KISS: Keep it Simple and Sequential

Summary

Steffen Juilf Smolka

July 19, 2012

1 Motivation

In 1965, Intel co-founder Gordon E. Moore published a paper in which he observed an exponential growth in the number of transistors on chips, a trend he predicted to continue. This observation later became known as Moore's Law and has proven to be very accurate for almost 50 years now, with the number of transistors in CPUs doubling approximately every two years. As shown in figure 1, this growth in the number of transistors used to translate to a proportional growth in CPUs' clock speed, but this correlation no longer holds. Rather, there has been an increase in the number of cores per socket in recent years.

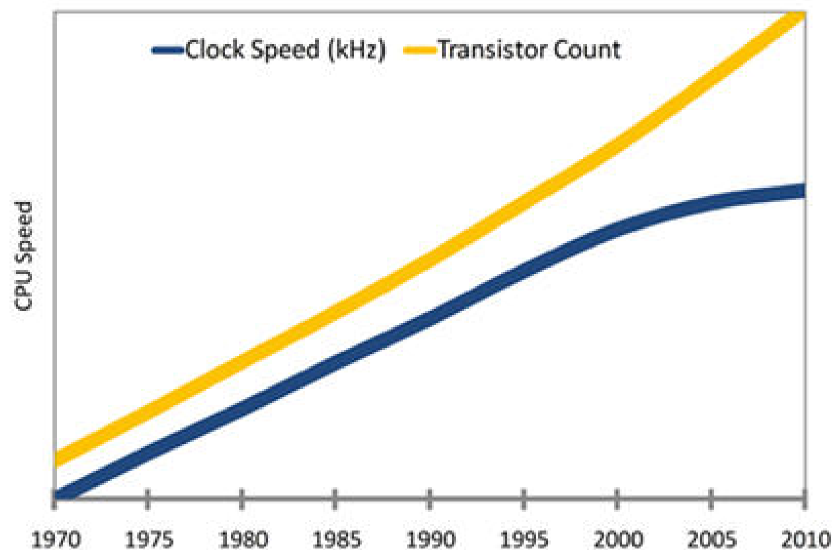

Figure 1: Moore's Law (y-axis uses exponential scale)

As Herb Sutter noted in his article "The Free Lunch Is Over" in 2005, this change has a deep impact on software development. While in the past, developers could just

wait for their programs to get faster as clock speeds increased, they now have to adapt their software in order to exploit the power of multiple cores. Thus arises the need for concurrent programs, especially in performance-critical systems. However, programming concurrent software has shown to be very error-prone in practise. For example, the access to shared variables has to be synchronized, and synchronization mechanisms can sometimes behave somewhat counterintuitive. To make things worse, the inherently non-deterministic nature of concurrent programs can make it very hard to find bugs in these programs. The problem is that every time a concurrent program is run, its threads may be scheduled in a different way, resulting in different thread interleavings which in turn may cause different program behaviours. In fact, there is a combinatorical explosion in the number of possible schedules as the number of threads increases.

This property of concurrent programs makes testing an ineffective method for checking program correctness, since testing fails to make any guarantees concerning the set of schedules explored, yielding low state space coverage in practice. Traditional model checkers for concurrent programs answer this deficiency by trying to explore all possible thread interleavings, thus ensuring high coverage. However, the high coverage comes at a cost: the complexity of model checking a concurrent programs is usually exponential in the number of threads [1], which severely restricts the scalability of this approach and makes it useless in many real-life scenarios.

2 The KISS Approach

KISS aims at providing a smart trade off between coverage and scalability. The idea is to *keep it simple and sequential*: Given a concurrent program P, it is transformed to a nondeterministic sequential program P_{seq} which simulates a subset of all possible executions of P. Basically, P_{seq} is a nondeterministic scheduler executing P. The transformation comes with several benefits. For one, transforming the input program to a sequential program means we no longer need to worry about the semantics of concurrent programs. In fact, the transformed program can be analyzed by an ordinary sequential model checker. From a theoretical point of view, KISS transforms the undecdiable problem of checking safety properties of a concurrent programs [2] to the decidable problem of checking safety properties of a sequential programs [3]. [1]

Of course, KISS cannot be used to proof the correctness of a program. As P_{seq} only simulates a subset of P's behaviors, the correctness of P_{seq} does not imply the correctness of P. However, the transformation preserves correctness, which means the implication does hold in the opposite direction. Forming the contrapositive of this implication, we find that a bug in P_{seq} implies a bug in P. Thus, we can utilize KISS for finding bugs in concurrent programs by first transforming P to P_{seq}, analyzing P_{seq} with a sequential checker, and finally translating error traces for P_{seq} back to error traces for P. The concept is illustrated in figure 2.

[1] Under the conventional assumption of an unbounded calling stack

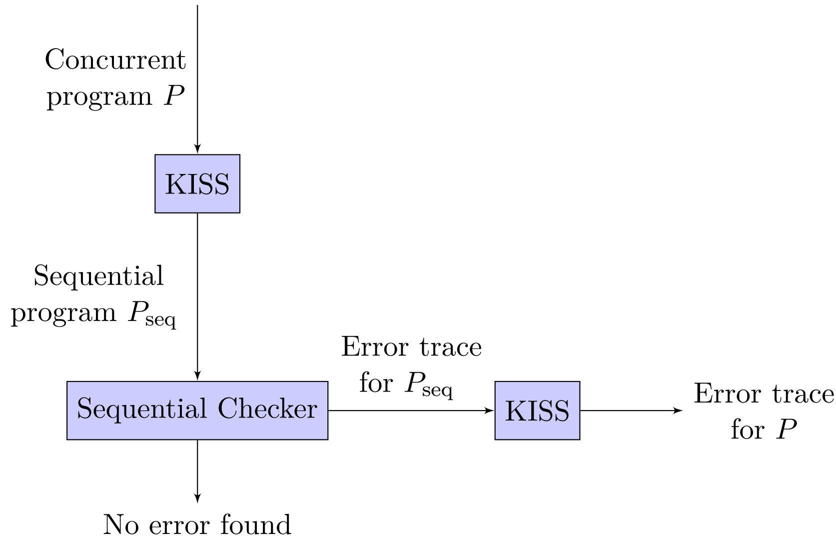

Figure 2: The KISS architecture

3 Transformation

3.1 Mechanics

The idea is that P_{seq} is basically P executed by a sequential scheduler. Whenever P_{seq} is run, the scheduler shall nondeterministically decide in which order the threads are scheduled and how they interleave. In a concurrent program, each thread holds its own stack and its own program counter, data we have to keep track of when simulating the execution of such a program. Storing this information in global variables gets us nowhere: Model checking sequential programs is exponential in the number of global variables, once again leading to exponential complexity in the number of threads. The remedy is to think of threads as functions. When a function is called, the machine pushes the information about the current state of execution, including the program counter, onto the stack. It also puts a stack frame for the function being called on top of the stack, which contains information such as the the functions parameters. The function is then executed. When it returns, the machine pops its stack frame and finds all the information needed to continue execution at the original point in the stack frame below. Note that from a high-level programming language's point of view, all this happens automatically and, in particular, without the use of any variables.

This mechanism can be exploited for scheduling threads using almost no variables (see figure 3). At any point during execution, a new thread can be scheduled simply by calling its starting function, or, from the machine's point of view, by putting it's stack frame on top of the stack. Likewise, a thread already running can be scheduled by executing one or more return statements, that is by popping all the stack frames above it. The thread will then continue running where it left of. To simplify explanations, let us consider a C-like

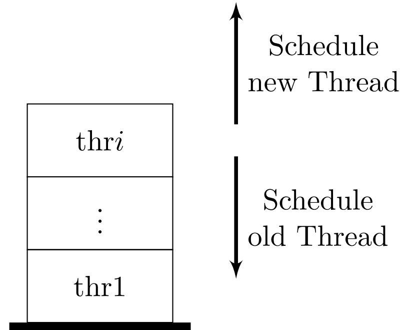

Figure 3: Scheduling Threads

parallel language with the statement `async(f)` for creating new threads, where `f` denotes the starting function of the thread. In order to keep track of the threads that have been created and are waiting to be scheduled, we introduce a global variable `ts`, which is a multiset of functions. To avoid exponential complexity in the number of threads, we fix the maximum size of the set `ts` to some number `MAX`. With these preperations, we can transform a given concurrent program P to a sequential program P_{seq} as follows, where `$` nondeterministically evaluates to true or false:

```
async(f);     ⇝     schedule_nondet_nr_of_threads();
                    if($) return;
                    if(|ts|<MAX) ts.add(f); else f();

stmt;         ⇝     schedule_nondet_nr_of_threads();
                    if($) return;
                    stmt;
```

We simulate creating a thread by adding its starting function to the set `ts`, thereby memorizing it for later scheduling, or by calling the thread's starting function immidiately if `ts` is already full. Additionally, before every statement, we add code that may schedule a nondeterministic number of new threads as well as code that may execute a nondeterministic return, thereby scheduling a thread already running.

There is one technical detail we have not mentioned so far. Of course, a thread's starting function may call another function, which in turn may call yet another function, and so on and so forth. As a result, the part of the stack belonging to one thread may consist of more than one stack frame. Let us consider the situation illustrated in figure 4, where the part of the stack associated with the currently executed thread i consists of two stack frames g() and h(), with h() currently running. In accordance with our idea of a scheduler and the technique described in figure 3, a nondeterministic return should cause thread i's stack frames to be popped and thread k to be continued. What actually happens when h() returns is that g() continues running. In order to solve this problem, we need a way to tell apart returns executed for scheduling reasons from regular returns. This can be accomplished by introducing a second global variable `raise`, which is initially

set to `false` and changed to `true` whenever a return is executed for scheduling reasons.

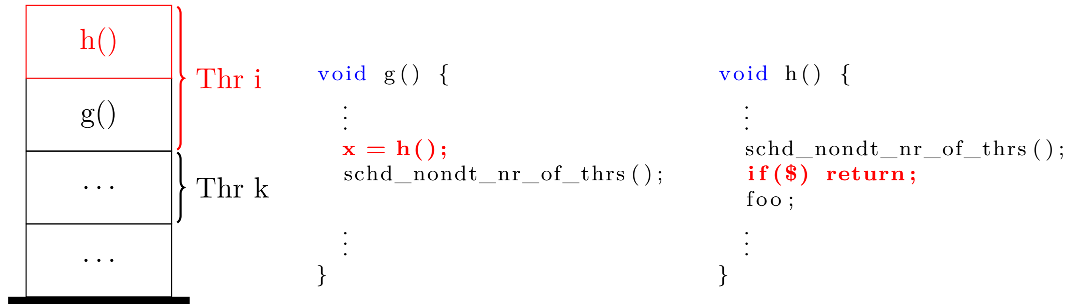

Figure 4: Nondeterministic Returns

We can now specify the whole transformation:

```
f();          ⇝    schedule_nondet_nr_of_threads();
                   if($) { raise=true; return; }
                   f(); if(raise) return;

async(f);     ⇝    schedule_nondet_nr_of_threads();
                   if($) { raise=true; return; }
                   if(|ts|<MAX) ts.add(f); else f();

stmt;         ⇝    schedule_nondet_nr_of_threads();
                   if($) { raise=true; return; }
                   stmt;
```

When executing a nondeterministic return, we set `raise=true` to communicate our scheduling decision. When getting back control after a synchronous function call, we propagate the return in case `raise` indicates it was executed for scheduling reasons, or continue regular execution in case it does not. The scheduling function is implemented as follows:

```
void schedule_nondet_nr_of_threads() {
    var f;
    while($) {
        if(ts.elements>0) {
            f = get();
            f();
            raise = false;
        }
    }
}
```

3.2 Observations

By exploiting the function concept, we are able to implement a nondeterministic scheduler with the use of only two global variables. Note that the stack frames of the currently executed thread are always located on top of the stack, giving the stack room to grow as necessary. However, we cannot simulate all possibles schedules: Once a thread's stack frames are popped, we cannot continue its execution since the necessary information is lost. Thus, we can simulate at most two interleavings between any two threads. For example, starting from thread 1, thread 2 might be scheduled and be executed for a while. Thread 2 might then execute a nondeterministic return, giving back control to thread 1. From that point on, continuing thread 2 is impossible, since all its stack frames have been popped. The maximum size `MAX` of the function multiset `ts` influences what schedules can be simulated as well. When setting `MAX=0`, for example, the transformation has the effect of replacing asynchronous with synchronous function calls.

4 Evaluation

Model checking a sequential program with boolean variables is of complexity $O(|C| \cdot 2^{g+l})$ [1], where $|C|$ denotes the size of the control flow graph, g the number of global variable, and l the maximum number of local variables in scope at any time. The described transformation enlarges the size of the control flow graph by a small constant factor and adds a constant number of global variables to the program. Using KISS on a concurrent program is therefore of about the same complexity as model checking a sequential program of the same size [1]. In particular, there is no correlation between the complexity and the number of threads in the concurrent program. Furthermore, the complexity can be dynamically controlled by altering the maximum size `MAX` of the set ts, allowing for even better scalability. KISS is fairly easy to implement and can basically be build on top of any existing sequential checker. As opposed to other comparable tools, KISS will never report false errors.

KISS has one serious flaw: It may miss errors. In fact, it can only simulate up to two interleavings between any two threads, and may therefore disregard many possible program behaviors. Nonetheless, the KISS author's experiments in collaboration with the Windows driver quality team suggest that KISS can be useful in practice. They used KISS to search for race conditions in 15 different drivers with a time limit of 20 minutes and a memory limit of 800 MB, and were able to detect 30 races. After investigating a subset of them, three of the race conditions were confirmed to be bugs [1].

References

[1] Qadeer, Shaz ; Wu, Dinghao: KISS: keep it simple and sequential. In: *PLDI*, 2004, S. 14–24

[2] Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. In: *ACM Trans. Program. Lang. Syst.* 22 (2000), Nr. 2, S. 416–430

[3] Reps, Thomas W. ; Horwitz, Susan ; Sagiv, Shmuel: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: *POPL*, 1995, S. 49–61