Utilizing Multi-cores for Reducing Response Times in Real-time Systems

Approved: ___JoachimWietzke_________________ Date: ____1/12/10_______
Committee Chair

Approved: ___ChristophWentzel________________ Date: ____1/7/10________
Committee Member

Approved: ___Joseph Clifton___________________ Date: ___1/6/10_________
Committee Member

Utilizing multi-cores for reducing response times in real-time systems

A Thesis Presented to

The Graduate Faculty University of Applied Science Darmstadt

And Presented to

The Graduate Faculty University of Wisconsin-Platteville

In Partial Fulfillment

Of the Requirements for the Degree Master

of Science in Computer Science.

By

Mario Shraiki

2009

DECLARATION OF CONFORMITY

I hereby declare that this thesis is written on my own solely by using the sources quoted in the thesis body. All contents (including text, figures, and tables), which are taken from sources, both verbally and in terms of meanings are clearly marked and referenced.

The thesis has not been previously submitted to any other examination board, neither in this nor in a similar form.

Signature: ____________________ Date: ____________

ACKNOWLEDGEMENTS

Eigentlich hatte ich vorgehabt eine kurze Arbeit zu schreiben, welche die Minimalanforderungen an Seitenzahl abdeckt. Da das leider nicht geklappt hat, habe ich mich entschlossen nun doch ein Vorwort zu schreiben.

Ich möchte mich an dieser Stelle bei den Personen bedanken die mich bei dieser Arbeit unterstützt haben.

Mein größter Dank gilt Samuel Arba Mosquera, der mich bei dieser Thesis hinsichtlich Struktur, Aufbau und formaler Schreibweise unterstützt hat und dem ich aus beruflicher Sicht viel zu verdanken habe. Sollten also noch Fehler vorhanden sein...

Desweiteren möchte ich mich bei meinen (langjährigen) Freunden Nico Triefenbach, Simon Wrobel (+Nora), Anton Daumlechner, Nathalie Schad und David Kramer bedanken. Sie machten sich nicht nur die Mühe diese Arbeit zu lesen, sondern versuchten auch diese zu verstehen.

Das Studium an der Hochschule Darmstadt hatte seine Höhen und Tiefen, von einigen Professoren blieb mir glücklicherweise einiges an Wissen erhalten.

Die für meinen Lebensweg wichtigsten werde ich kurz nennen. Herrn Prof. Dr. Wietzke, da er mein Interesse an Embedded-Systemen geweckt hat, und mich bei dieser Arbeit sehr gut unterstützt hat. Herrn Prof. Dr. Hahn verdanke ich das Wissen um Software strukturiert und geplant entwickeln zu können. Herrn Prof. Dr. del Pino, dessen Vorlesung mein Verständnis für Qualität erweitert hat. Herrn Prof. Dr. Wentzel, der sich immer die Zeit nahm als Fragen auftauchten.

One part of my study was the semester abroad. It was an interesting experience, hence culture, lessons and food were absolutely different. I want to thank Mike, Rob and Joe for the experience I have earned in the USA.

Special thank goes to Joe, for his sharp eyes and constructive criticism when reading my work. His constructive feedback triggered me to work harder on my thesis.

Ich möchte mich ebenfalls bei Herrn Schwind und bei Herrn Hollerbach bedanken, die mir diese Arbeit erst ermöglicht haben.

Utilizing multi-cores for reducing response times in real-time systems

Mario Shraiki
Under the Supervision of Prof. Dr. Joachim Wietzke, Prof. Dr. Joseph M. Clifton, and Prof. Dr. Christoph Wentzel

Statement of the Problem

The current trend for enhancing response times is parallelization. Reducing response-times can not be achieved with software written for single-core purposes. The software has to be *aware* of the additional cores, and must gain benefit from those. The shift from sequential programming toward multi-core programming influences not only the market for gaming and consumable devices, but also the one for medical and industrial applications.

The reduction of response time in single-core applications by code optimization is limited. By utilizing several cores, those limits can be overcome, because further resources can be allocated for that purpose. Furthermore, situations may exist in hard real-time environments, where it becomes impossible to keep all deadlines without the utilization of another core. But the usage of those additional resources and the shift to *real* parallel execution has its own difficulties. Problems will occur if cores are utilized incorrectly, and deadlines may not be kept.

Methods and Procedures

This thesis attempts to find efficient approaches for gaining additional benefit from several cores. For that purpose, a generic view at a multi-core system is introduced and evaluated. Each layer has its own impact on concurrency. Concepts and frameworks are evaluated for gaining benefit from multiple cores, i.e. whether those concepts and frameworks are suitable for a real-time environment

For being not limited to a special real-time system, their main limitations are evaluated as well.

Results

As this thesis will underline, utilizing several cores within a real-time environment is possible at different layers. Efficient frameworks exist, which can decompose work in order to gain benefit from additional cores. Frameworks like OpenMP, Intel TBB and Cilk can help to improve performance through work decomposition, and furthermore they are considered as scalable. For using them in a real-time environment, it has to be considered that attributes of the thread creation are done *automatically*. As such, it is not possible to influence the whole scheduling behavior related to those created threads. In OpenMP, Intel TBB and in CILK attributes like priority are inherited from the main thread. OpenMP and Intel TBB provide their own scheduling mechanisms. Those shall be set to static, to get a predictable result in time and work flow.

Contents

List of Figures

List of Tables

Listings

Part I

State-of-the-Art in real-time Multiprocessing

Chapter 1
Introduction

1.1 Outline

Advances in computer hardware technology improve system throughput and increase the computational speed in terms of millions of instructions per second (MIPS). Since costs to overcome physical limitations for enhancing single cores are higher than utilizing several cores, it is more economic to improve processing power with multi-core systems.[Chr]

Having a higher throughput, or having several cores on a system does not mean that timing constraints of an application will be met automatically. Those constraints are essential for real-time computing, where the individual timing requirement of each task has to be met. This objective is different, unlike fast computing where the average response time of a given set of tasks has to be minimized.

When several computational activities have different timing constraints, average performance has little significance for the correct behavior of the system. This situation is emphasized, when moving from single-core to multi-core hardware architectures.

1.2 Motivation

Multi-core systems are offering several benefits for developer. They are comparatively affordable compared to high-end single core platforms, have low power consumption, and good heat dissipation. Furthermore, higher frequency requires more power, making it harder and more expensive to cool down the system. In terms of robustness, it may be interesting to move from a single- to a multi-core platform.

Installing an application to a multi-core platform does not mean to get a benefit of those additional cores. In most cases, the application has to be modified or completely redesigned. The application, as well as the operating system must be *(*aware) of the additional cores.

CHAPTER 2

PURPOSES AND GOALS

2.1 Scope of this Thesis

This thesis covers issues, which must be considered for meeting real-time conditions, when moving from single- to multi-core hardware architectures. Techniques for reducing latency-, and computational-times by meeting real-time requirements are the main foci of this thesis.

2.2 Related work

Many approaches exist concerning multi-core technologies, most of them for the mass-market, not related to real-time systems.

This thesis tries to comprehend the essential requirements for real-time systems, and shows whether current state-of-the-art techniques exist for reducing the overall response time of the system in a multi-core system.

2.3 Style guide

In this work, quotations are written indented:

> Real-time systems are computing systems that must react within precise time constraints to events in the environment. As a consequence, the correct behavior of these systems depends not only on the value of the computation but also on the time at which the results are produced.[SR89]

The name of variables used in the text is written in courier font: `mNode`. Special words used in the text connoting a non-standard meaning are written in italic: *fast* Multiprocessing. Important words in the text emphasizing the focus are written in bold face: **Hazard pointer**.

References can be found at the end of the thesis and are formatted according to the AIP style guide. Citations in this work are written inline within square brackets with the reference in blue color: [SR89]. Footnotes are added at the end of each page in small fonts, and are numbered serially.

Part II

Fundamentals and Principles of *fast* real-time Multiprocessing

CHAPTER 3

FUNDAMENTALS

3.1 Definition of Real-Time

Timing requirements are especially important for real-time systems. External events, even when unpredictable for the system, must be handled in a predictable manner.

> Real-time systems are computing systems that must react within precise time constraints to events in the environment. As a consequence, the correct behavior of these systems depends not only on the value of the computation but also on the time at which the results are produced.[SR89]

An example for a predictable system with hard real-time constraints is the pacemaker. It is absolutely mandatory that every heartbeat comes in a controlled manner. Enhancing the overall computing time would not lead to the desired effect, but reducing the processing time T_p at a desired time T_0. In figure 3.1 an example for the time constraints is illustrated. *A* shows the desired heart beat for the patient, whereas *B* shows the approximation with processing time. If we enhance the computing time for the overall system[1] the heartbeat will change as shown in *C*. A real-time system has to be aware of its deadlines (see *D*), violating them is an error, which will lead to strict consequences to the real world. This example about hard real-time shows that real-time systems are related to the external environment, in that case the real world. It is reasonable to classify real-time systems into two domains:

hard real-time: A real-time task is said to be hard if missing its

[1]using a 1 GHz CPU instead of a 500 MHz one

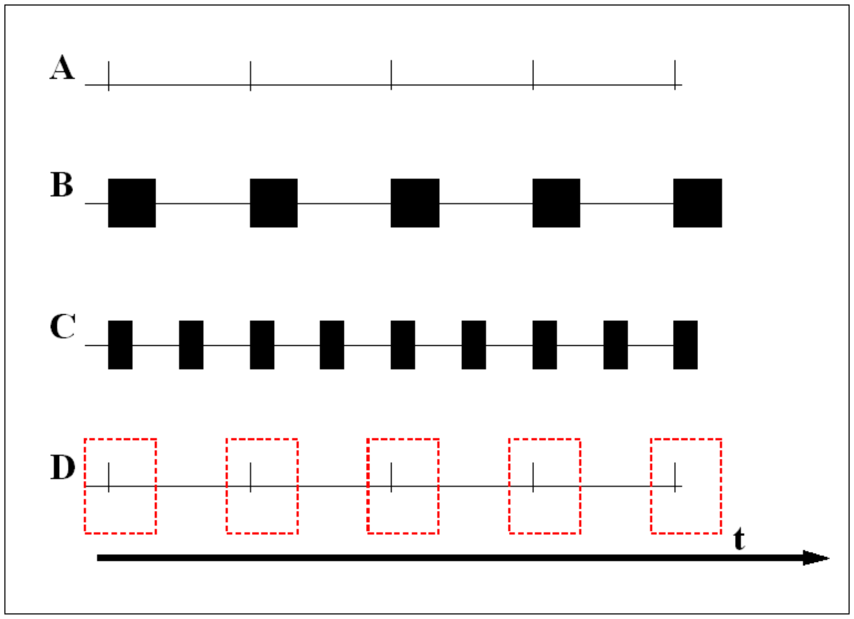


Figure 3.1: *Hard real-time requirements for a pacemaker.*

deadline may cause catastrophic consequences on the environment to control.[But04]

soft real-time: A real-time task is said to be soft if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment and does not jeopardize correct system behavior.[But04]

Typically, a real-time system consists of hard and soft real-time requirements. The system has to cover requirements where violating deadlines may cause catastrophic consequences and requirements where violating deadlines may not cause catastrophic consequences. These two aspects should be designed using different approaches. The system has to guarantee the in-

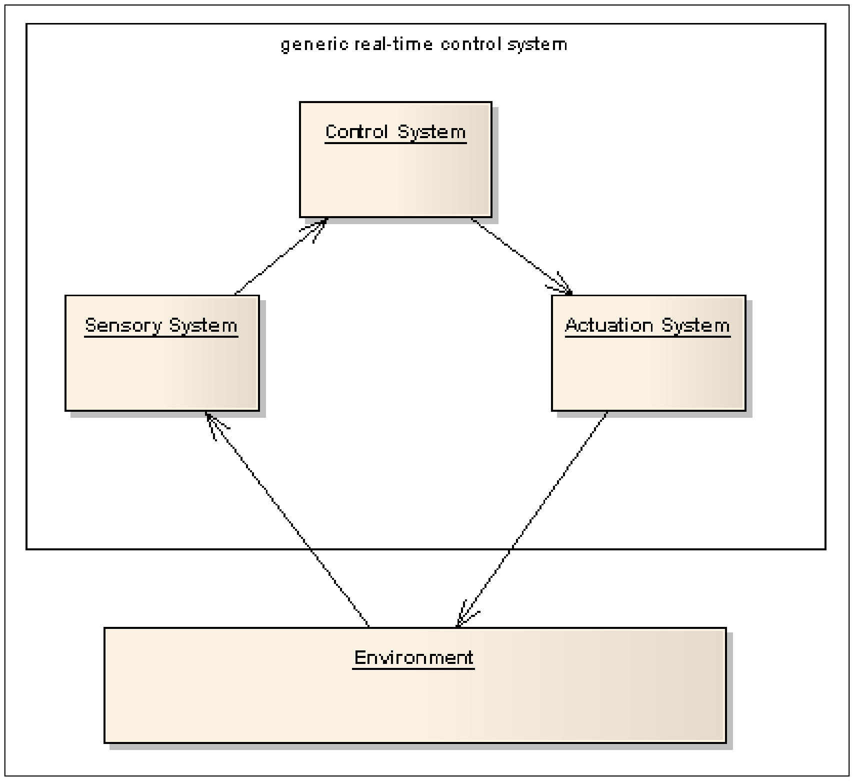


Figure 3.2: *Block diagram of a generic real-time control system (adapted from [But04]).*

dividual timing constraints of the hard tasks (see figure 3.2) while the average response time of the soft tasks should be reduced.

3.2 Multi-Core Technologies

Software is a language developed for communicating with hardware. Without having the proper hardware, real parallelism cannot be achieved. The purpose of this section is to describe the fundamentals of different hardware approaches and their impact on a real-time system.

> In order to achieve parallel execution in software, hardware must provide a platform that supports the simultaneous execution of multiple threads.[AR06]

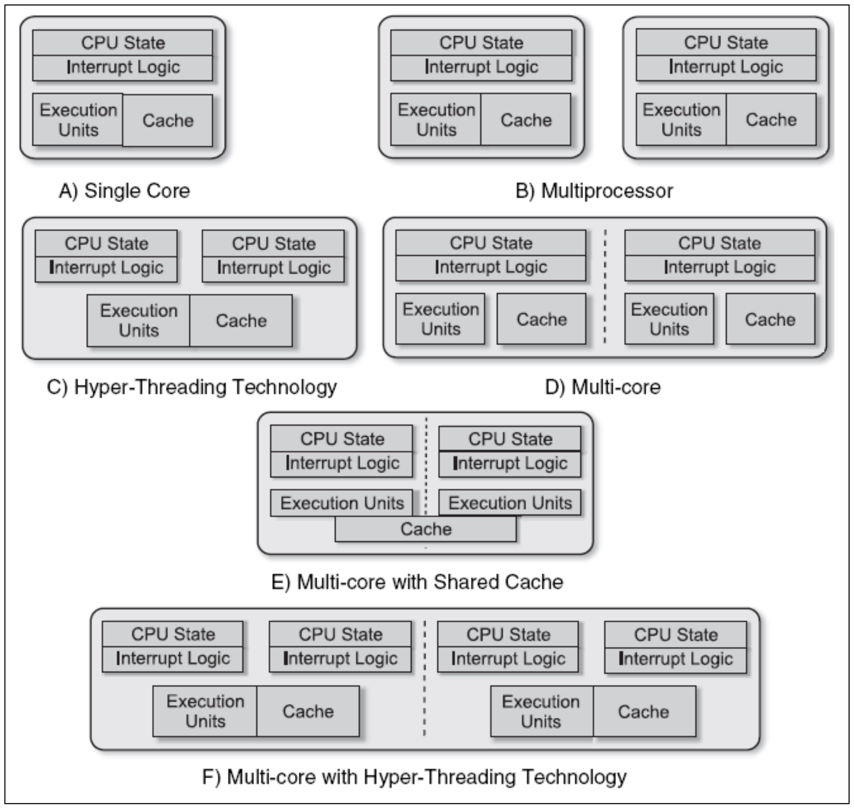

CPU State
Interrupt Logic
Execution Units
Cache
A) Single Core
B) Multiprocessor
C) Hyper-Threading Technology
D) Multi-core
E) Multi-core with Shared Cache
F) Multi-core with Hyper-Threading Technology

Figure 3.3: *Different hardware approaches for multiple cores.[AR06]*

For achieving real concurrency in terms of parallel tasks, hardware redundancy must exist. Without having redundant hardware, real parallelism will not be achieved.

Several programs can also be executed *at the same time* on single-core systems using multi-threading or multi-processing. One program will be stopped for a specific time and another one will be executed instead. Using this timesharing technique, a kind of pseudo parallelism is implemented. Real concurrency means that more than one program, application, or task is executed in parallel. This little difference has a significant impact on the whole design.

Depending on how much hardware is redundant or connected to each other and depending on the proposed application, the bus or the cache may become a bottleneck. Figure 3.3 shows different solutions for additional hardware in order to run programs in parallel.

Hyper-Threading: is a simultaneous multi-threading (SMT) technology. It enhances the utilization of processor pipelines by filling leaks within pipelines with commands from another thread. *Leaks* within a pipeline can occur when data in cache is missing and has to be recovered from main memory.

Cache: is an important part of the hardware multi-core architecture. Cache satisfies the need for fast data access. Current CPUs are beyond the range of 1GHz, whereas main memory lies in the range of several 100MHz. Cache provides fast access to already stored data compared to main memory. Efficient use of the cache can improve the accessing time for required data. This reduces the CPUs idle time and improves the effi-

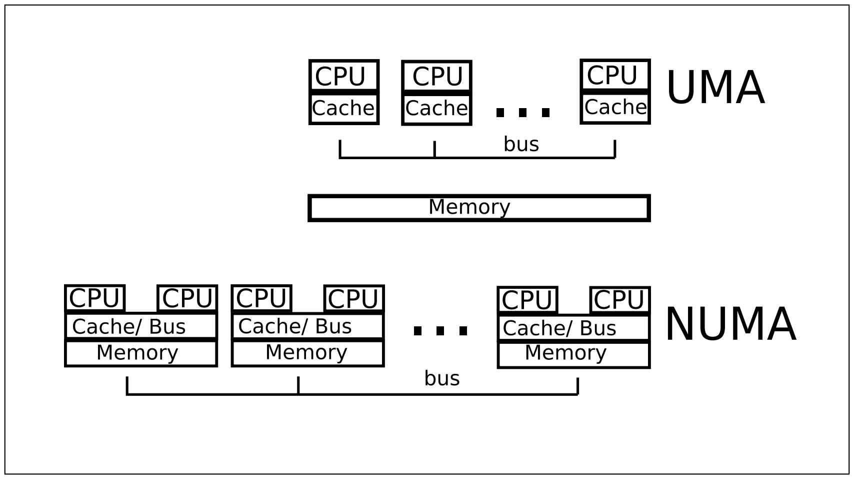


Figure 3.4: *Different approaches for memory access.*

> ciency of the system by optimizing the utilization of the CPU. This effect will be emphasized for each additional core in a multi-core system, because more cores have to acquire data for their further progress.

Data access is important for the behavior in the system. In terms of a multi-core system, data has to be modified and changed *in parallel*. The most of the data is stored in the main memory of a system. Figure 3.4 gives an overview on how main memory can be accessed. The principles are known as Non-Uniform-Memory-Access (NUMA) and Uniform-Memory-Access (UMA).[AR06]

> The term shared memory refers to the fact that the address space is shared; that is, the same physical address on two processors refers to the same location in memory. Shared memory does not mean that there is a single, centralized memory. In contrast to the symmetric shared-memory multiprocessors,

> also known as UMAs (uniform memory access), the DSM[2] multiprocessors are also called NUMAs (nonuniform memory access), since the access time depends on the location of a data word in memory.[HP03]

In a NUMA system, the response time of a memory access depends on the actual distance between the processor and the random-access-memory (RAM), although it is the same for each processor on the same node.

For the UMA system, the response time for memory accesses is the same all over the system.

[2]DSM stands for distributed shared memory.

3.3 Multi-Core Operating Systems

The work with additional cores is an additional overhead for an operating system (OS). The operating system must have specific features in order to interact with additional cores. Several approaches exist for that purpose.

AMP: Asymmetric multiprocessing uses a separate copy of an OS on a core. In most cases the software process is locked to that single core which provides an execution environment similar to an uniprocessor system. Code can be simply migrated, because the software process behaves like a uniprocessor system. One of the drawbacks is that all shared Input-Output (IO) devices, like an Ethernet port, have to be managed explicitly between the OS.

Each OS has full access to all IO (see figure 3.5) but is not aware of another OS.

SMP: Symmetric multiprocessing places the OS on one core, which handles the other ones. This means that one core suffers from additional overhead, but has as advantage that all shared resources are known by the OS, so that the access from several cores to e.g. Ethernet can be managed. Furthermore, the system becomes scalable through the OS because the scheduling of processes is distributed to the available cores.

One OS has full access to all IO (see figure 3.6).

BMP: Bound multiprocessing uses the same concept as SMP, but it is possible to bind a process to a specific core. The problem of the shared IO is handled as in SMP.

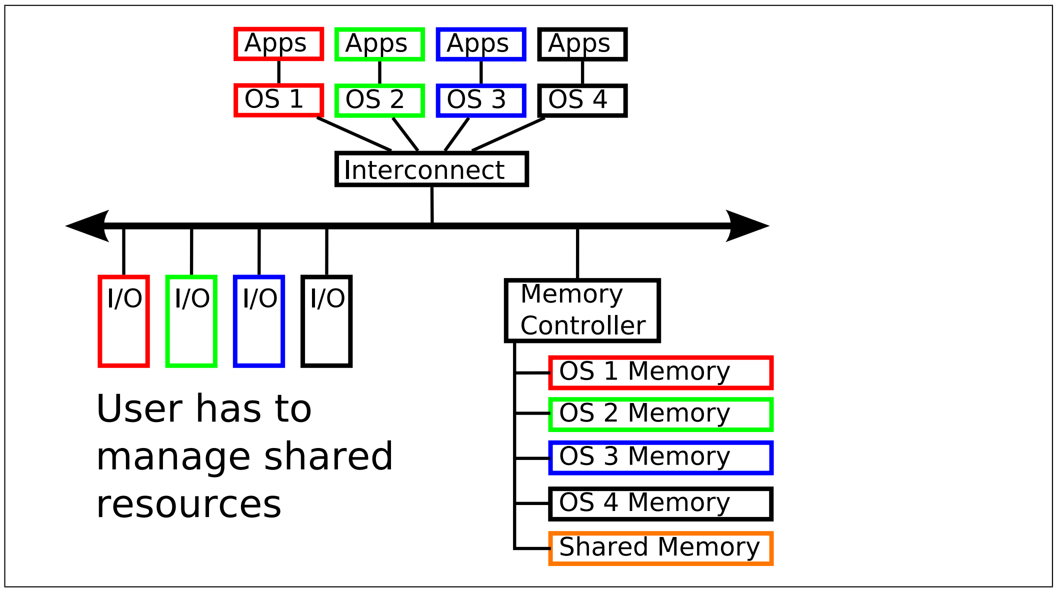


Figure 3.5: *AMP: User has to share IO and memory explicitly.*

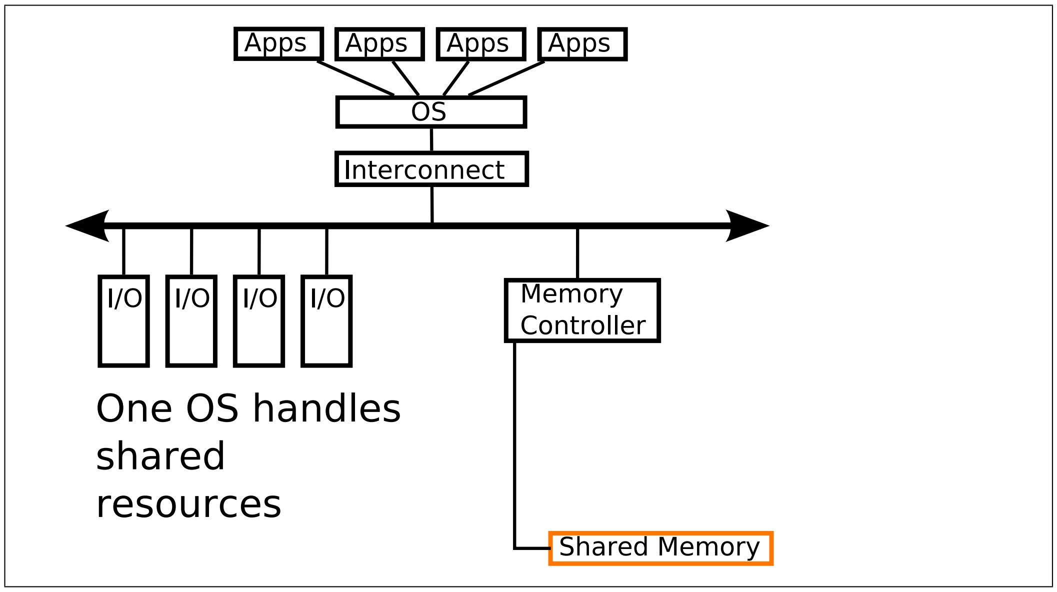


Figure 3.6: *SMP: OS handles shared resources.*

If resources like IO are not shared and there are no dependencies between the processes, AMP offers the most efficient approach for the utilizing the additional cores. In theory, the same amount of overhead is introduced from the OS at each core. Practically, it depends on the hardware architecture (especially UMA, NUMA). But migrating independent single core tasks to a multi-core system will enhance response times of the system. If deadlines are not violated, this will be the simplest and most efficient approach to utilize multi-cores.

Having shared IO will complicate the attempt to migrate to a multi-core platform. SMP introduces not only the overhead of the OS, but furthermore a scheduling overhead because the additional cores must be managed from the OS. So a set of tasks T has to be distributed to several cores C. Having only one core simplifies the management procedure, having several cores will increase the scheduling overhead. SMP will be the best choice if dependencies between processes exist, and if resources are shared. It introduces additional overhead because of the process management, but simplifies dependency management.

BMP adds an overhead but provides an additional advantage compared to classical SMP. The additional overhead comes from the fact that processes have to be bound to a special core. If e.g. a process A shall only run at core C_2, a mechanism has to be implemented to prohibit the use on e.g. C_0. This leads to the advantage that migrations[3] can be limited. For instance, if there is a migration overhead on a specific calculation, because several other processes are running in parallel, it is possible to assign one core for that critical

[3]simplified: a context switch between processors

operation, which will reduce scheduling overhead and therefore calculation time[4].

[4]Compared to classical SMP.

3.4 Dependencies between thread-safe, reentrant, atomic and recursive

Multi-processing or multi-threading has its assets, as well as its drawbacks. One of those drawbacks is the possibility of corrupting data during parallel or even semi-parallel[5] access.

A function executed by `Thread A` can be interrupted by `Thread B`. For this example, `Thread A` and `Thread B` will execute the same function `incCounter()`. If this function uses global variables or static variables the possibility of corrupting data exists, because data is shared between competing threads. Listing 3.1 shall show an example for the undesired behavior of a function.

The increment for `GLOB` is not atomic because it can be interrupted by the scheduler. This will occur whenever a time-slice is consumed or a higher priority thread arrives.

Listing 3.1: *Incrementing a single-word counter.*

```
Int64 incCounter(){
  static Int16 GLOB = 0;
  GLOB++;
  return GLOB;
}
```

Listing 3.2 shows the assembler operations for the single `GLOB++` increment statement.

Listing 3.2: *GLOB as a 16 bit variable at an x86 architecture with a 32 bit memory bus.*

[5]Through scheduling mechanisms like preemption.

```
mov [GLOB], eax
add 0x1, eax
mov eax, [GLOB]
```

Depending on the word-size of the variable `GLOB`, the amount of instructions may become more.

Listing 3.3: *GLOB as 64 bit variable at an x86 architecture with a 32 bit memory bus.*

```
mov  0x804c480,eax
mov  0x804c484,edx
add  0x1,eax
adc  0x0,edx
mov  eax,0x804c480
mov  edx,0x804c484
```

Each of those operations consumes time. The operation itself might be interrupted and data might be modified. In case of a 64 bit variable e.g. of type `long long`, the first bytes might be written, `GLOB` is changed and further progress is stopped because of preemption. This will lead normally to data corruption.

The listing 3.2 will not have data corruption, but if the value of `GLOB` is modified at the first instruction `mov [GLOB], eax`, the resulting value for that function might become invalid.

Functions as shown above are known as non-reentrant and non-thread-safe.

A solution within a single core system is to change the scheduling from round-robin to FIFO[6].[Sup]

[6]Also known as FIFO scheduling trick.

Since no time-slicing mechanism is used, a thread will only be interrupted by the scheduler if a higher priority event arrives. A thread has to complete its whole work, or it has to switch using calls as `sched_yield()`. This is an efficient approach on single core systems.

In a multi-core system this solution will mostly not work, because several threads might be running in parallel.

Shared data have to be protected by synchronization mechanisms or other techniques, when they are accessible by more than one thread. Different synchronization mechanisms exist for that purpose, each of them with their own advantages and disadvantages. Functions must fulfill special requirements in order to access shared data from different threads.

Thread-safe: A function is considered as thread-safe if it can be called by different threads without an unwanted interaction between the threads.

The attribute thread-safe in a function has no implications on whether this function is intended to be fast. Thread-safe is, in most cases, sequencing of concurrent access. This can lead to bottlenecks in a multi-threaded environment. If several threads are intended to perform modifications on a shared resource, one of the following must be ensured:

1. Only one thread has access rights for the shared resource at a time, other threads have to wait.

2. The modification must be completed before another thread starts modifying as well.

In both cases a kind of sequencing exists, because simultaneous access may lead to data corruption.

Providing *access rights* to shared resources is typically done by using synchronization mechanisms. Those have to be provided by the operating system, and have direct impact on the scheduling of the system. The most common synchronization primitives are:

- Semaphores
- Mutexes
- Memory-Lock[7]
- Spin-locks (busy-waiting)

The following listing shows one conceptual approach to make a function call thread-safe, by utilizing synchronization primitives.

```
Int64 incCounter(){
  mutex.take();
  static Int64 GLOB = 0;
  GLOB++;
  mutex.give();
  return GLOB;
}
```

Nearly each synchronization mechanism uses a system call, so thread-safe implementations have a cost of performance. Calling and releasing synchronization mechanisms cost time. During this time, other threads attempting access to the shared resource wait until execution has completed. Depending on the

[7]Commonly used in the embedded world

amount of threads waiting for that resource, and depending on the time for waiting, those costs could be too high for the overall system performance.

Another approach for solving that issue is the use of atomic primitives.

Atomic: Actions are atomic if they can be considered, so far as other processes are concerned, to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent.[BW01]

This approach, which works normally only for small data (single or double words), is typically performed in few processor cycles. Furthermore, system calls for the synchronization mechanisms do not exist. Those atomic calls are very fast if they are provided by the hardware architecture. The following example shows the use of such an implementation utilizing atomic primitives.

Listing 3.4: *Using atomic primitives.*

```
Int64 incCounter(){
  static Int64 GLOB = 0;
  atomic_add(&GLOB, 0x01);
  return GLOB;
}
```

The appliance of atomic primitives comprises new risks and problems. The listing 3.4 is using such a primitive, but is not thread-safe because a wrong value could be returned. Considering the following case:

1. Thread A and Thread B are executing function `incCounter()`.

2. The value of `GLOB` is 0.

3. Thread A executes `atomic_add(&GLOB, 0x01);` (`GLOB` value is 1).

4. Thread A is preempted by Thread B (`GLOB` value is 1).

5. Thread B performs function `incCounter()` and returns 2 (`GLOB` value is 2).

6. Thread A completes function `incCounter()` and returns 2 (`GLOB` value is 2).

The return value is needed for handling that issue and for making the function `incCounter()` thread-safe. The QNX implementation provides `atomic_add_value()` for that purpose. That implementation is slower then the pure `atomic_add()` implementation, but returns the original value before the atomic addition was performed. The value was incremented, so 1 is added *manually*. This will not lead to further data problems; hence `tmp` is allocated on the stack, and therefore just visible for the current context of the corresponding thread.

```
Int64 incCounter(){
  static Int64 GLOB = 0;
  Int64 tmp = atomic_add_value(&GLOB, 0x01)+1;
  return tmp;
}
```

A thread-safe function can be called simultaneously by multiple threads when each invocation references to shared data. Normally this access to the shared data is serialized.

Reentrant: A function is called reentrant if it is thread-safe and does not use any synchronization mechanism.

Recursive: A function is considered recursive if it calls itself.

Recursive functions shall be reentrant in order to work properly, but not all reentrant functions need to be recursive. The main difference between a thread-safe function and a reentrant function is that threads are not blocked in a reentrant function. Many algorithms exist which are implemented by the use of recursive functions (e.g. divide-and-conquer algorithms). The concepts used in those algorithms can be simply enhanced by multi-processing if their functions are reentrant.

If those functions are only thread-safe, serialization will be emphasized and can lead to efficiency issues. Other threads can not perform their work because it will lead to data corruption.

Having them reentrant means that they are executed with a less amount of serial code and may gain benefit by additional cores (e.g. in Cilk parallelized implementations). It has to be considered that not all problems can be efficiently solved in parallel. Some algorithms are faster if they are performed serially.

The proper understanding of those terms is crucial for handling complexity with multi-core systems. If not properly understood, a system may work properly in 99% of all cases, but sometimes might not. The example with the `incCounter` function shows how many additional circumstances shall be considered in order to build a thread-safe or a reentrant function. Atomic operations are very efficient calls, which must be supported by the hardware. If supported, these are by far faster than synchronization mechanisms like mutexes or semaphores. For gaining further achievements in terms of speed, frameworks or libraries for multiprocessing system should not only be thread-safe, they should be reentrant[8]. Otherwise, bottlenecks will occur in a manner that one running thread will block another one.

[8]At least frequently called functions.

3.5 Non-Blocking Algorithms

Synchronization primitives have their advantages as well as their disadvantages. An advantage is their *simplicity* to use, but hidden complexity and deadlocks, life-locks, priority-inversion and sequencing limit their use in terms of efficiency. Researchers have also looked for other solutions in order to minimize the use of synchronization mechanisms. Lock-free mechanisms are frequently used in kernel implementations (see [McK03] for read-copy-update (RCU) mechanisms in the Linux kernel implementation), or in the QNX kernel (like the scheduling trick for SMP scheduling) implementation.

> Although the Linux 2.6 kernel offers much improved real-time response, performance, and scalability when compared to the 2.4 kernel, it does not offer hard real-time response, nor has it been able to provide scheduling latencies below about 100 microseconds. However, this is changing with the advent of two new approaches.[MS05]

Those mechanisms are more complex than those for mutual exclusion, like shown for the reentrant functions in Part I. The implementations have to be reentrant in order to be non-blocking.

Traditionally, there are two basic levels of non-blocking algorithms, called lock-free and wait-free. Wait-free and lock-free algorithms can be used in advanced data-structures for multi-processing and multi-threading purposes. Those algorithms have several benefits in terms of scalability and performance. Wait-free algorithms are performed within a specific time-window.

This makes them interesting for their use in a real-time environment. In lock-free algorithms, at least one task is ensured to make progress.[Sun04]

Wait-free algorithms fulfill hard real-time conditions. They are deterministic and executed within a specific time-frame, regardless of the actual level of concurrency. This means directly that they are not only reentrant, but also terminate after a finite number of execution steps. Many implementations are using RCU, where an object is copied into different buffers and redirected into separate buffers for avoiding conflicts. The buffer with the latest value is activated using an atomic call like compare-and-swap (CAS) or another mechanism. An example for the wait-free RCU mechanisms, can be found in the Linux kernel implementation. Further details can be found at [Sun04]. Lock-free and wait-free algorithms promise huge performance since can be performed *in parallel*. Both types of algorithms are not using synchronization primitives like mutexes or semaphores.

Figure 3.7 shows the time behavior of wait-free algorithms. As shown in figure 3.7, each algorithm terminates at a specified time-frame, which make this kind of algorithms interesting for hard real-time systems.

Lock-free implementations ensure that at least one instance is making progress, whereas in wait-free implementations all instances are making progress. As shown in figure 3.8 their behavior is to poll if an operation cannot be performed. This is not desired in a hard real-time system, where those hard real-time conditions might lead to catastrophic circumstances. Solutions exist to overcome this problem, by e.g. implementing a counter for reducing the amount of spins.

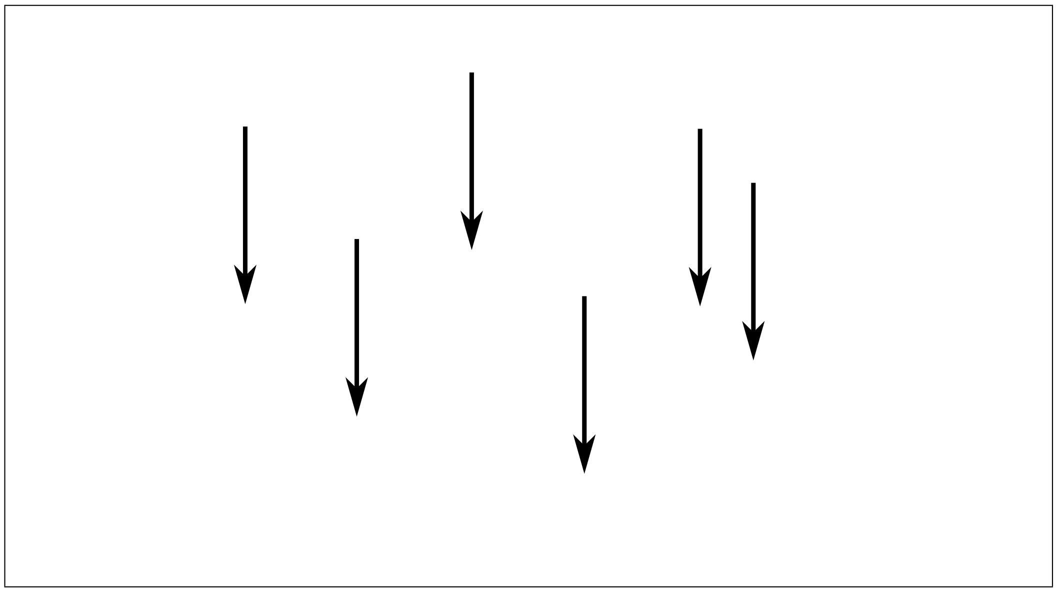


Figure 3.7: *Time behavior of wait-free algorithms (adapted from [Sun04]).*

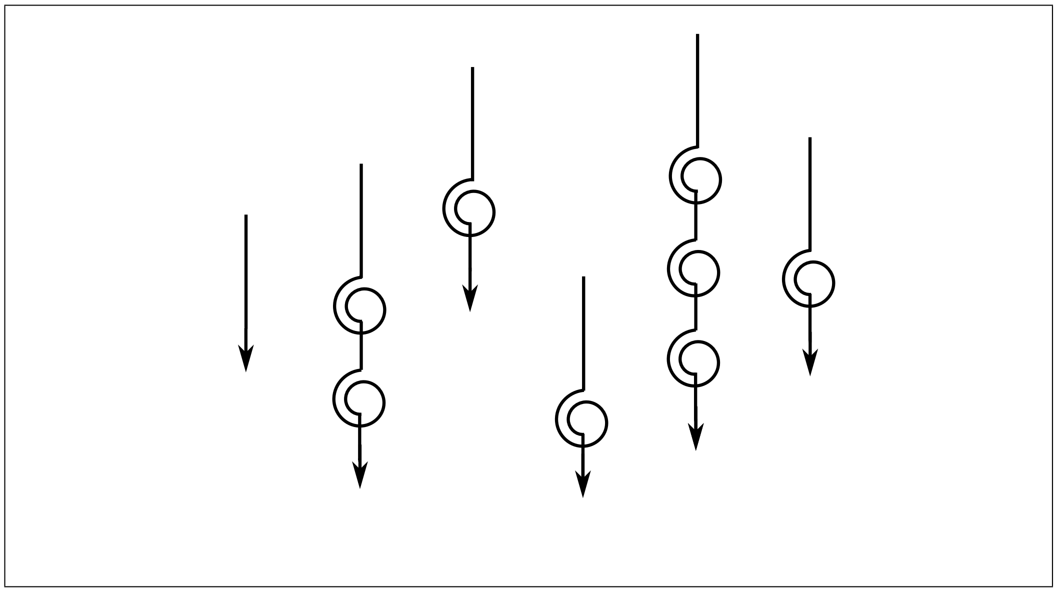


Figure 3.8: *Time behavior of lock-free algorithms (adapted from [Sun04]).*

Comparing lock-free algorithms with algorithms using locking primitives, the former have several benefits:

- They are scalable (several threads can execute them at the same time).
- They do not lead to priority inversion.
- They do not lead to deadlocks.
- They need no interaction with the operating system.
- They are fast.

By providing all these benefits, they also have their disadvantages:

- They are more complicated to program (ABA Problem).
- They are using (in most cases) atomic primitives, which must be supported by hardware.
- Lock-free implementation can lead to spinning.

ABA Problem: The ABA problem occurs when a thread reads a value A from a shared location, and then other threads change the location to a different value, say B, and then back to A again. Later, when the original thread checks the location, e.g., using read or CAS[9], the comparison succeeds, and the thread erroneously proceeds under the assumption that the location has not changed since the thread reads it earlier. As a result, the thread may corrupt the object or return a wrong result.[Mic04]

[9]CAS stands for compare-and-swap.

For illustrating the ABA problem, an example implementation of a simple linked list is shown. See figure 1 for a detailed work flow of the implementation. A basic node is defined, where `mNext` is a pointer to the next Node and `mData` a piece of information, which has to be stored into this linked list. A header `HEAD` exists, which points to the latest value. This header is stored on the data-segment (DS).

Listing 3.5: *Structure of the Stack (slightly modified from [Lan05]).*

```
struct CNode {
  CNode * mNext;
  Int32 mData;
};

CNode * HEAD;
```

Two operations are defined to add and remove elements from the stack. These operations are `push` and `pop`. `push` adds a new value onto the head of the stack, and `pop` retrieves a value from the head. Both operations are using the atomic call CAS for securing data integrity. The method `cas(a,b,c)` takes 3 parameters. The first parameter `a` is a memory region[10], which has to be compared against parameter `c`. If this comparison succeeds, the value of `a` is overwritten with the value of `b`. In case of a failure the return value of this function will be false, otherwise it will be true.

Listing 3.6: *Adding elements lock-free (slightly modified from [Lan05]).*

```
void push(Int32 t) {
  CNode* node = new CNode(t);
  do {
    node->mNext = HEAD;
  } while (!cas(&HEAD, node, node->mNext));
```

[10]Should not be stored in a register, so it has to be volatile.

```
}
```

As shown in listing 3.6, `node` is a pointer stored on the stack, pointing to a new value `t` on the heap. The pointer to the next Node `node->mNext` is allocated on the heap and will point to the same address as the global variable `HEAD` (after the assignment). In the example of the `push` function, the current location is where `HEAD` points to, compared (using `cas`) with the location where `node->mNext` points to. After the assignment `node->mNext = HEAD`, is the address stored in variable `HEAD` copied to `node->mNext`. So `cas` compares two addresses, not their values. If the addresses are equal, `HEAD` will be set to point to the new value on the heap `node`.

The same technique is used for the `pop` method.

Listing 3.7: *Removing elements lock-free (slightly modified from [Lan05]).*

```
bool pop(Int32& t) {
  CNode* current = HEAD;
  while(current) {
  // HEAD is compared with current
    if(cas(&HEAD, current->mNext, current)) {
      t = current->mData;

      delete current; // the node is returned to the heap
      return true;
    }
    current = HEAD;
  }
return false;
}
```

As shown in listing 3.7, a pointer of type `CNode` is allocated on the stack pointing to `HEAD`. It is checked whether `current` and `HEAD` are equal. If those values are not equal, it means that another thread has already updated the stack structure (using `push` or `pop`). When the structure is changed, `current` will be updated with the new value of `HEAD`. Otherwise, if both values (`current` and `HEAD`) are equal, the value will be retrieved from `HEAD` and returned. As already explained for the `push()` method, comparison is done for pointers (locations) and not for values. Problems can occur if locations in memory are reused by the memory manager. See figure 2 for a detailed work flow, which will lead to the so called ABA problem.

Listing 3.8: *Explaining the ABA problem.[Lan05]*

```
Thread 1:                    Thread 2:
pop()
read A from HEAD
store A.next "somewhere"
                             pop()
                             pops A, discards it
                             First element becomes B
                             memory manager recycles
                             "A" into new variable
                             pop(): pops B
                             push(HEAD, A)
CAS with A succeeds
```

If a location is reused (see listing 3.8) before a `cas` operation, the location could be the same one, but the value might be changed. This will not lead to a program error, but might lead to a semantic or logic error, which cannot be recovered from that easily.

Those kinds of errors are in most cases sporadic and difficult to find. It might be that the system performs well, but the logical error might lead to serious issues.

3.6 Summary

This chapter showed that real-time systems have different requirements than non real-time systems. The relation to time in order to meet deadlines is important for those systems. Violating those deadlines can lead to catastrophic consequences. In order to meet them, the system has to be deterministic and well defined.

Several tasks can only be performed in parallel when hardware is redundant. There is a difference between the time-sharing of two processes on a single core, and the time-sharing of two processes on a dual-core system.

This scheduling behavior is defined by the use of an operating system. Without the support of the operating system to exploit several cores, it can become difficult to use several cores as efficient as only one single core.

The terminology for thread-safe, reentrant, atomic and recursive was discussed. Having a thread-safe function does not mean that this function scales well in a multi-core environment. By using locking mechanisms, this function will be locked while in use and minimizes the parallel access for that time.

New types of algorithms were introduced, which do not suffer from locking mechanisms. With those algorithms parallel access is possible, but they are complicated to create. New problems might occur like the ABA problem, which was analyzed to give a hint about the complexity of those algorithms.

Chapter 4
Analysis

4.1 Layers influencing concurrency

Layers encapsulate functionality in order to reduce complexity. This is also the case when using an operating system. The operating system provides the Software Developer with services and functions. It encapsulates the direct hardware access (see figure 4.1). This enlightens programming efforts but reduces the overall system performance by adding an initial overhead (layer) to the application.

It is mandatory to have additional layers for encapsulating functionality toward a specific domain. Otherwise the time-to-market and robustness of the system would suffer. There are at least two approaches for improving the response-time of the overall application. The first one is a technical approach using different hardware or faster primitives. The second one is a conceptual approach using a different concept e.g. scattering and gathering data. Both have their advantages, but also their drawbacks. Figure 4.1 shall give an overview of the specific domains from the perspective of a Software Developer.

Layer 0: Represents the hardware. Hardware is the first step for achieving real concurrency. Modifications of this layer can be achieved by e.g. BIOS settings[1], or jumper switches. It requires specific knowledge about the hardware technology (e.g. hyper-threading, caching, or power-save options) and its related impact on the system.

Layer 1: Represents the Operating System. It is important to choose the correct operating system for the intended use. Real-time operating sys-

[1]BIOS settings are changing the behavior of the hardware.

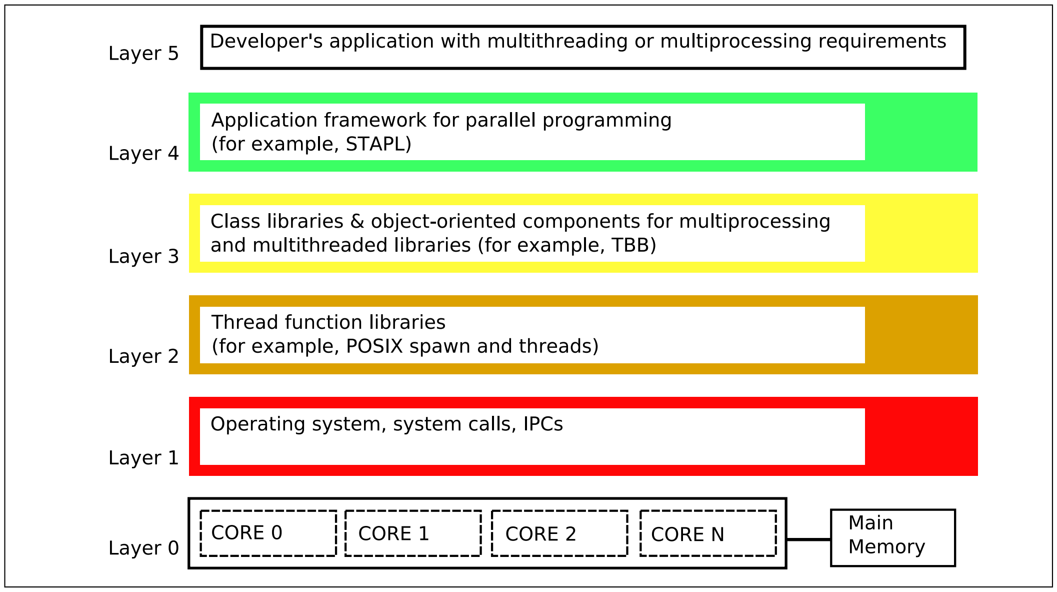

Layer 5
Developer's application with multithreading or multiprocessing requirements
Layer 4
Application framework for parallel programming (for example, STAPL)
Layer 3
Class libraries & object-oriented components for multiprocessing and multithreaded libraries (for example, TBB)
Layer 2
Thread function libraries (for example, POSIX spawn and threads)
Layer 1
Operating system, system calls, IPCs
Layer 0
CORE 0
CORE 1
CORE 2
CORE N
Main Memory

Figure 4.1: *Different layers for the interaction with the OS (adapted and slightly modified from [HH08]).*

tems meet additional requirements related to determinism, stability and communication. From the Software Developer perspective, system calls can be used for providing access to hardware. It requires specific knowledge about operating system internals, hardware interfaces and kernel-level interfaces.

Layer 2: Represents core thread library functions. Libraries like POSIX offer the possibility to create concurrency using threads or processes. Those libraries interact directly with the operating system, e.g. for registering processes at the process manager. Using libraries of this layer requires detailed knowledge about process and thread management, interprocess communication and about synchronization techniques. Libraries and frameworks use this layer for adapting the ability to create threads and processes. It provides an encapsulation to the operating system. So applications using libraries (e.g. POSIX) at this layer are mainly portable[2].

Layer 3: Represents customized libraries for an easy use of multi-processing or multi-threading. They make use of direct operating system calls (layer 1), and of the thread function libraries (layer 2) as well. They offer specialized solutions for their domain (e.g. Cilk for using nested parallelism). Those customized libraries reduce much of the parallelization issues by introducing their own primitives. They provide high-level language constructs or simple function calls, whereas synchronization objects and direct system interaction for the low-level work are in most cases hidden.

Layer 4: Represents application frameworks for parallel programming. Those application frameworks are more specialized in terms of their own domain. Parallelization is encapsulated in a set of rules defined by the framework, which has to be followed. Problems with synchronization are handled from the framework itself, but there may be dependencies when using other layers as well. Using this layer requires knowledge of the framework.

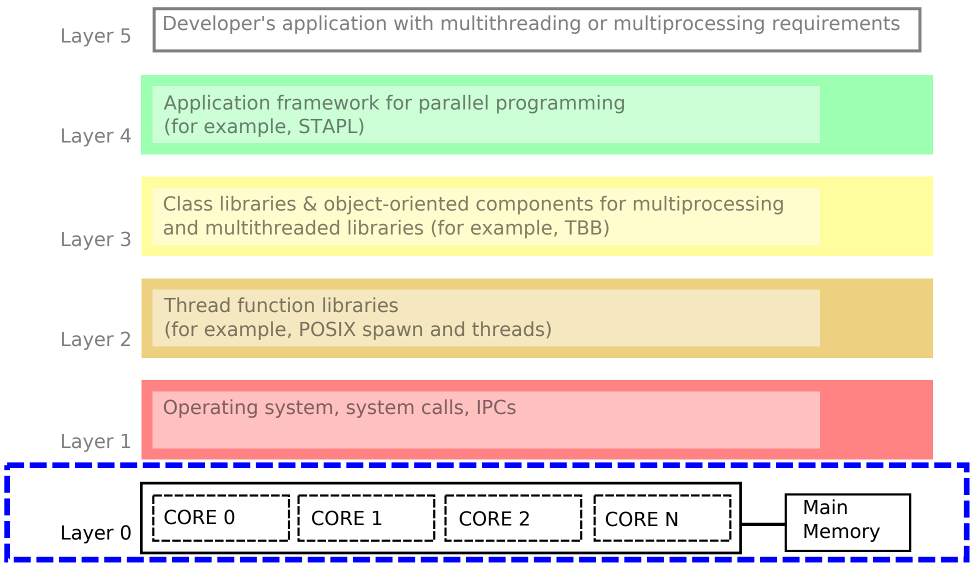
Layer 5: Represents the developers' environment, which has multi-threading or multi-processing requirements. From the perspective of a Software Developer, modifications of the source-code are related to one of the

[2] if ported libraries exist as well

previous layers. Knowledge of the appropriate layer is necessary depending on which layer is accessed.

The layers will be described in the following sections. Each layer has influence in terms of latency-time and response-time of the overall application, as well as influence on a real-time environment. Important key technologies of each layer will be explained and analyzed in terms of usability, performance and influence concerning real-time requirements.

4.2 Layer 0: Hardware


Layer 5
Developer's application with multithreading or multiprocessing requirements
Layer 4
Application framework for parallel programming (for example, STAPL)
Layer 3
Class libraries & object-oriented components for multiprocessing and multithreaded libraries (for example, TBB)
Layer 2
Thread function libraries (for example, POSIX spawn and threads)
Layer 1
Operating system, system calls, IPCs
Layer 0
CORE 0
CORE 1
CORE 2
CORE N
Main Memory

Hardware is the first layer for entering the *parallelized world*. Without the possibility to e.g. calculate several equations at the same time, real parallelism becomes pseudo-parallelism. Mastering hardware is crucial in gaining more benefit of several cores. An application will not be *n-times* faster, just because of using *n-cores*. Those cores can compute in parallel, but they need to exchange data as well. In a shared memory environment (SM) the main memory may become a bottleneck, depending on the hardware architecture and on the utilization of the bus, which links processors with main memory. From the developer perspective it is important to know which different factors exist for affecting the performance of the overall system. However, determinism is one of the most important requirements for a real-time applications, which shall not be affected by improving performance.

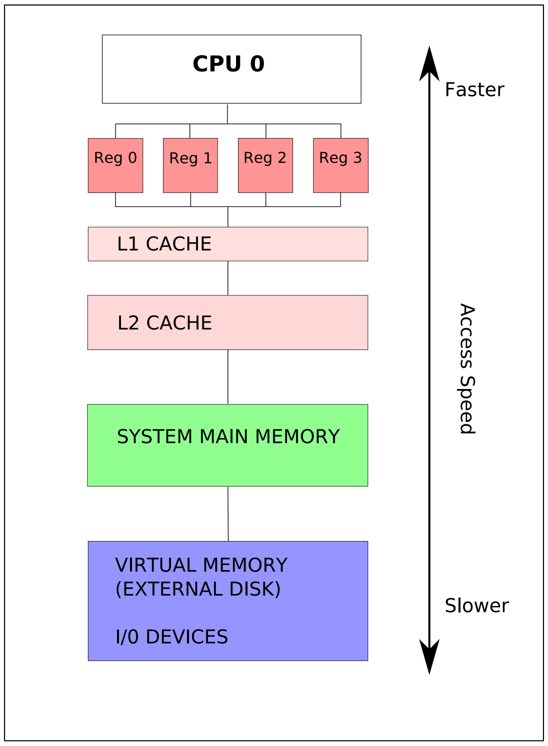


Figure 4.2: *Different access times for memory.[HH08]*

Memory

Memory is one of the main components of a system. The importance of the memory and its management increases with additional number of cores.

Figure 4.2 shows the different types of memory.

- Register
- Cache
- Random-Access-Memory (RAM)
- Virtual Memory

In order to reduce *costly* operations by accessing RAM, further knowledge for exploiting registers and cache is necessary.

Register

A register is a fast memory with almost no latency-time[3]. Registers are usually implemented directly into the core. This means that each core has its own set of registers. The registers bound on one core are independent from registers of other cores. Registers can be accessed directly via assembler code, or by using the keyword `register`[4] in C++. Read and write operations to registers are very fast. That is one reason for their use in interrupt handling routines. Registers are important for the overall performance of a system:

- Stack Pointer (ESP)
- Base Pointer (EBP)
- Instruction Pointer (EIP)

This improves the system performance by utilizing hardware resources efficiently.

Cache

Cache is especially important for multiprocessor systems. The purpose is to preserve the memory bus, so that the average throughput can be increased to a significant level. Memory will become a bottleneck if the bus or the memory

[3]In terms of accessing and modifying.

[4]This does not mean that the variable will be always stored in the register, but compiler will do so if enough space exist.

itself is not fast enough for providing direct response. Cache is a fast memory that *glues* registers and the RAM together. The main purpose of caching was originally to enhance the core utilization through data buffering, since accessing RAM is costly in terms of clock cycles. Having a faster layer, which provides the same data as stored in RAM, can enhance the overall system performance significantly[5]. In a multi-core environment caching is more important, because the access to a bus by several cores can be minimized by utilizing buffering techniques.

Caching disrupts determinism, because if data is missing, it has to be restored from slower RAM, which consumes extra processing cycles, which is not desirable for fulfilling hard real-time conditions. In most cases, it is mandatory to disable caching options e.g. in the BIOS settings for achieving determinism. Situations may exist where non-determinism of caching plays a lower significant role, so that it becomes an interesting feature for enhancing system processing.

Cache Trashing: For a full utilization of a single core it is important for data structures to fit into the processors cache memory, so that no calls or data accesses to RAM are employed. Whenever data from cache is modified, it has to be invalidated for updating data stored in RAM. The updated information must be re-cached. This will be the case if a call to the operating system (system or kernel call) is performed.

In addition to thrashing the cache memory, the processor
has to change its internal mode of operation and raise the

[5]Statistically the most frequent accesses to the main memory are limited to a small address space (known as program locality)

> privilege level. The operating system needs to save the process' execution context and establishes the kernel context to perform the OS' call (and vice versa on return). Thus this is a relatively costly operation compared to a regular in-program function call.[Som02]

Each operation which needs interaction with additional periphery, outside the memory architecture, adds another order of magnitude overhead. Reducing the number and frequency of such expensive system calls can boost up performance and drop the use of system resources.

Compiler Options

Compiler options may have a serious impact on the response behavior of the overall system. Depending on those options, written source code might be interpreted differently. Even a *resort* of instructions is possible, which can cause under specific circumstances (e.g. using the `-parallel` statement from the Intel compiler [GN]), a different behavior of the system. This kind of resort might eventually work in a serial program utilizing one core, but might not work within a multi-core system, because *real* parallelism exists. A compiler cannot understand all semantics, whether a piece of code shall be run in parallel or not. Using a synchronization mechanism like the mutex might cause a deadlock when being automatically resorted. This might lead to a problem, because current compilers are not (or only partly) aware of such issues.

A compiler can definitely enhance response times using hardware dependent optimization techniques. Instructions like MMX or SSE3 exist, which are implemented directly in hardware, and can perform several operations (normally 2-3 operations) at once. Those instructions can be utilized by setting appropriate options. The compiler will *search* for specific instructions which can be replaced by those calls. Knirsch utilized those options for enhancing the speed-up time for his system[6] using the following instructions:

Listing 4.1: *Gaining benefit from compiler options.*

```
gcc -Wall -O1 -msse3 -march=core2 -mfpmath=sse -pipe -fstrength-
    reduce
-fexpensive-optimizations -finline-functions -funroll-loops
-foptimize-register-move
```

Using options for loop-unrolling or for using in-line functions, etc. instructed the compiler to interpret the written source-code more efficiently, where the start-up time is improved by about 30%.

> When using the default flags the system would need up to 2.15 seconds from starting the kernel instead of the given 1.58 seconds.[Kni09]

This example showed that even modifications at the lowest level can have serious impact on the processing time. It is also possible to get an additional speed-up by creating hardware specific assembly through recompilation, in order to exploit more resources from the target hardware. This will work if the hardware architecture and its features are known; otherwise, there will be no benefit from using those options.

[6]Using the gnu compiler for an Intel architecture.

HyperTransport (HT)

Cores need to communicate with each other; otherwise, it is not possible to share work across cores, and to merge results. Implementing a communication through cache may be possible for one dual core processor where this cache may be shared, but not for two separate processors. Implementing a communication through *slower* RAM may be easy in UMA systems, where the address of the memory is shared across all cores, but becomes an additional overhead in NUMA systems. Sharing mechanisms may lead to bottlenecks, depending on scheduling overhead. Interconnect technologies provide fast communication among cores, and help to simplify inter-core communication.[Hyp]

HyperTransport is such an interconnect technology, providing low latency-times and a high bandwidth[7] for, not only inter-core, but also core to periphery communication.

> HyperTransport (HT) is a state-of-art packet-based, high-bandwidth, scalable, low latency point-to-point interconnect technology that links processors to each other, processors to co-processors and processors to I/O and peripheral controllers.[Hyp]

This kind of technology offers several features, which are suited for hard real-time conditions, not only because of the high bandwidth, but also because of the low latency-times and its predictability. Figure 4.3 shows a NUMA architecture using HyperTransport.

[7]It can reduce the magnitude of additional overhead, for expensive IO operations.

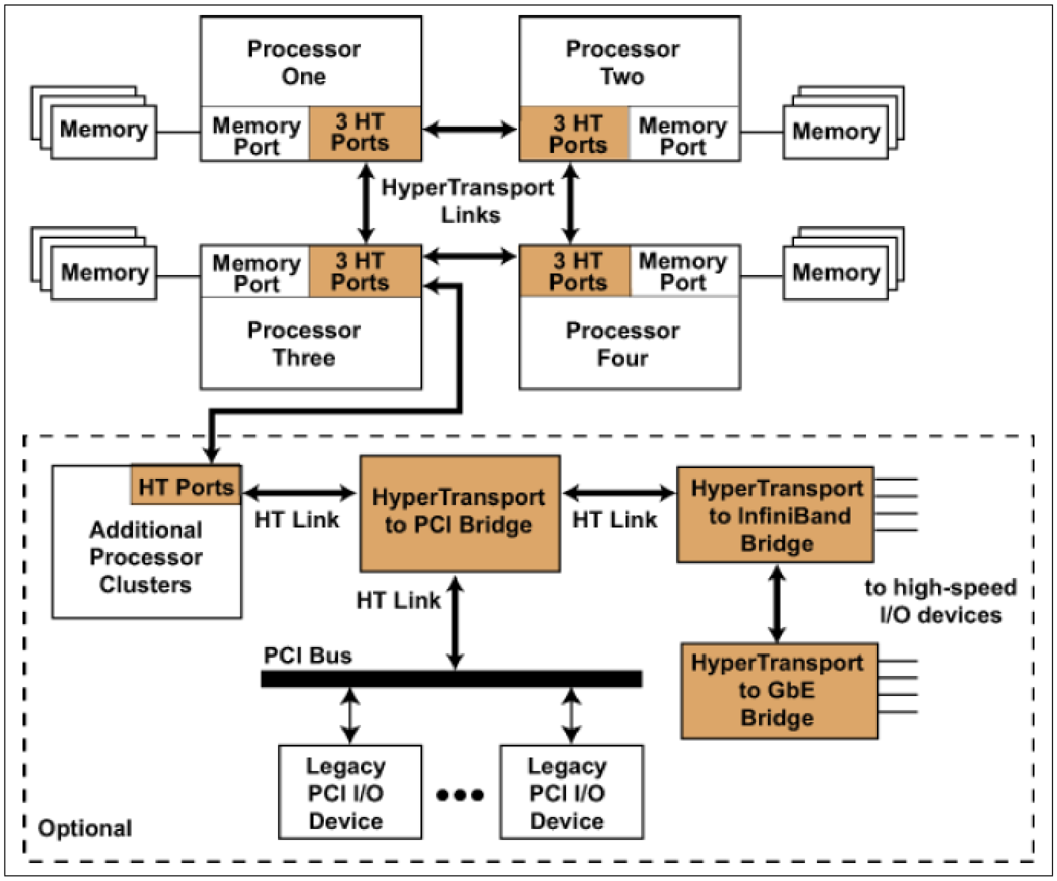
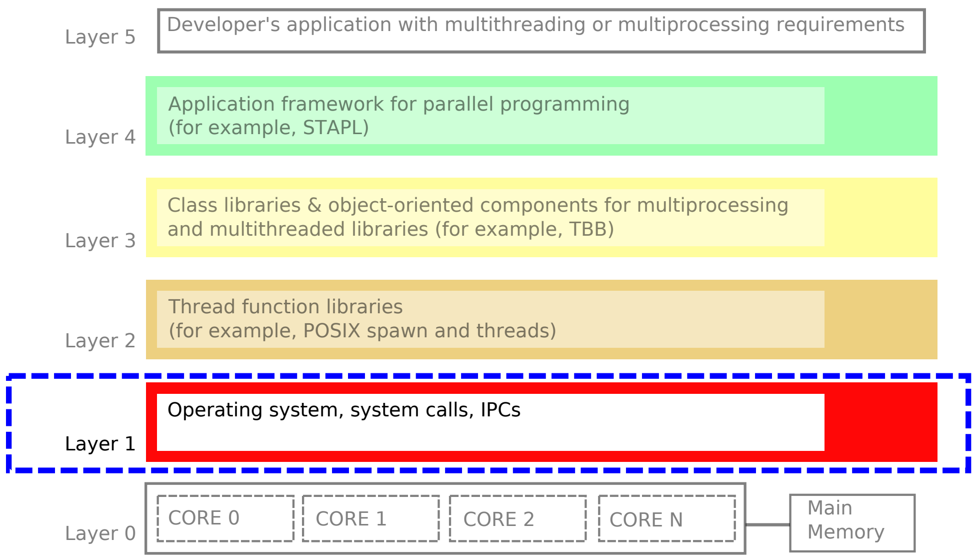


Figure 4.3: *Processor communication using HyperTransport.[Hyp]*

The illustrated processors have their own memory. For an application written on top of an operating system there, is just one shared memory. From a developers perspective, it appears as an UMA architecture (one shared memory). The main difference is that data may be saved in several memory locations, which means that accessing has to be done directly, or in worst case over several processors, which will *cost* more time. In order to enhance response times, scheduling has to consider this issue. As already mentioned, accessing times are different between an UMA and a NUMA architecture. In a NUMA architecture, those times are related to the number of *hops* from one processor to another one. This number of *hops* may differ, depending on the number of processors and on the defined strategy[8] for the access.

[8]Like in a larger Ethernet network where several ways to a destination may exist.

4.3 Layer 1: Operating System


Layer 5
Developer's application with multithreading or multiprocessing requirements
Layer 4
Application framework for parallel programming (for example, STAPL)
Layer 3
Class libraries & object-oriented components for multiprocessing and multithreaded libraries (for example, TBB)
Layer 2
Thread function libraries (for example, POSIX spawn and threads)
Layer 1
Operating system, system calls, IPCs
Layer 0
CORE 0
CORE 1
CORE 2
CORE N
Main Memory

Several operating systems are available, which provide multi-core support. They behave differently in terms of determinism, real-time and efficiency. They may also differ in their way to react when external events arrive. The implementation for scheduling (e.g. FIFO) may differ. So it is possible that two operating systems with different scheduling algorithms react differently to the external world. In a hard real-time environment, it is important to know those details when choosing the operating system for a commercial use.

An operating system offers several services. One service is to provide an abstraction layer for the interaction with hardware. Another service is the task management for process and thread scheduling. Scheduling helps to orchestrate decomposed tasks[9]. Different methods exist for changing the scheduling

[9]Chunks of work, to be processed by one of the cores.

behavior. The most important methods for this thesis will be introduced and explained.

Thread-Affinity

A long running thread in a multi-core system will be executed by different cores. A SMP scheduler may migrate the work of a task to another core for balancing purposes. Each core has its own set of registers and its own cache (see Layer 0). A core switch of a thread will lead to the following consequences:

1. Current processed data, which might be cached on a specific core will become invalidated from that cache, when *touched* by another core.
2. Registers and contents bound on a specific core have to be migrated to the other core.

All of these operations consume time. So, applications should minimize frequent core switching, because core migration consumes additional time. In a SMP environment, context switching may occur through improper use of synchronization mechanisms, unnecessary system calls or other interrupts, which can lead to massive context switches. Those can lead indirectly to core switching, adding further overhead to the context switching overhead. In order to reduce this kind of migration, it is possible to assign threads to a subset of cores by changing the affinity of a thread.

The core scheduling is an additional layer for scheduling the cores only. In other terms, SMP scheduling comes with an additional overhead for utilizing cores and for changing thread-affinity. Different implementations for SMP scheduling exist. Those implementations try to get additional benefit from the cache. They try to schedule threads to the last utilized core, in order to *keep the cache hot.* Even frameworks like Intel TBB try to get benefit from affinity by reducing memory access using the cache. Thread-affinity is a tool for optimizing core utilization and can also help to minimize response times by reducing overhead caused by core migrations.

System Calls

System calls highly depend on the operating system implementation. The timing behavior and the anatomy of a system call are important for the response times of a system. Typically, the kernel is completely locked during a system call. Depending on the duration of the system call, every additional request from another thread is serialized. This induces an additional amount of non-determinism to the overall system. If the kernel is blocked, and another system call cannot be executed, additional latency will be added to the call. The correct consumed time may suffer due to system calls, when considering round-robin, where each thread has its own quantum[10].

Having several threads executing a system call will block the kernel for all threads, except for the first one. Depending on the scheduler, the next access will be provided. The kernel is a kind of bottleneck, and system calls must be minimized in order to anticipate a bottleneck behavior.

Figure 4.4 shows the anatomy of a system call implemented in the QNX Neutrino kernel. The interrupts are locked as short as possible. It is also possible that a current operation of the kernel is preempted (after enabling interrupts again) e.g. by a critical task, which leads to a more predictable system.

[10]A quantum has the same meaning as a time-slice, and has a value of typically 4 ms.

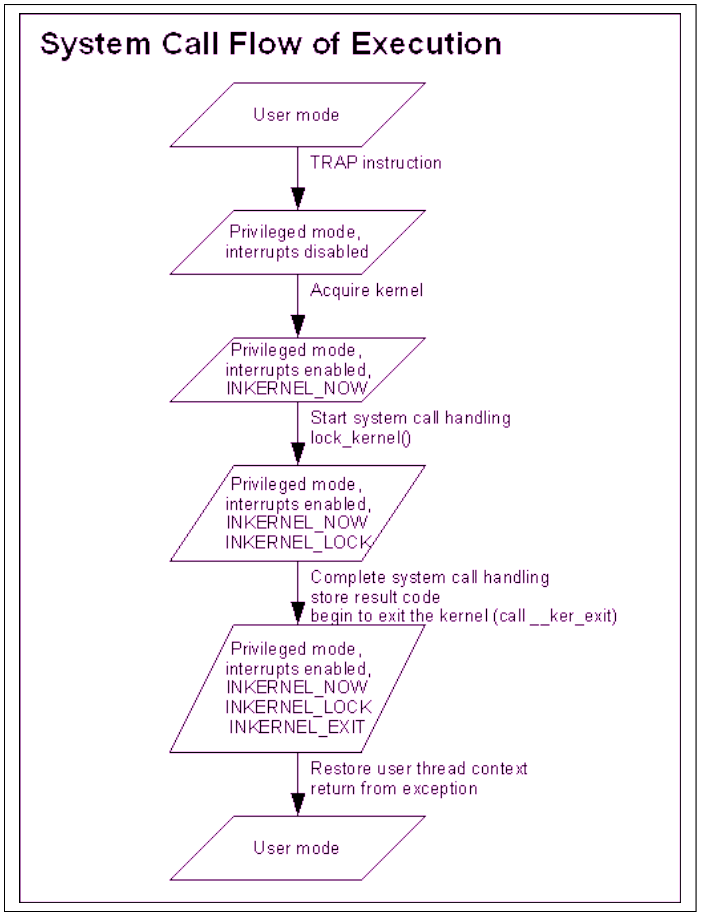

System Call Flow of Execution
User mode
TRAP instruction
Privileged mode, interrupts disabled
Acquire kernel
Privileged mode, interrupts enabled, INKERNEL_NOW
Start system call handling lock_kernel()
Privileged mode, interrupts enabled, INKERNEL_NOW INKERNEL_LOCK
Complete system call handling store result code begin to exit the kernel (call __ker_exit)
Privileged mode, interrupts enabled, INKERNEL_NOW INKERNEL_LOCK INKERNEL_EXIT
Restore user thread context return from exception
User mode

Figure 4.4: *Preemptable system calls implemented in QNX Neutrino.[Resa]*

Whenever a higher priory thread is consuming its quantum, it can preempt kernel operations[11].

[11]This is one of the QNX Neutrino features.

Interrupts

Interrupts are generated for notifying the software of external events. They are[12] the *glue* between peripherals and the operating system. So interrupts are normally related to device drivers. Interrupts normally have the highest priority in the system. Each time an interrupt occurs, a working thread with lower priority will be preempted. Two different ways for working with interrupts exist. The first one is to write code directly into an Interrupt Service Routine (ISR). The second one is to delegate work from an ISR to a working thread. Depending on the amount of work and the frequency of the interrupt, one of the strategies must be chosen.

Writing code directly into an ISR is probably the easiest approach, but has its drawbacks. The size of the ISR is limited and often requires additional code within an ISR for protection. In some implementations, the ISR handler must be disabled to prevent further interruptions on crucial tasks. Secondly, the ISR must be performed as fast as possible; hence it takes ownership of a processor. As long as the ISR is executed, a lower priority thread will not work[13].

Lengthy interrupts can degrade overall system performance, hindering the execution of other critical processes. Hindered processes may have impact on meeting deadlines for *each piece* of work. Violating those deadlines will lead to catastrophic consequences in a hard real-time system. For that reason it is important to delegate time consuming operations from the ISR to a preemptable (user-level) thread or process. The data used by an ISR can be delegated to a thread, whereas the ISR and the thread have to share data. If an

[12]Polling mechanisms exists as well.
[13]On this core.

interrupt arrives, information may be preprocessed and delegated to a thread.

An ISR and thread can run in parallel in a multi-core system. An interrupt and a thread can share data. If that is the case, the access has to be secured because of the possible parallel execution of the ISR and the thread. When data is provided to the user-level thread, it is possible, depending on the priority of the thread, that the current task can be preempted.

Preemption supports a kind of quality of service mechanism for those routines. This enables reducing the latency-time for more important tasks

Scheduling

Scheduling is one of the most important properties of a real-time system. Scheduling is *the main utility* to be exploited to map internal system workflow to real world events. Different scheduling methods are proposed from the POSIX standard and can be used within the most POSIX conform operating systems:

- FIFO
- Round-robin
- Sporadic

The kind of scheduling is important, because the scheduler is responsible for ensuring that all processes can be orchestrated so that they can meet their deadlines. The focus of this thesis will be on round-robin and FIFO scheduling mechanisms, so sporadic scheduling will not be discussed any further in this thesis. Sporadic scheduling[14] uses a kind of budgeting for performing timing decisions.

FIFO and round-robin can be used if different processes or threads are sharing the same priority. Otherwise, only the highest thread will work and other ones will be preempted if all resources (cores) are allocated. FIFO and round-robin are preemptive mechanisms. So, if a higher priority thread starts its operation, a lower priority thread may be preempted. This preemption is done through the scheduler.

FIFO implementations add another significant overhead to medium scaled implementations, because scheduling has to be triggered using `sched_yield` or another system call. Moreover, FIFO implies a kind of state-machine at the scope of processes and threads. Implementing it in a multi-core system will increase the complexity of this state-machine significantly, and, furthermore, it will increase the risk of having deadlocks, live-locks or other problems.

The task performance in round-robin is different. It has an additional characteristic which is related to time. The quantum in which each thread (in QNX Neutrino threads and processes are nearly the same) may run is fixed and is usually triggered from an external hardware timer. In most cases, this timer has a resolution at around 1 ms.

[14]It is old fashioned and can not efficiently be used in a larger development team, hence each team has to maintain their own budget, which may interfere with the budget of another team.

Synchronization Primitives

Synchronization primitives are offered by the OS to provide mechanisms against data-races. Data-races occur if data dependencies over several tasks exist. This also may happen on a single-core-machine, but having a multi-core machine emphasizes this effect.

An example for those data-races can be shown through the client and observer mechanism, where the client writes the information at a specific place in memory and the observer has to update this information on a screen. Running the observer and client in parallel or even quasi-parallel will lead to an error when either the update or the write procedure is not an atomic operation. If the writer has not completed the intended work[15] and the observer tries to update, a wrong information will be displayed or an error might occur.

Synchronization primitives can be used to make this operation thread-safe. Those synchronization primitives are atomic, but their time consumption may not be constant. Problems may occur like deadlocks, live-locks or priority inversion. Improper use of many synchronization primitives can lead to a relevant amount of overhead. These primitives heavily rely on their implementation within the OS and on the used target hardware.

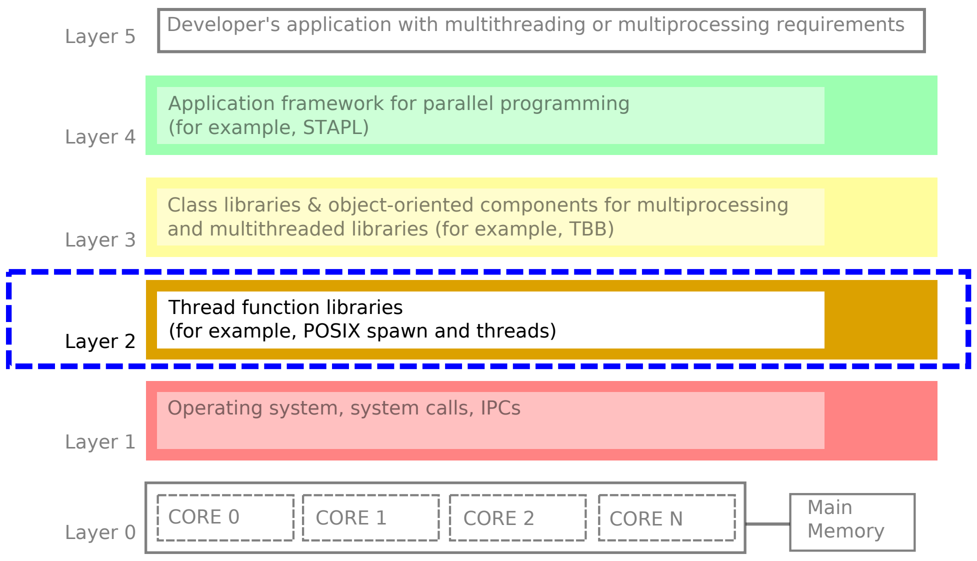
To improve system performance and in order to reduce latency-times, proper usage of those primitives must be ensured. Synchronization primitives were mainly used for dealing with pseudo parallelism. It may be that those primitives, which are often interacting with the kernel, are not suitable for multi-core applications. Having 1000 threads sharing the same resource, which is protected through a shared semaphore, may decrease the perfor-

[15]See increment example (listing 3.3).

mance significantly, depending on the frequency of accessing this resource. Semaphores lead to system calls, so the operating system has to perform work. A double-check mechanism in user-space may influence the performance drastically, but also lock-free mechanisms exist where the operating system will not be influenced.

Synchronization primitives must be handled with special care on a multi-core system, because false wake-ups can occur. They have to be placed inside a loop and their conditions must be checked.[Resb] So it is not safe to use the standard primitives (mutexes, semaphores) within a multi-core system, without the proper consideration of the impact of real concurrency.

4.4 Layer 2: Thread function libraries


Layer 5
Developer's application with multithreading or multiprocessing requirements
Layer 4
Application framework for parallel programming
(for example, STAPL)
Layer 3
Class libraries & object-oriented components for multiprocessing
and multithreaded libraries (for example, TBB)
Layer 2
Thread function libraries
(for example, POSIX spawn and threads)
Layer 1
Operating system, system calls, IPCs
Layer 0
CORE 0
CORE 1
CORE 2
CORE N
Main
Memory

This layer is mainly used for providing portability to applications which have the need for parallelization. In other terms, if several OS are sharing only layer 2, then applications and frameworks based on this layer become OS independent. One realization of that is the POSIX library.

Thread creation

> POSIX defines how the application talks to the library and how the library and underlying operating system behave in response. Each implementation can have its own way of dividing the work between the library and the operating system.[LD91]

With this library it is possible to create and to destroy threads and processes, which are necessary for the creation of an OS independent parallelized application. This OS independence comes at the cost of an additional layer to the application. For reducing the overhead, which comes from the additional layer, a direct system call might be performed. The listing 4.2 shows the different signatures between the system call `ThreadCreate()` and the POSIX library call `pthread_create()`:

Listing 4.2: *Different layers can be accessed through a simple selection of the function calls.*

```
#include <sys/neutrino.h>   #include <pthread.h>

int ThreadCreate(           int pthread_create(
  pid_t pid,                  pthread_t* thread,
  void* (func)(void*),        const pthread_attr_t* attr,
  void* arg,                  void* (*start_routine)(void*),
  const struct                void* arg );
      _thread_attr* attr );
```

Using those calls can reduce overhead, but not without cost.

Synchronization Primitives

Calling `ThreadCreate` instead of `pthread_create` will reduce time for creating a thread, but the time the function will consume, which is passed through `void* (func) (void*)` or through `void* (*start_routine)(void* )` will be almost the same. More interesting are functions which are called frequently, like synchronization mechanisms as the mutex (see listing 4.3).

Listing 4.3: *Achieving higher benefits utilizing lower synchronization primitives.*

```
#include <sys/neutrino.h>           #include <pthread.h>

```

```
int SyncTypeCreate(                 int pthread_mutex_init(
  unsigned type,                      pthread_mutex_t* mutex,
  sync_t * sync,                      const pthread_mutexattr_t*
      attr );
  const struct _sync_attr_t * attr );
```

The implementation from `pthread_mutex_init` is utilizes `SyncTypeCreate` for gaining access to a kernel internal mutex. This is suitable for a mutex with the need for further scheduling information in order to implement advanced accessing protocols for reducing priority inversion. The POSIX implementation might also have direct access to the hardware. Using an atomic statement, in this case `test_and_set`[16], helps to implement a mutex in *user-space.* A mutex which does not rely on a kernel interaction[17] saves the time necessary for a context switch.

Listing 4.4: *POSIX mutex implementation in user-space.*

```
static __inline int __attribute__((__unused__)) _smp_cmpxchg(
    volatile unsigned *__dst, unsigned __cmd, unsigned __new) {
  __asm__ __volatile__(
    "lock; cmpxchgl %3, (%2)"
    :"=m" (__atomic_fool_gcc(__dst)), "=a" (__cmd)
    :"d" (__dst), "c" (__new), "1" (__cmd)
    :"memory");
  return __cmd;
}
```

As shown in listing 4.4, the memory was locked in order to prevent issues derived from CPU migration and parallel access. This comes from a compare, which is necessary and uses additional instructions. A mutex in user-space has its advantages and its disadvantages. A great advantage is the indepen-

[16]usually implemented in x86 hardware

[17]It will need a system call, when *advanced protocols* like for priority-ceiling are used.

dence of the OS, which could be also interpreted as a disadvantage; hence the OS does not know anything about this synchronization mechanism. Mutexes and semaphores are sources of priority inversion and deadlock situations. Accessing protocols exist, which retrieve information from the scheduler, for implementing sophisticated protocols in order to reduce deadlock situations and priority inversion.

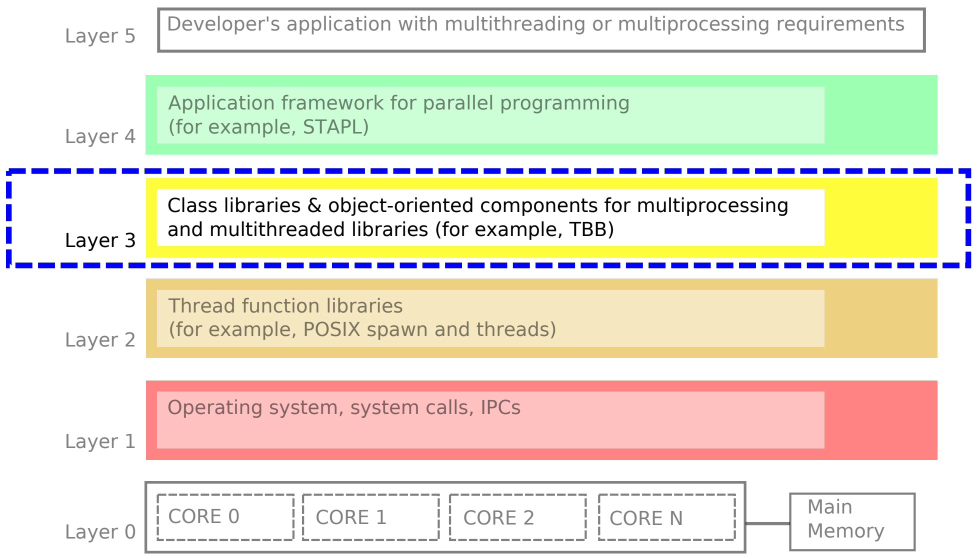
POSIX proposes several synchronization primitives, most of them are implemented in a POSIX-conforming operating system. QNX Neutrino offers mutexes, condvars, barriers, sleepon-locks, reader- and writer-locks and semaphores for that purpose.[Resb]

Almost every operating system extends this standard with its own primitives. QNX Neutrino provides its own mechanism for Interprocess communication (IPC), which works quite efficiently, and Linux provides RCU synchronization mechanisms. QNX Neutrino uses its own internals for gaining additional performance. Libraries are not limited to system calls; they can interact with the hardware directly. This direct interaction can improve performance (see mutex as example). It is important to know when a mechanism will work, and how those mechanisms will interact with their environment.

> Raw threads and MPI expose the control of parallelism at its lowest level. They represent the assembly languages of parallelism. As such, they offer maximum flexibility, but at a high cost in terms of programmer effort, debugging time, and maintenance costs.[Rei07]

The POSIX library helps to provide an additional layer for operating system independence. Using this layer requires knowledge about the operating system, and if performance becomes important, deeper understanding about implementation details is needed.

4.5 Layer 3: Libraries and Components for multiprocessing and multithreading


Layer 5
Developer's application with multithreading or multiprocessing requirements
Layer 4
Application framework for parallel programming (for example, STAPL)
Layer 3
Class libraries & object-oriented components for multiprocessing and multithreaded libraries (for example, TBB)
Layer 2
Thread function libraries (for example, POSIX spawn and threads)
Layer 1
Operating system, system calls, IPCs
Layer 0
CORE 0
CORE 1
CORE 2
CORE N
Main Memory

It was shown in the previous chapters that the utilization of hardware can be improved using specific hardware dependent compiler settings, an optimized operating system, or faster synchronization primitives. The current layer offers a new, more general possibility for enhancing response times, called *work decomposition*. It is often the case that bigger chunks of work, which are executed serially, can be decomposed into smaller chunks of work, which can be parallelized. This work decomposition offers the opportunity to utilize several cores. In other terms, work decomposition is one of the most important aspects for parallelism.

In some cases, it is not possible to decompose work because of data dependencies. It is possible to resolve many of those dependencies by restructuring data, or by using techniques of redundancy. Frameworks exist for work decomposition, but frameworks for removing all data dependencies do not exist. Those frameworks may aid the developer for utilizing several cores, but the developer has to remove those dependencies for gaining maximal benefit. It is an interesting fact that the developer needs to know less about scheduling or synchronization, because the frameworks establish a kind of automatism, which is exploited for gaining better predictable results in a more generic way. One author introduced his book about frameworks for parallelization (in this case Intel Thread Building Blocks (TBB)) with the following statement:

> You do not need to have any experience with parallel programming or multi-core processors to use this book. Whether you have a great deal of experience with parallel programming, or none at all, this book will be useful. No prior understanding of threading is required.[Rei07]

For a better understanding of those frameworks, three of the most important[18] will be discussed:

- Cilk
- OpenMP
- Intel Thread Building Blocks (TBB)

Cilk and OpenMP need their own compiler support. OpenMP is included in the gcc compiler, and Cilk uses the functionality of the gcc compiler, but has

[18]From my perspective.

to be compiled with its own Cilk compiler. TBB behaves differently; the philosophy of TBB is to provide a library for enhancing parallelization, whereas Cilk and OpenMP use their own special keywords. All have in common that they utilize the POSIX library for gaining access to threads (`pthread`) and synchronization mechanisms.

Cilk and OpenMP differ in implementation detail and work-flow, but provide a similar interface to the developer. Both frameworks use keywords for the creation of threads and for explicit synchronization. In general, it has to be distinguished between *explicit* and *implicit* synchronization. This will be noted in each discussed framework.

Cilk

Recursive functions are often used for a simplified implementation of backtracking or special brute-force mechanisms. They are used in real-time systems[19] as well. Many algorithms like divide-and-conquer are implemented as recursive functions. Divide-and-conquer algorithms require a kind of work decomposition, which can be greatly utilized with Cilk. An often used example for exploiting parallelism is the computation of the Fibonacci numbers. The keyword `cilk` has to be written in front of a function, which shall be parallelized. Each *Cilk function* has an implicit barrier at the end of the function. Listing 4.5 shows a parallel version of a recursive Fibonacci algorithm.

Listing 4.5: *One of the advantages of Cilk lies in the parallelization of recursive functions.*

```
cilk int fib (int n){
```

[19]With some special considerations for security.

```
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib (n-1);  // can be parallelized
    y = spawn fib (n-2);  // can be parallelized
    sync;                 // wait for both parallelized calls
    return (x+y);
  }
}
```

The developer has to *sign* which parts of the source code can be performed in parallel. The Cilk compiler translates those sections with corresponding library calls (from the Cilk library). It is interesting that the use of high numbers (n > 10000) does not lead to poor system performance, because 10000 threads could be spawned. Spawn does not create any threads; it marks that a command can be executed in parallel. Cilk uses a work-stealing algorithm for utilizing the cores in a system. Cilk has proven that it is suitable even for hard real-time conditions, because its efficiency is predictable.

> Cilk is a runtime system whose work-stealing scheduler is efficient in theory as well as in practice. Moreover, it gives the user an algorithmic model of application performance based on the measures of *work* and *critical path* which can be used to predict the runtime of a Cilk program accurately.[BJK+95]

Cilk achieves nearly linear speedup for problems with the same structure as Fibonacci.[Blu92] The Cilk scheduler uses randomized work stealing, which means *randomized determinism* in some cases.

OpenMP

OpenMP uses an iterative approach for work decomposition, especially for parallelizing loops. The big advantage of OpenMP is that only little changes on the original code must be performed. The changes are done through compiler options (pragmas), which, if not supported, are ignored by the compiler.

The standard way of parallelizing a program using OpenMP is to write it sequentially. After that, the program can be parallelized by setting appropriate pragmas.

Listing 4.6: *With OpenMP it is possible to parallelize loops efficiently.*

```
#pragma omp for
for( i = 0; i < n; i++ ){
/*xxx*/
}
```

As shown in figure 4.6, parallelization can easily be achieved by using pragmas. The iterations for a loop are divided into chunks, where each chunk is performed in an own thread. At the end of the loop, all threads wait for each other until all work is completed. The pragma `omp for` creates an implicit lock at the end of the scope[20].

The POSIX synchronization construct *barrier* is used for that purpose. This construct is used for implicit synchronization within OpenMP. Listing 4.7 shows the implementation details of the used barrier.

Listing 4.7: *The POSIX barrier is used for implicit synchronization in OpenMP.[Bab]*

```
void __ompc_barrier (omp_team_t *team) {
  pthread_mutex_lock(&(team->barrier_lock));
```

[20]In C++ a scope can be defined using brackets.

```
  team->barrier_count++;
  barrier_flag = team->barrier_flag;

  if (team->barrier_count == team->team_size) {
    team->barrier_count = 0;
    team->barrier_flag = barrier_flag ^ 1; /* Xor: toggle*/
    pthread_mutex_unlock(&(team->barrier_lock));
    return;
  }
  pthread_mutex_unlock(&(team->barrier_lock));

  OMPC_WAIT_WHILE(team->barrier_flag == barrier_flag);
}
```

Threads are managed as teams, where each thread team maintains a barrier counter and a barrier flag. Teams increment the barrier counter when they enter the barrier and wait for the barrier flag to be set by the last thread[21]. The team size is equal to the counter when the last thread enters the barrier. As such, the barrier flag is set and all other waiting threads can then proceed.

Intel Thread Building Blocks (TBB)

Similar to the iterative approach of OpenMP is the TBB approach from Intel. A thread pool is initialized at the beginning, which can be used for completing different tasks. The keyword *Task* has a special meaning for TBB[22]. Using a thread pool has several advantages, because the overhead for the creation and deletion of threads is minimized. TBB not only adapted the work stealing concept from Cilk for the scheduling of the own thread pool, but also the possi-

[21] It is not defined in the POSIX standard which thread will finish the work.

[22] It has a special meaning for Cilk and OpenMP as well, but it is the main conceptual approach from TBB.

bility to mix TBB semantics (library calls) with OpenMP semantics (pragmas). One descendant from this approach is the *Intel Compiler*[23], which offers the possibility to use OpenMP and TBB. TBB is based on library calls, which implies that additional *source code* has to be included in order to gain benefit of this framework. This is different than the Cilk or the OpenMP approach, where those additional calls are hidden from the Software developer by introducing new keywords. Listing 4.8 shows the required source code for creating a parallel version of a loop.

Listing 4.8: *Using TBB templates for work decomposition.*

```
class CFunctor {
public:
  void operator( )( const blocked_range<size_t>& r ) const {
    for( size_t i=r.begin(); i!=r.end(); ++i ){
      //Perform the chunk of work between r.begin() and r.end()
      //This is calculated through myStart, myEnd, myChuckSize
    }
  }
  CFunctor() { }  //constructor
};
void tbb_parallel(){
  parallel_for(blocked_range<size_t>(myStart,myEnd,myChunkSize),
       CFunctor() );
}
```

As shown, a complete new class has to be created in order to create a parallel version of a loop. This class is required for working with *STL like* templates as the `parallel_for` call. A class has to overload the bracket operator with the defined signature, as shown in listing 4.8, for being adapted from the TBB framework. It is also possible to add further data as attributes to the `CFunctor`

[23]And its related library.

class, which could be used during parallelization. The introduction of a new class for such a simple construct might be seen as too much overhead. With the upcoming C++ standard called C++1x, it will be possible to use lambda expressions. Those lambda expressions offer the possibility of temporary objects, without the definition of a class. This will have a significant impact for writing TBB enhanced applications. Listing 4.9 shows that much of the introduced overhead can be eliminated by using lambda expressions.

Listing 4.9: *Using TBB templates with lambda expression*

```
void tbb_parallel(){
  parallel_for(blocked_range<size_t>(myStart,myEnd,myChunkSize),
    //--- start of the lambda expression
    [=]( const blocked_range<size_t>& r ){
      for( size_t i=r.begin(); i!=r.end(); ++i ){
        //Perform the chunk of work between r.begin() and r.end
            ()
        //This is calculated through myStart, myEnd, myChuckSize
      }
    }
    //--- end of the lambda expression
  );
}
```

In their current versions, the syntax for writing lambda expressions is different between the Intel compiler and the gcc compiler. Listing 4.9 gives a good example for the effectiveness of lambda expressions.

Listing 4.10 gives an introduction to the nested task concept in TBB. For a more comprehensive understanding, the calculation of the Fibonacci numbers will be used to show the differences between Cilk nested parallelism and the TBB approach.

Listing 4.10: *The task principle implemented in Intel Thread Building Blocks (adapted and slightly modified from [Rei07]).*

```
class FibTask: public task {
public:
  const long n;
  long* const sum;
  FibTask( long n_, long* sum_ ) : n(n_), sum(sum_) {}

  task* execute() {
    if( n < 2 ) {
      *sum = n;
    } else {
      long x, y;
      FibTask& a = *new( allocate_child() ) FibTask(n-1,&x);
      FibTask& b = *new( allocate_child() ) FibTask(n-2,&y);
      set_ref_count(3);
      spawn( b );
      spawn_and_wait_for_all( a );
      *sum = x+y;
    }
    return NULL;
  }
};
```

Two important steps must be done for implementing nested parallelism with TBB. The first step is to inherit from the `task` class, which is part of the TBB framework. As a second, the `task* execute` method has to be implemented. This is the part where the *work decomposition* has to be defined. In case of the Fibonacci numbers, it has to be defined whether the recursive calls are done in parallel or not. The command `set_ref_count(3)` defines that 3 actions are running in parallel. That are the two `spawned` nodes a and b, as well as the task execute method itself. The keyword for parallelizing (spawn) is in this case equal to Cilk. The synchronization is done via spawn_and_wait_for_all(a).

The call within the main function looks different, because nodes were defined for parallelization purposes. Listing 4.11 shows the call for spanning the tree.

Listing 4.11: *Calling the FibTask.*

```
long ParallelFib( long n ) {
  long sum;
  FibTask& a = *new(task::allocate_root()) FibTask(n,&sum);
  task::spawn_root_and_wait(a);
  return sum;
}
```

The first node is the root node (in listing 4.11 defined as a), the following are sub-nodes, which are created for each recursive call.

Listing 4.10 showed the TBB implementation for nested parallelism. This implementation is not performance optimized. It is important to have an adequate grain size for the tasks. The overhead induced for the orchestration of the task may not be useful for small tasks. In case of the Fibonacci example, the overhead comes from the recursive computation of small n-values (see listing 4.12).

Listing 4.12: *Increase performance through changed work size.*

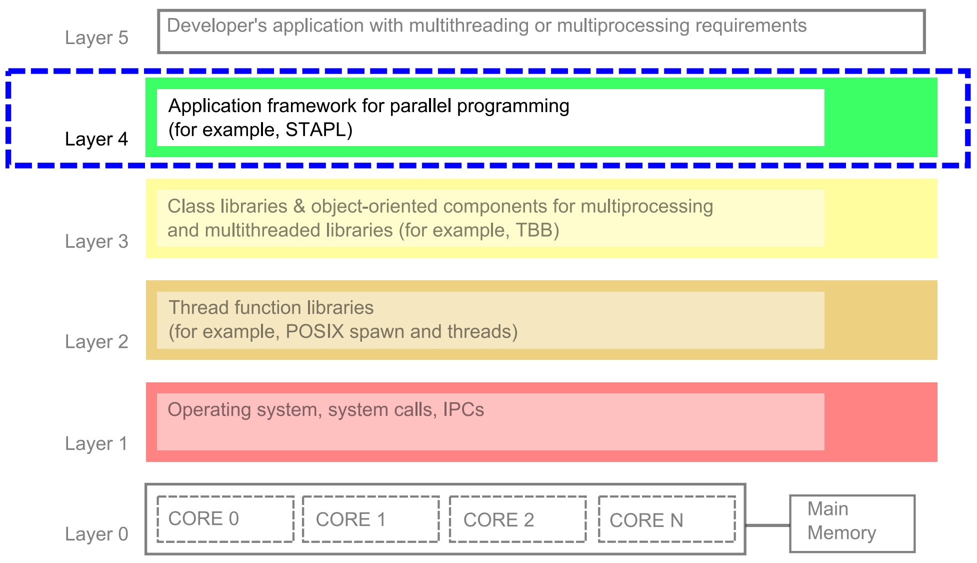
```
// fine granularity          // use different strategies
//                           // for enhancing performance
task* execute() {            task* execute() {
  if( n < 2 ) {                if( n < 16 ) {
    *sum = n;                    *sum = SerialFib(n);
  } else {                     } else {
```

Use the serial version for small n-values (e.g. $n < 16$) to minimize overhead introduced through parallelism and create parallel versions for larger n-values (e.g. $n >= 16$). It is an interesting fact that two different approaches for the calculation must be implemented to gain an improvement. The Software developer has to be aware of this concept.

The TBB framework handles the deletion of those nodes automatically[24]. In case of hard real-time conditions, those techniques are not allowed. Dynamic memory allocation will lead to memory fragmentation, which will end in a non-deterministic system. Using the default memory allocation routine of TBB should not be allowed without considering the effects for the overall system.

[24]TBB comes with its own memory allocator.

4.6 Layer 4: Application framework for parallel processing


Layer 5
Developer's application with multithreading or multiprocessing requirements
Layer 4
Application framework for parallel programming (for example, STAPL)
Layer 3
Class libraries & object-oriented components for multiprocessing and multithreaded libraries (for example, TBB)
Layer 2
Thread function libraries (for example, POSIX spawn and threads)
Layer 1
Operating system, system calls, IPCs
Layer 0
CORE 0
CORE 1
CORE 2
CORE N
Main Memory

With layer 4 it becomes possible to decouple the level of parallelism from shared memory toward to distributed memory. OpenMP, Cilk and Intel TBB implementations require (at least at the moment) parallelization in shared memory. Libraries exist, which can perform parallelism in shared memory as well as in distributed memory.

Standard Template Adaptive Parallel Library (STAPL)

This library is designed to work on both shared and distributed memory parallel computers. It allows a developer to work at a higher level of abstraction as in Cilk, OpenMP and TBB. It provides interface classes and interface algo-

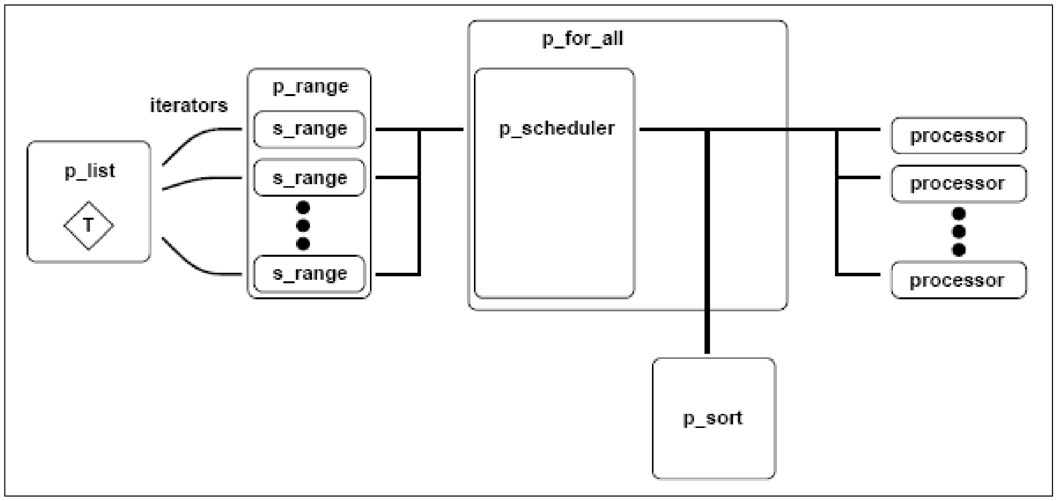


Figure 4.5: *STAPL components.[AJR+ 03]*

rithms, which hide many details for the work decomposition. The architecture (see figure 4.5) of this library allows implementations at different scopes.

The keyword *Adaptive* is originated from a feature of the STAPL framework. STAPL tries to adapt[25] the best possible algorithm for increasing performance, based on a performance model.

> Build-in performance monitors can measure actual performance and, using an extension of the BSP model predict the relative performance of the algorithmic choices for each library routine.[AJR+03]

One important aspect is the support of recursive (nested) parallelism, which is supported by Cilk. With the new standard 3.0 of OpenMP, another feature is introduced for gaining nested parallelism support by introducing the task ob-

[25]It provides an interface for that.

ject. The task concept is driven from the TBB implementations, where tasking is used as generic concept (e.g. `parallel_for`).

STAPL consists of five major components: parallel containers (`p_container`), thread safe iterators, ranges for parallel work decomposition (`p_range` and `s_range`), a function for managing parallelism (`p_for_all`) in conjunction with algorithms for defining processor scheduling (`p_scheduler`) and order of the next range to be used (`p_sort`).

Decomposable work has to be defined by setting its range. This range is divided into different subranges, which can be processed by a single processor (see figure 4.5) in a defined order.

The separation into the mentioned *major components* offers flexibility in arranging tasks. Depending on which method for scheduling and work decomposition is used, it may be become usable for hard real-time implementations.

4.7 Summary

In this chapter 6 different layers were introduced for showing opportunities and problems in multi-core systems. *Real concurrency* can only be achieved by having the appropriate hardware. Hardware and its configuration settings have direct impact on the system, especially for one with several cores. Usually, each core has its own cache and an own set of registers. These types of memory are fast, but using them improperly can lead to performance issues. One example for that case is cache-trashing, which may occur by the frequent use of system calls.

RAM may become a bottleneck in multi-core systems, because each core needs access to the memory bus. In order to relieve this issue, an additional bus like HyperTransport is used in modern systems. HyperTransport reduces the traffic on the main memory bus.

The impact of the operating system was discussed in Layer 1. As shown, scheduling has a big impact on the core utilization. This scheduling can be influenced not only by synchronization primitives and system calls, but also by the thread affinity, which can be used to reduce scheduling overhead. System calls are expensive because they *block* the kernel for a short time. This blocking mechanism leads to serial access to the kernel in a multi-core system, which is not the case in a single-core system.

Programming in a multi-core environment is different from a single-core, because real concurrency data has to be protected from *all sides*. The data access has to be secured because data may accessed in parallel (from ISR and thread).

Layer 2 showed that, depending on the implementation, even calls from a layer above the operating system may be fast. Therefore, a developer has to be aware of this implementation details, to know e.g. whether a call is related to a kernel interaction or not.

Frameworks for parallelizing were introduced in Layer 3. Those frameworks automate a *big part* to be done for work splitting. The parallel loop was an example where this *big part* was performed by the OpenMP framework.

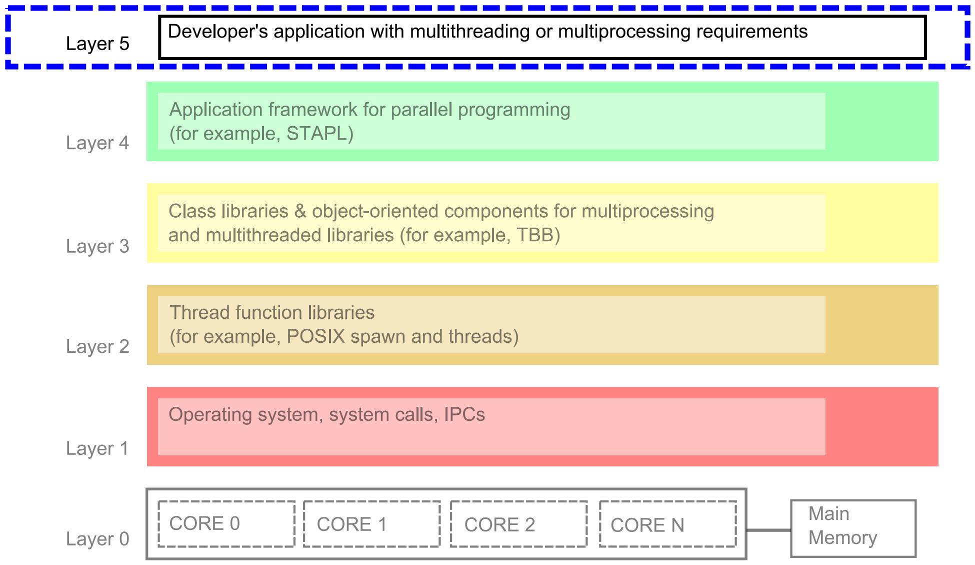
OpenMP, Cilk and Intel TBB use different approaches for gaining benefit from additional cores. TBB uses a thread-pool, whereas OpenMP creates threads and delegates the work to those sub-threads. Nested parallelism, which is the easiest way to gain parallelism in Cilk, and tasks were introduced. These attributes serve for comparative purposes among parallelization capable frameworks.

STAPL introduced at layer 5 is a kind of framework, which goes one step further. Different methods of work splitting and scheduling can be adapted during runtime for maximizing the performance of the overall system. This layer uses frameworks like OpenMP for parallelization reasons.

CHAPTER 5

CONCEPTS AND MEASUREMENTS

5.1 Layer 5: Application Layer


Layer 5
Developer's application with multithreading or multiprocessing requirements
Layer 4
Application framework for parallel programming (for example, STAPL)
Layer 3
Class libraries & object-oriented components for multiprocessing and multithreaded libraries (for example, TBB)
Layer 2
Thread function libraries (for example, POSIX spawn and threads)
Layer 1
Operating system, system calls, IPCs
Layer 0
CORE 0
CORE 1
CORE 2
CORE N
Main Memory

Several ways for utilizing performance are possible, and were partly discussed in the previous part. The use of frameworks can help to exploit additional cores more efficiently. Cilk, Intel TBB and OpenMP may have their advantages for multi-core systems, but deep knowledge is mandatory to use them in a hard real-time environment. Those frameworks use advanced primitives to eliminate synchronization overhead or to resolve latency issues.

It is interesting that the previously shown frameworks did not provide any further information of the lock-free or wait-free mechanisms shown in Part II.

The implementation of a concurrent lock-free Queue

The class `CCommQueue` of the `H_DA` framework (based on [WT05]) is reimplemented to give an example for the lock-free approach and to provide a work-

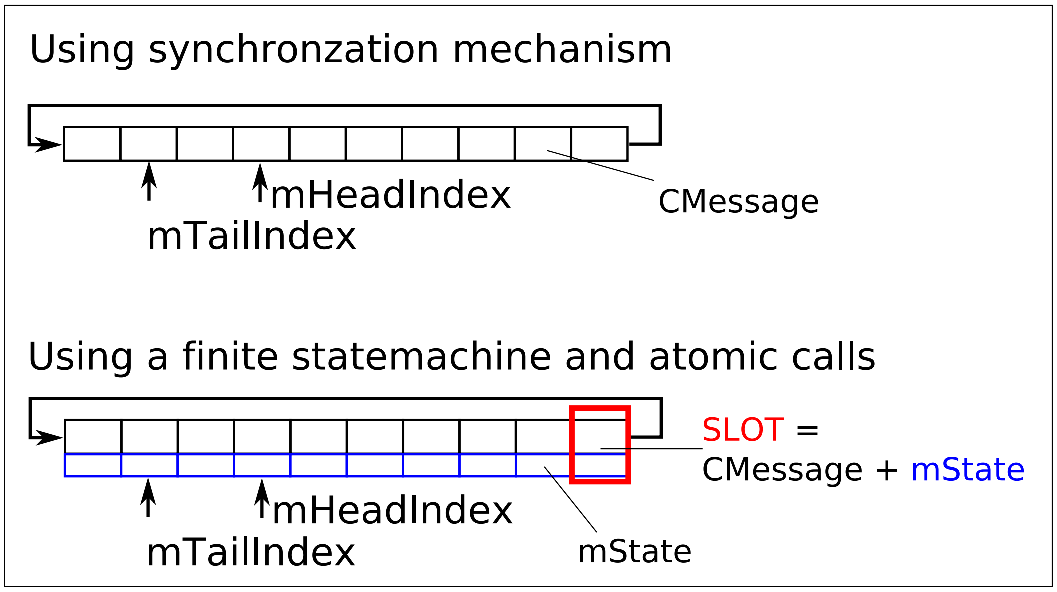


Figure 5.1: *Two different concepts for implementing a queue.*

ing solution for those algorithms. `CCommQueue` is an implementation of a fixed sized Ring-Buffer class. Within this framework, several processes and threads have access to this buffer and are writing and reading elements in parallel. In order to have a clean data access, methods for adding and getting elements are protected through a mutex. The current implementation uses semaphores to signal whether the queue is empty or filled, and a mutex for read-, write-protection. For further information, take a look at [WT05]. The new implementation uses a finite state-machine and the atomic call compare-and-swap. As shown in figure 5.1, two indexes are used for accessing current data. The index `mTailIndex` is used for getting data from the end of the queue. The index `mHeadIndex` is used for adding new data to the front of the queue. It is important that the `mTailIndex` does not step over `mHeadIndex`, in which case data will be lost.

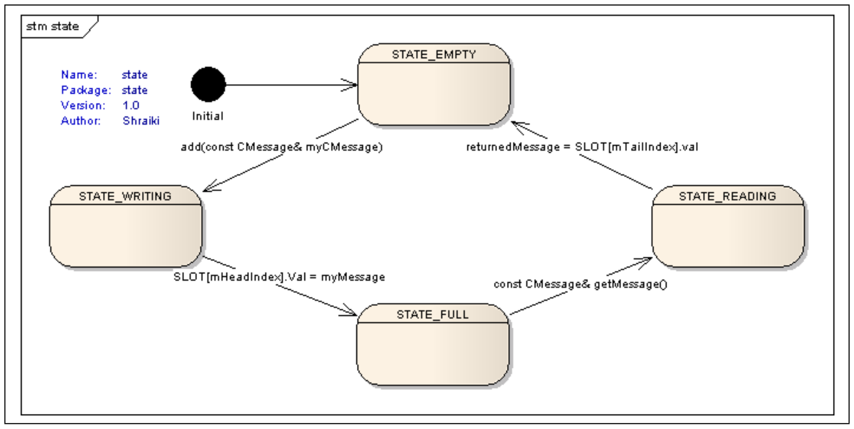

stm state
Name: state
Package: state
Version: 1.0
Author: Shraiki
Initial
STATE_EMPTY
add(const CMessage& myCMessage)
returnedMessage = SLOT[mTailIndex].val
STATE_WRITING
STATE_READING
SLOT[mHeadIndex].Val = myMessage
const CMessage& getMessage()
STATE_FULL

Figure 5.2: *Different states for providing structured access to one CMessage in parallel.*

The fixed size buffer is an array defined in a shared memory. This shared memory is accessible by different processes. The data messages stored in the queue are of type `CMessage`. In the new implementation, a wrapper was added for storing the state of each `CMessage` (see figure 5.1). The combination of `CMessage` and the corresponding state (`mState`) will be called SLOT. It is a kind of wrapper, which includes `CMessage` and the current state (see figure 5.2) of this message. The parallel access to a SLOT has to be serialized within a multi-core system. The atomic compare-and-swap shall be used for that purpose. In case of a parallel add, the current `mHeadIndex` has to be retrieved, and has to be made secure against parallel accesses. This can be done by making a copy from `mHeadIndex` to the stack. This copy has to be protected against data races if the access is not atomic (single word instruction). Using an Int16[1] on an x86 system will work in most cases, but this approach depends highly on the instruction set and architecture of the hardware. The variable on the

[1]Int16 means a primitive data type with the size of 16 bit.

stack is thread-safe, because it is thread-private (saved on the threads stack).

The next step is to check whether the current index (`mHeadIndex`) is already used by another thread. The current state has to be retrieved for this purpose. The problem here lies in the fine details. Whenever the current state is checked, it is possible that it is modified at the same time or just one or two cycles later. In the case of adding a new `CMessage`, it has to be checked if the SLOT has the state `STATE_EMPTY`. If the state is `STATE_EMPTY`, a message can be added. The state has to be changed to `STATE_WRITING` before the data for a message is added. This will prevent other threads from accessing this `SLOT` in parallel. The check of the state (`STATE_EMPTY`) and setting it to `STATE_WRITING` has to be done without any interrupts so that data-races cannot occur[2]. The slot was acquired by another thread if the check was not successful; as such the new updated `mHeadIndex` index has to be retrieved and be used in the thread's own stack to perform the check once again.

The SLOT belongs exclusively to the calling thread after the state is set to `STATE_WRITING`. Then the thread has enough time to complete its operation. For performance reasons, `mHeadIndex` should be incremented directly after a successful compare-and-swap within the add method. This enables another thread to use an updated (incremented) version of the `mHeadIndex`.

The work for adding a message is done when the message is added (stored) to the queue, the current temporary index is stored into `mHeadIndex` and the final state is set to `STATE_FULL`. The following list summarizes how a new `CMessage` is added to the queue.

1. Retrieve the current `mHeadIndex` and store it into a thread-private work-

[2]If the operation is completed there can not be a race with more then one participant.

ing index `tmpIndex`, which will be used for all index related operations.

2. Increment the thread-private working index `tmpIndex`.
3. Check whether the SLOT of this index has the state `STATE_FREE`
   a) If successful, take the slot immediately by setting the state to `STATE_WRITING` and proceed.
   b) If not successful, start from the beginning again.
4. Store `tmpIndex`
5. Finish algorithm by setting the state of the SLOT to `STATE_FULL`

The same approach works similar for the get method.

Using atomic calls without interaction with the kernel should lead to a significant improvement in terms of performance. The current implementation is not using any synchronization mechanisms that are involved with the kernel.

5.2 Strategy for Measuring Performance

For measuring performance and the repeatability of the different strategies and methods, a test bench is created. This test bench has an uniform interface to the function calls which are analyzed. This was achieved using a function pointer. The functions must have the form `void name(void)` in order to be measured from the test bench. Tests were implemented in order to measure:

1. Operating System Primitives
2. POSIX library Primitives
3. OpenMP
4. Intel TBB

The Cilk compiler, which is based on the gcc, was unable to understand OpenMP extensions. Because of that unavailability, tests were not performed with Cilk.

Tests were created for measuring differences between OpenMP, Intel TBB and their serial counterparts. Figure 5.3 gives a schematic overview of the implemented test bench. As shown in the figure, tests can be repeated and T_{Delta} is measured for each iteration. This is done for measuring the average time T_{mean}, the standard deviation T_{std}, the minimum time T_{min} as well as the maximal time T_{max}. These are important metrics for a real-time system, and help to determine whether a deadline will be violated or not.

As shown in the figure, new tests are easily be adapted by adding new functions with the defined signature. Furthermore, the use of a common test bench is a systematical attempt for measuring; as such systematical errors will

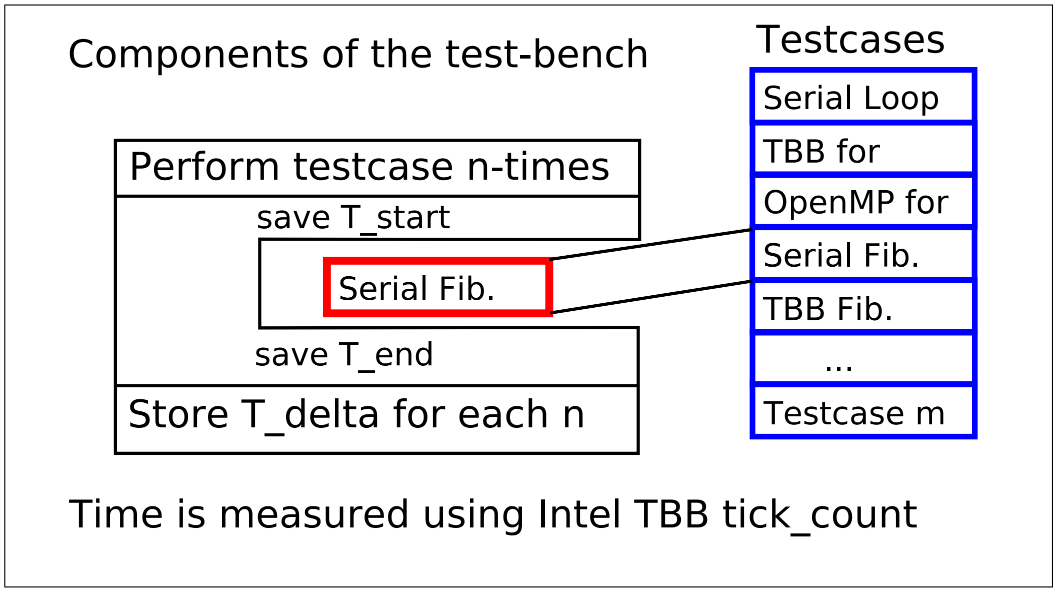


Figure 5.3: *Strategy for testing different approaches for parallelization.*

dominate[3]. This kind of regression test is useful for checking primitives or for computations like the calculation of a Fibonacci number. This kind of test bench is not suitable for testing a whole framework, but becomes reasonable for testing primitives, hence the initial state for a test is easier to achieve.

5.3 Results for measured Test-Cases

Table 5.1 is a summary of the results for the performed tests. As expected, parallelization has a negative impact on determinism, as the standard deviation of the table shows. The TBB framework has superior values concerning standard deviation (T_{std}) and average processing time (T_{mean}).

The time for creating a thread-pool for the TBB is not mentioned in the

[3]in most cases...

		T_{mean}	T_{std}	T_{min}	T_{max}	T_{p-v}
A	Serial Loop	19.618	0.000	1.961	1.963	0.002
	OpenMP For dyn.	3.631	0.002	0.357	0.371	0.014
	OpenMP For stat.	3.646	0.001	0.359	0.369	0.010
	OpenMP For sect.	3.706	0.001	0.364	0.377	0.013
	TBB For	2.595	0.002	0.257	0.274	0.017
B	Serial Fib.	9.163	0.000	0.914	0.917	0.003
	TBB Fib.	49.779	0.001	4.971	4.982	0.011
	Hybr. Fib.	1.192	0.000	0.119	0.120	0.001
C	Nto. Mutex	64.639	0.002	6.448	6.470	0.022
	Pos. Mutex	24.055	0.000	2.404	2.407	0.003

Table 5.1: *Comparison of the performance and the standard deviation between the frameworks.*

tests. Differently than the OpenMP approach, where threads are created (see kernel-trace 3 for more details) and attached after each iteration (see kernel-trace 4 for more details), this additional overhead does not apply for the TBB framework. Performance measurements for parallelizing loops are shown in table 5.1 labeled with A. Intel TBB shows superior results, but as mentioned before, the coding overhead is several times higher in the TBB implementation. The repeatability is better in the OpenMP implementation by using static scheduling.

Performance measurements for nested parallelization are shown in table 5.1 labeled with B. It is interesting to see that the improper use of the TBB framework will lead to poor performance (T_{mean} was approximately 5 times higher) by 100% core utilization. This happened through synchronization overhead in user-level (see kernel-trace 5 for more details). A hybrid approach, using Intel TBB for *larger chunks* of work, and the serial one for *smaller chunks* showed a good performance result.

Performance measurements for the comparison of selected POSIX (using the POSIX library) and Neutrino primitives (accessing the OS features directly) are shown in table 5.1 labeled with C. The POSIX implementation of the mutex is faster than direct system calls for the mutex primitive provided by the Neutrino operating system. It might be possible to increase (slightly) the performance of the Neutrino calls by using other parameters for that call[4]. Unfortunately, the interface for calling the kernel directly is not recommended and not well documented (e.g. for changing the attributes of called synchronization primitive).

The additional overhead for creating threads or for synchronization increases the standard deviation. The performance is increased when using the frameworks correctly, as the example with the Intel TBB framework showed. It depends on the requirements of the application itself whether a standard deviation can be accepted for meeting a deadline or not.

5.4 Strategy for Measuring the lock-free implementation

The queue implementation will be handled differently; hence it shall be tested under *real world conditions* to fit into the existing `H_DA` framework. For this purpose, the `H_DA` framework was adapted and components were created. The `H_DA` framework has several elements for creating a parallel version of an application. It includes object oriented wrappers for using mu-

[4]E.g. enabling a inheritance protocol or not.

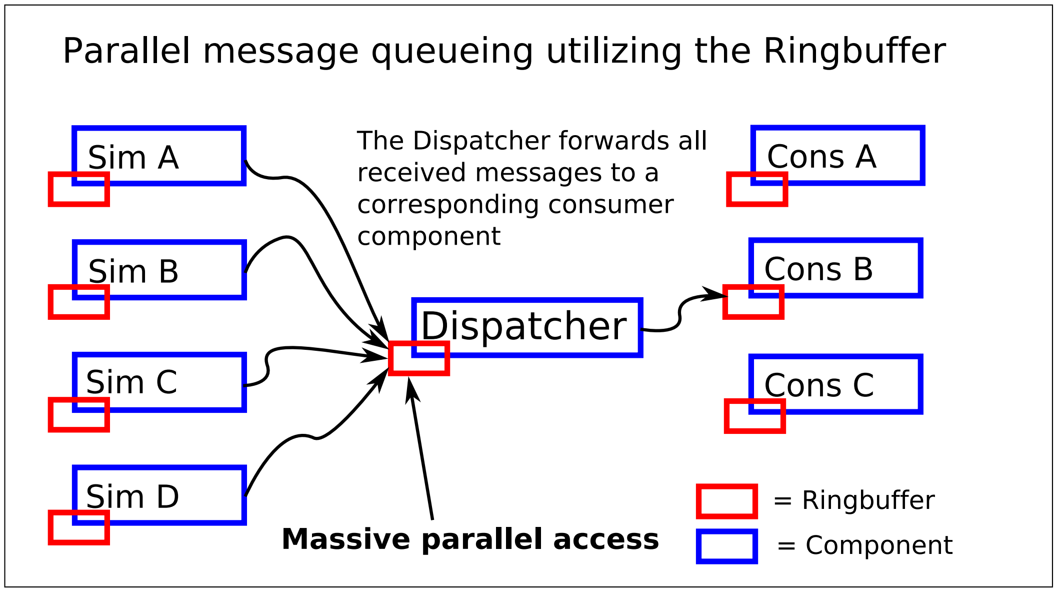

Parallel message queueing utilizing the Ringbuffer
Sim A
Sim B
Sim C
Sim D
The Dispatcher forwards all received messages to a corresponding consumer component
Dispatcher
Cons A
Cons B
Cons C
Massive parallel access
= Ringbuffer
= Component

Figure 5.4: *The queue (class CCommQueue) as important element for communication.*

texes, semaphores, threads, shared-memory and several more. The advanced elements are the components in this framework. They can be used as processes or threads[5] and provide a communication interface, by default. This communication interface uses a queue for storing incoming messages and for forwarding messages into another component. Figure 5.4 shows the importance of the queue. Several components access this queue in parallel, so the access to this element has to be protected (critical section). The original implementation uses a mutex for protecting the access to the queue, and several semaphores. They can signal, in case of a full queue, that related receiver-components have to consume the messages and that producer-components have to wait until new space is available. They signal as well, in case of an

[5]If the software developer implements *a switch*.

empty queue, that producer-components can forward their messages and consumer-components can sleep.

The lock-free implementation does not have such a signaling mechanism, so that in case of a full queue, the add method will attempt to add a new message in a loop, and will tend to busy-wait, which will prevent lower priority threads from gaining processing time.

5.5 The impact of the queue-size

Figure 5.5 shows the weakness of the lock-free implementation for small queue-sizes. The scheduling for this test is set to round-robin with a quantum of 4 ms. Having a full queue during those 4 ms will lead to a spinning behavior, because the lock-free algorithm tries to add the message until the messaged is accepted. The scheduler will not be influenced by a full queue, which is the case within the locked implementation. Here the semaphores are used to indicate the queue status and to change rescheduling behavior. Further scheduling overhead was minimized by limiting the Dispatcher and Receiver to one core on the first processor, and by setting one core of the second processor to one producer.

With increasing queue-size the performance benefit of the lock-free implementation is superior to the locked one. The drop in performance for smaller queue-sizes has its origin from the round-robin scheduling. This limits the processing time for the Dispatcher and for the Consumer to 4 ms, so that the queue-size will become full in about 1 ms. The lock-free algorithm will *spin* the remaining 3 ms. It is interesting that the standard deviation for the

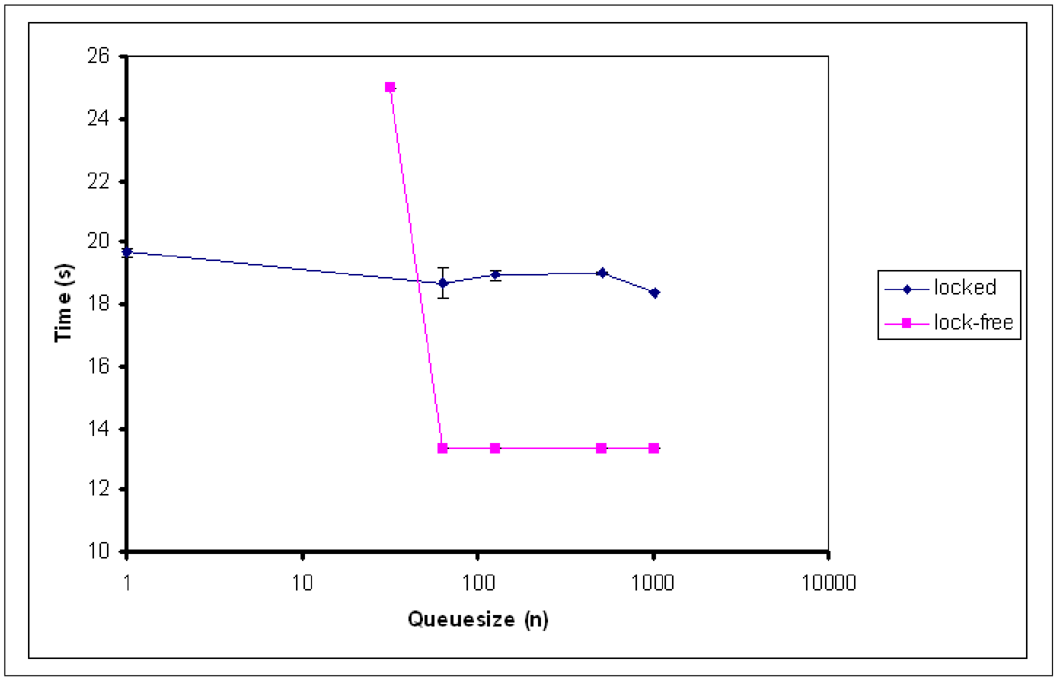


Figure 5.5: *Comparison between the locked and the lock-free implementation related to the queue-size.*

	locked				lock-free			
Size (n)	T_{mean}	T_{std}	T_{min}	T_{max}	T_{mean}	T_{std}	T_{min}	T_{max}
1	19,691	0,369	19,406	20,318	na	na	na	na
32	na	na	na	na	24,997	0	24,997	24,997
64	18,703	0,686	17,677	19,349	13,326	0	13,326	13,326
128	18,953	0,430	18,290	19,460	13,326	0,001	13,325	13,326
512	19,004	0,191	17,677	19,366	13,334	0,003	13,326	13,336
1024	18,379	0,088	18,304	18,531	13,350	0,001	13,348	13,351

Table 5.2: *The lock-lock free implementation has a smaller standard deviation and faster response times for larger queue-sizes.*

lock-free implementation is several orders of magnitude smaller than for the locked one. Table 5.2 shows the result of the measurements. Measurements were performed for six different queue sizes. As shown in the table, larger queue sizes did not significantly affect performance.

5.6 The impact of parallel access

The tests were performed on a motherboard with two Intel XEON 5300 quad-cores. Also the impact of the number of producer was measured. The lock-free implementation showed good scalability in terms of a small overhead (see kernel-trace 9 for more details). The table 5.3 shows that the number of producers has no significant impact on the performance in the lock-free implementation. Like in the queue-size test, further scheduling overhead was minimized by limiting Dispatcher and Receiver to one core of the first processor, and by limiting one Producer to one core of the second processor. All producers tried to access the ring-buffer in *parallel*. Figure 5.6 compares the scaling behavior of the lock-free and the locked implementation when additional producers are added. Additional producers lead to further overhead induced through system calls for locking access to the shared ring-buffer from the Dispatcher. The overhead induced through the state-machine seems reasonable compared to the amount of time spent for a system call and for consumed time through rescheduling (see kernel-trace 6 for more details). Even in this case the standard deviation of the lock-free implementation is small (see table 5.3).

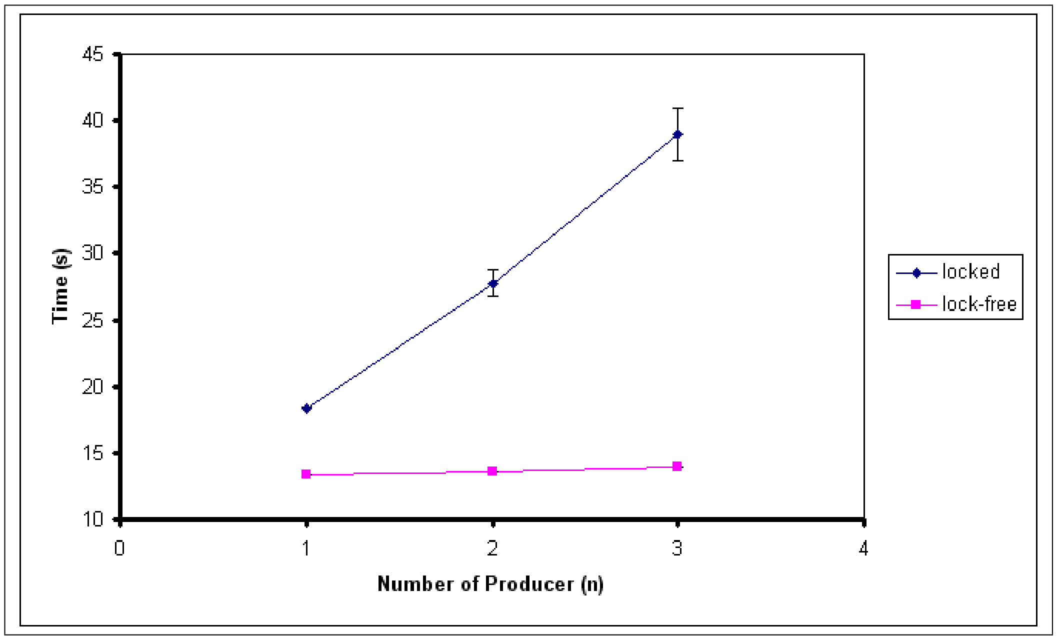


Figure 5.6: *Comparison between the locked and the lock-free implementation related to the number of producers.*

	locked				lock-free			
Prod. (n)	T_{mean}	T_{std}	T_{min}	T_{max}	T_{mean}	T_{std}	T_{min}	T_{max}
1	18,379	0,088	18,304	18,531	13,350	0,001	13,348	13,351
2	27,788	0,975	26,522	28,768	13,628	0,003	13,624	13,631
3	38,980	1,956	36,991	42,095	13,907	0,007	13,898	13,917

Table 5.3: *Additional communication overhead is lower in the lock-free implementation.*

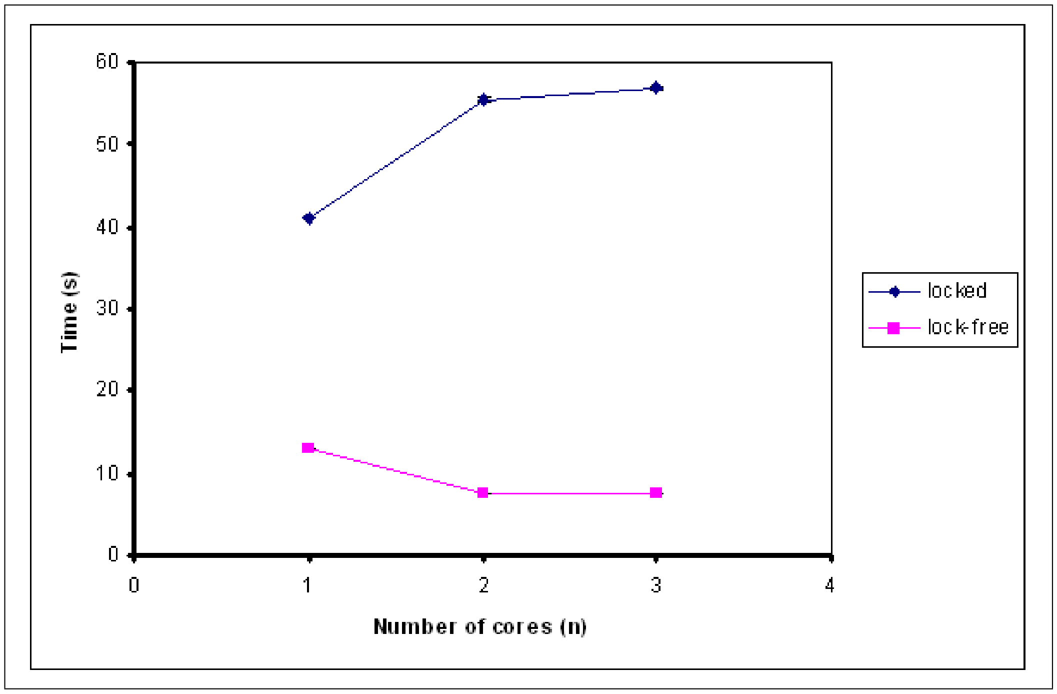


Figure 5.7: *Comparison between the locked and the lock-free implementation related to the number of cores.*

5.7 The impact of additional cores

This test compares how well the implementations will scale when additional cores are added. The scheduling overhead was minimized by limiting each producer to one (of the four) core(s) of the second processor. The Dispatcher and Receiver[6] were the only components that were able to gain benefit from additional cores. The current framework uses only one thread for each component. As such the Dispatcher has only one thread for delegating messages to other components (Receiver). This limits the overall performance because the Dispatcher communicates only with one Receiver in parallel. The fig-

[6]Only one Receiver is being used during the tests.

	locked				lock-free			
Core (n)	T_{mean}	T_{std}	T_{min}	T_{max}	T_{mean}	T_{std}	T_{min}	T_{max}
1	41,178	0,029	41,122	41,201	13,086	0,000	13,086	13,086
2	55,471	0,247	55,217	55,792	7,573	0,001	7,572	7,573
3	56,968	0,176	56,711	57,191	7,573	0,001	7,572	7,574

Table 5.4: *Linear increase in performance for the lock-free implementation, when adding additional cores.*

ure 5.7 shows the limits of the test environment for core utilization. Further threads must be added to the Dispatcher component for enhancing the message throughput. However, the lock-free implementation scales with the number of cores almost linearly (see kernel-trace 7 for more details), because performance is approximately doubled. Similar to the previous comparison, the standard deviation is small compared to that of the locked implementation. Another interesting point is the measured time for the locked implementation. The performance seems to decrease because of the message overhead induced through locks of the four producers (see kernel-trace 6 for more details).

The comparison of the locked and the lock-free implementation showed that even on a single-core system the performance improvement is significant. The implementation overhead seems to be higher, but the results showed a good standard deviation and a performance benefit as well for the lock-free implementation. The current implementation does not use any signaling semaphores. This will increase performance for smaller queue-sizes on single-core systems, since components with full or empty queues will not spin, but sleep. It is an interesting result that the locked implementation does not scale on a multi-core system; furthermore, performance seems to decrease at a small ratio when further cores are utilized. Hence, the standard deviation is much smaller for the lock-free implementation. It will become important for real-time and for hard real-time systems. However, the spinning behavior does not fit into a hard real-time environment. This spinning behavior must be limited, e.g. through an additional counter, which can be easily implemented in the methods `add` and `get` of the lock-free implementation. It reduces the amount of allowed *spins* during acquiring a `SLOT`.

Another point is that the *spins* may be performed several times in parallel, which will lead to an overhead of the amount of *spins*. This overhead could be reduced by executing a `usleep(rand()%100)` if compare-and-swap fails. However, modifications which change the scheduling behavior will have an impact on the standard deviation. So it has to be measured and evaluated for use within a real world system.

5.8 Summary

This chapter introduced the test environment in which the performances of the frameworks, from the previous chapter, were measured. The benchmarks for Intel TBB were superior to OpenMP. OpenMP creates threads and delegates to them the work from the main thread, whereas TBB uses an initial thread-pool and assigns tasks directly to it. The delegation of the work in OpenMP introduces more overhead in terms of time, compared to the TBB approach. TBB has additional capabilities like nested parallelism. Nested parallelism reduces (when properly implemented) the time for recursive calculations dramatically.

Additionally, a comparison between the lock-free and the locked mechanism was performed. The lock-free implementation has superior results, which comes from the fact that the involvement of the kernel is minimized and scheduling of the operating system is not changed. The lock-free queue has not only a significant better benchmark, but also a reduced standard deviation (i.e. a better determinism), which is important for real-time systems.

Part III

Evaluation

CHAPTER 6

DISCUSSION AND OUTLOOK

An improvement of performance through core utilization is possible, as shown in the previous chapter.

Lock-free and wait-free algorithms are complicated and not commonly used for real-time applications. Recent implementations of the LINUX kernel include some of those constructs. The open source community was able to reduce the response times of the Linux kernel below 1 ms by using read-copy-update primitives for synchronization.

Lock-free algorithms are not suited for hard real-time systems, because of their characteristic to spin if an operation was not successful. A worst-case analysis will not be possible if this spinning is not performed in a controlled manner. It is possible through a simple stack based counting variable within the loops to restrict this spinning behavior to, for example, 12 iterations. So, whenever the compare-and-swap operation fails, the counting variable has to be incremented to reduce the amount of spins to e.g. 12.

The spinning overhead can also be reduced through the introduction of a sleep operation. E.g. whenever compare-and-swap fails, the thread sleeps a randomized time `usleep(rand()%100)`, giving another thread the possibility to aquire the `SLOT`.

Those mentioned techniques may enable the use of lock-free algorithms in a hard real-time environment.

From the overall system perspective, a spinning thread will utilize 100% of the core computation. As such, it is possible, depending on the number of cores and on the priority of the spinning thread, that lower priority threads will not have time to perform their work. In case of the implemented lock-free ring-buffer, this case will occur whenever the ring-buffer is filled or busy (e.g. `STATE_WRITING`). So, for the sake of efficiency and stability, a signaling semaphore should be included to indicate whether the ring-buffer is full or not.

Another interesting point is that locked mechanisms suffer from rescheduling. A thread, when being blocked from the scheduler, is added at the end of the schedule list. Even if this thread is marked as `READY`, the whole schedule list must be iterated until a processor is assigned to this thread.

This time for rescheduling does not exist in the lock-free implementation. Hence, each thread is always busy and therefore ready for the CPU. So the scheduling list will not change in that case.

The compare-and-swap operation has to be protected in a multi-core system using a hardware memory lock. Otherwise, several cores may perform the compare-and-swap mechanism for the same referenced data at the same time, which can lead to an error. So, a hardware memory lock is used, which consumes approximately 3-5 processor cycles, to prevent core interaction at the same time. This locks the memory access, which reduces the system performance significantly depending on the number of cores and on the frequency of the memory lock. It is an interesting point that the double-check paradigm increases the overall system performance.[SH], [WT05]

It has to be checked whether the condition for enabling the memory-lock becomes true by using an if-statement. The hardware memory lock is set only if the condition is true. This minimizes the calling frequency of the hardware memory-lock, hence other cores are enabled to access memory over *a longer period of time*.

Wait-free algorithms are *desired* for a hard real-time system, since they will complete in a defined number of steps. Such algorithms can not be implemented for all requirements within a real-time system, because a solution may simply not exist. In case of a multi-core system these algorithms are interesting because of their possibility to scale (more cores add more overhead) and their efficiency.

The introduced frameworks solve the work decomposition differently. Intel TBB uses an initial thread pool for the reduction of the thread creation overhead, whereas threads in OpenMP are created and destroyed after their usage, and Cilk uses the approach of task stealing. All of those frameworks utilize additional cores and are scalable. It is also possible to change the scheduling behavior for improving load-balancing[1], which is not important for a real-time

[1] Each core will process the same amount of work.

system, especially when those load-balancing algorithms may lead to lower efficiency and to decreased predictability.

So, for hard real-time tasks, those parallelization frameworks might not be suitable for all cases. But, for the soft real-time domain of a system (maybe a process at a lower priority), they might be suitable and help to exploit additional cores.

QNX Neutrino provides the possibility to log user events at the system-level through an instrumented kernel. This is a useful feature for measuring times, for giving a snapshot[2] of the current scheduling and for analyzing the predictability of the overall system.

I was able to get Cilk, Intel TBB and OpenMP to run on QNX Neutrino. All of those frameworks are based on the POSIX library. QNX Neutrino claims to be a POSIX conform operating system, so most of the port was achieved through pure POSIX conformity. QNX Neutrino uses its own compiled version of the GNU compiler, which has no OpenMP support. By retrieving the sources for this compiler, pre-compiled libraries for QNX Neutrino, and some modifications to the build-hooks file, I was able to compile my own version of the GNU compiler, which works for QNX Neutrino.

It was not necessary to change sources for Cilk, it compiled directly within QNX. In order to get Intel TBB to run, some lines must be changed for the direct hardware access.

The example of the Intel TBB Fibonacci Numbers showed that use of work decomposition, grain-size and related threading overhead must be understood to gain performance. The parallel TBB implementation for n=40 was

[2]Kernel traces for more then 3 seconds, where in most cases larger then 1 GByte.

5 times slower in finding the Fibonacci number than the serial approach. This could be interpreted as scheduling overhead at a single-core system, but not at an 8-core system where all cores are utilized to 100%, which was the test bench.

Frameworks reduce the overhead induced by parallelization. It is my opinion that parallelism should be gained through the use of those frameworks for the sake of stability and transparency. It might become difficult to get specialized frameworks for all problems[3], but they reduce the programming time for exploiting additional cores.

For hard real-time systems, the additional overhead that comes from a generic solution of a framework might be unacceptable. In these cases, one must integrate their own solution into the system.

Intel TBB claims to work in combination with OpenMP and vice versa. Both frameworks are based on the POSIX library and are compatible to POXIX conform calls. But it is possible to interfere with those frameworks by setting the affinity or blocking a team of threads by using mutexes. OpenMP and Cilk use their own compiler, which might lead to problems within different environments.

The current trend goes towards parallelization; frameworks are being improved to get more benefit from additional cores. Cilk++, which is a commercial version of Cilk, adds new features to the framework for parallelizing loops. Intel TBB improved the task construct in their new version and claims to have a significant increase in performance, and OpenMP introduces the task construct.

[3]And they might be not compatible to each other.

Since each new version of a framework may change its implementation details and its scheduling behavior, the used version must be understood in order to gain performance benefit from those frameworks in a real-time environment.

Chapter 7

Conclusions

Gaining benefit through multi-core processing by fulfilling hard real-time conditions is a very complex topic. An additional layer of complexity is added when moving to real parallelism, because data can be processed *in parallel*. For meeting hard real-time conditions, each of the introduced layers must be considered in detail. This thesis gives an overview of the new complexity and the possible benefits when dealing with multi-core systems.

The QNX Neutrino instrumented kernel provides an easy tracing capability for monitoring system events. The Intel TBB framework provides measuring tools for a thread-safe time measurement in user space. Cilk, Intel TBB, and an OpenMP enabled GNU compiler for QNX were ported to QNX Neutrino (see appendix: *Setup of the Environment*). It was not possible to get a direct timing comparison between Cilk and the other frameworks, because the Cilk compiler was not accepting OpenMP constructs and vice versa.

In general, knowledge is **the key** for dealing with multi-core systems. A developer has to know about different technologies like *wait-free* and *lock-free* algorithms. It is also important to choose the correct frameworks for the determined problem.

Cilk exploits nested parallelism, whereas Intel TBB may lead to coding overhead. This coding overhead will be reduced with the introduction of lambda constructs in the new upcoming C++ standard. OpenMP provides an easy interface for loop-parallelism, whereas Cilk does not have such a construct. The strategy for choosing a correct approach for parallelization depends not only on the specific problem but on the grain size as well.

This problem was emphasized in the Intel TBB Fibonacci example. Two different solutions for the problem were implemented, and the appropriate solution was chosen during runtime. STAPL goes one step further by adding an interface for adapting the correct method during runtime.

It is important to know that those frameworks hide much of the implementation details. Using those frameworks without mastering implementation details will lead, for sure, to an unpredictable behavior of a real-time system. The chunk size and the scheduling must be changeable by the developer for enhancing determinism and efficiency. The result of the different measured frameworks Cilk, Intel TBB and OpenMP shows that Intel TBB is the *more matured* approach. The features of Intel TBB cover not only parallelism and nested parallelism, as well as for-loop constructs, but also parallel containers, constructs for pipelining and a thread-safe exception environment. For that reason, Intel TBB should be preferred in real-world applications, even when introducing code overhead.

For proving the efficiency of lock-free algorithms, the ring-buffer implementation from the `H_DA` framework was modified, and the throughput was measured. The result showed approximately 8 times superior performance of the lock-free implementation compared to the original one. Secondly, the standard deviation is significantly better in the lock-free implementation. Lock-free implementations behave more deterministic compared to locked implementations. They should also be considered for hard real-time systems, when a limitation for the spinning behavior is guaranteed. A hybrid approach between the lock-free and the original implementation seems to be feasible for avoiding deadlocks and reducing core load simultaneously.

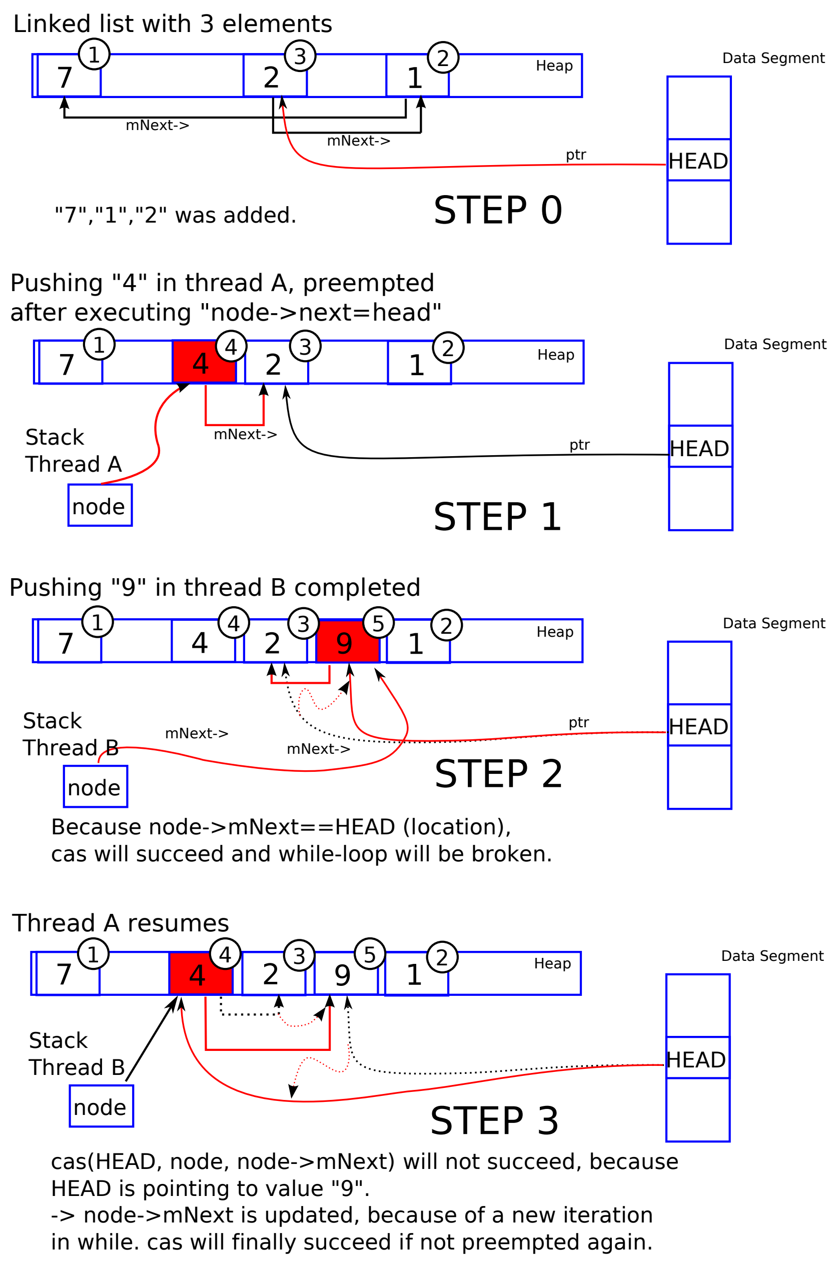
As such, it is possible to create efficient frameworks or primitives for exploiting multiple cores in a real-time environment. In my opinion, multi-core systems should only be exploited by the use of frameworks. The analyzed frameworks provide a generic solution to many problems and are efficient in distributing work to additional cores. The possibility to adapt those frameworks to many problems improves the maintenance of the overall system.

It has been shown that response times can be shortened in real-time systems by a correct utilization of multiple cores, hence the hypothesis is confirmed.

APPENDIX

Illustrating the ABA problem

B


Linked list with 3 elements

"7","1","2" was added.

Pushing "4" in thread A, preempted after executing "node->next=head"

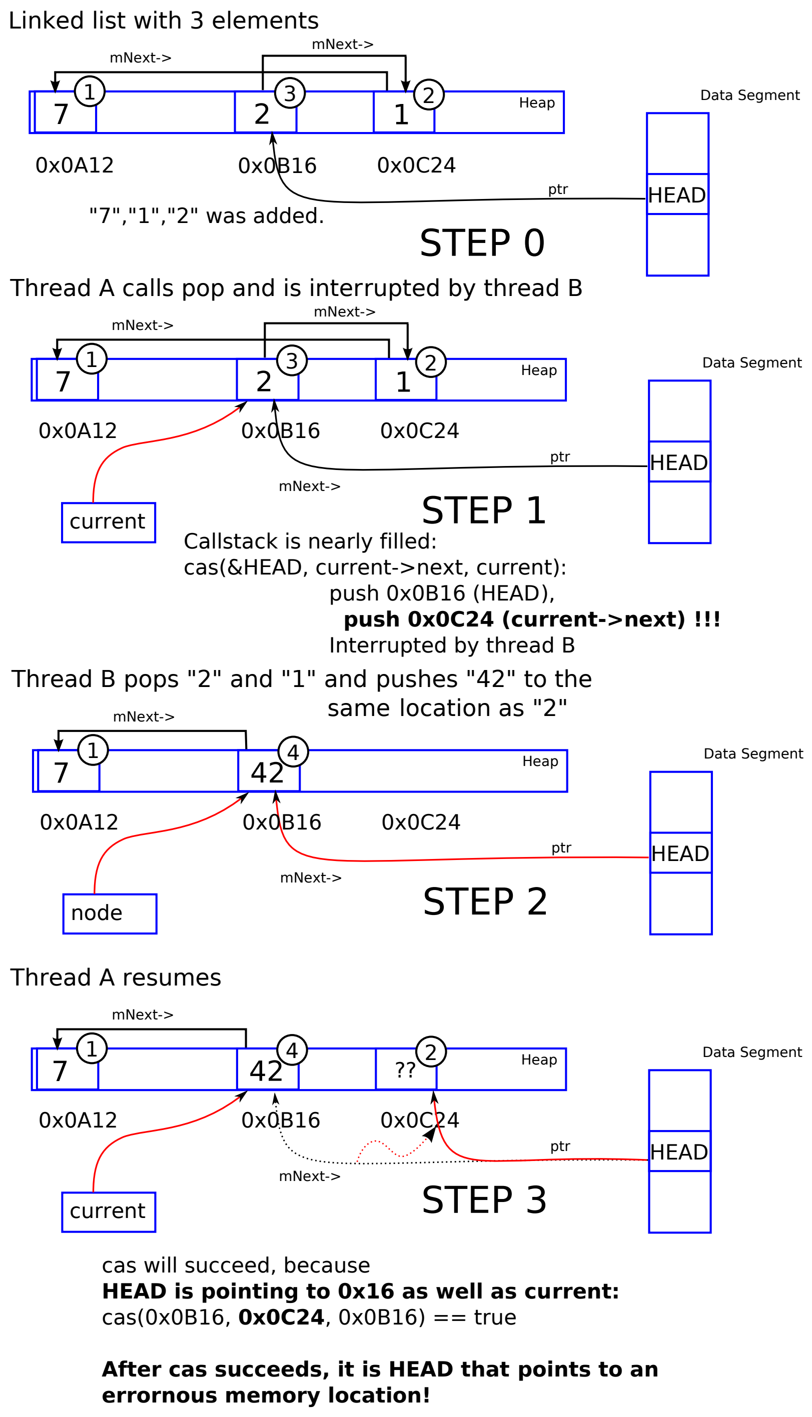
Pushing "9" in thread B completed

Because node->mNext==HEAD (location), cas will succeed and while-loop will be broken.

Thread A resumes

cas(HEAD, node, node->mNext) will not succeed, because HEAD is pointing to value "9".
-> node->mNext is updated, because of a new iteration in while. cas will finally succeed if not preempted again.

Figure 1: *The work flow of the Node implementation.*



Linked list with 3 elements

"7","1","2" was added.

Thread A calls pop and is interrupted by thread B

Callstack is nearly filled:
cas(&HEAD, current->next, current):
push 0x0B16 (HEAD),
push 0x0C24 (current->next) !!!
Interrupted by thread B

Thread B pops "2" and "1" and pushes "42" to the same location as "2"

Thread A resumes

cas will succeed, because
HEAD is pointing to 0x16 as well as current:
cas(0x0B16, **0x0C24**, 0x0B16) == true

After cas succeeds, it is HEAD that points to an errornous memory location!

Figure 2: *The ABA problem illustrated by example.*

Listings for the test bench

Listing 1: *Function pointer for test decoupling.*

```
METRIC performTest(void (*fkt_ptr)(void), const int& myLoop){
  METRIC m;
  MEASURE t;
  for(int i=0; i < myLoop; i++){
    t.start();     // start measurement for iteration i
    (*fkt_ptr)();
    t.end();       // end measurement for iteration i
    if (0==i){     // save the initial time
      m.min=t.delta().seconds();
      m.max=t.delta().seconds();
      m.sum=t.delta().seconds();
    }else{         // save current min, max and the sum
      if (t.delta().seconds()<m.min)m.min=t.delta().seconds();
      if (t.delta().seconds()>m.max)m.max=t.delta().seconds();
      m.sum+=t.delta().seconds();
    }
  }
  return m;
}
```

The listing 2 shows how functions are called and the results stored.

Listing 2: *Loop tests.*

```
  m = performTest(&serial, LOOP);
  printResult(m);
  m = performTest(&TBB, LOOP);
  printResult(m);
  m = performTest(&Cilk, LOOP);
  printResult(m);
  m = performTest(&OpenMP, LOOP);
  printResult(m);
```

The function `performTest` is used for measuring as listing 1 shows. All measured times are added together to get the average value. As a second, the minimum and maximum time for each iteration are saved as well.

Listing 3: *Testing the Neutrino mutex primitive.*

```
void mutex_test_nto(){
  int i,j,k, tmp;

  sync_t sync;
  sync_attr_t attr;

//  fill_n(&attr, sizeof(sync_attr_t), 0);

  //attr.__prioceiling = _NTO_SYNC_NONRECURSIVE;
  attr.__clockid=0;
  attr.__flags=_NTO_ATTR_MUTEX||_NTO_SYNC_PRIONONE ||
      _NTO_SYNC_NOERRORCHECK || _NTO_SYNC_NONRECURSIVE;
  attr.__prioceiling=1;
  attr.__protocol = 1;


  int tmps = SyncTypeCreate_r(
      _NTO_SYNC_MUTEX_FREE,
          &sync,
          &attr);
       printf("Error during creation %d\n",tmps);


  for (i=0; i< STRESS; ++i){
    for (j=0; j< STRESS; ++j){
      for (k=0; k< STRESS; ++k){
        // nop, see compiler settings
        SyncMutexLock_r(&sync);
        tmp = tmp + sin(i) + cos(j) + log(k);
        SyncMutexUnlock(&sync);
      }
    }
  }
  SyncDestroy(&sync);
}
```

Listing 4: *Testing the parallel for construct from TBB.*

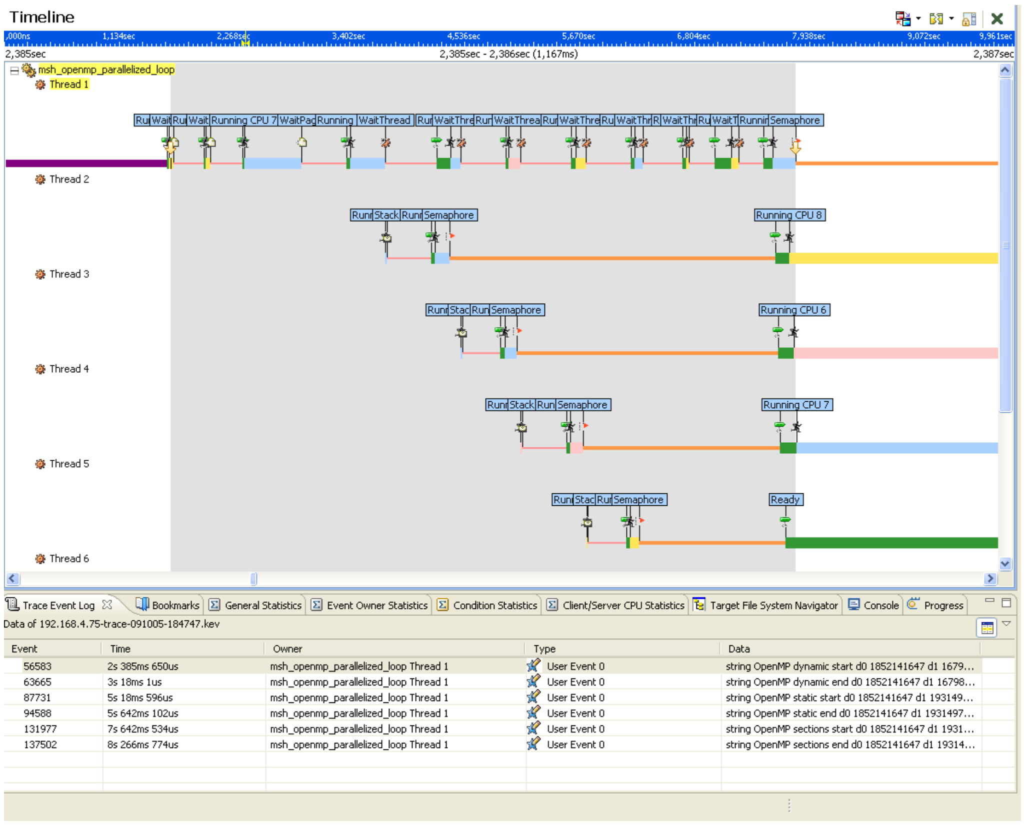
```
class CFunctor {
public:
  void operator( )( const blocked_range<size_t>& r ) const {
    int tmp(0);
    for( size_t i=r.begin(); i!=r.end( ); ++i ){
      for (int j=0; j< STRESS; ++j){
        for (int k=0; k< STRESS; ++k){
          // nop, see compiler settings
          tmp = tmp + sin(i) + cos(j) + log(k);
        }
      }
    }
  }

  CFunctor()
  {}
};

void tbb_parallel(){
  parallel_for(blocked_range<size_t>(0,STRESS,STRESS/8),
      CFunctor() );

  #ifdef DEBUG
  printf("%20s%10qd \n", "tbb_parallel:", t.delta());
  #endif
}
```

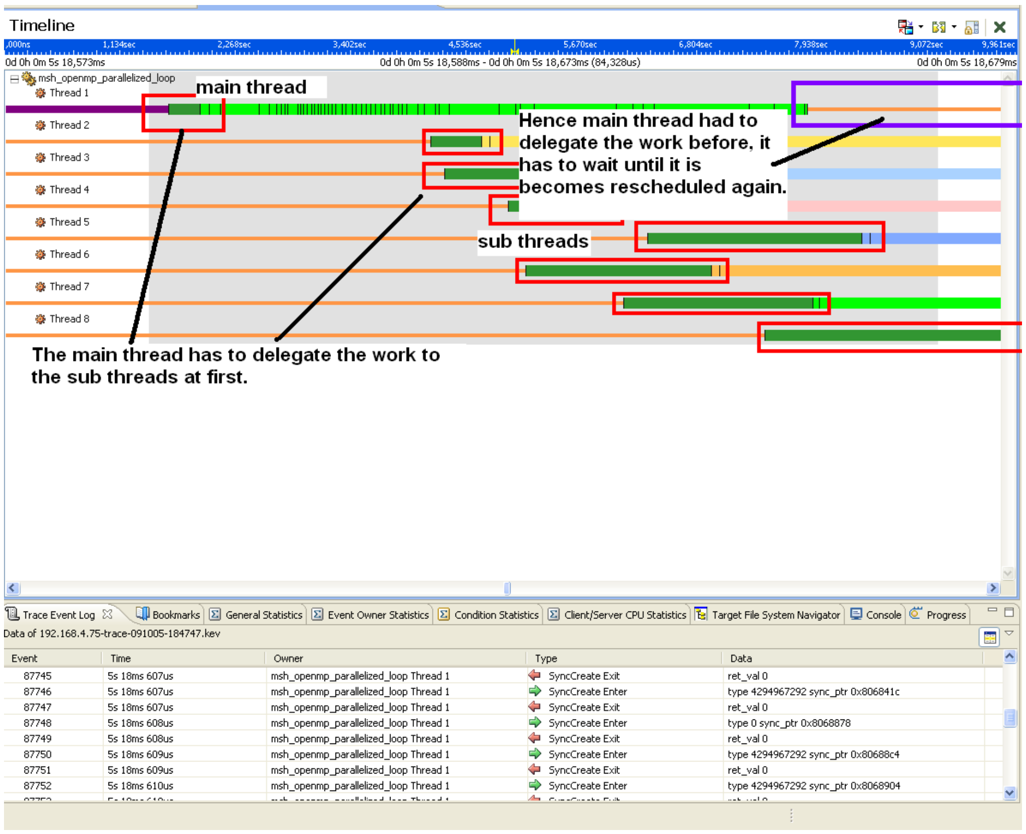
Kernel traces



Data of 192.168.4.75-trace-091005-184747.kev

Event	Time	Owner	Type	Data
56583	2s 385ms 650us	msh_openmp_parallelized_loop Thread 1	User Event 0	string OpenMP dynamic start d0 1852141647 d1 1679…
63665	3s 18ms 1us	msh_openmp_parallelized_loop Thread 1	User Event 0	string OpenMP dynamic end d0 1852141647 d1 16798…
87731	5s 18ms 596us	msh_openmp_parallelized_loop Thread 1	User Event 0	string OpenMP static start d0 1852141647 d1 193149…
94588	5s 642ms 102us	msh_openmp_parallelized_loop Thread 1	User Event 0	string OpenMP static end d0 1852141647 d1 1931497…
131977	7s 642ms 534us	msh_openmp_parallelized_loop Thread 1	User Event 0	string OpenMP sections start d0 1852141647 d1 1931…
137502	8s 266ms 774us	msh_openmp_parallelized_loop Thread 1	User Event 0	string OpenMP sections end d0 1852141647 d1 19314…

Figure 3: *OpenMP creates new threads and synchronizes them using a semaphore.*



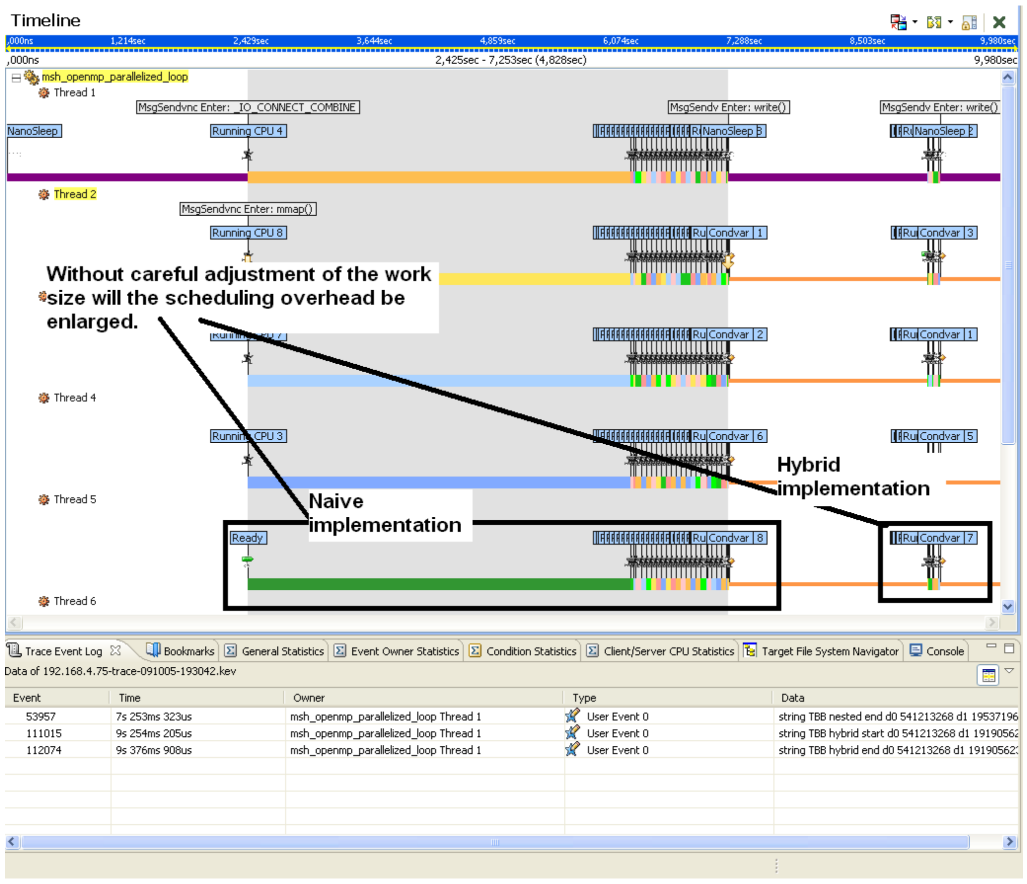
Hence main thread had to delegate the work before, it has to wait until it is becomes rescheduled again.

The main thread has to delegate the work to the sub threads at first.

Data of 192.168.4.75-trace-091005-184747.kev

Event	Time	Owner	Type	Data
87745	5s 18ms 607us	msh_openmp_parallelized_loop Thread 1	SyncCreate Exit	ret_val 0
87746	5s 18ms 607us	msh_openmp_parallelized_loop Thread 1	SyncCreate Enter	type 4294967292 sync_ptr 0x806841c
87747	5s 18ms 607us	msh_openmp_parallelized_loop Thread 1	SyncCreate Exit	ret_val 0
87748	5s 18ms 608us	msh_openmp_parallelized_loop Thread 1	SyncCreate Enter	type 0 sync_ptr 0x8068878
87749	5s 18ms 608us	msh_openmp_parallelized_loop Thread 1	SyncCreate Exit	ret_val 0
87750	5s 18ms 609us	msh_openmp_parallelized_loop Thread 1	SyncCreate Enter	type 4294967292 sync_ptr 0x80688c4
87751	5s 18ms 609us	msh_openmp_parallelized_loop Thread 1	SyncCreate Exit	ret_val 0
87752	5s 18ms 610us	msh_openmp_parallelized_loop Thread 1	SyncCreate Enter	type 4294967292 sync_ptr 0x8068904

Figure 4: *OpenMP attaches to already spawned threads.*

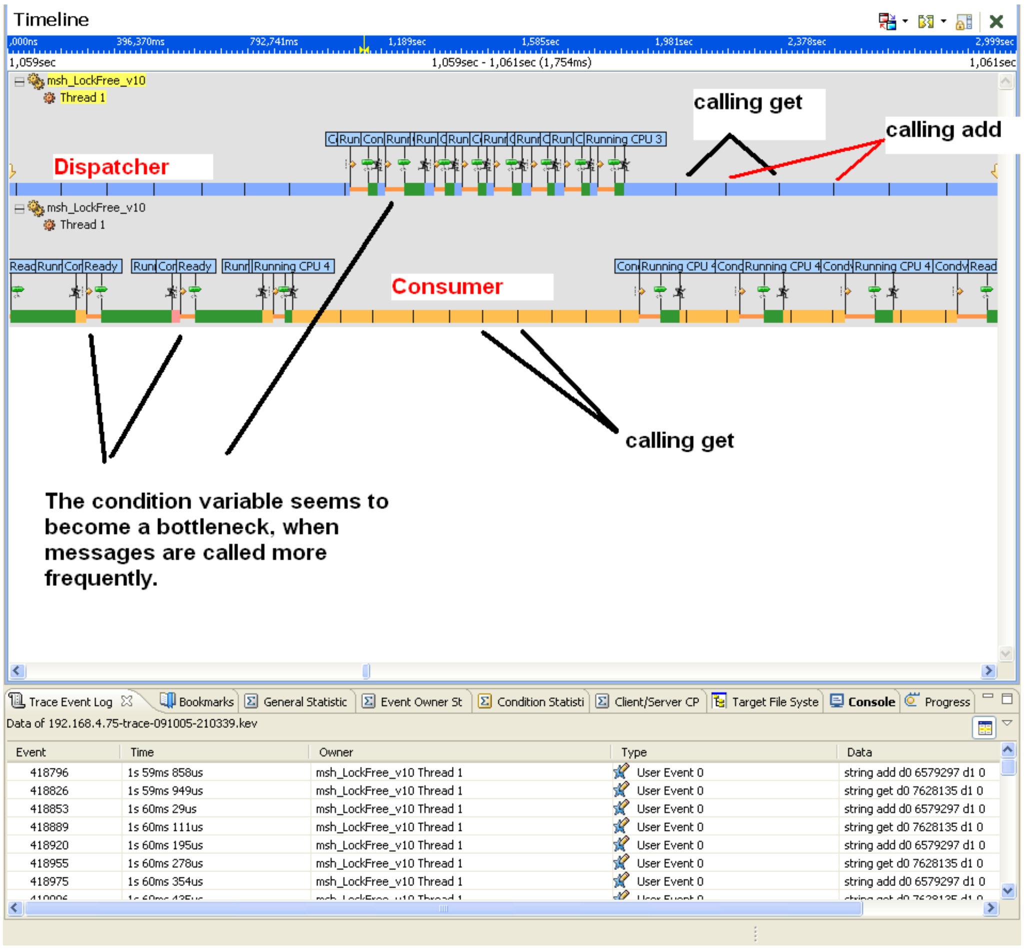


Trace Event Log | Bookmarks | General Statistics | Event Owner Statistics | Condition Statistics | Client/Server CPU Statistics | Target File System Navigator | Console

Data of 192.168.4.75-trace-091005-193042.kev

Event	Time	Owner	Type	Data
53957	7s 253ms 323us	msh_openmp_parallelized_loop Thread 1	User Event 0	string TBB nested end d0 541213268 d1 19537196
111015	9s 254ms 205us	msh_openmp_parallelized_loop Thread 1	User Event 0	string TBB hybrid start d0 541213268 d1 19190562
112074	9s 376ms 908us	msh_openmp_parallelized_loop Thread 1	User Event 0	string TBB hybrid end d0 541213268 d1 19190562

Figure 5: *Comparison between the naive TBB implementation and the hybrid approach.*

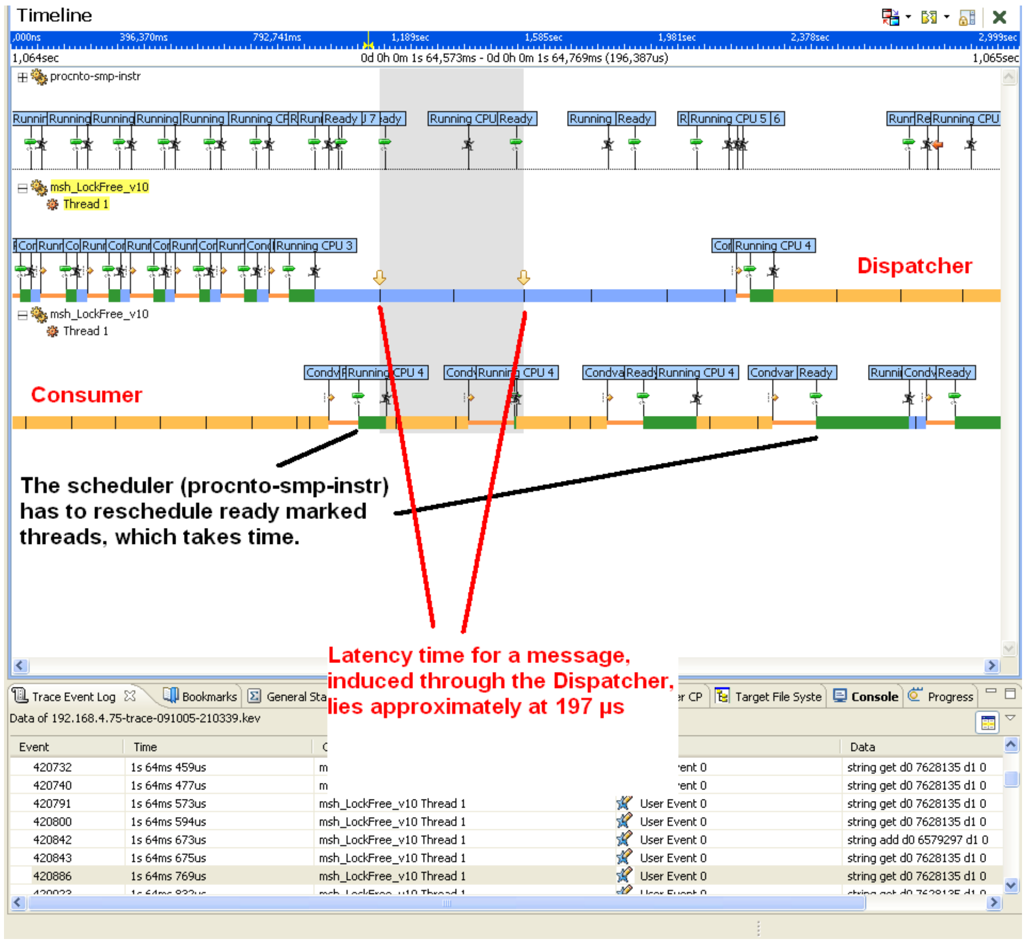


The condition variable seems to become a bottleneck, when messages are called more frequently.

Data of 192.168.4.75-trace-091005-210339.kev

Event	Time	Owner	Type	Data
418796	1s 59ms 858us	msh_LockFree_v10 Thread 1	User Event 0	string add d0 6579297 d1 0
418826	1s 59ms 949us	msh_LockFree_v10 Thread 1	User Event 0	string get d0 7628135 d1 0
418853	1s 60ms 29us	msh_LockFree_v10 Thread 1	User Event 0	string add d0 6579297 d1 0
418889	1s 60ms 111us	msh_LockFree_v10 Thread 1	User Event 0	string get d0 7628135 d1 0
418920	1s 60ms 195us	msh_LockFree_v10 Thread 1	User Event 0	string add d0 6579297 d1 0
418955	1s 60ms 278us	msh_LockFree_v10 Thread 1	User Event 0	string get d0 7628135 d1 0
418975	1s 60ms 354us	msh_LockFree_v10 Thread 1	User Event 0	string add d0 6579297 d1 0

Figure 6: *Overview of the queue (CCommQueue) implementation using locking mechanisms.*



The scheduler (procnto-smp-instr) has to reschedule ready marked threads, which takes time.

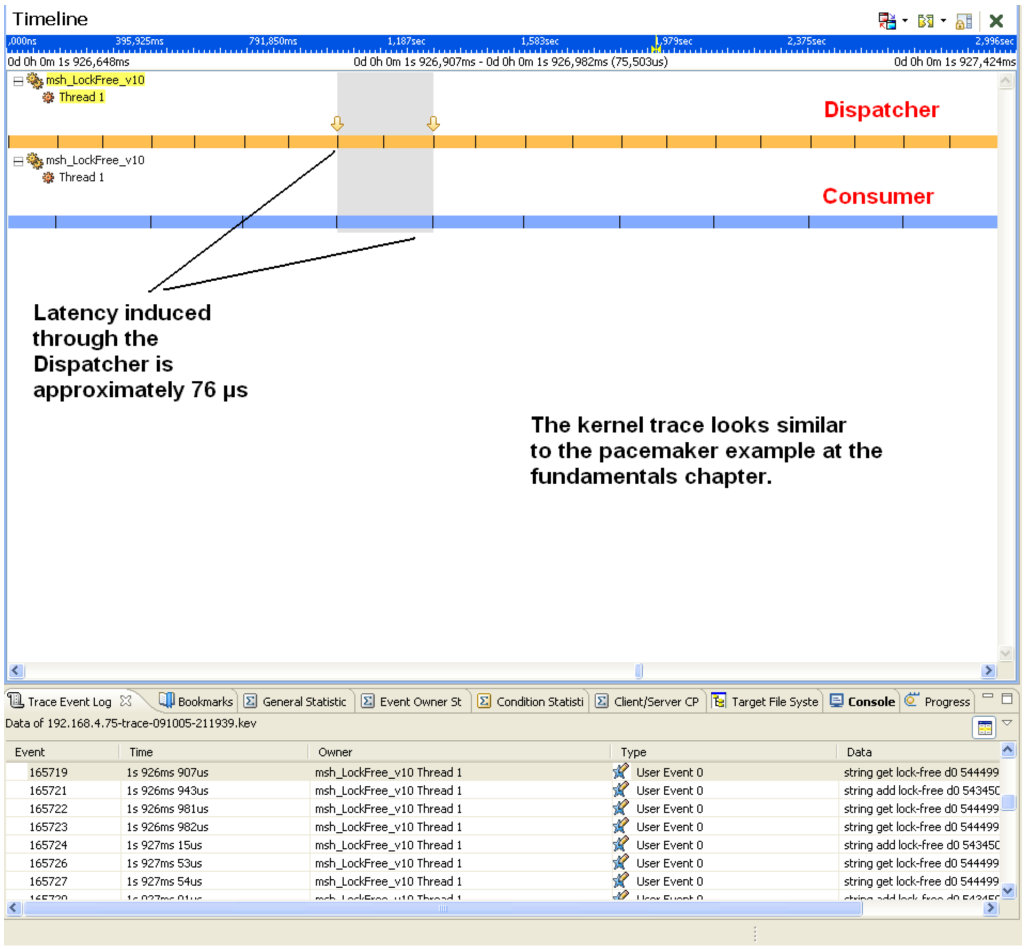
Latency time for a message, induced through the Dispatcher, lies approximately at 197 µs

0d 0h 0m 1s 64,573ms - 0d 0h 0m 1s 64,769ms (196,387us)

Data of 192.168.4.75-trace-091005-210339.kev

Event	Time	C		Data
420732	1s 64ms 459us	m	ent 0	string get d0 7628135 d1 0
420740	1s 64ms 477us	m	ent 0	string get d0 7628135 d1 0
420791	1s 64ms 573us	msh_LockFree_v10 Thread 1	User Event 0	string get d0 7628135 d1 0
420800	1s 64ms 594us	msh_LockFree_v10 Thread 1	User Event 0	string get d0 7628135 d1 0
420842	1s 64ms 673us	msh_LockFree_v10 Thread 1	User Event 0	string add d0 6579297 d1 0
420843	1s 64ms 675us	msh_LockFree_v10 Thread 1	User Event 0	string get d0 7628135 d1 0
420886	1s 64ms 769us	msh_LockFree_v10 Thread 1	User Event 0	string get d0 7628135 d1 0

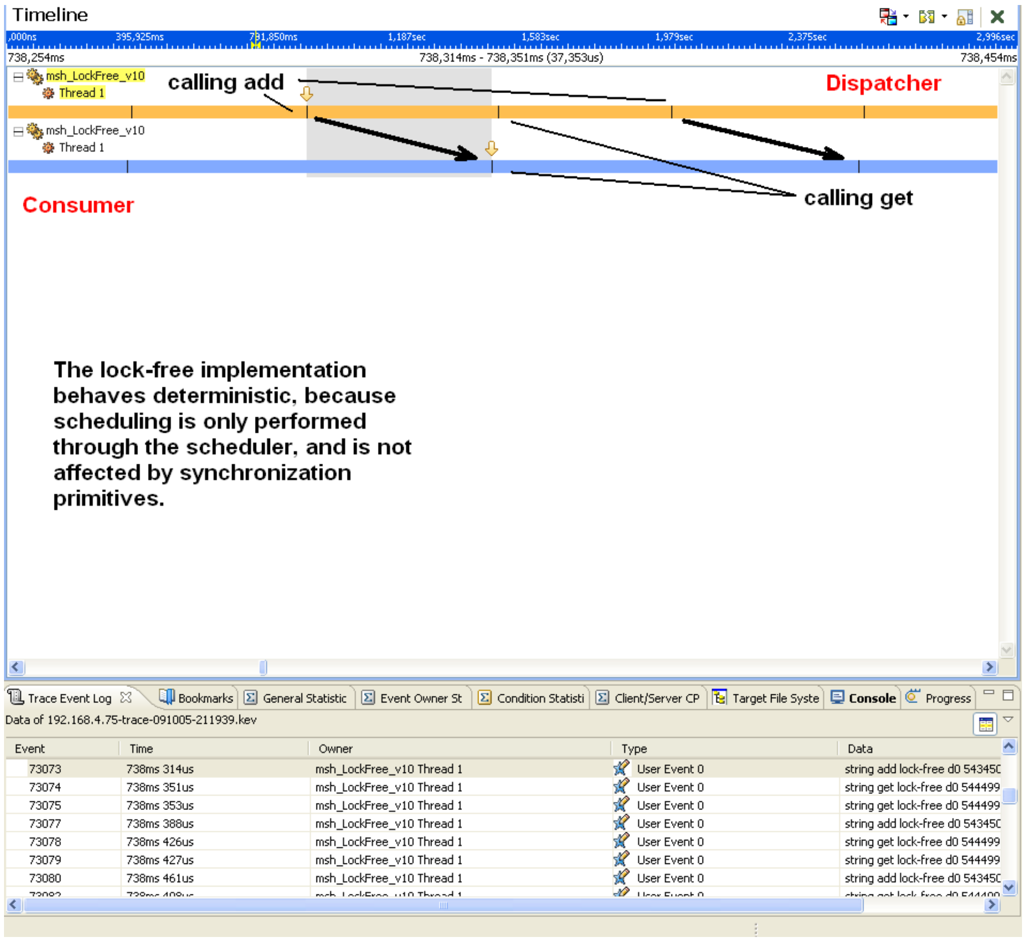
Figure 7: *Latency-time, induced through the Dispatcher, in the locked implementation.*



Data of 192.168.4.75-trace-091005-211939.kev

Event	Time	Owner	Type	Data
165719	1s 926ms 907us	msh_LockFree_v10 Thread 1	User Event 0	string get lock-free d0 544499
165721	1s 926ms 943us	msh_LockFree_v10 Thread 1	User Event 0	string add lock-free d0 54345C
165722	1s 926ms 981us	msh_LockFree_v10 Thread 1	User Event 0	string get lock-free d0 544499
165723	1s 926ms 982us	msh_LockFree_v10 Thread 1	User Event 0	string get lock-free d0 544499
165724	1s 927ms 15us	msh_LockFree_v10 Thread 1	User Event 0	string add lock-free d0 54345C
165726	1s 927ms 53us	msh_LockFree_v10 Thread 1	User Event 0	string get lock-free d0 544499
165727	1s 927ms 54us	msh_LockFree_v10 Thread 1	User Event 0	string get lock-free d0 544499
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Figure 8: *Overview of the queue (CCommQueue) implementation using the lock-free approach.*



The lock-free implementation behaves deterministic, because scheduling is only performed through the scheduler, and is not affected by synchronization primitives.

Data of 192.168.4.75-trace-091005-211939.kev

Event	Time	Owner	Type	Data
73073	738ms 314us	msh_LockFree_v10 Thread 1	User Event 0	string add lock-free d0 54345C
73074	738ms 351us	msh_LockFree_v10 Thread 1	User Event 0	string get lock-free d0 544499
73075	738ms 353us	msh_LockFree_v10 Thread 1	User Event 0	string get lock-free d0 544499
73077	738ms 388us	msh_LockFree_v10 Thread 1	User Event 0	string add lock-free d0 54345C
73078	738ms 426us	msh_LockFree_v10 Thread 1	User Event 0	string get lock-free d0 544499
73079	738ms 427us	msh_LockFree_v10 Thread 1	User Event 0	string get lock-free d0 544499
73080	738ms 461us	msh_LockFree_v10 Thread 1	User Event 0	string add lock-free d0 54345C

Figure 9: *Latency-time, induced through the Dispatcher, in the lock-free implementation.*

Setup of the Environment

The target operating system used was QNX Neutrino 4.1. The following steps must be executed under QNX Neutrino 4.1.

Cilk The used version was `Cilk-5.4.6`. The source code is included in the DVD of this thesis. In order to get Cilk operational under Neutrino, the source code had to be extracted. At the root directory `Cilk-5.4.6`, `make` was executed. Two errors appeared, mainly because of missing modules which were not ported to Neutrino (e.g. pearl). At the example directory `Cilk-5.4.6\examples`, `make clean` was executed, so that all examples were deleted. If e.g. `make queens` is executed within this directory, the syntax of the command is shown for building the Cilk version of the queens problem. Analog syntax was used for building other Cilk programs.

The build process for an OpenMP enabled compiler was more complex.

OpenMP The compiler used in Neutrino is a modified version of the gcc compiler. The features for OpenMP are disabled in this modified version. So, the compiler had to be recompiled with OpenMP support enabled.

The following steps must be performed in order to get OpenMP runnable at QNX Neutrino 4.1.

1. the gcc `4.3.3` version was downloaded from QNX branch `svn:core-dev-tools` (now included in the DVD).
2. already ported libraries mpfr, ppl, gmp for Neutrino were downloaded (now included in the DVD)
3. the source was extracted to a destination path `/home/gcc-4.3.3`.
4. the compiled libraries for mpfr, ppl, gmp were extracted to their destination path `/usr/local/`.
5. the file `build-hooks` in the root directory was modified by adding the line `--enable-libgomp` directly after `--enable-libmudflap`.
6. the `libmudflap/Makefile.in` was modified by changing the line `QNXLDFLAGS = -Wl,-Bstatic -Wl,-lbacktraceS -Wl,-Bdynamic` to `QNXLDFLAGS = -lbacktrace`.
7. unnecessary directories in the path `/home/gcc-4.3.3/`, like sh, ppc were deleted.
8. the compiler was compiled using

   ```
   make 2>&1 | tee CC.log
   ```
9. the status of the compilation was stored at `CC.log`.

The following steps must be executed under a Linux operating system.

Intel TBB Intel TBB is a library which had to be ported to QNX Neutrino. A part of this library is hardware and OS dependent, and must be changed to use this library under QNX Neutrino.

The following steps were performed in order to get Intel TBB (`tbb21_009`) runnable at QNX Neutrino 4.1.

1. the source was retrieved from `http://www.threadingbuildingblocks.org/ver.php?fid=122` (now included in the DVD) and extracted into the destination folder `/home/TBB/`.

2. a patch was created (now included in the DVD (file:`new.patch`)) for enabling TBB under Neutrino.

3. the patch had to be applied by using `patch -p0 < new.patch` in the destination folder.

4. the library was compiled using `make OS=QNX` in the destination folder.

In order to use this library, the whole content of `/home/TBB/` was to copied to the operating system Windows where the development environment was stored. The include path `c:\TBB\include` was registered to the QNX development environment (Windows). The compiled libraries i.e. the directories within `c:\TBB\build` were also registered. The libraries within this build directory must be deployed to the targets library path. Those libraries were used by adding tbb, tbbmalloc to the compiler settings of the environment.

Bibliography

[AJR+03] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. *Lecture notes in computer science*, pages 193–208, 2003.

[AR06] S. Akhter and J. Roberts. *Multi-Core Programming: Increasing performance through software multi-threading*. Intel Press, 2006.

[Bab] Babara Chapman. How OpenMP is compiled. Website. Available online at `http://hpc.cs.tsinghua.edu.cn/research/cfp/iwomp2007/slides/tut-compilerrev-chapman_new.pdf` visited on July 12th 2009.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[Blu92] Robert D. Blumofe. Managing storage for multithreaded computations. Master's thesis, Department of Electrical Engineering and

Computer Science, Massachusetts Institute of Technology, September 1992. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-552.

[But04] G.C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications.* Springer-Verlag New York Inc, 2004.

[BW01] A. Burns and A.J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX.* Addison Wesley, 2001.

[Chr] Christopher Diggings. Three Reasons for moving to Multicore. Website. Available online at `http://www.ddj.com/hpc-high-performance-computing/216200386` visited on October 4th 2009.

[Gal95] B.O. Gallmeister. *POSIX. 4: programming for the real world.* Oreilly & Associates Inc, 1995.

[GN] I. Guide and R. Notes. Intel® C++ Compiler Professional Edition 11.0 for Linux.

[HH08] C. "Hughes and T." Hughes. "Professional Multicore Programming: Design and Implementation for C++ Developers". *"Wrox Programmer To Programmer"*, page 648, 2008.

[HP03] J.L. Hennessy and D.A. Patterson. *Computer architecture: a quantitative approach.* Morgan Kaufmann, 2003.

[Hyp] HyperTransport™Consortium). HyperTransport™IO Technology Overview. Website. Available online at `http://www.hypertransport.org/docs/wp/HT_Overview.pdf` visited on Juny 23th 2009.

[Kni09] Andreas Knirsch. Fast-Startup Concept for Embedded Systems. Master thesis, Hochschule Darmstadt, September 2009.

[Lan05] Geoff Langdale. Lock-free programming. Website, 2005. Available online at `http://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf`; visited on Juli 16th 2009.

[LD91] D.A. Lewine and D. Dougherty. *POSIX programmer&s guide*. O&Reilly, 1991.

[McK03] Paul E. McKenney. Using RCU in the Linux 2.5 kernel. *Linux Journal*, 1(114):18–26, October 2003.

[Mic04] M.M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[MS05] P.E. McKenney and D. Sarma. Towards hard realtime response from the Linux kernel on SMP hardware. In *linux. conf. au*, 2005.

[Rei07] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[Resa] QNX Community Resources. Anatomy of a system call. Website. Available online at `http://community.qnx.com/sf/wiki/do/`

viewPage/projects.core_os/wiki/KernelSystemCall visited on May 17th 2009.

[Resb] QNX Community Resources. The qnx neutrino microkernel. Website. Available online at http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/kernel.html visited on Juli 16th 2009.

[SH] D.C. Schmidt and T. Harrison. Double-checked locking. *Pattern languages of program design*, 3:363–375.

[Som02] P. Sommerlad. Performance Patterns. In *Proceedings of the 7 th European Conference on Pattern Languages of Programs*, 2002.

[SR89] J.A. Stankovic and K. Ramamritham. *Tutorial: hard real-time systems*. IEEE Computer Society Press Los Alamitos, CA, USA, 1989.

[Sun04] H. Sundell. *Efficient and practical non-blocking data structures*. Department of Computer Engineering, Chalmers University of Technology, 2004.

[Sup] QNX Developer Support. Developing multicore systems. Website. Available online at http://www.qnx.com/developers/docs/6.3.0SP3/multicore_en/user_guide/how_to.html; visited on July 14th 2009.

[WT05] J. Wietzke and M.T. Tran. *Automotive Embedded Systeme effizientes Framework-vom Design zur Implementierung*. Springer, 2005.