ANALYSIS AND EVALUATION OF PERFORMANCE ISSUES OF PARALLEL SOFTWARE ON MULTI-CORE PROCESSORS

Franz Wiesinger[(a)], Mustafa Tunca[(b)], Michael Bogner[(c)]

[(a), (b), (c)] University of Applied Sciences Upper Austria – Embedded Systems Design,
Softwarepark 11, A-4232 Hagenberg, AUSTRIA
(a) franz.wiesinger@fh-hagenberg.at, (b) mustafa.tunca@fh-hagenberg.at, (c) michael.bogner@fh-hagenberg.at

ABSTRACT
For decades, the processor manufacturers have attempted to achieve performance gains by increasing the clock frequency on single-core processors. But physical problems – such as the high power dissipation – lead to the release of the first multi-core processors on the market in 2005.
To benefit from the multi-core architecture, parallel programming is required. However, this programming model requires a different approach and is associated with certain risks and pitfalls.
This paper focuses on modelling of certain test scenarios for two common multi-core specific problems, namely oversubscription and false sharing. Various simulations and tests offered solutions and design patterns to avoid such problems. Results have shown that the problems have a fatal impact on the execution time, so that the performance gain on the multi-core system is nearly nonexistent. Thence, any software developer must have in-depth knowledge of the used hardware and software to benefit as much as possible from multi-core architectures.

Keywords: multi-core, parallel software, performance analysis, oversubscription, false sharing

1. INTRODUCTION

For decades, processor manufacturers have tried to achieve more performance of their processors mainly by increasing the clock frequency. But despite technical countermeasures, this methodology of improving the performance of single-core processors has led to an increase of the power consumption and to an unmanageable heat generation when using conventional cooling techniques.
This fact and the growing demands of customers induces the large processor manufacturers such as Intel, IBM and AMD to further improve the development of single-core processors and to focus on the development of multi-core processors.
This was achieved by increasing the number of transistors – the main component of a processor – by improving the manufacturing techniques. As a result, these additional transistors serve in the processor development for many architectural advantages (Rauber and Rünger 2012). Instead of further increasing the chip packing density on a single-core processor and making it even more complicated, the available surface area on the silicon is now used for multiple cores which allows the distribution of the workload across these cores. Thereby, it is possible to increase the throughput and in further consequence, the performance referring to the power consumption is improved (Gove 2010).
The first multi-core processors reached the end-user in 2005 – thus, a new era of information processing was born. However, this resulted also in a new challenge for software developers. Because so far, the processor manufacturers were primarily responsible for performance improvements on their processors. Existing software was automatically faster with each generation of processors due to technological advances. But now having multi-core chips, it is no longer possible to get such improvements by only replacing the processor on a system. Instead, the software engineers have to use parallel software development to benefit from the new hardware architecture and to gain from performance improvements.
The parallel software development existed long before the era of the multi-core processors and therefore had been used by a minority of software developers in multi-processor environments. But now the multi-core processors being the new standard, all software developers have to deal with the issue of parallelism. Generally spoken, parallelism requires, inter alia, the decomposition of a given problem to a set of threads, which process the problem on different cores of the processor. However, this approach of development differs from the former sequential execution on a single-core processor, so in turn the software development requires a new way of thinking in modelling parallel software, which further yield to various difficulties and pitfalls.
As mentioned above, these problems and pitfalls are not really new and are known from former multi-processor systems. For this reason, the main focus of this paper is to check whether the performance-reducing problems also have an impact on current multi-core systems. Therefore, we have modelled some typical test scenarios which are most common. In this paper, we present our results for the issues oversubscription and false sharing. We have simulated and analyzed the run-time behavior of our software implementations and have derived some solutions, which will be explained in detail in this paper.

2. RELATED WORK

As the multi-core systems became the new standard, numerous well-documented papers about these systems are available, especially from the processor manufacturers. Since the described issues already existed before the multi-core processors got popular, there are many well-documented works which deal with the general issues on parallel systems (Duffy 2008, Tanenbaum 2009).

Since the developers tend to use too many threads in their applications, the common problem of oversubscription arises. Blumofe and Leiserson (1999) analyze and discuss a way of scheduling multithreaded applications on parallel systems and create the so-called work-stealing technique. Yanyong Zhang et al. (2003)provide a deeper analysis of the oversubscription problem, by comparing existing techniques and migrating the solutions into large scale parallel systems.

In contrast to the active community for scheduling problems on parallel systems, there is sparely research regarding the problems on hardware-level on such systems. The work of Torrellas et al. (1994)are one of the first in this field analyzing the issue of false sharing, one of the problems occurring in shared-memory parallel systems.

However, regarding problems on hardware and software in parallel systems, most of them do not deal with the new hardware platform and present no in-depth discussion according the special issues of concurrent multi-core systems. On the other hand, there are works which discuss this issues also for multi-core systems, but they are mainly theoretically, do not focus on special hardware and derive no suggestions or design patterns (Akhter and Roberts 2006, Gove 2010, Gleim and Schüle 2012, Rauber and Rünger 2012, Williams 2012).

3. TEST ENVIRONMENT

To analyze the problems precisely, the measurements have to be made according to specific criteria, which are defined in the following sections.

3.1. Test System

The test system uses a multi-core processor with Sandy Bridge microarchitecture with the model name Intel Core i7 2670QM. The processor has four physical cores and supports eight logical processors. Furthermore, the test system has 6 GB of memory and three cache levels, wherein a cache line in each cache-level has the size of 64 bytes. The L3-cache is shared among all cores and allows inter-core communication on the chip. Windows 7 Professional (64-bit) has been used as the operating system.

3.2. IDE and Profiler

For the implementation of the test scenarios the integrated development environment Microsoft Visual Studio 2013 has been used.

The analysis of the run-time behavior has been done with the profiler integrated in the IDE, the Visual Studio Profiling Tools.

In addition to the integrated profiler, an official plug-in from Microsoft for Visual Studio 2013 called Concurrency Visualizer has been used for visualization.

3.3. Libraries and Programming Language

The elaboration of the test scenarios takes place in the programming language C++. For managing the threads the *std::thread* class has been used, which is part of the C++11 language standard.

The renunciation of the operating system interface e.g. to create threads or synchronize them, means that the source code is independent of the operating system. Thereby, it is possible to run the test scenarios on any system with a compiler that supports the C++11 language standard. In our research, we used the compiler Visual C++ 2013.

3.4. Class Design

For our simulations, we decided to use an extensible software design to allow further implementations of other critical problems. The software design encapsulates the concrete implementations by using the strategy design pattern. So the developer can easily exchange the implementation of the concrete problem at run-time.

Figure 1 illustrates this structure represented in a UML class diagram. There are two important interfaces. The interface *CriticalProblem* offers methods which can be overridden by the implementation of a test scenario for a specific problem. And the class *ScenarioStarter* has methods, which have a concrete strategy such as for oversubscription to run its operations.

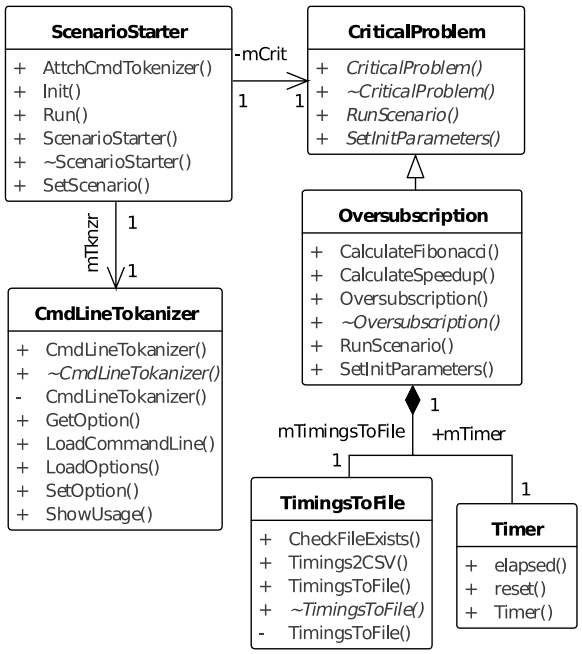

Figure 1: Class diagram modelling a general structure for implementing the test scenarios.

Listing 1 shows an example where the design pattern illustrated in Figure 1 is applied. The object *tokenizerObj* gets the user-given command line arguments respectively initial values for the test scenario and ana-

lyzes it for correctness. Then a test scenario *oversubscriptionObj* is attached to *starter*. The test scenario is finally initialized with the user-given options from the *tokenizerObj* and can be started with the *Run()* method. After the test scenario has finished another type of test scenario can be started directly afterwards.

Listing 2: Program code showing a typical usage of the class design.

```
int main(int argc, char** argv)
{
        CmdLineTokenizer tokenizerObj;
        ScenarioStarter starter;
        Oversubscription oversubscriptionObj;
        FalseSharing falsesharingObj;

        tokenizerObj.LoadCommandLine(argc, argv);
        starter.AttchCmdTokenizer(&tokenizerObj);
        starter.SetScenario(&oversubscriptionObj);
        starter.Init();
        starter.Run(); // run and measure run-time

        starter.SetScenario(&falsesharingObj);
        starter.Init();
        starter.Run(); // run and measure run-time
        return 0;
}
```

3.5. Time Measurement

In addition to the integrated profiler, time measurements have been done on specific sequences in the source code. To gain detailed and accurate measurement results, we have taken the class *std::chrono* with nanosecond precision from the C++11 language standard (simple high resolution timer in C++ 2015). The class *Timer* encapsulates the time measurement functionality. It has been used by the concrete strategy respectively by the test scenario for every specific problem.

4. CRITICAL PROBLEMS

This section discusses the performance degradations caused by oversubscription and false sharing in the parallel software development. These will, of course, lead to correct program results and also to a certain progress in the program execution. However, by false assumptions and a wrong way of thinking belonging the development of parallel software for these multi-core systems, the performance potential given by the hardware cannot be fully exploited.

4.1. Oversubscription

Since the operating system is responsible for creating and managing the threads, this circumstance is very important for the performance of a parallel application (Akhter and Roberts 2006).

Depending on the concrete implementation of the scheduler, the user simply misses the desired performance in a program.

Oversubscription is a problem that occurs when there are more active threads – which are managed by the operating system and referred as software threads – than hardware threads, which define the number of virtual processors (Akhter and Roberts 2006).

Most of the operating systems use the time slicing technique, also known as round-robin scheduling, which negatively influences the situation of oversubscription. Although blocked threads are not part of the active round-robin scheduling, since these threads are dequeued from the waiting queue, the problem still exists. This scheduling technique leads to context switches of all active threads in which the footprint of the interrupted process, such as the used registers are saved and the register data from the next thread that has received a time slice, is loaded (Tanenbaum 2009).

Since each thread gets a time slice, none of them suffers from starvation. However, the context switch causes a certain overhead by recovering and storing the thread conditions, which results – when having a high number of software threads – in poor performance (Akhter and Roberts 2006).

Not only by the context switches, but also at memory-level, oversubscription causes problems. Processors try to use the caches for storing frequently used data to avoid accessing the slower main memory. But as the cache memory is – compared to the main memory – very small, oversubscription forces to outsource the data from previous time slots to a slower memory. Accordingly, this critical problem forces the threads to compete for the cache memory, which in turn affects the performance of the program badly (Akhter and Roberts 2006).

4.1.1. Simulation

To reproduce the oversubscription problem, the following test scenario has been modelled:

The goal is to measure the time needed to count up a variable from zero to a user-specified limit in the main thread. During this counting up, a user-defined amount of worker threads calculate the Fibonacci numbers. By this simulation, we can measure how the worker threads impact the task of the main thread.

4.1.2. Measurement Analysis

A summary of the measurement results testing up to 70 threads is shown in Table 1, supplemented by the execution time and the speedup – a value for describing the ratio between the serial and multithreaded execution time. The results indicate that although the workload of the main thread does not change, the run-time behavior starts to get worse at 8 active threads.

Table 1: Summary of results of the oversubscription test

Run-time behavior of the test scenario		
Threads	Run-time [sec]	Speedup
1	0.0600001	1
8	0.155009	0.387074944
17	0.35402	0.169482233
25	1.00306	0.05981706
33	2.3088	0.025987569
49	5.60041	0.010713519
57	7.69081	0.007801532
65	10.3428	0.005801147
70	12.2304	0.004905817

Since 8 logical processors exist on the test system and hence 7 worker threads and the main thread can run simultaneously, oversubscription starts to occur only at 8 worker threads. This in turn describes the increase of the run-time exactly at this number of active threads.
The graphical representation of the simulation results is shown in Figure 2. The horizontal axis shows the number of active threads and the vertical axis represents the execution time in seconds. The diagram shows that the execution time progressively rises up in a curve for an increasing number of threads. A detailed view is illustrated for better readability in Figure 3, which shows the execution time up to 10 threads, where the run-time is almost nearly constant up to 7 threads.

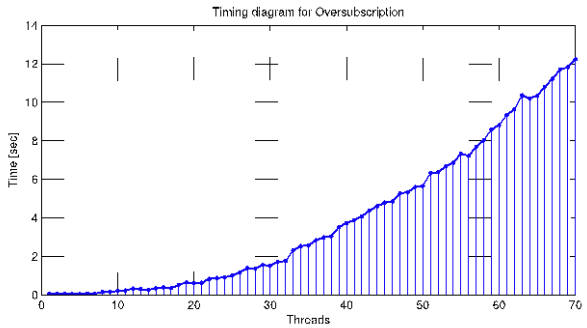

Figure 2: Run-time behavior of the oversubscription test scenario illustrated in a diagram.

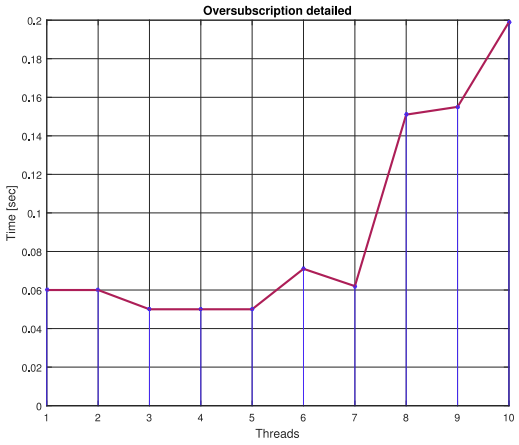

Figure 3: Detailed view of the execution time up to 10 threads.

To analyze what exactly happens when too many threads are active, a simulation with 40 threads has been started in the Concurrency Visualizer. The result illustrated in Figure 4 shows the various thread states at program execution. It shows that 77 % of the execution time is spent for the preemptions caused by the context switches and only 21 % of the run-time is actually used for the execution of the worker threads.

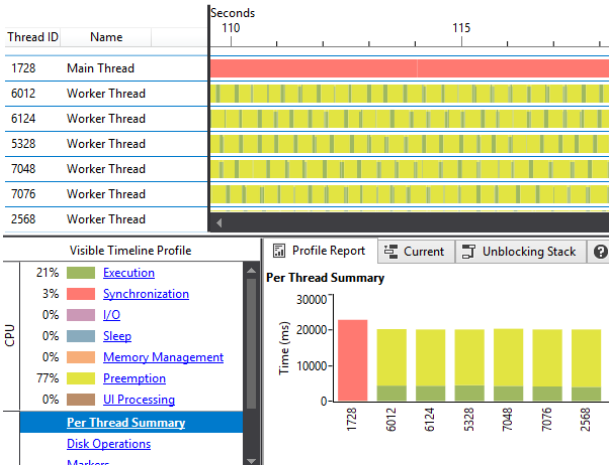

Figure 4: Concurrency Visualizer analyzing the test scenario for oversubscription.

4.1.3. Problem and Solution

The simulations show, how drastically the round-robin scheduling technique affects the run-time behavior. Generally spoken, oversubscription can be eliminated by setting the maximum amount of active threads to the actual number of logical processors.
Although, this does not affect the implemented test scenario, but a reason for an intentional or unintentional high number of active threads used in a program can also be due to poor load distribution. To avoid the side effects of the time slicing technique one can make use of the so-called work-stealing technique. The aim of this technique is to avoid unloaded threads as much as possible. The specific procedure based on (Hwu 2011) is shown in Figure 5 as a flow chart.
In the illustration, all the threads have their own work-pool. If a thread has already processed a task, then it tries to take another one of its own tasks. If during run-time new subtasks have been created, then these are also stored into the thread's collection. Since the subtasks do often share data with the main task a better cache utilization is achieved (Hwu 2011).

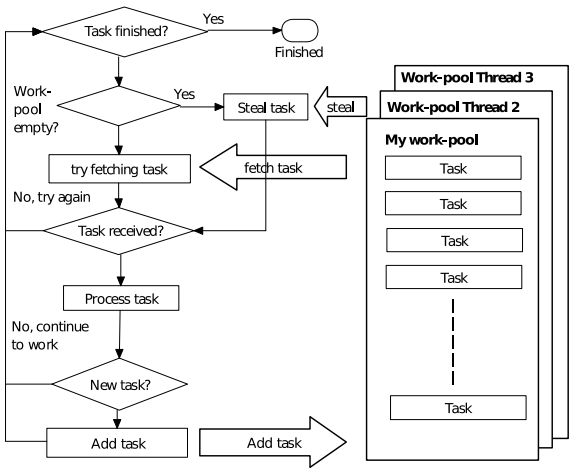

Figure 5: Flow chart illustration of the work-stealing technique.

If a thread has already executed all tasks in its own collection, then it tries to "steal" tasks from another thread by randomly looking into the collections of other threads. Since this search of tasks causes a certain amount of effort, it is the goal of this technique to provide the "thieves" big tasks to keep them busy as long as possible.

4.2. False Sharing

Due to the special hardware architecture of multi-core systems and the limited amount of memory, performance bottlenecks can also lead to problems on software-level or operating system level, and also on the hardware-level.

The cache offers a very fast memory for outsourcing frequently used data of the processors. Since the modern cache-memories are well suited for sequential programming, these fast memories are one of the main causes for substantial side effects in the parallel software development. One of these effects respectively problems is cache line ping-pong (Akhter and Roberts 2006).

This problem occurs when multiple hardware threads try to perform operations on data in the same cache line. Therefore, it can happen that during an ongoing operation on two or more cores, the cache line gets copied like in a ping-pong game between the different cores of the processor through the memory bus (Akhter and Roberts 2006).

One of the causes of cache line ping-pong is false sharing. Since the cache line is the smallest storage unit in data transfer through the memory bus, multiple hardware threads have often a local copy of exactly the same cache line. This does not represent an issue when only a reading access is performed on the cache-line, but is very critical on write operations (Gleim and Schüle 2012).

Although, in false sharing each hardware thread works in disjoint memory positions on their own copy of the cache line, all changes get invalidated in all copies to maintain the cache coherency (Gleim and Schüle 2012).

Thus, the cache coherence protocol ensures that the cache line gets exchanged between the cores on every change of the cache line.

For this purpose, the modified line has to be written back to the main memory, so it can be re-copied to all cache memories of the other hardware threads (Tian and Chiu-Pi Shih 2012).

This sequence is shown with reference to (Tian and Chiu-Pi Shih 2012) in Figure 6:

Two hardware threads, T1 and T2, load a cache line from the shared L2-cache in their own L1-cache. Thread T1 modifies a part of the cache line. This line gets invalidated by the cache coherency protocol and set to dirty (Tian and Chiu-Pi Shih 2012). Thread T2 wants to change an element in a disjoint memory location in the same copy of the cache line, but to maintain the cache coherency, the cache line from the core 0 is copied through the memory bus into the shared L2-cache. T2 loads the updated line from the L2-cache in its own L1-cache and writes its data into the memory location (Tian and Chiu-Pi Shih 2012).

False sharing, therefore, strains the memory bus, although looking at the source code no dependencies of the data can be seen at the first glance.

Furthermore, the advantage of using the cache for fast data storage gets nullified, whereupon subsequently there is a substantial performance loss (Gove 2010).

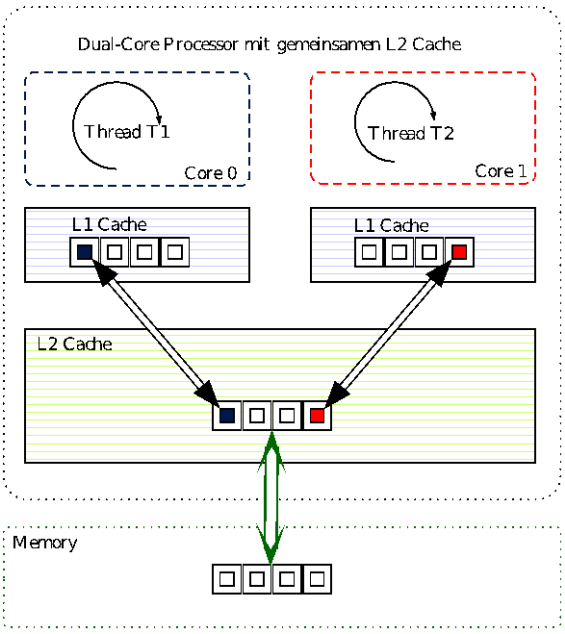

Figure 6: Visualization of the false sharing problem on a dual-core processor with shared L2-cache.

4.2.1. Simulation

In order to reproduce the false sharing problem, the created threads have to share the same cache line but work in different memory locations. For this reason, the following scenario has been modelled:

Several robots in a factory which are simulated by threads can only move in a rectilinear direction, which are controlled by an external program. This controlling

program fetches at certain intervals the current distance compared to the last measurement and stores for each robot the received data into disjoint memory locations into a random access container. In case of failure, these received values are used to reset all the robots back to their starting position, so the robots can proceed with their task again.
The data type for the distance is *int64_t* which has the size of 8 bytes on the test system. The cache line size is 64 bytes. Thus, up to 8 threads can work in the same cache line.

4.2.2. Measurement Analysis

In Table 2 a summary of the run-time behavior and the speedup can be found. The measured values show that the run-time increases linearly with the increasing number of threads, although the threads – of a semantic point of view – work on independent variables having write access to their memory locations.
This worsening of the run-time is also shown in Figure 7, in which on the horizontal axis the number of threads and the run-time in seconds on the vertical axis is plotted.

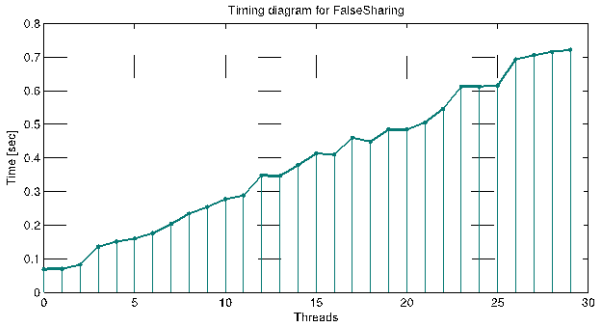

Figure 7: Run-time behavior for the false sharing test scenario illustrated in a diagram.

Table 2: Summary of results of the false sharing test

Run-time behavior of the test scenario		
Threads	Run-time [sec]	Speedup
0	0.0690473	1
1	0.070049	0.98570001
2	0.0820556	0.841469686
4	0.151098	0.456970311
7	0.203136	0.339906762
8	0.234155	0.294878606
10	0.277185	0.249101863
20	0.484326	0.142563686

4.2.3. Problem and Solution

The biggest disadvantage with false sharing is that it is not easily recognizable at the first glance. Figure 8 shows the measurement results of the Concurrency Visualizer. The profiler indicates that the program is up to 72 % busy executing the program and the remaining 20 % of the run-time is lost due to the console output through the main thread. Even further analysis with the Concurrency Visualizer still does not give any hints about false sharing.

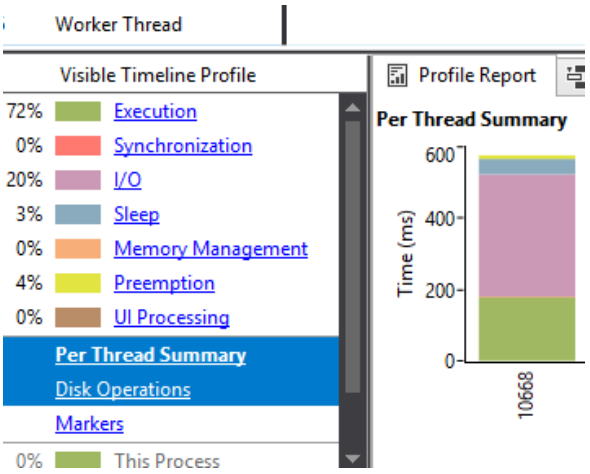

Figure 8: Results of the Concurrency Visualizer does not give any hints for false sharing.

Therefore, there is still the possibility to make use of the so-called hardware performance counters which are supported by many modern processors. Using suitable tools like the Visual Studio profiling tools, this counter can be used to determine the amount of cache misses. A high value of cache misses indicates false sharing.
The general solution for false sharing referring to the simulated test scenario is the following:
The implemented example uses a container of type *std::vector*, in which each thread respectively robot has write access to an element in the container. But each element is not large enough to contain a whole cache line, which in turn results in the cache line being constantly copied between the hardware threads on every change to preserve the cache coherency. For this reason, the software developer must take care of that each thread gets its own cache line for its changes (Gleim and Schüle 2012).
The solution to fix this problem is illustrated in Figure 9. In the implemented example, the basic data type of the container has been *int64_t*, which comprises 64 bits respectively 8 bytes on the test system. Since the test system has a cache line size of 64 bytes, to solve the problem we have to use the technique of spacing, in which we have to enlarge the container at least by a factor of 8. Thereafter, each thread can change an element with the multiplicative increment of 8 and with the ascending index of the threads. So the threads can access their own cache line without getting in conflict with the other threads.

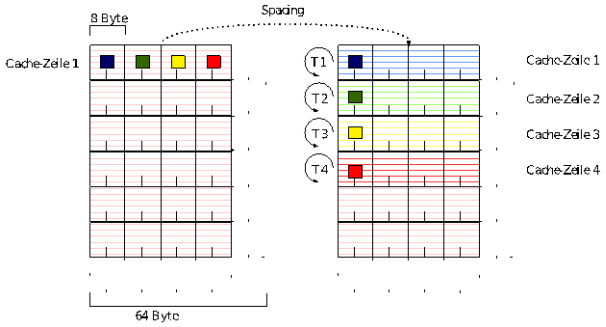

Figure 9: One possible way to solve the false sharing problem by using the spacing technique.

5. CONCLUSION

The detailed problem analysis of the selected multi-core specific problems has on the one hand shown that the well-documented problems of the former multi-processors are still applicable also for multi-core processors. On the other hand, it has to be remarked that it is not sufficient to only know and avoid general problems such as deadlocks and race conditions. The software developers need also to know about the processor, software and operating system used to avoid unconscious performance bottlenecks. They have to be familiar with modern parallel software development and must also have in-depth knowledge distributing tasks to multiple threads. Furthermore, the developers must be able to identify data dependencies and have to skillfully coordinate the involved threads by using appropriate synchronization techniques. The development trend of multi-core processors also shows that the number of such processors will increase further in the coming years. Thus, the developer must also learn to cope with this trend, so that the application does not only scale well with the problem size, but also with the increasing number of cores on such processors.

REFERENCES

Akhter S. and Roberts J. 2006. Multi-Core Programming: Increasing Performance through Software Multi-Threading. USA:Intel Press.

Blumofe R. and Leiserson C., 1999. Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM) 46:720-748.

Duffy J., 2008. Concurrent Programming on Windows. Boston:Addison-Wesley.

Gleim U. and Schüle T. 2012. Multicore-Software: Grundlagen, Architektur und Implementierung in C/C++, Java und C. dpunkt. Heidelberg:dpunkt.

Gove D., 2010. Multicore Application Programming: For Windows, Linux, and Oracle Solaris. Crawfordsville:Addison-Wesley Professional.

Hwu W.W., 2011. GPU Computing Gems, Volume 2. Boston:Elsevier.

Rauber T. and Rünger G. 2012. Parallele Programmierung. Heidelberg:Springer-Verlag.

Simple high resolution timer in C++ Available from https://gist.github.com/gongzhitaao/7062087 [March 2015]

Tanenbaum A.S., 2009. Moderne Betriebssysteme. München:Pearson Deutschland GmbH.

Tian T. and Chiu-Pi Shih., 2012. Software Techniques for Shared-Cache Multi-Core Systems. Available from https://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems [July 2015]

Torrellas J., Lam H.S., Hennessy J.L., 1994. False sharing and spatial locality in multiprocessor caches. IEEE Trans. Comput. 43:651–663.

Williams A., 2012. C++ Concurrency in Action: Practical Multithreading. Shelter Island:Manning.

Yanyong Zhang Y., Franke H., Moreira J., Sivasubramaniam A., 2003. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. IEEE Trans. Parallel Distrib. Syst. 14:236–247.