KAdvice: Infering Synchronization Patterns From an Existing Codebase

Alexander Schmidt          Andreas Polze
Hasso Plattner Institute  
Operatings Systems and Middleware Group 
Prof.-Dr.-Helmert-Str. 2-3 
14482 Potsdam, Germany 
{alexander.schmidt, andreas.polze}@hpi.uni-potsdam.de

ABSTRACT

Operating system kernels are complex software systems. The kernels of today’s mainstream OSs, such as Linux or Windows, are composed from a number of modules, which contain code and data. Even when providing synchronous interfaces (APIs) to the programmer, large portions of the OS kernel operate in an asynchronous manner. Synchronizing access to kernel data structures therefore is a central problem in OS kernels running on todays multcore and multiprocessor hardware. With the need to utilize future multi- and manycore processors, managing the synchronization problem becomes central to all multithreaded control-parallel applications. Since only little software is written from scratch, understanding the intended use of locking constructs and their relation to shared data structures will become critical to all programmers.

Built upon our experiences with developing code inside the Windows kernel, we have developed the KAdvice approach, which helps to analyze locking structures in an existing codebase. KAdvice applies static analysis to call graphs and code dependencies to recommend appropriate locking patterns when accessing certain data structures. KAdvice has itself proven very useful in context of students’ programming projects based upon the Windows Research Kernel (WRK). However, our approach is more general and applicable not only to OS kernels but to control-parallel software in general.

Categories and Subject Descriptors

D.2.2 [Design Tools and Techniques]: Computer-aided software engineering

General Terms

Algorithms, Languages, Theory

Keywords

KAdvice, data-flow analysis, lock patterns

1. INTRODUCTION

The class of recommendation systems for software engineering (RSSE) help programmers or developers to navigate through several dimensions in the information space in their software projects [16]. This is necessary as developers are confronted with an ever increasing complexity of frequently changing code bases. The problem with frequent changes in a code base is that, most of the times, documentation is not updated at the same pace. There is a variety of RSSE systems, which are capable of assisting programmers with questions like: what methods in other components are dependent to my method and thus require an update, how are classes interrelated, or what code locations might have caused the bug I am trying to fix.

With chip multi-processors (CMP) or multi-core architectures becoming mainstream, developers are faced with another challenge: writing parallel applications. This has been a well-established research topic on server systems and parallel computers, but with multi-core processors being omnipresent in desktop computer systems, and even mobile phones, average programmers are now faced with the complexity of designing and implementing parallel applications. It is still an open research question of how to address that issue best at the programming level [3,8,13].

However, the biggest challenge with parallel applications, besides identifying feasible parallel code paths, is to protect shared state from corruption, by avoiding data races. In general, several locking patterns can be identified to address that issue [10]: sequential programming, code locking, data locking, parallel fastpath, and data ownership. These approaches vary in the degree of complexity, the potential speedup and the relative synchronization overhead.

A sequential program, i.e., a non thread-safe program – while not lending itself to parallel execution at all – is also the simplest approach to prevent data races on program-wide global (shared) state. To increase speedup through parallelization, code locking is used to lock regions of code, for instance through monitors or by enclosing non-thread-safe library functions in an acquire-release-bracket. To further increase the degree of parallelization, data locking can be used to protect individual objects from simultaneous modification, which allows parallel execution of one and the same code region when operating on different data objects. Finally, data ownership means to uniquely assign shared data (partitions) to individual processors, i.e., a certain shared data partition is only accessed by one dedicated processor. While keeping the degree of parallelization high, this approach does not incur any synchronization overhead. A parallel fastpath pattern is an approach, where code paths are differentiated by their frequency of occurrence. The synchronization strategy is optimized to incur a lower overhead in the more probable cases while incurring a high synchronization overhead if a less probable code path is taken. Instances of this pattern are for example the usage of reader/writer locks and hierarchical locks [7].

One could easily argue that the more elaborate approaches, which allow for a higher degree of parallelism, are yet the more complex to implement, understand, and maintain,
as they increase the chance of incurring unwanted side effects like deadlocks or data corruption. Unfortunately, documentation about the locking rationale might be insufficient, outdated, or even non-existent. In all those cases, it is up to the developer to re-discover the rationale from the existing code base, which is tedious, time-consuming, and error-prone. We thus argue to extend the dimensions for RSSEs in such a way that an RSSE may assist a developer in identifying suitable locks to protect shared data. Thus, the contributions of this paper are as follows:

We present KAdvice, our tool for statically analyzing locking patterns in a given software codebase. This analysis employs the data-flow analysis body of techniques, for which we developed a sound set of data-flow equations, which we apply in an optimized, inter-procedural, and context-sensitive algorithm.

The remainder of the paper is organized as follows. We first present a study of existing research approaches suitable for the analysis of locking patterns. Second, we present our model to analyze the code base. Third, we present our algorithm before we finally conclude the paper.

2. RELATED WORK

To assist developers in finding important locking patterns, or more specifically, what particular locks are necessary to acquire before accessing shared data, it is crucial to determine the set of locks that are held whenever an access to some specific object occurs. To the best of the author’s knowledge, the KAdvice approach is new in the context of recommendation systems. However, KAdvice shares that key pre-requisite with well-known research approaches focusing on the automatic verification of properties of parallel programs. Such properties are for example the freedom of deadlocks or the absence of race conditions.

Two general approaches can be distinguished: dynamic race detection tools and static race detection tools, which will be covered separately. Both classes share with KAdvice the basic necessity to detect (1) when an access occurs, (2) what lock is currently held, and (3) what locks had been held on previous accesses to correlate locks and accesses. Race detection tools forget that information, as soon as they found a violation. KAdvice, however, keeps this information to build an information data base for developers.

Although dynamic tools, at first sight, may seem inappropriate for RSSEs, they provide however valuable information that may not be gathered by static alternatives.

2.1 Dynamic Detection Tools

Dynamic concurrency verification tools try to detect access anomalies during runtime of the system under investigation. The typical approach of dynamic tools is the lockset approach [17]. That is, each shared field of a data structure is associated with a set of locks that guard the respective data structure. On initialization, the set contains all locks of the system. Whenever a thread accesses a shared object, its current lockset is intersected with the current lockset of the object and stored as the new lockset of the object. Thus, if the lockset becomes empty, an access anomaly has been found. The original approach by Savage et al. has been improved and modified over time [5][24].

An alternative approach is based on Lamport’s Happened-Before relation [9]. The basic idea here is to determine, if accesses to a shared object or a member of a shared object happens in a partial order. As approaches based on the Happened-Before relation usually require more time and space, hybrid approaches have been developed [3][11][22].

All these dynamic approaches usually impose a significant overhead, both with respect to execution time and memory consumption, to the system under investigation. And, although dynamic approaches only track feasible paths, they cannot guarantee that the system is free of races, as long as their code coverage is less than 100%, which is very likely in complex software systems.

2.2 Static Detection Tools

Static race detection tools try to detect data races during the compilation phase, just as KAdvice does. They have the benefit that they cover the whole source code and run off-line, i.e., they do not impose any runtime overhead.

Among the first concurrency analysis tools was that of Young and Taylor [23]. In their approach, besides static analysis, they leveraged symbolic execution to detect any violations to the consistency model. However, the state space explosion makes this approach unfeasible for complex software projects. The same “flaw” holds for Warlock [19], one of the first tools using solely static analysis. However, it requires thorough annotations throughout the code to retrieve any useful result, which renders it impractical on large software system like operating systems.

Recent approaches, like RacerX [6], or Locksmith [15], require less annotations. However, they do not consider recursive locks in their model. Also, RacerX, in some cases, is too conservative with respect to analyzing recursive function calls. That might be suitable for race detection, but not for identifying the synchronization model. Von Praun and Gross [20], finally, present also a static analysis framework consisting of optimized super control-flow graphs as in KAdvice. However, they focus only on object-oriented languages and the Java language in particular.

3. SYSTEM MODEL

A particular locking pattern manifests itself in such a way that whenever a data object is accessed, the respective lock(s) are held accordingly to the locking pattern. That means, for a developer, it is sufficient to know what locks need to be acquired prior to a field access instead of knowing the precise locking pattern. For example, if a program is designed to comply with the sequential program locking pattern, there are no locks at all. Also, if a system has been designed with the data locking pattern in mind, there must be a particular lock that protects a particular field, which can be either a global lock or a per instance lock.

These observations lead to the conclusions that it is sufficient for a developer to know the set of locks held when accessing a particular data structure field. The previous section has shown that, in principle, there are two approaches on how to compute the set of locks: static and dynamic. Dynamic approaches have the disadvantage that they require a functioning runtime system or at least a trace of a run of the system for some period of time. Especially with identifying locking patterns, it is unclear how long such a trace must be recorded in order to gather sufficient information. Thus, KAdvice will implement a static approach that bases on abstract interpretation of the program [2]. Static approaches are also handy for the developer as they merely need the source code and can easily be run in the background.

Before KAdvice performs the lock analysis, we need to build up a model of the given code base. That model comprises the complete call graph of the system, context-sensitive control-flow graphs, and an appropriate lock state model. All of these are covered in the subsequent sections.

3.1 Call Graph

KAdvice abstractly interpretes the given code base in order to compute the set of locks and their respective states.
As we will point out in the next section, we must perform an inter-procedural analysis. To minimize the number of analysis steps, it is crucial to understand the sequential structure of the code base. KAdvice therefore creates a call graph, which is a directed graph where each node represents a function and each edge represents a function call. That means, a call graph is a directed graph \( G = (F, E) \), where \( F \) is the set of functions and \( E = F \times F \) the set of function calls. In that call graph, KAdvice determines a set of nodes that have no caller, which we denote as root functions. Our analysis will start from those functions, as they have no incoming dependencies to care about.

### 3.2 Context-Sensitive Control-Flow Graphs

KAdvice splits up each function into basic blocks. A basic block is a consecutive stream of instructions, where the beginning of a basic block is the only part of the block that may be the target of another jump instruction and the stream is terminated by a branch instruction. For technical reasons, call instructions are not considered as jump/branch instructions.

All basic blocks of a function comprise the context-sensitive control-flow graph (cCFG). A context-sensitive control-flow graph is a directed graph, where each node represents a basic block of the code. Furthermore we define two types of edges: control-flow edges and call edges. A control-flow edge represents control flow through the function itself, i.e., from one basic block to another. Additionally, a cCFG contains two special nodes that represent the function entry and exit point.

The reason, why a cCFG is necessary, becomes clearer, when considering the C code given in Listing 1. There are two different code paths in the system: the code path that calls InitData just once on line 23 and the code path that also calls InitData on line 19. The important fact to consider here is that depending on the actual path, a different constraint holds for the field accessed in InitData. For the call in line 19, we must say that field bar::data is protected by the bar::lock lock, while no lock actually protects the local variable data2. That means, it is crucial for our analysis to distinguish call sites and the respective context.

To distinguish between different callers of a function, a simple approach would be to have a sub-flow graph for each call site. Instead, we create a flow graph only once for each function. This cloning based approach is necessary to minimize memory consumption during the analysis phase. As each callee may itself call functions, recursively creating a flow graph for each call site would result in an exponential number of flow graphs. In order to support context-sensitivity, we introduce call edges to the control-flow graph, i.e., we annotate each call edge from a call site to the respective called function with the current context that must be considered when analyzing the function.

We define the context at a call site as the list of formal parameters that are passed to the callee. Furthermore, we need to extend the context by the list of locks that we are aware of at the call site. During analysis of the callee, the list of locks may be updated. We thus need to propagate that list back to the caller. To minimize execution time, the context is stored in a per-function summary cache. A function is only analyzed, if the context at the call site is new. This approach greatly reduces the amount of analyses.

### 3.3 Lock States

In order to detect available locks in the system, KAdvice needs to know the fundamental synchronization primitives used in the system. To minimize efforts for a developer to use our tool, fundamental synchronization primitives are expressed as a list of functions that have acquire and release semantics, respectively. A function has acquire semantics, if it acquires a lock and the lock is held after returning from the function. Similarly, a function has release semantics, if it releases a lock held on function entry.

Acquire and release functions transform the state of a lock, which can be in one of the following states that comprises the set \( V \) of lock states:

1. **Acquired**: The lock has been acquired by a function call.
2. **Released**: The lock has been released by a function return.
3. **Recursively Acquired**: A lock has been acquired and subsequently released within the same function.
4. **Recursively Released**: A lock has been released and subsequently acquired within the same function.
5. **Unknown**: The state of the lock is unknown.
6. **Conflict**: A lock has been acquired and released by different functions.

A lock is in the acquired state, if it has been acquired by a function and the lock is still held after the function return. A lock is in the released state, if it has been released by a function return. A lock is in the recursively acquired state, if it has been acquired and released within the same function. A lock is in the recursively released state, if it has been released and acquired within the same function.

### 4. THE LOCK DETECTION PROCESS

The lock detection process itself is a multi-phase operation. In a first phase, KAdvice constructs the call graph of the system. For each root function in the call graph, KAdvice constructs a cCFG and performs an inter-procedural

---

**Listing 1:** This listing demonstrates the necessity for context-sensitivity.

```
struct bar{
  Lock lock;
  int data;
};

void InitData(int *data)
{  
  *data = 0;
}

void Sample(BOOLEAN DoBranch)
{
  struct bar s1;
  int data2;

  if (DoBranch)
  {
    AcquireLock(s1.lock);
    InitData(&s1.data);
    ReleaseLock(s1.lock);
  }
  InitData(&data2);
}
```
data-flow analysis. During analysis, the cCFG is adaptively expanded as necessary. When the analysis finishes, KStruct determines the locations of data structure accesses in the given code and tracks for each access the set of locks visible at that point and their respective state into a database. At the moment of writing, the database must be queried by an extra tool, the KAdvisor. In a later prototype, we are working on IDE integration for at least Microsoft Visual Studio.

### 4.1 Lock Semi-Lattice

The foundation for our analysis is the lock semi-lattice presented here. It allows KAdvice to determine the state of any lock at any point in the system. We define our lattice as \( L = (V, \wedge_L) \), where \( V \) is the set of lock states.

We define the meet operator \( \wedge_L \), \( V \times V \rightarrow V \), as follows. For all \( x \in V \):

- \( \text{UNKNOWN} \wedge_L x = x \)
- \( \text{CONFLICT} \wedge_L x = \text{CONFLICT} \)

For any two constants \( c_1 \) and \( c_2 \), \( c_1, c_2 \in V \) the \( \wedge_L \) operator is defined as follows:

\[
c_1 \wedge_L c_2 = \begin{cases} 
\text{CONFLICT} & \text{if } c_1 \neq c_2 \\
\text{Unknown} & \text{otherwise}
\end{cases}
\]

It can easily be shown that our meet operator \( \wedge_L \) is idempotent, commutative, and associative. Based on the meet operator, we define a partial order \( \leq_L \) on the lock semi-lattice. For all \( x \) and \( y \) in \( V \):

\[
x \leq_L y \iff x \wedge_L y = x.
\]

Because \( \wedge_L \) is idempotent, commutative, and associative, it can be easily shown that our lock order \( \leq_L \) is reflexive, anti-symmetric, and transitive. We thus have defined a sound semi-lattice, and poset on lock values, respectively.

When analyzing a system, we do not want to run the computation again for each and every single lock. Suppose \( \Lambda \) is the set of all locks in the system. We compose a product lattice \( L = (V^{|\Lambda|}, \wedge_L) \), where each value represents a map from each lock variable in the system to one of the values described in the previous section. A map \( m \in V^{|\Lambda|} \) is a tuple \((v_1, v_2, ..., v_{|\Lambda|})\), \( v_i \in V \). The \( i \)-th entry denotes the value of lock \( i \) in the system. To index a certain value in the map, we introduce \( m(\lambda) \) to denote the value of lock \( \lambda \). The meet operator \( \wedge_L \) on the product lattice is just the meet operator \( \wedge_L \) applied component-wise on a map \( m \).

### 4.2 Transfer Functions

In a cCFG, we assign each statement in the system an entry and an exit state. The entry state of the first instruction of a basic block is also the entry state of the basic block itself. The same holds for the last statement and the exit state of a basic block. The entry state of a basic block is computed by applying the meet operator \( \wedge_L \) on the exit state of all predecessors of that block in the cCFG.

The exit state of an instruction is computed by applying a transfer function on the entry state. Computing the exit state of a basic block is thus a composition of transfer functions. For the sake of analyzing lock states, we propose the following transfer functions \( f_s \) by correlating \( m \) and \( m' \), where \( m' = f_s(m) \).

1. If the statement \( s \) is an acquire operation on lock \( \lambda \), then \( \forall \gamma \in \Lambda, \gamma \neq \lambda : m'(\lambda) = m(\gamma) \), i.e., all the other locks not affected by the statement remain their state.

\[
m'(\lambda) = \begin{cases} 
m(\lambda) + 1 & \text{if } m(\lambda) \text{ is already a number} \\
1 & \text{if } m(\lambda) \text{ is UNKNOWN} \\
\text{CONFlict} & \text{otherwise}
\end{cases}
\]

2. If the statement \( s \) is a release operation on lock \( \lambda \), then \( \forall \gamma \in \Lambda, \gamma \neq \lambda : m'(\gamma) = m(\gamma) \).

\[
m'(\lambda) = \begin{cases} 
m(\lambda) - 1 & \text{if } \lambda \text{ has been acquired} \\
1 & \text{at least once.} \\
\text{CONFlict} & \text{otherwise}
\end{cases}
\]

3. If the statement is a call instruction, the transfer function is the composition of all transfer functions for the called function, the callee. Let \( \Delta \) be a subset of \( \Lambda \) that contains all the locks \( \lambda \) affected by the callee. For all locks \( \lambda \) in \( \Delta \), the transfer function is

\[
m'(\lambda) = \text{OUT}[F_c](\Lambda) \]

where \( \text{OUT}[F_c] \) is the output map of callee \( F \) in context \( c \). \( \text{OUT}[F_c] \) is computed by analyzing the callee itself. We leverage the partial transfer function approach to optimize analysis time [22]. For all other locks not affected by \( F \) the lock value remains the same, i.e., \( \forall \gamma \in \Lambda - \Delta, m'(\gamma) = m(\gamma) \).

4. For all other statements the transfer function is just the identity function \( I(m) = m \), i.e., \( \forall \Lambda \in \Lambda : m'(\lambda) = m(\lambda) \).

Beginning with the set of previously defined root functions, KAdvice iterates over the instructions of the basic blocks in the cCFGs of the respective functions until a fixed point is reached, i.e., neither an entry nor an exit state changes during an iteration. The careful construction of the lock product-lattice as well as the design of the transfer functions guarantees that such a fixed point will be reached, as in each step of the analysis a data-flow value can only remain the same or gets less with respect to the partial order we defined on the lattice. As the lattice is of finite height, the algorithm must terminate.

Once KAdvice has finished computing the fixed point of lock states, it determines for each and every field access the state of locks at this point. This information can in a first step be used to guide a developer on what locks must be acquired for some data object. In some cases, it may happen that multiple locks are indicated to be held at similar field accesses, some of which are just held by coincidence. We are currently trying to refine those to indicate only the most relevant for the developer.

### 4.3 Limitations

The proposed approach is sound and works reasonably well on our sample software system, the Windows Research Kernel (WRK) [14]. The WRK comprises the majority of the source code of the Windows Server 2003 operating system kernel and is freely available to academia. Its 400+ C files and 200+ header files sum up to a total of roughly 1 million lines of code or 22 MB of source code [15] which we consider a sufficiently complex software system in order to test our prototype implementation. However, we encountered the following problems during our evaluation:

At the current state, our approach requires a statically computable call graph to resolve function calls. If there are indirect function calls, or more specifically, if there are indirect function calls that depend on runtime values, we cannot reason about the functions called. Fortunately, indirect function calls are less often used in C programs than for instance in object-oriented programs. Also, the identified parts of the WRK that rely on indirect function calls are less critical with respect to locking patterns. In most cases, these functions are tracing and performance related callbacks that do not contribute to the functionality of the kernel itself.
Another problem results from missing code paths or imported functionality through late binding that is, for instance, implemented in a DLL. In such cases, all that KAdvise can statically derive is that some function with a certain function signature is called, but not precisely which function. Also, if code paths are implemented in static libraries that are not available as source code, KAdvise cannot reason about the innards of those libraries. In the latter case, we argue that if functionality is imported from opaque libraries, the inherent implementation details should not affect design decisions of the code relies on that library. Thus, whatever locking pattern a library implements, it is independent of the locking pattern of its user.

In the former case, we propose a hybrid approach to compute a more precise call graph. This approach requires a runtime system in order to execute the system under development (if its has reached an executable state). Then, one could use dynamic instrumentation, for instance [12], to intercept function calls. The static call graph could then be extended with additional runtime information to create a more specific call graph. Although not all possible code paths may have been traced, the result yields feasible code paths and a more precise view on the system. If source code is available for the intercepted function calls, those code can easily be integrated into our analysis phase.

5. CONCLUSIONS

In this paper we presented KAdvise, a tool for automatically detecting locking patterns in an existing code base in order to assist developers in figuring out what locks protect which data structure fields. This information becomes more and more important as, in the foreseeable future, applications will only get faster if they can benefit from multiple processors. KAdvise builds up an abstract model of the system and performs a data-flow analysis to gather locking information, which is then related to data structure access information.

We have implemented our algorithm in a prototype implementation that has been used on the Windows Research Kernel to analyze existing locking patterns in the Windows kernel. This is difficult as the WRK implements many locking patterns at various granularity levels. We are currently evaluating the results of our analysis in a study where students who are novice to the Windows kernel must modify parts of the kernel with and without KAdvise. First results are promising, which is why we consider KAdvise a valuable addition to the RSSE domain.

6. REFERENCES