Facing the Challenges for Real-Time Software Development on Multi-Cores

Dr. Fridtjof Siebert
aicas GmbH
Haid-und-Neu-Str. 18
76131 Karlsruhe, Germany
siebert@aicas.com

Abstract

Multicore systems introduce new challenges for the developers of embedded systems. C, C++, and Java developers will profit from this class by understanding where caution is required in the use of certain programming language mechanisms. Examples include the memory model provided by the language implementation, effects of compiler optimizations, atomic operations, and use of the volatile modifier. Multicore often does not produce the expected performance gain due to synchronization problems; practical alternative multithreading programming approaches, such as lock-free algorithms, will be presented to help developers maximize the performance potential of multicore processors.

Categories and Subject Descriptors C.3 [*Computer Systems Organization*]: SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS—Real-time and embedded systems; D.3.3 [*Software*]: PROGRAMMING LANGUAGES—Language Constructs and Features; D.4.7 [*Software*]: OPERATING SYSTEMS—Organization and Design: Real-time systems and embedded systems

General Terms algorithms, languages, performance, reliability

Keywords multicore, parallel, concurrent, real-time, Java, memory

1. Introduction

The emergence of multicore computer architectures will have a profound effect on the software development process and the implementation of programming languages. With multicore systems, parallel systems become the norm even for low-end computers such as embedded controllers.

1.1 Typical problems

The typical problem on multicore systems arise from missing synchronization primitives. The basic synchronization primitives available are locks (mutexes, semaphores) and compiler hints such as the volatile flag.

1.1.1 Synchronization

As a very simple initial example (in C or Java), I take the code from figure 1 which declares a counter variable and a routine to increment it.

```
1: int counter;
2:
3: void increment()
4: {
5:   counter++;
6: }
```

Figure 1. Non-synchronized code to increment a counter variable

Assume that the routine *increment()* is used concurrently by different threads. Incrementing an *int* using the post-fix ++ operation is not atomic, so we have a race condition in this code, concurrently executed calls to *increment()* may result in some increments getting missed. However, it may well be that this code never showed any problems on a single core system.

Assume that *increment()* is called about 100 times every second after one of several possible events occurred. On a single-CPU system, the code may work fine all of the time: If all events are handled by threads of the same priority the code will not be preempted and the race condition here is merely theoretical.

On a two CPU system, the event handlers may execute concurrently. Assuming that *increment()* requires 100 cycles and we run at 500MHz, the likelihood of a conflict is about 10% each second. If *increment()* was

called 1000 times per second, the likelihood of failure within one second is 99.99%!

Fixing this problem is easy, add synchronization as in figure 2 (in Java).

```
1: int counter;
2:
3: synchronized void increment()
4: {
5:   counter++;
6: }
```

Figure 2. Synchronized code to increment a counter variable

The overhead required for the synchronization is relatively low, and all the problems described above are fixed. In Java, it has therefore become practice to add *synchronized* keywords generously whenever multiple threads may simultaneously access modifiable data.

1.2 Theoretical Limits

The performance to be gained on multicore systems by parallel execution has an important theoretical bound defined by Amdahl's law [2]. It is plain and simple, you gain only on the sections of the application that can be parallelized, sequential code does not profit from parallel CPUs. Unfortunately, if the number of CPUs increases, the remaining sequential code becomes more and more dominant, such that, e.g., having only 25% of sequential code will limit the performance gain on a multicore system to a factor of four, no matter how many CPUs are available.

However, this paper does not address Amdahl's law, but addresses issues of inter-thread communication that result in a situation that is much worse in practice: If not designed well and correctly, thread interactions result in failures or severe performance bottlenecks that are much worse than what Amdahl's law would predict.

1.3 Memory Model

The basis for software development on a multicore system (and even on a multi-threaded system running on a single core) is the memory model defined in the programming language. The memory model defines the semantics that parallel accesses to memory location have, and the diversions from straight sequential semantics that the compiler and the CPU implementation is allowed to make. The memory model is therefore fundamental for the development of applications with a well defined behaviour.

Java is very advanced in this respect as the Java community early on discovered the importance of the memory model and the flaws of the original specification [4], to come up with a robust memory model [3] the developer can rely on. The C++ programming languages is catching up by also clarifying its memory model [1], but the C++ community's approach is more relaxed requiring more care by the developer. The memory model of the C programming language is undefined for multi-threaded applications, which allows the largest number of compiler optimisations, but it makes this language unsuitable for the development of multi-threaded code. In particular with the advent of more and more multicore systems, the undefined semantics in C render this language a major risk for the development of safety-critical applications.

The main differences between these memory models will be explained below in this paper.

2. Correctness on Multicore Systems

Code that was developed for single CPUs and that was verified and tested extensively on single CPU systems may be incorrect on a multicore system. Even though a single CPU system uses multi-threading that gives the illusion to the software developer that several CPUs are present, the run-time semantics on multicore systems are significantly different and allow for interleaving of code execution that is impossible on a single core and that can cause fatal failure when moving single-core code to a multicore system.

```
 1: volatile int index;
 2: byte[] data;
 3:
 4: void highPriTask()
 5:   {
 6:     if (index >= 0)
 7:       {
 8:         data[index] = readData();
 9:       }
10:   }
11:
12: void lowPriTask()
13:   {
14:     if (enabled())
15:       {
16:         index = currentIndex();
17:       }
18:     else
19:       {
20:         index = -10000;
21:       }
22:   }
```

Figure 3. A high- and a low-priority task that interact correctly on a single-core, but that can cause a crash on a multicore since the high priority task may be preempted by the low priority task

Figure 3 shows a simple code sequence that runs correctly on a single CPU system, but that may fail on

a multicore system. This example uses two real-time threads that perform a high- and a low-priority task regularly. This code was developed to run on a real-time system with preemptive fixed-priority scheduling, i.e., the low priority task may always be preempted by the high priority task, while the high priority task will always complete without being preempted by the low priority task.

These two tasks communicate via a shared variable *index* that identifies a data slot to be read by the high priority task. A negative value for *index* indicates that no data should be read. The high priority task reads *index* twice, first to check whether it is non-negative, and, in case this test was successful. to write the read data into the slot referred to by index. This works perfectly on a single CPU system since the low priority task cannot modify *index* once the high priority task has started execution.

However, on a multicore system, these two tasks may actually run in parallel on different CPUs resulting in the execution interleaving shown in Figure 4. The low priority task may modify the value of *index* at any point during the execution of the high priority task resulting in using a negative array index.

```
   CPU 1 (highPriTask):          CPU 2 (lowPriTask):
1:                               if (enabled())
2: if (index >= 0)
3:                               index = -10000;
4: data[index] = readData();
```

Figure 4. Possible interleaving of example from Figure 3 that results using the illegal array index -10000.

The problem with this kind of code is that even very extensive testing might not be sufficient to discover these kinds of problems. An in-depth review of single-CPU code is required to ensure that it runs properly on multicore systems.

The situation becomes worse when the original code was already poorly designed and contains bugs. Imagine the two tasks from this example run at the same priority. The application contains a race conditions that may result in failures during run-time on a single core systems. However, these failures are very unlikely since they may only occur in case of a thread switch right after the first time *index* was read.

On a multicore system, such a thread switch is not required, the two tasks will just naturally be assigned to different CPUs by the OS such that many more possible interleavings of memory accesses will occur at run time. This means that race conditions that may exists in multi-threaded software, but that never manifested in run-time failure on a single core systems may become orders of magnitude more likely to result in application failure on a multicore.

3. Performance on Multicore Systems

When moving a multi-threaded software system from a single core to a multicore platform, one would hope to see an increase in performance due to the possibility to run several threads in parallel on different CPUs, compared to being limited to one thread at each point in time on a single CPU system.

Unfortunately, instead of a performance increase, the result of moving to multicore is a severe performance decrease in many cases. The reason for this lies in different thread communication patterns that may occur on parallel systems.

Figure 5 illustrates a piece of real-time code that performs well on a single-CPU system. A low priority task regularly checks a shared variable *index* that is occasionally updated by a high priority task. To avoid race conditions, all accesses to the shared variable are synchronized using a Java monitor.

```
 1: int index;
 2: byte[] data;
 3:
 4: void highPriTask()
 5:   {
 6:     while (index < 10000)
 7:       {
 8:         synchronized (data)
 9:           {
10:             data[index] = readData();
11:             index++;
12:           }
13:       }
14:   }
15:
16: void lowPriTask()
17:   {
18:     int lastIndex;
19:     while (true)
20:       {
21:         synchronized (data)
22:           {
23:             if (lastIndex != index)
24:               {
25:                 lastIndex = index;
26:                 updateIndex(index);
27:               }
28:           }
29:       }
30:   }
```

Figure 5. Two communicating threads that perform well on a single CPU, but that may cause a severe slowdown in the high priority thread on a multicore system

When running this code on a single CPU system, the high priority task may preempt the low priority task at any time. If the high priority task is unlucky, this preemption will occur while the low priority task is within its synchronized block. Then, the high priority task will have to wait for the low priority task to leave that section the first time the high priority task enters the same lock. Standard priority inheritance techniques ensure that this happens in bounded time. On any subsequent iterations of the loop in the high priority task, no synchronization is required: The low priority task cannot run and re-acquire the lock such that no further blocking of the high priority task will occur.

On a single core, there is therefore very little inter-thread communication in this application. The behaviour is shown in Figure 6.

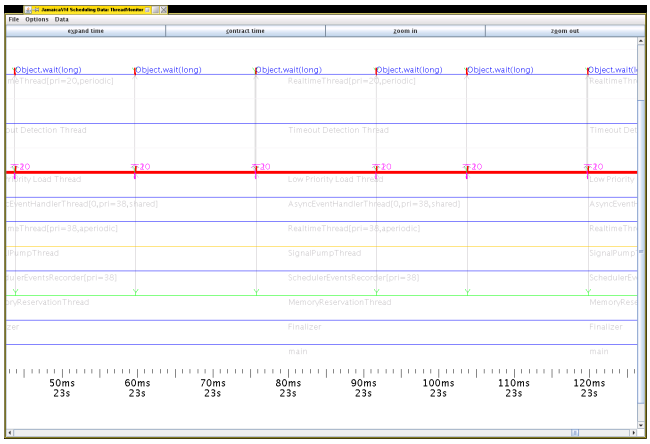

Figure 6. Locking in the example from Figure 5 between a low-priority and a high-priority thread causes little thread interaction on a single core.

When running on a multi core, however, the high priority task can see an extreme slowdown as shown in Figure 7. In this case, the low priority thread can run in parallel to the high priority thread, such that it can re-acquire the lock again and again, forcing the high priority thread to block repeatedly.

4. The Memory Models in Java, C++ and C

To understand well what kind of synchronization mechanism are required in the code, the developer of parallel software first needs to understand the memory model of the underlying programming language that is implemented by the compiler and run-time system in use. The memory model must allow for sufficient compiler and hardware optimisations, while it also must give the developer a good understanding of how the execution environment may execute the code.

Java was the first mainstream programming language for which a reliable memory model was de-

Figure 7. Locking in the example from Figure 5 between a low-priority and a high-priority thread causes extreme slowdown for the high priority thread on a multicore.

fined [3, 6, 5]. Since this language has built-in support for threads and synchronization mechanisms, a clear specification of the behaviour is required. The C++ community is currently working on their own memory model [1], based on the Java approach, but with important relaxations making execution of C++ software less predictable.

For the C language, the language specification does not define the memory model at all. Consequently, only single-threaded applications can be written safely in C. For multi-threaded applications, the developer must understand the particular C compiler in use well enough to know what code this compiler will generate and possibly include processor-specific memory fences manually into the code.

The memory model is a contract between the hardware, compiler and the developers. It describes the legal behaviours of multi-threaded code with respect to shared memory. This is the basis to reason about multi-threaded programs, to give security guarantees and to allow optimisations in the compiler.

4.1 The Java memory model

An important part of the Java memory model is the definition of so-called ordering operations. These ordering operations are entering or exiting synchronized blocks and accesses to volatile fields. The Java memory model permits certain reordering of memory accesses, but it is very strict on the reordering over ordering operations. Finally, final fields require specific handling in the Java memory model

The following subsection will describe the aspects of the memory model in more detail.

4.1.1 Reordering

The Java memory model allows compilers to reorder independent statements. It in particular allows optimisations that are

- trace preserving, i.e. result in the same memory access patterns, e.g.

```
if (r == 1)
  x = r;
else
  x = 1;
```

  can be replaced by

```
x = 1;
```

- removal of irrelevant reads
- memory access reordering, e.g.,

```
r1 = x;  r2 = y;  <=> r2 = y;  r1 = x;
x  = r1; y  = r2; <=> y  = r2; x  = r1;
r1 = x;  y  = r2; <=> y  = r2; r1 = x;
```

- removal of redundant read after write, e.g.,

```
x = r1;
r2 = x;
```

  can be replaced by

```
x = r1;
r2 = r1;
```

- removal of redundant reads, e.g.,

```
r1 = x;
r2 = x;
```

  can be replaced by

```
r1 = x;
r2 = r2;
```

- removal of redundant write before write, e.g.,

```
x = r1;
x = r2;
```

  can be replaced by

```
x = r2;
```

  Not allowed are the following modifications

- additions of irrelevant reads
- removal of redundant writes after read, e.g.,

```
r1 = x;
x = r1;
```

  can not be replaced by

```
r1 = x;
```

- moving memory accesses into a synchronized section, e.g.,

```
x1 = r1; r2 = y1;
synchronized (o)
  {
    ...
  }
x2 = r3; r2 = y2;
```

  can not be replaced by

```
synchronized (o)
  {
    x1 = r1; r2 = y1;
    ...
    x2 = r3; r2 = y2;
  }
```

4.1.2 volatile fields and synchronization

Entering a synchronized block or reading a volatile field ensures that all reads of shared memory are completed before the synchronization operation is performed. Symmetrically, leaving a synchronized block or writing a volatile field ensures that all writes have been performed before the synchronization operation is performed.

The memory model guarantees that certain memory accesses will not be moved over an ordering operation by the compiler or the underlying hardware. In particular, no memory reads from shared memory that occur before entering a synchronized block or reading a volatile field will be performed after this ordering operation. Symmetrically, all memory writes to shared memory that occur before exiting a synchronized block or writing a volatile field will become visible to other threads before executing the ordering operation.

4.1.3 final fields

A special treatment of final fields is required by the Java memory model. If an object's constructor initialises a final field before the object reference is handed over to other threads, it is guaranteed that this initialisation becomes visible to other threads before they can access this field, even in the absence of any synchronization mechanisms.

4.2 The C++ memory model

The memory model used for C++ is more generous when it comes to permitted optimisations by the compiler. This flexibility for the compiler comes at the cost of loss of predictability: Unlike Java, a C++ application with data races has an undefined behaviour. It therefore lies in the responsibility of the developer to avoid all race-conditions or to have an in-depth knowledge about the compiler.

5. Lock-free algorithms – the solution?

Lock-free algorithms avoid the use of high-level synchronization mechanisms such as locks and use more primitive operations such as compare-and-swap (CAS) instead. A CAS is an atomic operation with the semantics shown in Figure 8.

```
 1: CAS(VAR var,expected,new)
 2: {
 3:   word result;
 4:   atomic
 5:     {
 6:       result = var;
 7:       if (result == expected)
 8:         {
 9:           var = new;
10:         }
11:     }
12:   return result;
13: }
```

Figure 8. Semantics of *CAS* as used in this paper

The use of lock-free algorithms may help to overcome the performance problems that explicit synchronization introduces. Coming back to the simple example from figure 1 that increments a counter: This code can be made thread safe through usage of lock free operations such as compare-and-swap (CAS) as shown in figure 9.

```
 1: volatile int counter;
 2:
 3: void increment()
 4: {
 5:   int old, new, result;
 6:   do
 7:     {
 8:       old = counter;
 9:       new = old + 1;
10:       CAS(counter,old,new,result);
11:     }
12:   while (old != result);
13: }
```

Figure 9. Use of Compare-And-Swap to increment a counter variable

However, this code based on CAS is difficult to understand, even for this very simple case. More complex algorithms using such lock-free mechanisms quickly become extremely difficult to develop and practically impossible to maintain. It is therefore a good idea to restrict the use of lock-free algorithms to individually developed and thoroughly tested libraries. In the case of Java, the package *java.util.concurrent* provides such library code. Figure 10 illustrates how a lock free counter can be implemented using class *AtomicInteger* from this package.

```
1: AtomicInteger counter = new AtomicInteger();
2:
3: void increment()
4: {
5:   counter.incrementAndGet();
6: }
```

Figure 10. Use of lock-free library code to increment a counter variable

When developing real-time code, it is required that we can give reasonable worst-case execution time bounds for all time-critical code. However, the use of a CAS instruction typically requires a possibly infinite number of retries. E.g., for the CAS that is executed in the loop shown in figure 9 it is difficult to find an upper bound on the number of iterations of this loop. Therefore, this code cannot be used in an environment that requires strict timing bounds.

The following sub-sections explain under which conditions CAS-based lock-free code can be used even for time-critical tasks.

5.1 Compare-And-Swap for One-Way State Changes

An application may make use of state variables whose value can change only in one specific direction, i.e., if the current state is A, the state variable can only be changed to B. A phase change from A to B may be attempted by several CPUs in parallel using compare-and-swap. Only one CPU will succeed with this operation. However, the failed CPUs do not need to retry the CAS since the operation failed because some other CPU made exactly the state change that the failed CPUs attempted to make. In contrast, a CAS in the general case can fail and require one or several retries in case another CPU in parallel writes the same word. One-way state changes do not require a retry even when performed in parallel.

Care is needed to ensure that a CAS on a state change does not fail due to a cycle of states (the ABA problem): e.g., imagine CPU1 attempts to change the state from A to B and this unlucky CPU is so slow (it

may be pre-empted by the OS to perform some completely different task) that another CPU2 will perform the state change to *B*, will finish phases *B*, *C*, *D*, etc. and finally put the system back into phase *A*. If now CPU1 will execute its compare-and-swap, it will perform the state change to *B* even if the newly started phase *A* is not complete yet. To solve this, for any such cycles in state changes, it has to be ensured that at some point during the cycle, no CPU is in the middle of attempting a state change.

In general, a CAS for a one-way state change can be used in a real-time system straightforwardly, a worst-case execution time can be found since no retries are needed.

```
 1: volatile States state = INIT;
 2: for (i=0; i<N; i++)
 3:   {
 4:     new Thread()
 5:     {
 6:       boolean running = false;
 7:       public void run()
 8:       {
 9:         CAS(state,INIT,STARTING);
10:         running = true;
11:         [..]
12:       }
13:     }.start();
14:   }
```

Figure 11. Use of CAS for a one-way state change

An example of such a one-way state change is shown in Figure 11. Here, a shared volatile variable *state* is changed from the original state *INIT* to state *STARTING* by several concurrent threads. Only one of theses threads will actually change the variable, for all others, the CAS will fail since they will already see the new value. However, this failed CAS has no negative effect since the desired state change has been performed by a different thread.

5.2 Compare-And-Swap with Retry

Less useful for real-time systems are CAS-loops that retry the CAS operation in case it failed. This approach is sufficient for non-real-time systems that are optimised for average throughput, but not being able to limit the number of CAS-retries is not acceptable in a real-time system.

The number of CAS-loop iterations can, however, be limited to *n* if the number of CPUs is limited by *n* and the number of CPU cycles spent in between two successive executions of such a loop is at least *n-1* times larger than the time required for one CAS-loop iteration. The reason for this is that if a CAS failed on one CPU, a competing CAS on another CPU must have been successful. Hence, this other CPU will then perform work outside the CAS-loop. The same will happen for each following iteration with a failing CAS. After *n-1* failed CAS-loops, all other CPUs will be in the CAS-free code section and the *n*th CAS is guaranteed to succeed[1].

In consequence, a CAS with a retry-loop is only usable in a real-time system if the CAS-free code section after the CAS-loop can be made large enough. Ideally, the length of the CAS-free part should be configurable by a run-time constant to permit scaling for arbitrary numbers of CPUs.

```
 1: AtomicInteger counter = new AtomicInteger();
 2: static final int GRANULARITY = 64;
 3: [..]
 4: for (i=0; i<N; i++)
 5:   {
 6:     new Thread()
 7:     {
 8:       int local_counter;
 9:       public void incCounter()
10:       {
11:         local_counter++;
12:         if (local_counter >= GRANULARITY)
13:         {
14:           local_counter = 0;
15:           counter.addAndSet(GRANULARITY);
16:         }
17:       }
18:     }.start();
19:   }
```

Figure 12. Use of a thread-local counter to limit the number of retries required for a CAS

An example of bounding the number of retries for a CAS operation is shown in Figure 12: Here several threads provide an operation *incCounter* to increment a global counter. To ensure that every thread will execute this operation in bounded time, the global counter is updated lazily using a given *GRANULARITY*, which is set to 64 in this example. Assume this code runs with 8 threads on a 8 CPUs and one thread *T1* that has performed 64 increments now updates the global counter using the CAS operation within *addAndSet*. If this CAS fails and requires a retry, this means that another thread *T2* successfully changed the global counter. This other thread therefore will now have to perform 64 increments of the local counter before it will call *addAndSet* again. Consequently, if the next CAS performed by *T1* will fail as well, it cannot be due to *T2*, but due to another thread *T3* that was

[1] On real systems, determining the precise time spent in the loop is more complex, including interrupts or cache effects. Nevertheless, using a large enough code sequence between two successive CAS-loops can be used to reduce the probability of exceeding *n* iterations to an arbitrarily low value.

successful and will then perform 65 local increments. If the time required to perform 64 local increments is larger than the time required to perform 8 *addAndSet* operations, all other threads *T2* through *T8* will perform local counter increments after 7 failed CAS performed by *T1*. Consequently, the eighth attempt by *T1* will successfully perform the CAS since no other thread will change the global counter simultaneously.

Of course, the value contained by the global counter is not accurate using this technique, but it may be off by the counter values stored locally in the threads.

6. Conclusions

The upcoming multicore systems have a deep impact on the software development for real-time and embedded systems. It is not possible to port existing single-processor code to a multicore system without thorough analysis of the code. In particular real-time software may show failure or severe performance degradation as a result of the move to multicore systems.

A thorough understanding of the memory model provided by the programming language is required for the development of multi-threaded software that should benefit from multicores. Java currently provides the most reliable memory model, while the C++ community is currently working on a less strict model. C does not give any guarantees on the memory model and multi-threaded execution.

The use of lock-free algorithms may solve some important performance issues. However, the techniques are extremely complex and require skilled personnel and careful engineering to an extend that lock free algorithms will likely remain the realm of a few highly specialised developers.

An application architecture that avoids most of these problems while benefiting from multicore systems would try to be parallel at the large scale: Large independent code sections that may run in parallel will not suffer from problems related to synchronization or inter-thread communication. The goal has to be to design applications this way.

Acknowledgement

This work was partially funded by the European Commission's 7th framework program's JEOPARD project, #216682.

References

[1] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 68–78, New York, NY, USA, 2008. ACM.

[2] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[3] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. *SIGPLAN Not.*, 40(1):378–391, 2005.

[4] William Pugh. The java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455, 2000.

[5] Jaroslav Ševčík. The java memory model (tutorial). September 2009.

[6] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the java memory model. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 27–51, Berlin, Heidelberg, 2008. Springer-Verlag.