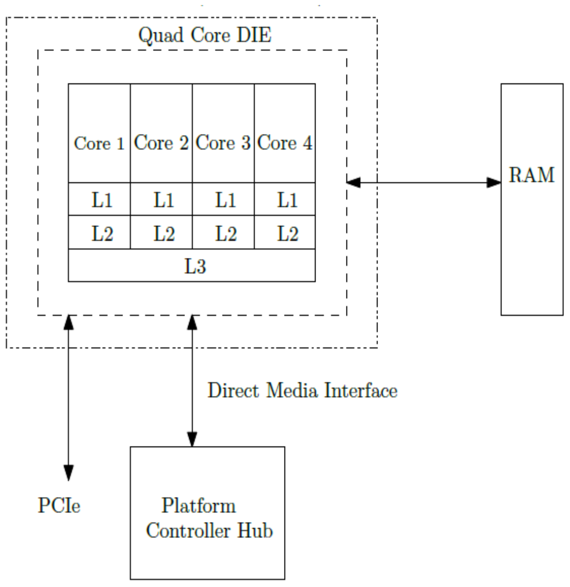


Figure 2: Intel Core i7 architecture

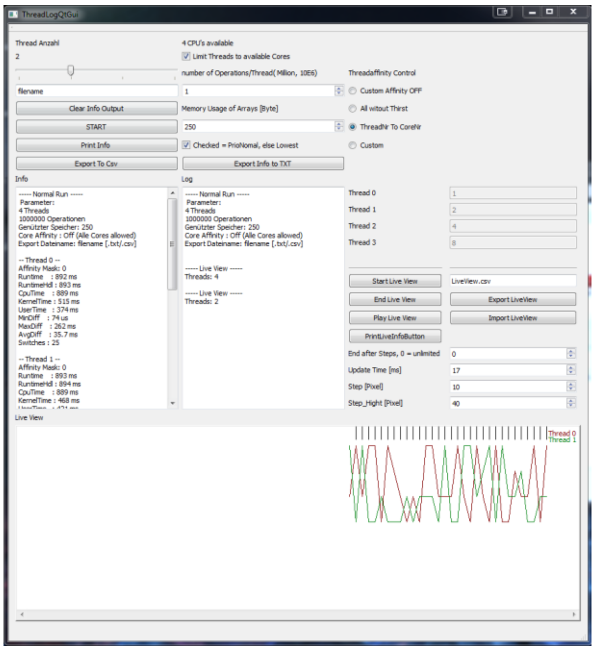
2. TEST

2.1. Main idea

The information that is used to find weak spots is the runtime and the thread-to-core allocation applied over the time. To get the information a test application is needed. This application is able to produce a heavy work load for every available processor core by using multiple threads. This application has to collect the thread core allocation and the runtime of each thread. The threads execute simple integer operations. After some operations the thread checks the allocation and stores the information.

Figure 3 shows the testing application. It uses the Nokia QT 4.7 Framework (Nokia, 2008-2011). On the top of the application it is possible to define the number of threads and the thread priority for the next test run. After a run has finished the logged data can be exported into a file in the well-known comma separated values (csv) format.

It also supports a simple live view of the thread-to-core allocation. But its accuracy is far away from the standalone test with the csv export, because it only samples the threads at a defined times slice. The sample rate is in the range of milliseconds. This means if the cores are faster switched, the live view will not show all cores switches.


ThreadLogQtGui

Thread Anzahl
2
filename

4 CPU's available
Limit Threads to available Cores [✓]
number of Operations/Thread(Million, 10E6): 1
Memory Usage of Arrays [Byte]: 250
Checked = PrioNomal, else Lowest [✓]

Threadaffinity Control
Custom Affinity OFF
All witout Thirst
[✓ ThreadNr To CoreNr]
Custom

Info
```
------ Normal Run ------
 Parameter:
4 Threads
1000000 Operationen
Genützter Speicher: 250
Core Affinity : Off (Alle Cores allowed)
Export Dateiname: filename [.txt/.csv]

-- Thread 0 --
Affinity Mask: 0
Runtime    : 892 ms
RuntimeHdl : 893 ms
CpuTime    : 889 ms
KernelTime : 515 ms
UserTime   : 374 ms
MinDiff    : 74 us
MaxDiff    : 262 ms
AvgDiff    : 35.7 ms
Switches : 25

-- Thread 1 --
Affinity Mask: 0
Runtime    : 893 ms
RuntimeHdl : 894 ms
CpuTime    : 889 ms
KernelTime : 468 ms
```

Log
```
------ Normal Run ------
 Parameter:
4 Threads
1000000 Operationen
Genützter Speicher: 250
Core Affinity : Off (Alle Cores allowed)
Export Dateiname: filename [.txt/.csv]


------ Live View ------
Threads: 4

------ Live View ------
Threads: 2
```

Thread 0	1
Thread 1	2
Thread 2	4
Thread 3	8
End after Steps, 0 = unlimited	0
Update Time [ms]	17
Step [Pixel]	10
Step_Hight [Pixel]	40

LiveView.csv

Figure 3: Screenshot of the test application

2.2. Test algorithm

To produce heavy workload a filter kernel calculates over a picture. The picture is represented by a simple two dimensional array. The array gets divided into stripes. Every thread has to calculate a single stripe of the picture. Figure 4 is showing this procedure. The integer calculation produces the work load. The size of the image is adjustable. Because of that this is not fair dividable, a small adjustment is made. Every thread has to run a fixed number of steps and calculate the filter kernel from the top left to the bottom right of their image part. If the thread reaches the end it restarts from the beginning until the number of calculate steps are reached. This provides the same workload for every thread.

To retrieve usable information at runtime it is needed to collect the thread-to-core affinity and the time. Figure 5 shows how this is done. The number of threads and the thread affinity is adjustable before the test run. To retrieve the thread-to-core affinity every thread has to retrieve the number of the core that is actually executing code. This is done by calling the Win32 API function GetCurrentProcessorNumber. This function returns the number of the processor core, from which core the function is called. Then the actual time is retrieved (GetActualTime). Therefore the Query performance counter of Windows is used (Microsoft, 2011). This counter makes it possible to retrieve the system ticks. These ticks can be converted into the time by using the Query Performance Frequency factor. This factor depends on the clock frequency of the processor. Before starting the threads a base time is stored. This time is the same for every thread. After retrieving the core number and the time, the core number is compared with the core number from the previous check (CheckCoreChange). If the core number is different, the core number and the time get stored. Then the calculation continues. This procedure starts from the beginning until the needed number of calculation steps is reached. At the end all the information of all threads are collected and exported into the csv file. By using this timestamps it is possible to calculate the number of all core changes that happened while the thread was running. It is also possible to calculate the average time between the core changes.

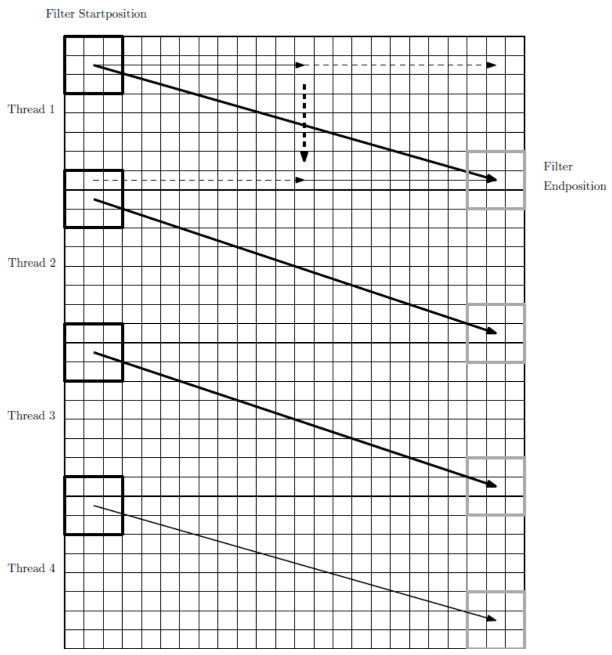


Figure 4: Picture of filterkernel

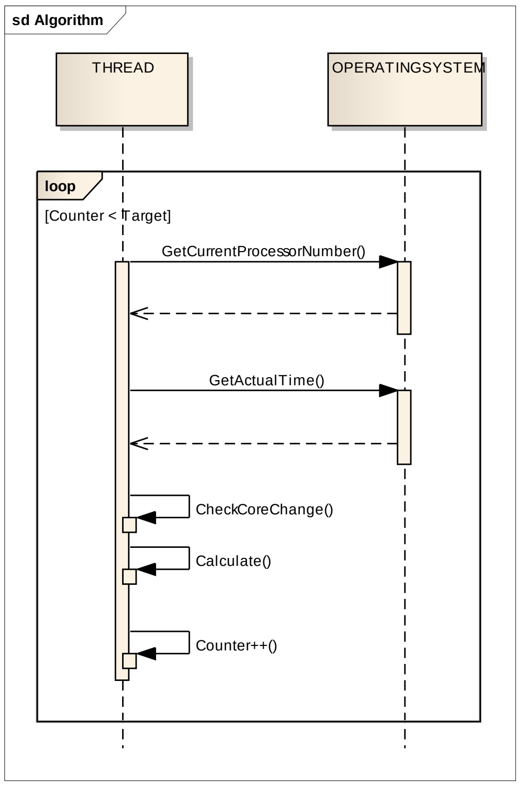


Figure 5: Algorithm of data determination

As a summary a thread retrieves the information this way:

1. Calculate the filter kernel for its position
2. Retrieve the actual core number
3. Get the actual time
4. Compare the retrieved core number (step 2) with the core number from the last check (stored in step 5)
5. Store the core number if the core number has changed
6. Continue with the calculation

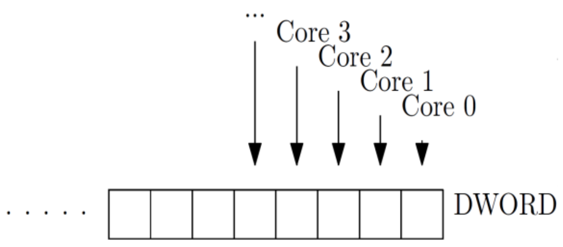


Figure 6: Thread affinity mask

Figure 6 shows the thread affinity mask. It is a DWORD which is 32 bit wide. This mask is used to set the affinity for every worker thread. This is the first thing that is done when the thread gets started. Therefore this mask is passed to the WIN32 API function SetThreadAffinityMask. As shown in the picture, every bit represents a processor core. Bit zero is for core zero and so on. If the bit for the core is true the thread can run on the core. If the bit is false, the core is denied. On a quad core processor only the lower four bits are important. If the mask is not set, all cores are allowed.

2.3. Test System

The main specification of the test system:

- Intel Core i7 860 quad core
  - Turbo Boost Technology disabled
  - Hyper threading disabled
- Asus P7P55D Evo mainboard
- 12 GB DDR3 Ram
- Windows 7 Professional 64 bit
- Microsoft Visual Studio 2010
  - C++
  - 32 bit Compiler

Due to better comparison between the scenarios with different number of threads the Intel Turbo Boost Technology is disabled. Otherwise a single thread test will run with higher clock rates than a test with four threads due to dynamic frequency scaling. Hyper threading is disabled, because it provides for every core one additional core. But it is wanted to compere only real quad core processors. So to disable this feature of an Intel Core i7, this is the main difference between a Core i7 and a Core i5 quad core processor, was the better choice.

2.4. Test cases

The test cases vary by the number of running threads from one up to four threads. For every different thread number the thread affinity is changed. The default thread affinity is that the operating system can execute every thread on every processor core. The second case is that every thread can only be executed on one single core. Thread zero can only run on core zero, thread one can only run on core one and so on. This will avoid that the operating system moves the threads from one core to a different core. The third main case is that every thread can be executed on every core, except core zero. This should demonstrate that core zero is preferred by the operating system. To start four threads in this situation was skipped, because there are only three runnable cores.

Furthermore there are some special cases tested. These cases are that the application starts more threads than the number of available processor cores, known as oversubscription. Other special cases are that only one or two threads are used. These threads are tested with different thread affinity. This means the thread gets bound to core zero or core three. This test should show the interference with the preferred core of the operating system.

3. RESULTS

Table 1 shows an example of the result of a single test run. It represents the average core change time for one to four threads, by using the systems default thread affinity and the exclusion of core zero. The average core change time describes the time between a thread gets pushed from one core to a different core.

Table 1: Average core change time (time in ms)

Threads	OS default	Core zero excluded
1	70.1	87.2
2	8.8	9.4
3	7.9	29.8
4	51.0	-

If the number of threads on a quad core processor is less than four and higher than one, the average core change time is below 10 ms. If the number of threads is one or equal the number of cores the average core change time is above 50 ms. This means the number of core switches with two or three threads is about 5 times higher than the number of core switches with one or four threads. If core zero is excluded, the averages core change rate of two threads is at least 3 times higher than the average core change rate of one or three threads. This behavior is shown in Figure 7. The upper screenshot of the taskmanager shows two hard working threads, where the threadaffinity is left at the default settings (all cores are allowed). The load overall load stays at 50 percent, but it is shared between all four available cores. To produced this behaviour Windows 7 has to move the threads from one core to an other core in small time slices as shown in Table 1 at two threads. When the two threads are fixed to two cores, both cores are loaded by 100 percent and the other two cores are idle. This is shown in Figure 7 at the bottom screenshot. This scenario is comparable with a scenario where four hard working threads are executed on a quad core processor, because from the view of the process the number of threads is the same as the number of available processor cores.

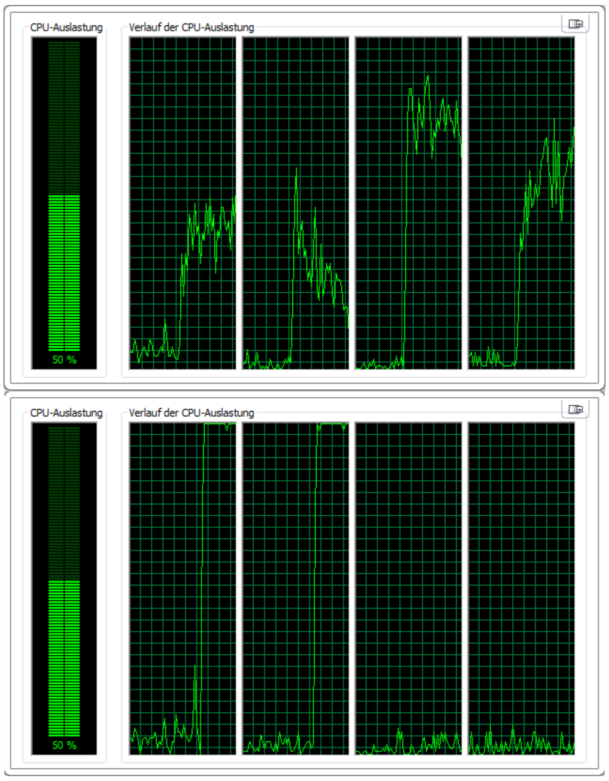


Figure 7: Comparison between default assignment and fixed assignment to core zero and core one, two threads

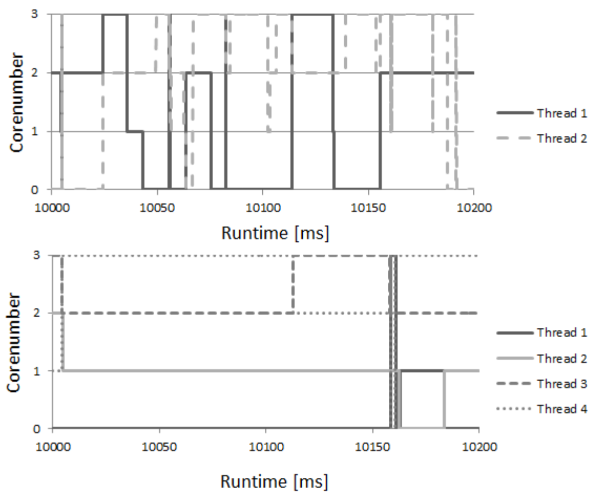


Figure 8: Core change comparison between 2 and 4 threads

Figure 8 shows a snapshot of the traced data. The time slice with 10 seconds until 10.2 seconds was randomly chosen. The vertical axis stands for the core number where the thread was assigned. The top diagram shows the two thread situation. And the bottom diagram shows the four thread situation. It is obvious that the two threads get pushed from one core to another core much more often, than four threads. This diagram confirms the results from Table 1.

Table 2: Runtime in seconds

Threads	OS default	Core zero excluded	thread fixed to core
1	41.8	41.70	43.7
2	48.73	47.40	48.03
3	46.63	47.20	49.23
4	51.57	-	53.40

Table 2 shows the runtime in the different situations. The time is the maximum runtime of all running threads, this means this time describe the timespan between the start of all threads and the stop of all threads. The runtime of the threads don't decrease by the number of threads, because by increasing the number of threads the workload gets increased by the same factor as describe in 2.2. In every situation, except one and two threads with core zero excluded, the default affinity mask is the fastest.
The Table also shows that it is not possible to gain any performance by binding every thread to a single core. Nearly every situation provided a significantly decrease in performance.

Table 3: special cases with two threads

assignment	Runtime [s]	Difference [%]
default	48.73	0
fixed to core 0 and 1	48.03	-1.44
Core zero excluded	47.40	-2.74
Core 2 and 3	47.33	-2.87
Fixed to core 2 and 3	46.80	-3.97

Table 3 shows a special case. Therefore in every situation only two threads are started. The thread affinity mask is the only difference between the test cases. The runtime is the time when both threads have finished their work. At default the operating system is allowed to use every processor core. This time is the base for the difference compared to the other situations. At fixed to core the thread zero was bound to core zero and thread one was bound to core one. This modification provides a slight increase in performance by 1.44%. The exclusion of core zero is 2.74 percent faster than the default setting. The last two situations are, that only core two and core three are used. If the operating system is only possible to choose between core two and three the performance gain is about 2.87

% compared to the default setting. If one thread gets bound to core two and the other thread gets bound to core three, the increase in performance is about 3.97 percent. It is obvious that Windows 7 prefers core zero and that it is better to bind the threads to other cores if the number of threads is smaller than the number of available cores.

Table 4: Average and maximum runtime in seconds at oversubscription and default thread affinity

Threads	Average runtime	Maximum runtime
3	46.62	46.63
4	50.57	51.57
6	69	76.77
8	97	97.93

Table 4 shows the scenario of oversubscription. In comparison to three and four threads the oversubscription situations with six and eight threads shows interesting results. The average runtime is the average time of all threads that is needed to finish the work, while the maximum runtime is the time that is needed to finish all threads. If we take the average time it is obvious that six and eight threads are slightly faster than four threads, by including the factor that the work increases by the same factor as the number of threads. This looks like it would be really good to use oversubscription, but depending on the application it is important to take a look at the maximum runtime. When using three, four or eight threads on a quad core processor the difference between the average and the maximum runtime is very low, but with six threads the difference is much bigger. For this scenario the whole work is done after the maximum runtime. With six threads some threads would finish earlier than the others.

4. CONCLUSION

Windows 7 handles most of the situations of standard applications really well. But there are also special cases where it is possible to increase the performance by modifying the thread affinity to gain even more performance. If the processor should run only one or two threads on a quad core processor, it is possible to increase the performance by excluding core zero. For three or more threads no modification is needed.

If the number of threads is lower than the number of available processor cores Windows 7 tries to balance the work load over all processor cores. The balancing operation causes heavy core switches and challenges the cache coherence protocol of the CPU. In comparison to the situation with four threads on a quad core, the core change rate is about 5 to 10 times higher. It is possible to decrease this rate by modifying the thread affinity.

Oversubscription is slightly faster than using the same number of threads than available processor cores. But if the number of threads is no multiple of the number of cores, the runtime between the threads differs very much. Depending on the used scenario is recommended to use a number of threads that is a multiple of the number of cores.

REFERENCES

Akhter S., R. J. (2006). *Multicore Programming: Increasing Performance through Software Multithreading.* Intel Press.

AMD. (2009). *Key Architectural Features of AMD Phenom X4 Quad-Core Processors*. Retrieved 2011, from http://www.amd.com/us/products/desktop/processors/phenom/Pages/AMD-phenom-processor-X4-features.aspx

AMD. (2011). *AMD Phenom II Key Architectural Features*. Retrieved 2011, from http://www.amd.com/us/products/desktop/processors/phenom-ii/Pages/phenom-ii-key-architectural-features.aspx

Becchi, M. C. (2006). Dynamic thread assignment on heterogeneous multiprocessor architectures. New York: ACM.

David A. Patterson, J. L. (2007). *Computer organization and Design.* Burlington: Morgan Kaufmann.

David Harris, S. H. (2007). *Digital Design and Computer Architecture. From Gates to Processors.* Morgan Kaufmann.

Intel. (2011, 5). *2nd Generation Intel Core Processor Family Desktop Datasheet, Vol. 1*. Retrieved from http://www.intel.com/content/www/us/en/processors/core/2nd-gen-core-desktop-vol-1-datasheet.html

Intel. (2011, 6). *Intel Core2 Extreme Quad-Core Processor QX6000 Sequence and Intel Core2 Quad Processor Q6000 Sequence Datasheet*. Retrieved from http://download.intel.com/design/processor/datashts/31559205.pdf

Microsoft. (2011). *Microsoft Developer Network*. Retrieved from http://msdn.microsoft.com/en-us/ms348103

Nokia. (2008-2011). *QT 4.7 reference Documentation*. Retrieved from http://doc.qt.nokia.com/4.7/index.html

Sondag, T. R. (2009). Phase-guided thread-to-core assignment for improved utilization of performance-asymmetric multi-core processors. Iowa: ACM.

Tanenbaum, A. S. (2003). *Moderne Betriebssysteme.* München: Pearsons Studium.

Tanenbaum, A. S. (2006). *Computerarchitektur.* München: Pearsons Studium.