Chapter 1

Introduction

Multicore processors have become standard in modern hardware platforms that are used in desktop workstations, servers, clusters, mobile devices, and other embedded systems that need a lot of computing power but that do not have hard real-time requirements.

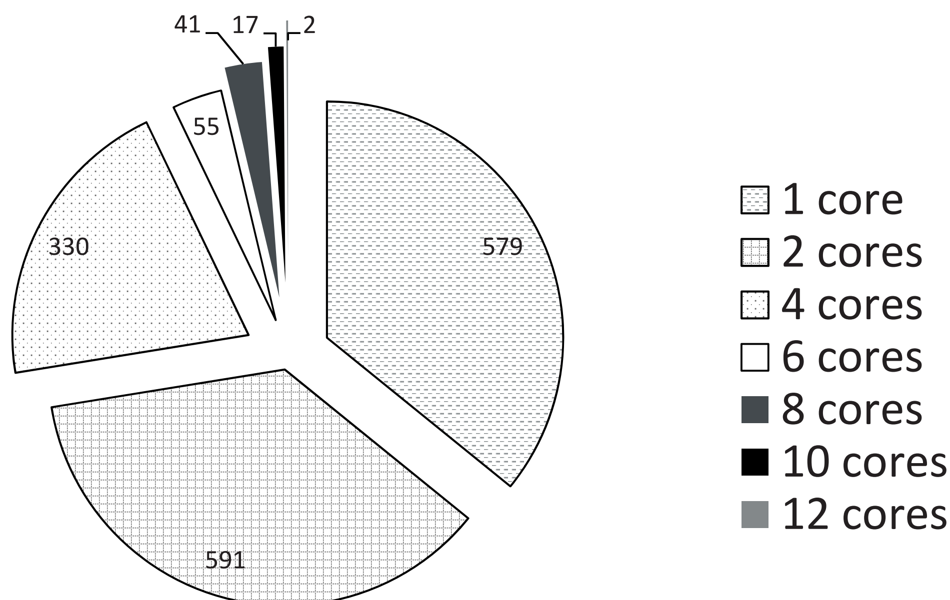


Figure 1.1: Number of Intel processor models separated by core count. The data for this chart was gathered on December 05, 2013 from `http://ark.intel.com/Search/Advanced`.

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

For example, if we look at Intel's Intel 64, IA-32, and Itanium microprocessors starting with the Pentium III and count singlecore and multicore processors separately as shown in Fig. 1.1, we see that multicores dominate the product portfolio of this major microprocessor manufacturer for the desktop, server, and cluster market segment. Since the figure also includes now outdated models from the pre-multicore era, the dominance of multicore products is even stronger than it appears at first.

Multicore processors are also about to become the standard in many embedded systems. In the mobile segment, the Samsung Exynos 5250 [97], Qualcomm Snapdragon, and Apple A6[1] are system-on-chips (SoC) that are commonly used in smartphones or tablets. These SoCs use multicores as well. In the field of industry automation, we performed research to find out whether it is possible to automatically parallelize legacy applications in order to make them ready for future generations of programmable logic controllers that are also going to use multicores [65].

To guarantee that an application makes efficient use of the multiple cores provided by modern processors, programmers are nowadays required to develop parallel code instead of sequential programs. Although concurrent programming is now an important skill, writing parallel code has shown to be difficult for mainly two reasons. First, it is not always obvious how an efficient parallelization of a certain problem would look like. In fact, complexity theoreticians have identified problems for which it is unknown whether they can be solved with efficient parallel algorithms [43]. Second, while it is at times hard to write efficient parallel programs, it is frequently just as easy to produce incorrect parallel code, because programmers think sequentially and bugs in concurrent programs are hard to identify and debug because of the non-determinism of parallel code [74].

To make concurrent programming easier for non-experts, we are therefore interested in software development techniques that involve high-level language constructs and programming paradigms. Building abstractions from low-level primitives and providing these abstractions as a language construct or within a runtime system allows programmers to focus on the problem to parallelize, while developers that are forced to write their programs using tedious low-level primitives in order to accomplish the same task are more likely to introduce concurrency bugs in their code. Although higher levels of abstraction reduce the risk of errors and increase programmer productivity, these abstractions come with a cost. If a compiler or runtime systems implements them naïvely, a program that uses high-level constructs will perform much worse than a hand-

[1] Unfortunately, there are no publicly available datasheets for the latter two SoCs.

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

optimized program that uses low-level primitives. We therefore need compiler and runtime optimizations that reduce the overhead of high-level paradigms in order to make them attractive to developers.

This thesis focuses on compiler and runtime techniques for an existing high-level concurrent programming construct for shared-memory parallel programs: the atomic block. To motivate our work, we will first give some reasons why atomic blocks have become worthy of studying by discussing race conditions as one root cause of concurrency bugs. We also quickly recapitulate how locks are used to avoid race conditions, but show how an erroneous application of this technique either does not prevent them or how it leads to deadlocks, which are another type of concurrency bug, and how a correct but inept use of locks leads to performance problems. Next, we present atomic blocks and their semantics, and show why this construct is helpful to avoid race conditions and deadlocks. We will also argue why atomic blocks cannot eliminate race conditions entirely if a programmer manually adds them to the code. After that, we introduce the two major techniques that are used to implement the semantics of atomic blocks, and compare their advantages and disadvantages to see why no single implementation technique is best. Following the introduction of atomic blocks, we propose our contributions to remedy the problems identified and to improve upon the state of the art. We describe a compiler analysis that automatically infers atomic blocks in parallel code, which helps developers to eliminate race conditions in their programs, and a compiler and a runtime technique that improve upon existing approaches to implement atomic blocks, leading to better runtime performance and less memory consumption. The end of this chapter provides the outline of this thesis.

1.1 Race Conditions

In a multi-threaded program, a race condition arises when concurrent accesses to shared memory are not properly synchronized [90], and at least one of those accesses is a write. Race conditions can lead to unexpected program behavior if programmers assume that the unsynchronized memory accesses will happen in a particular order. Since the order of the concurrent memory accesses depends on the timings of the threads performing them, program failures due to race conditions seem to appear randomly, which makes them hard to reproduce and to debug.

Let us examine the code in Fig. 1.2 to illustrate this concept. The code contains two functions, f and g that are concurrently executed by two different

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

```
int a = 3, b = 3;

void f() {
  a--;
  b++;
  assert((a + b) == 6);
}

void g() {
  b--;
  a++;
  assert((a + b) == 6);
}
```

Figure 1.2: Multi-threaded code with race conditions.

```
int a = 3, b = 3;

void f() {
  int tmp1 = a - 1;
  a = tmp1;
  int tmp2 = b + 1;
  b = tmp2;
  assert((a + b) == 6);
}

void g() {
  int tmp3 = b - 1;
  b = tmp3;
  int tmp4 = a + 1;
  a = tmp4;
  assert((a + b) == 6);
}
```

Figure 1.3: Transformed multi-threaded code with race conditions.

threads. They increment and decrement two shared variables a and b that are both initialized to 3, and at the end of each function, we want the invariant $a+b = 6$ to hold. This program contains several race conditions since due to the concurrent execution of the functions, the memory accesses can be interleaved in any order. To see this, we transform the original code into the code in Fig. 1.3 in order to make the intermediate steps of the increment/decrement operations visible. If we now assume that f and g are executed concurrently, there are interleavings where the code behaves correctly, but there is also at least one order of memory accesses that leads to values for a and b that violate the invariant that we stated before. For example, if the memory accesses happen in the order given in Fig. 1.4, the invariant is satisfied and the program runs correctly. However, if the memory accesses are performed in the order shown in Fig. 1.5, the invariant is violated.

To make the program behave correctly in all cases, we need to ensure that the memory accesses of one function are not interleaved with the accesses of another function. The accesses of every function form *critical sections*. At any time, at most one thread may execute within a critical section.[2] If we declare each of the bodies of f and g as a critical section, then the accesses to the shared variables are coordinated in a way that eliminates interleavings that

[2]We can relax this definition a little: we can allow two critical section to execute concurrently if they access disjunct parts of the shared data.

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

```
int a = 3, b = 3;

void f() {               void g() {
                           tmp3 = b - 1;       // a: 3, b: 3
                           b = tmp3;           // a: 3, b: 2
  tmp1 = a - 1;                                // a: 3, b: 2
  a = tmp1;                                    // a: 2, b: 2
                           tmp4 = a + 1;       // a: 2, b: 2
                           a = tmp4;           // a: 3, b: 2
  tmp2 = b + 1;                                // a: 3, b: 2
  b = tmp2;                                    // a: 3, b: 3
                           assert(...);        // a: 3, b: 3, inv. satisfied
  assert(...);           }                     // a: 3, b: 3, inv. satisfied
}
```

Figure 1.4: Memory access ordering that leads to correct behavior for the program of Fig. 1.3. The functions f and g execute concurrently and time progresses from top to bottom. The comments show the values of the variables after every statement.

```
int a = 3, b = 3;

void f() {               void g() {
                           tmp3 = b - 1;       // a: 3, b: 3
                           b = tmp3;           // a: 3, b: 2
  tmp1 = a - 1;                                // a: 3, b: 2
                           tmp4 = a + 1;       // a: 3, b: 2
                           a = tmp4;           // a: 4, b: 2
  a = tmp1;                                    // a: 2, b: 2
                           assert(...);        // a: 2, b: 2, inv. violated
  tmp2 = b + 1;          }                     // a: 2, b: 2
  b = tmp2;                                    // a: 2, b: 3
  assert(...);                                 // a: 2, b: 3, inv. violated
}
```

Figure 1.5: Memory access ordering that leads to incorrect behavior for the program of Fig. 1.3. The functions f and g execute concurrently and time progresses from top to bottom. The comments show the values of the variables after every statement.

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

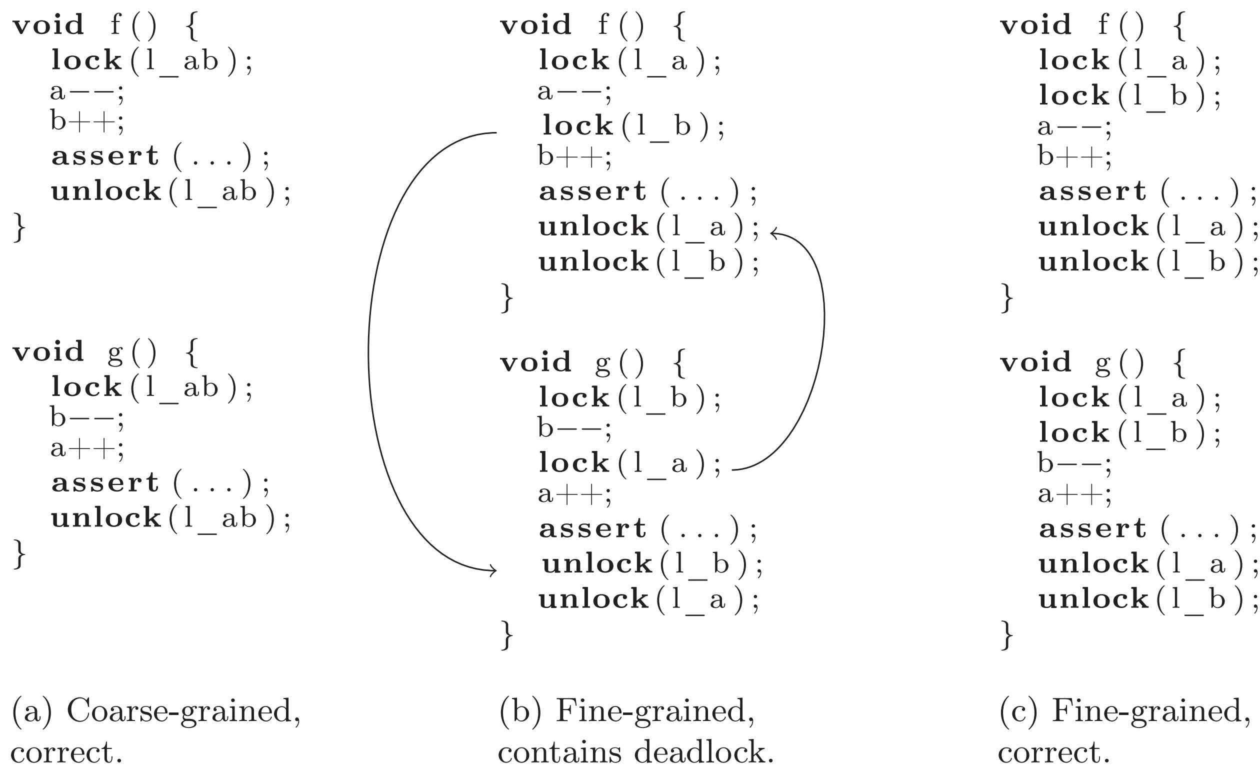
violate the invariant, so that race conditions are impossible. Unfortunately, it is hard for programmers to know where critical sections in the code are. It may not immediately be obvious which statements of a program access shared variables and even if developers, after careful reasoning, identified all critical sections, modifications to the code force them to re-think about what influences the changes have to the critical sections. The fact that the detection of all race conditions in a program is NP-hard [90] leads us to the conclusion that reasoning about critical sections is equally difficult indeed. As we will see in the next section, it is also difficult to properly implement critical sections, i.e., to use techniques in order to coordinate accesses to shared data such that all race conditions are avoided.

1.2 Locking

A common way to coordinate accesses to shared memory is to use locks [27]. In their most basic variant, locks have two states, *available* and *held* and provide the two operations *lock* and *unlock* to calling threads. A *lock* operation performed by a thread waits until the lock becomes *available* and then attempts to atomically switch the state of the lock to *held*. If the state-switch fails due to a successful intervening *lock* by another thread, the entire operation restarts until it eventually succeeds. After that, the lock is said to be *acquired*. The *unlock* operation *releases* a *held* lock by marking it as *available*. Locks allow threads to mutually exclude each other from entering a critical section at the same time and are therefore useful to synchronize memory accesses in order to prevent race conditions.

To coordinate memory accesses with locks, a programmer must know which data needs to be protected against concurrent accesses and must decide upon a locking solution. This includes the number of locks that should be used, which locks to acquire in order to access certain items of shared data and the order in which the locks are to be acquired and released. Of course, the choice of the locking solution has influence on the performance of the software. One extreme of this process would be to protect all shared data by a single global lock, while the other extreme would be to use one lock per shared variable. Locking solutions that tend to the first extreme and use few locks are coarse-grained, while solutions with a large number of locks that are closer to the second extreme are fine-grained. We will see that finding a good locking solution for a given program is difficult.

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.



(a) Coarse-grained, correct.

(b) Fine-grained, contains deadlock.

(c) Fine-grained, correct.

Figure 1.6: Examples of correct and incorrect locking approaches for the code of Fig. 1.2.

Let us illustrate this theoretical concept with the code of Fig. 1.2. The entire bodies of f and g each form a critical section of their own. Fig. 1.6 contains one incorrect and two correct implementation attempts for the critical sections with different granularities and placement strategies for the *lock* and *unlock* statements. The first decision we need to make is how many locks to use. With a coarse-grained approach (one lock for all shared variables) as in Fig. 1.6(a), the placement of the *lock/unlock* statements becomes straightforward: they must be at the start/end of the critical section to ensure that the data accesses are coordinated. It is relatively easy to verify that the invariant $a+b=6$ always holds when all threads are at the end of their critical section and that the coarse-grained implementation attempt is therefore correct. On the other hand, using fine-grained approaches (one lock per shared variable) as in Fig. 1.6(b-c) leads to more possibilities of placing the *lock/unlock* statements and as we shall see, increases the probability that a programmer writes incorrectly synchronized code.[3] The code in Fig. 1.6(b) acquires a lock just before the corresponding

[3] Admittedly, using one lock per variable in this example is not necessary, since acquiring a second lock is redundant: whenever l_a is acquired, l_b will be acquired as well. But the example nevertheless allows us to illustrate some important concepts.

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

variable is accessed and releases all locks at the end of the critical section. This code however, contains a bug known as *deadlock* [21]. In our setting, a deadlock is a situation where two or more threads mutually wait for each other to release a lock that is held by the other thread, such that all threads are blocked forever. For example, assume that both f and g start simultaneously. Function f immediately acquires l_a and g immediately acquires l_b. Now both functions proceed and f tries to acquire l_b, which is currently held by g. Therefore, the thread that executes f will block until l_b becomes *available*. We picture the situation that the *lock* operation of l_b in f blocks until the *unlock* operation of l_b in g is performed with the arrow on the left. However, g also blocks, waiting for l_a to become *available*, which we depict with the arrow on the right. Since f and g circularly wait for each other, both threads block forever. Thus, it is important that the locking strategy is designed in a way that the order of the locking operations does not lead to deadlocks. However, it is easy to introduce deadlocks in the code, since the order in which locks will be acquired is not always obvious. As deadlocks occur only sporadically due to the non-deterministic timing-behavior of threads, they are hard to debug.

Fig. 1.6(c) shows a correct fine-grained locking solution that avoids deadlocks. The key is to *lock* and *unlock* all locks at the start and end of the critical section. Additionally, the locks are acquired in a fixed order. Since both critical sections acquire l_a first and l_b afterwards, there is no circular dependence between the locking operations, and therefore f and g will never both at the same time infinitely wait for the other thread to release a lock.

Unfortunately, the technique we use to avoid deadlocks in Fig. 1.6 is not applicable in general since it might not be known up-front which shared variables the critical section accesses. Furthermore, deadlocks are not the only bugs that an incorrect locking strategy may cause. Other mistakes programmers could make are to forget acquire a lock, which leaves the critical section vulnerable to race conditions or to forget an *unlock* operation, which would cause that the lock could never be acquired again, causing threads to block forever. Furthermore, as software evolves, e.g., we change a critical section to access an additional shared variable, we need to rethink our locking solution, which may introduce new errors. Although coarse-grained locking tends to reduce the risk of introducing bugs, it is not a satisfying approach. Since we want to exploit the computing power of multicore processors, we need to be concerned about performance as well. If a program contains a large number of critical sections, coarse-grained locking limits the number of threads that can execute concurrently. In the worst case, there is no concurrency at all and most of the computing power of the parallel hardware is wasted. In general,

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

fine-grained locking leads to more performance since two critical sections that access disjunct areas of the shared data can execute concurrently if the critical sections use different locks.[4]

While we have focused on locking as an implementation technique for critical sections, alternative implementations for concurrent data structures such as non-blocking algorithms are not suited for average programmers either. We will discuss non-blocking algorithms in more detail in Chapter 2, but in short, non-blocking algorithms use sequences of atomic read-modify-write instructions to update shared data and thus several threads can operate on the same data structure without waiting for each other. Non-blocking algorithms are attractive because they never deadlock and they allow more concurrency than coarse-grained locking, but it is hard to design algorithms that are free of race conditions, even for simple data structures [28]. A preferable approach would be to only mark regions in the code that must be protected against race conditions, but leave their correct and efficient implementation to the compiler or runtime. This idea leads us to the discussion of atomic blocks.

1.3 A Part of the Solution: Atomic Blocks

Since locking and other synchronization techniques are difficult to use properly, atomic blocks have been invented to simplify synchronization matters in parallel programs. The semantics of an atomic block state that all accesses to shared memory within the atomic block appear to happen at once, such that intermediate states within the blocks are invisible to other threads. This causes the execution of atomic blocks to be serialized, i.e., if two atomic blocks A and B are executed concurrently by two threads, then A and B do not interleave: A cannot observe any intermediate changes that B makes to shared variables and vice versa, since one of the atomic blocks is guaranteed to finish before the other one starts. Since the implementation of these semantics is left to the compiler or runtime system, programmers do not have to worry about deadlocks. We can therefore easily synchronize the memory accesses in our example code of Fig. 1.2 if we put each of the bodies of f and g in a separate atomic block as shown in Fig. 1.7. Then both functions instantly update a and b and the invariant holds at all times.

[4] Since the critical sections of our example do not access disjunct data, fine-grained synchronization will not perform better in this case.

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

To the best of our knowledge, the concept of atomic blocks has its origins in a 1976 publication from Eswaran that introduces transactions in data base systems [34]. Lomet was the first to add atomic blocks to a programming language in 1977 [72]. However, the implementation of atomic blocks that Lomet describes does not include a method to prevent deadlocks. But since then, atomic blocks have recently gained more momentum since new implementation techniques have been invented that prevent deadlocks and therefore make atomic blocks more useful.

```
void f() {
  atomic {
    a--;
    b++;
    assert((a + b) == 6);
} }

void g() {
  atomic {
    b--;
    a++;
    assert((a + b) == 6);
} }
```

Figure 1.7: Example code of Fig. 1.2 synchronized with atomic blocks.

While atomic blocks do simplify concurrent programming and increase programmer productivity [92], we still cannot be satisfied with the current state of the art for two reasons. First, it is still too difficult to add atomic blocks to all places in the code where they are necessary to make a concurrent program behave correctly and second, we would like the implementation of atomic blocks to be more efficient than what current techniques achieve.

The problem with atomic blocks is that it still requires programmers to add them to the code. But to place them in the software, developers need to know where race conditions in their programs may happen, so knowing where atomic blocks are is the same as knowing where the critical sections are in the code. Forgetting to add atomic blocks in the right places can therefore still lead to concurrency bugs. As we have seen, correctly identifying all critical sections in a program is difficult. What we really need is tool support that helps programmers to add atomic blocks to the code. While there are approaches that automatically detect atomic blocks, they still require specifications from the programmer to work, such as marking which fields of an object need to be accessed atomically. Ideally however, such a tool needs as little intervention from the programmer as possible.

Besides that, we are also concerned with an efficient implementation of the atomic blocks that a tool or a programmer added to the code. Two current techniques to implement atomic blocks are software transactional memory (STM) [100] and lock inference [17]. Transactional memory is an optimistic parallelization approach that can be implemented in hardware or software. Regions of code that need to access data atomically run as transactions: all reads or writes are performed tentatively and updates to memory are only com-

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.