Addressing the Performance Bottlenecks of Device Driver Synthesis

Alexander Legg

Supervised by
Leonid Ryzhyk, Nina Narodytska, and Gernot Heiser

May 31, 2016

Contents

Abstract

Device drivers are notorious for their contribution to the failure of software systems. A driver forms the interface between two complicated systems: an operating system and a hardware device. As a result, driver development is complicated and highly prone to human error. Driver reliability has been the focus of much research over the past two decades. One result of this research is device driver synthesis.

Device driver synthesis aims to remove human error from the development process by removing humans. A synthesis tool can take a formal specification of requirements and produce a system. Thus, a driver can be synthesised by providing the tool with a specification of the OS and the hardware that it must interface with. The resulting driver is guaranteed to be correct with respect to the specifications.

The drawback of this approach is that synthesis is a computationally intensive task belonging to 2-EXPTIME. In practice synthesis tools are capable of producing drivers though there remain cases where synthesis is impossible. In this thesis I will present an algorithm that provides results on many of these previously unattainable cases.

Acknowledgements

Acknowledge some people

1 Introduction

This is the introduction.

1.1 Device Drivers

Device drivers are the software that allows the operating system to interface with hardware. The role of the driver is to manipulate the inputs of the device so that it remains in a error-free state and correctly handles the requests of the operating system. By way of example consider an ethernet driver. The driver accepts requests to send and receive data packets from the OS and acts on those requests by reading from and writing to buffers on the device. It must ensure that those buffers are maintained in a usable state by correctly updating a register containing the location of the head of the buffer queue.

According to a study performed in 2011 [5], drivers account for approximately 57% of the lines of code in the Linux kernel and subsequently is the largest source of bugs. The study also analysed the staging directory of the kernel, which contains all in-progress drivers, and found it to contain the highest fault rate (faults per line of code) out of any directory in the kernel. The results of the study give evidence to the widely held belief that correct drivers are hard to produce.

Consequences of buggy drivers

This thesis focuses on automatic construction of correct drivers as a solution to the driver problem. Alternate approaches, of which there are many, will be discussed in Chapter 2.

1.2 Synthesis

Explain what synthesis is

- Briefly describe the current state of the art
- Explain state space explosion

1.3 Synthesis with SAT

Explain SAT solvers

- Explain how they might help with state space explosion
- (Explain that there is room for both approaches in the world)
- Explain QBF and/or why this is nontrivial

2 Related Work

3 Formalisation of Software

Synthesis is a process that demands mathematical formalisation in order to provide a strong guarantee of the correctness of the resultant software. As such we require a mathematical language to describe the system we wish to produce, the environment in which it operates, and the properties we want the system to adhere to. This chapter will outline that language and the ways we can reason about what we describe in it.

3.1 Reactive Systems

Devices drivers are an example of a reactive system. A reactive system is a system that is in a continuous process of responding to input from its environment. A driver accepts requests from the operating system and state information from the device, and it responds by sending commands to the device and reporting to the operating system.

3.2 Kripke Structures

A reactive system can be thought of as a sequence of *states*. The system *transitions* between these states as it responds to its inputs. A Kripke structure [4] formalises this notion and provides us will the language to reason about a reactive system.

A Kripe structure M is defined by the tuple $M = (S, S_0, R, L)$ with respect to a set of atomic propositions AP.

- A finite set of states, S,
- a subset of initial stats $S_0 \subseteq S$,

- a transition relation $R \subseteq S \times S$, and
- a labelling function $L : S \to 2^{AP}$.

The transition relation defines how the system moves between states. It must be left-total, i.e. for every $s \in S$ there is an $s' \in S$ s.t. $R(s, s')$. The labelling function maps every state in S to a set of atomic propositions that hold in that state of the system.

We often consider *paths* or *runs* of a Kripke structure. A path is a sequence of states $\pi = s_0, s_1, s_2, ...$ such that $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

3.3 Temporal Logic

Kripke structures lay the groundwork for reasoning about reactive systems. Using the labelling function we may define desirable properties for the system. What is now lacking is a means of bringing states together to reason about the system as a whole. This requires a logical language that can express temporal properties.

Temporal logic takes propositional logic and provides additional semantics for the concept of time. In a Kripke structure this refers to the expressiveness to reason about runs of the system. This allows us to define properties that must be true for the entire execution of a reactive system.

Linear temporal logic (LTL) allows for statements that refer to a single run of a Kripke structure. Pnueli introduced LTL in 1977 [6] to succinctly describe the outcomes of program execution by referring to global invariants and eventualities. The syntax is:

- ϕ is a propositional formula referring to the current state,
- $X\phi$ - ϕ is true in the next state of the execution,
- $F\phi$ - Eventually (finally) ϕ will be true, and
- $G\phi$ - ϕ is always (globally) true.
- $\phi_1 U \phi_2$ - ϕ_1 holds until ϕ_2 holds.

These operators are semantically defined with respect to a Kripke structure $M = (S, S_0, R, L)$. We use $M, s \models \phi$ to denote ϕ holds true at state $s \in S$ of structure M. We definte $\models$ recursively:

- $M, s \models \phi$ iff $\phi \in L(s)$.
- $M, s \models \neg\phi$ iff not $(M, s \models \phi)$.
- $M, s \models \phi_1 \wedge \phi_2$ iff $(M, s \models \phi_1) \wedge (M, s \models \phi_2)$.
- $M, s \models \phi_1 \vee \phi_2$ iff $(M, s \models \phi_1) \vee (M, s \models \phi_2)$.
- $M, s \models X\phi$ iff for some state s', $R(s, s') \wedge M, s' \models \phi$.
- $M, s_0 \models F\phi$ iff for some path $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi))$.
- $M, s_0 \models G\phi$ iff for some path $(s_0, s_1, ...)$, $\forall i(i \geq 0 \wedge (M, s_i \models \phi))$.
- $M, s_0 \models \phi_1 U \phi_2$ iff for some path $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi_2) \wedge \forall j(j \geq 0 \wedge g < i \rightarrow (M, s_j \models \phi_1)))$.

Throughout this thesis will use F and G to represent the *finally* and *globally* operators. Elsewhere in the literature $\Diamond$ and $\Box$ are sometimes used to represent the same.

In addition to LTL, which is used to formalise properties about a single execution trace, we may need the ability to talk about aggregations of traces. In 1981 Clarke introduced computation tree logic (CTL) [3], which has additional syntax and semantics for exactly that. The syntax of CTL is as follows:

- $A\phi$ - ϕ is true on all paths
- $E\phi$ - there exists a path on which p is true

We again define the semantics of CTL with respect to a Kripke structure $M = (S, S_0, R, L)$.

- $M, s \models \phi$ iff $\phi \in L(s)$.
- $M, s \models \neg\phi$ iff $\neg(M, s \models \phi)$.
- $M, s \models \phi_1 \wedge \phi_2$ iff $(M, s \models \phi_1) \wedge (M, s \models \phi_2)$.
- $M, s \models \phi_1 \vee \phi_2$ iff $(M, s \models \phi_1) \vee (M, s \models \phi_2)$.
- $M, s \models EX\phi$ iff for some state s', $R(s, s') \wedge M, s' \models \phi$.
- $M, s \models AX\phi$ iff for all states s', $R(s, s') \rightarrow M, s' \models \phi$.

- $M, s_0 \models A[\phi_1 U \phi_2]$ iff for all paths $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi_2) \wedge \forall j(j \geq 0 \wedge g < i \rightarrow (M, s_j \models \phi_1)))$.
- $M, s_0 \models E[\phi_1 U \phi_2]$ iff for some path $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi_2) \wedge \forall j(j \geq 0 \wedge g < i \rightarrow (M, s_j \models \phi_1)))$.
- $(M, s \models AF\phi) \Leftrightarrow (M, s \models A[\top U \phi])$
- $(M, s \models EF\phi) \Leftrightarrow (M, s \models E[\top U \phi])$
- $(M, s \models AG\phi) \Leftrightarrow (M, s \models \neg EF(\neg\phi))$
- $(M, s \models EG\phi) \Leftrightarrow (M, s \models \neg AF(\neg\phi))$

In CTL, each A or E must be paired with an LTL operator. For example $AG\phi$, which says that ϕ must always hold on all paths. Alternatively, CTL* allows for free mixing of operators from LTL and CTL. This allows for terms such as $E(GF\phi)$, which is true iff there exists a path where ϕ will always be true at some future state.

3.4 Model Checking

Once the desirable properties of a system have be laid out in temporal logic, the next step is to verify that a program adheres to its specification. One approach is to manually (or with the assistance of a proof checker) construct a proof for the program. However, this costs the verifier in both time and mental effort and so a mechanised approach is desired. The first such automatic *model checker* was proposed by Clarke et al. [2] to verify temporal properties of finite state programs. The algorithm they proposed was a search based labelling of a finite state-transition graph, representing the program, with subformulas of the temporal logic specification.

Another approach is based on the notion that temporal logic properties can be expressed in terms of automata theory. Specifically, a finite state automaton over infinite words can be used to represent a temporal logic formula. Büchi automata [1] are ω-automata, i.e. finite automata that accept an infinite stream of input, which may be constructed such that the automaton will accept exactly the inputs allowable by a temporal logic formula.

In [7], the authors propose a model checking approach using this connection between temporal logic and automata theory. They propose the

construction of a finite state, infinite word generator representing the program P, and an acceptor of the same, ϕ, constructed from the temporal property to be checked. Thus the program may be model checked by determining whether $P \cap \neg\phi$ is empty.

Like all ω-automata, the language of a Büchi automaton is ω-regular. A regular language over the alphabet Σ is

- The empty language , *or*
- A singleton language $\{a\}$ for $a \in \Sigma$, *or*
- For two regular languages A and B:
  - $A \cup B$ the union of those languages, *or*
  - $A \,.\, B$ the concatenation of those languages, *or*
  - $A*$ the Kleene operation on that language.

An ω-regular language is a regular language with infinitely long words.

3.5 Reactive Synthesis

Synthesis as a Game

GR(1) Games

3.6 Binary Decision Diagrams

3.7 Abstraction

3.8 SAT

3.9 Interpolation

4 Synthesis with SAT

5 Evaluation

6 Conclusion

Bibliography

[1] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science (Proc. 1960 Internat. Congr .)*, pages 1–11. Stanford Univ. Press, Stanford, Calif., 1962.

[2] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

[3] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs*, pages 52–71, 1981.

[4] Saul Kripke. Semantical considerations on modal logic. In *Acta Philosophica Fennica*, volume 16, pages 83–94. 1963.

[5] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 305–318, New York, NY, USA, 2011. ACM.

[6] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[7] Moshe Y. Vardi. *Logics for Concurrency: Structure versus Automata*, chapter An automata-theoretic approach to linear temporal logic, pages 238–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.