Realizing Iterative-Relaxed Scheduler in Kernel Space

Master-Arbeit
Sreeram Sadasivam
2662284

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik

Fachgebiet Dependable Embedded Systems and Software
Prof. Neeraj Suri Ph.D

Realizing Iterative-Relaxed Scheduler in Kernel Space
Master-Arbeit
2662284

Eingereicht von Sreeram Sadasivam
Tag der Einreichung: 16. Marz 2018

Gutachter: Prof. Neeraj Suri Ph.D
Betreuer: Patrick Metzler

Technische Universität Darmstadt
Fachbereich Informatik

Fachgebiet Dependable Embedded Systems and Software
Prof. Neeraj Suri Ph.D

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Master-Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die schriftliche Fassung stimmt mit der elektronischen Fassung überein.

Darmstadt, den 16. Marz 2018 Sreeram Sadasivam

1 Background

1.1 Software Verification

Software programs are becoming increasingly complex. With the rise in complexity and technological advancements, components within a software have become susceptible to various erroneous conditions. Software verification have been perceived as a solution for the problems arising in the software development cycle. Software verification is primarily verifying if the specifications are met by the software[12].

There are two fundamental approaches used in software verification - dynamic and static software verification[12]. Dynamic software verification is performed in conjunction with the execution of the software. In this approach, the behavior of the execution program is checked- commonly known as Test phase. Verification is succeeding phase also known as Review phase. In dynamic verification, the verification adheres to the concept of test and experimentation. The verification process handles the test and behavior of the program under different execution conditions. Static software verification is the complete opposite of the previous approach. The verification process is handled by checking the source code of the program before its execution. Static code analysis is one such technique which uses a similar approach.

The verification of software can also be classified in perspective of automation - manual verification and automated verification. In manual verification, a reviewer manually verifies the software. Whereas in the latter approach, a script or a framework performs verification.

Software verification is a very broad area of research. This thesis work is focused on automated software verification for multithreaded programming.

1.2 Multithreaded Programming

Computing power has grown over the years. Advancements are made in the domain of computer architecture by moving the computing power from single-core to multi-core architecture. With such advancement, there were needs to adapt the programming designs from a serialized execution to more parallelizable execution. Various parallel programming models were perceived to accommodate the perceived progression. Multithreaded programming model was one of the designs considered for the performance boost in computing[5].

Threads are small tasks executed by a scheduler of an operating system, where the resources such as the processor, TLB (Translation Lookaside Buffer), cache, etc., are shared between them. Threads share the same address space and resources. Multithreading addresses the concept of using multiple threads for having concurrent execution of a program on a single or multi-core architectures. Inter-thread communication is achieved by shared memory. Mapping the threads to the processor core is done by the operating system scheduler. Multithreading is only supported in operating systems which has multitasking feature.

Advantages of using multithreading include:

- Fast Execution
- Better system utilization
- Simplified sharing and communication

- Improved responsiveness - Threads can overlap I/O and computation.
- Parallelization

Disadvantages:

- Race conditions
- Deadlocks with improper use of locks/synchronization
- Cache misses when sharing memory

1.3 Concurrency Bugs

Concurrency bugs are one of the major concerns in the domain of multithreaded environment. These bugs are very hard to find and reproduce. Most of these bugs are propagated from the mistakes made by the programmer[14]. Some of these concurrency bugs include:

- Data Race
- Order violation
- Deadlock
- Livelock

Non-deterministic behavior of threads is one of the reasons for having the among mentioned bugs. Data race and order violation are classified as race condition bugs. Whereas, deadlock and livelock are classified as lack of progress bugs.

1.3.1 Race Condition

Race condition is one of the most class of common concurrency problems. The problem arises, when there are concurrent reads and writes of a shared memory location. As stated above, the problem occurs with non-deterministic execution of threads.

Consider the following example, you have three threads and they share two variables x and y [5]. The value of x is initially 0.

Thread 1	Thread 2	Thread 3
(1) x = 1	(2) x = 2	(3) y = x

Table 1.1: Race condition example

If the statements (1), (2) and (3) were executed as a sequential program. The value of y would be 2. When the same program is split to three threads as shown in the above Table 1.1, the output of y becomes unpredictable. The possible values of y = {0,1,2}. The non-deterministic execution of the threads makes the output of y non-deterministic. Table 1.2 depicts possible executions for the above multithreaded execution.

The above showcased problem is classified as race condition bug. Ordered execution of reads and writes can fix the problem.

1.3.2 Lack Of Progress

Lack of progress is another bug class observed in multithreaded programs. Some of the bugs under this class include deadlocks and livelocks.

Execution Order	Value of y
(3),(1),(2)	0
(3),(2),(1)	0
(2),(1),(3)	1
(1),(3),(2)	1
(1),(2),(3)	2
(2),(3),(1)	2

Table 1.2: Possible executions

Deadlock

Deadlock is a state in which each thread in thread pool is waiting for some other thread to take action. In terms of multithreaded programming environment, deadlocks occur when one thread waits on a resource locked by another thread, which in turn is waiting for another resource locked by another thread. If a thread is unable to change its state indefinitely because the resource requested by it are being held by another thread, then the entire system is said to be in deadlock[6].

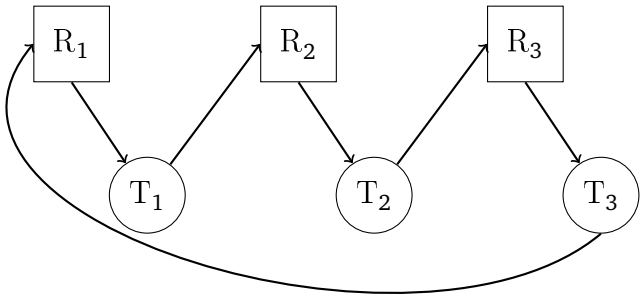


Figure 1.1: Dead Lock Example

In the example depicted in Fig 1.1, we have three threads T_1, T_2, T_3 and three resource instances R_1, R_2, R_3. The figure depicts hold and wait by each threads. Thread T_1 holds resource R_1 and waits for the acquisition of resource R_2 from thread T_2. T_2 cannot relinquish resource R_2, unless it acquires resource R_3 for its progress. But, resource R_3 is acquired by T_3 and is waiting for R_1 from T_1. Thus, making a circular wait of resources. This example clearly explains the dependency of resources for the respective thread progress.

Deadlock can occur if all the following conditions are met simultaneously.

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

These conditions are known as Coffman conditions[8].

Deadlock conditions can be avoided by having scheduling of threads in a way to avoid the resource contention issue.

Livelock

Livelock is similar to deadlock, except the state of threads change constantly but, with none progressing. Livelock is special case of resource starvation of threads/processes. Some deadlock detection algorithms are susceptible to livelock conditions when, more than one process/thread tries to take action[14][6]. The above mentioned situation can be avoided by having one priority process/thread taking up the action.

1.4 Model Checking

From section 1.3, it is very clear that there needs to be verification for multithreaded programs. The verification solutions range from detecting causality violations to correctness of execution[10]. Model checking is an example of such a technique. It is used for automatically verifying correctness properties of finite-state concurrent systems[7][2]. This technique has a number of advantages over traditional approaches that are based on simulation, testing and deductive reasoning. When solving a problem algorithmically, both the model of the system and the specification are formulated in a precise mathematical language. Finally, the problem is formulated as a task in logic, namely to verify whether a given structure adheres to a given logical formula. The technique has been successfully used in practice to verify complex sequential designs and communication protocols[7]. Model checker tries to verify all possible states of a system in a brute force manner[1]. Thus, making state explosion as one of the major challenges, which is discussed in detail in section 1.4.1. Model checking tools usually verify partial specification for liveness and safety properties[10]. Model checking algorithms generate set of states from the instructions of a program, which are later analyzed. There is a need to store these states for asserting the number of visits made them are atmost once. There are two methods commonly used to represent states:

- Explicit-state model checking
- Symbolic model checking

Advantages of using model checking:

- Generic verification approach used across various domains of software engineering.
- Supports partial verification, more suited for assessment of essential requirements for a software.
- Not vulnerable to the likelihood that an error is exposed.
- Provides diagnostic information thus, making it suitable for debugging purposes.
- Based on graph theory, data structures and logic thus, making it 'sound and mathematical underpinning'.
- Easy to understand and deploy.

Disadvantages of using model checking:

- State explosion problem.
- Appropriate for control-intensive applications rather than data-intensive applications.
- Verifies system model and not the actual system.
- Decidability issues when considering abstract data types or infinite state systems.

1.4.1 State explosion problem

The state space of a program is exponential in nature when it comes to number of variables, inputs, width of the data types, etc,. Presence of function calls and dynamic memory allocation makes it infinite[10]. Concurrency makes the situation worse by having interleaving of threads during execution. Interleaving generates exponential number of ways to execute a set of statements/instructions. Thus, having an explosion in state space. There are various techniques used to avoid the state explosion problem.

1.4.2 Explicit-state Model Checking

Explicit-state model checking methods recursively generate successors of initial states by constructing a state transition graph. Graphs are constructed using depth-first, breadth-first or heuristic algorithms. Erroneous states are determined 'on the fly' thus, reducing the state space. A property violation on the newly generated states are regarded as erroneous states. Hash tables are used for indexing the explored states. If there is insufficient, memory lossy compression algorithms are used to accommodate the storage of hash tables[10]. Explicit-state techniques are more suited for error detection and handling concurrency.

1.4.3 Symbolic Model Checking

Symbolic model checking methods manipulate a set of states rather than single states. Sets of states are represented by formulae in propositional logic. It can handle much larger designs with hundreds of state variables. Symbolic model checking uses different model checking algorithms: fix-point model checking(mainly for CTL), bounded model checking(mainly for LTL), invariant checking, etc,. Two main symbolic techniques used - Binary Decision Diagrams(BDD) and Propositional Satisfiability Checkers(SAT solvers). BDDs are traditionally used to represent boolean functions. A BDD is obtained from a Boolean decision tree by maximally sharing nodes and eliminating redundant nodes. However, BDDs grow very large. The issues in using finite automata for infinite sets are analogous. Symbolic representations such as propositional logic formulas are more memory efficient, at the cost of computation time. Symbolic techniques are suitable for proving correctness and handling state-space explosion due to program variables and data types.

1.4.4 Partial Order Reduction

Partial Order Reduction(POR) is a technique used for reducing the size of state space to be searched by a model checking algorithm[17]. This technique exploits the independence of concurrently executed events. Two events are independent of each other when executing them either order results in the same global state[7]. A common model for representing concurrent software is to have it depicted as interleaving model. In interleaving model, we have a single linear execution of the program arranged in an interleaved sequence. Concurrently executed events appear to be ordered arbitrarily to each other. Considering all interleaving sequences would lead to extremely large state space. Constructing full state graph would make the fitting into the memory difficult. Therefore, a reduced state graph construction is used in this technique.

POR exploits the commutativity of concurrently executed transitions, which would result in the same state. Fig 1.2 depicts the commutativity behavior. S, S_1, S_2 and R are various states of a given program and α_1, α_2 represents various transitions. Consider two paths P_1 and P_2. $P_1 = S \rightarrow S_1 \rightarrow R$ and $P_2 = S \rightarrow S_2 \rightarrow R$. P_1 and P_2 reaches the same final state R. Thus, showing us that commutativity of transitions α_1, α_2 on the given example.

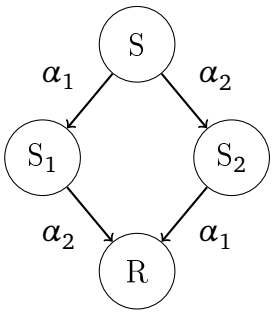


Figure 1.2: Commutativity Example

$B_r \subset B_f$

B_f represents the behaviors of full state graph and B_r represents set of behaviors of reduced graph. Partial order reduction derives its motivation from the early versions of algorithms used for partial order modeling of program execution. POR is described as model checking using representatives[16]. Verification is performed using representatives from equivalence classes of behaviors.

The transitions of a system play a major role in the POR. POR is based on the dependency relation that exists between the transitions of a systems. A transition $\alpha \in T$ is enabled in a state s, if there is a state s' such that $\alpha(s,s')$ holds. Otherwise, α is disabled in s. The set of transitions enabled in s is $enabled(s)$. A transition α is deterministic, if for every state s there is at most one state s' such that $\alpha(s,s')$.

A path π from a state s_0 is a finite or infinite sequence.

$\pi = s_0 \rightarrow s_1 \rightarrow ...$

$\alpha_0(s_0,s_1)$, $\alpha_1(s_1,s_2)$ are transitions on the states in path π such that for every i, $\alpha_i(s_i,s_{i+1})$ holds. If π is finite, then the length of π is the number of transitions in π and will be denoted by $| \pi |$. Purpose of POR is to reduce the number of states, while preserving the correctness of the program. A reduced state graph is generated using depth-first or breadth-first search methods. Model checking algorithm is applied to the resultant graph, which has fewer states and edges.

An independence relation $I \subseteq T \times T$ is a symmetric, anti-reflexive relation such that for $s \in S$ and $(\alpha, \beta) \in I$:

- Enabledness If $\alpha, \beta \in enabled(s)$ then $\alpha \in enabled(\beta(s))$.
- Commutativity $\alpha, \beta \in enabled(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$.

The dependency relation D is the complement of I, namely $D = (T \times T) \setminus I$. The enabledness condition states that a pair of independent transitions do not disable one another. However, that it is possible for one to enable another. Stuttering refers to a sequence of identically labeled states along a path. In fig 1.2, we have two paths P_1 and P_2 which are stuttering equivalent. Thus, the reduced graph would have fewer number of states and retains the correctness property of the model.

Two main POR techniques which are commonly considered: persistent/stubborn sets and sleep sets. Persistent set technique computes a provably-sufficient subset of the set of enabled transitions

in each visited state such that unselected enabled transitions are guaranteed not to interfere with the execution of those being selected. The selected set is called a persistent set. Whereas, the most advanced algorithms are based on stubborn sets. These algorithms exploit information about "which communication objects in a process gets committed to in a given set of operations in future"[11]. Such an information is generally obtained from static code analysis. The sleep set technique exploits information on dependencies exclusively among transitions enabled in the current state along with information recorded about the past of the search. Both the techniques can be used simultaneously and are complementary. Unfortunately, existing persistent/stubborn set techniques suffer from a severe fundamental limitation in the context of concurrent software systems. The non-determinism in the execution of the concurrent programs makes the computation of precision difficult. Sleep sets could be used but, it cannot avoid state explosion. To overcome the above limitations, we have Dynamic POR, which is discussed further in the next section.

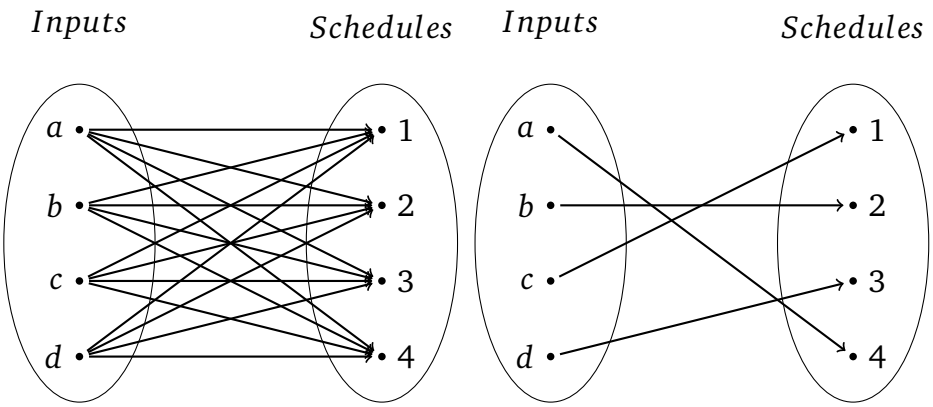
1.4.5 Dynamic POR

Dynamic POR is a technique, which dynamically tracks interactions between processes and then exploits this information to identify back tracking points where alternative paths in the state space need to be explored[11]. The algorithm works on depth first search in the reduced state space of the system. Dynamic POR helps to calculate dependencies dynamically during the exploration of the state space. It is able to adapt the exploration of the program's state graph to the precision of having another read or write operation accesses on the same memory location in the same execution path. Dynamic POR algorthm by Flanagan and Godefroid [11], explores single transitions and performs recursive calls subsequently. A persistent set is calculated at each state in the state graph of a system.

1.5 Deterministic Multi-Threading

In section 1.4, we primarily dealt with various ways to verify a program and suggested various techniques, which could be used in a multi-threaded environment. In this section, we discuss about a different approach to deal with the verification of multithreaded programs. In sections 1.2 and 1.3, we discussed about the non-determinism offered by multithreaded programming designs and the bugs associated with them. One way to detect and avoid bugs is to have a constrained scheduling of threads thus, adhering to deterministic execution. Deterministic multi-threading is an approach used to bring in determinism in the execution of multi-threaded programs.

Fig 1.3a depicts a mapping of inputs to possible scheduling pattern adopted by the multi-threaded program. By bringing in deterministic mapping as shown in fig 1.3b, we have direct mapping between inputs and schedules. Having such mapping provides us the opportunity to determine erroneous executions. Such a mapping facilitates to determine concurrency bugs in the program execution. There are many frameworks - CoreDet[3], Parrot[9], Kendo[15], DThreads[13], Grace[4], which adheres to this principle. Some of these frameworks are discussed further in the next chapter.



(a) Traditional Mapping (b) Deterministic Mapping

Figure 1.3: Input to Schedule mappings in multithreaded execution

Bibliography

[1] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. Principles of model checking. MIT press, 2008.

[2] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. Systems and software verification: model-checking techniques and tools. Springer Science & Business Media, 2013.

[3] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In ACM SIGARCH Computer Architecture News, volume 38, pages 53–64. ACM, 2010.

[4] Emery D Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for c/c++. In ACM sigplan notices, volume 44, pages 81–96. ACM, 2009.

[5] Richard H Carver and Kuo-Chung Tai. Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs. John Wiley & Sons, 2005.

[6] Sagar Chaki, Edmund Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. Concurrent software verification with states, events, and deadlocks. Formal Aspects of Computing, 17(4):461–483, 2005.

[7] Edmund M Clarke, Orna Grumberg, and Doron Peled. Model checking. MIT press, 1999.

[8] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. ACM Computing Surveys (CSUR), 3(2):67–78, 1971.

[9] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A Gibson, and Randal E Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 388–405. ACM, 2013.

[10] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 27(7):1165–1178, 2008.

[11] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In ACM Sigplan Notices, volume 40, pages 110–121. ACM, 2005.

[12] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. Fundamentals of software engineering. Prentice Hall PTR, 2002.

[13] Tongping Liu, Charlie Curtsinger, and Emery D Berger. Dthreads: efficient deterministic multithreading. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pages 327–336. ACM, 2011.

[14] Carmen Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and Elisa Gonzalez Boix. A study of concurrency bugs and advanced development support for actor-based programs. arXiv preprint arXiv:1706.07372, 2017.

[15] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. ACM Sigplan Notices, 44(3):97–108, 2009.

[16] Doron Peled. All from one, one for all: on model checking using representatives. In International Conference on Computer Aided Verification, pages 409–423. Springer, 1993.

[17] Doron Peled. Ten years of partial order reduction. In Computer Aided Verification, pages 17–28. Springer, 1998.