Automatic Synthesis of High-Assurance Device Drivers

GERNOT HEISER, NICTA, Australia
LEONID RYZHYK, MICHAEL STUMM, University of Toronto, Canada
PAVOL CERNY, University of Colorado Boulder, USA
ALASTAIR F. DONALDSON, Imperial College London, UK

1 Executive summary

Device drivers are hard to develop and are notoriously unreliable [13,20]. While constant innovation in the area of electronic design automation has led to dramatic improvements in the IC design process, device driver development practices have not changed much since the days of mainframe computers. As a result, it is common nowadays for a product release to be delayed by driver rather than silicon issues [42].

To address this long-standing problem, we propose a new driver development methodology that will allow *faster* creation of device drivers with *fewer defects*. The innovation at the heart of our methodology is the **automatic synthesis** of **correct-by-construction** device drivers from a formal model of the hardware device and a specification of the driver/OS interface.

Our methodology has the following crucial properties.

- **Dramatic reduction in cost of driver development, maintenance and QA.** This is achieved by synthesizing drivers automatically from existing device specifications provided by hardware designers as *transaction-level device models* (TLMs).
- **Support for hardware/software co-design and co-verification.** Our approach allows device drivers to be created, tested, and formally verified early in the product design workflow, before the hardware device is available in silicon. The availability of the device driver in turn facilitates evaluation and validation of the hardware design.
- **Strong functional correctness guarantees.** Our correct-by-construction approach to synthesis, combined with formal verification to detect bugs in input specifications and the synthesis algorithm, yields a high degree of confidence in the resulting device driver code.
- **Broad application.** Our approach is applicable to a wide range of modern I/O devices, including SoC IP blocks, communication devices, media and storage adapters, etc.

Research vectors addressed

This project will primarily attack the following research vectors identified in the Intel RFP.

- **Driver synthesis** This project will advance state of the art in game-based reactive synthesis resulting in the first practical driver synthesis tool applicable to a wide range of complex I/O devices.
- **Driver validation** We combine synthesis and verification to achieve a high degree of assurance that the produced drivers are functionally correct. This goes well beyond what is possible with existing driver analysis tools, which are limited to detecting specific classes of defects.
- **Hardware/software co-design & co-verification** By reusing hardware device specifications in driver synthesis, we achieve tighter integration between driver and hardware design workflows to the benefit of both software and hardware designers.

Expected impact

This work will lead to a radical change in the way device drivers are developed. It will provide a solid scientific and engineering foundation that is missing in the current development practice, and thus enable the creation of consistently high-quality drivers with predictable, and faster, development times.

Quantitatively, we require that our tools can synthesize and verify drivers for high-end devices within 20 minutes, yielding code whose efficiency and size is within 10% of manually crafted drivers. We aim to reduce overall driver development time from the current average of 6-12 months to less than one month.

Team and partners

The investigatory team's combination of complementary skills covers the wide range of research areas necessary to execute this ambitious project. Principally, these are Computer Systems (Heiser, Ryzhyk, Stumm), Program Synthesis (Cerny and Ryzhyk), Software Verification (Donaldson, Cerny and Ryzhyk) and Concurrency (Donaldson and Cerny). We have also secured the support (see attached letter) of Byron Cook, Principal Researcher at Microsoft Research, who was a key player in Microsoft's SLAM project on device driver verification.

2 Background and details of proposed technology

2.1 State-of-the-art and its limitations

Device driver reliability problems have drawn a lot of attention in both industry and academia. Previous research identified several key sources of driver defects [13, 29, 34, 36]. First, driver development is currently not well integrated into the hardware design process: driver writers typically work from device data sheets provided by hardware designers, which are often ambiguous or incomplete, and are prone to errors. These flaws adversely affect the quality of resulting drivers. Further, driver development typically starts after hardware design has been completed, putting driver development on the critical path to product delivery and limiting the time that can be spent on driver quality assurance.

The second major source of device driver bugs is the complex interface between drivers and the OS. Modern OSs define complex rules on driver-OS interactions that are neither well documented not stable across different OS releases. Furthermore, the OS can invoke the driver from multiple concurrent threads, which leads to numerous possibilities for race conditions and deadlocks.

Previous research has made major progress in using automatic verification methods to detect, or prove absence of, defects in driver code related to the driver-OS interface [3,14,15,19,21]. While this is a major step forward, current work cannot detect defects in device management code, which constitutes the core of any device driver, and does not scale to large device drivers which make extensive use of program features that are challenging for verification, such as sophisticated pointer manipulation, indirect function calls and bit-level arithmetic. Existing work has thus had an important but modest impact on the quality of device drivers.

2.2 Overview of our methodology

We put forward a radically different approach to solving device driver reliability problems. Instead of fixing or isolating bugs in existing drivers, we propose to automatically derive correct-by-construction driver implementations from device specifications. By reusing existing device specifications created by hardware designers, this approach allows creating drivers early in the product design cycle and with minimal manual effort. To achieve strong functional correctness guarantees, we propose formal techniques for automatically validating the synthesized driver.

Our proposed methodology comprises two main components, *driver synthesis* and *driver verification*, which are illustrated graphically in Figure 1. We now summarize the role and contributions of each component.

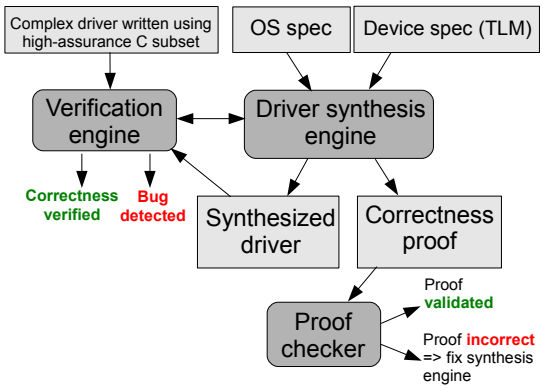

Figure 1: Overview of the driver synthesis methodology.

Driver synthesis The core innovation underpinning our method is a technique and tool for automatic synthesis of correct-by-construction device drivers, indicated by the *driver synthesis engine* in Figure 1. The key insight behind this innovation is that the problem of synthesizing device drivers from device specifications can be precisely formalized using the apparatus of *game-based reactive synthesis*. To this end, we interpret interactions between the driver, the device, and the OS as a game, where in order to win the driver must satisfy any possible sequence of OS requests given any feasible device behavior. Driver synthesis then consists of finding a *winning strategy* in the game on behalf of the driver.

The synthesis process starts with two input specifications: the *device specification* and the *OS specification*. The device specification is a transaction-level model of device behavior created by hardware designers as part of the standard design workflow. The OS specification describes the communication protocol between the driver and the OS. This specification is developed by driver framework designers and is *generic* across a class of similar devices, e.g., Ethernet controller devices, thus a single OS specification for a device class can be used to synthesize many drivers for devices from this class.

Given the input specifications, the synthesis tool computes a winning driver strategy and generates a complete driver implementation in C. Optionally synthesis can be guided by user input to improve the structure of synthesized code, making it more more human readable and easier to maintain.

The driver implementation synthesized at this step assumes a *sequential* OS environment that serializes all requests to the driver. The next step in this project is

to automatically extend this sequential implementation with *synchronization code* required to handle concurrent driver invocations from multiple OS kernel threads, satisfying the requirement that invocations of a driver return the same results in the multi-threaded setting as they would if all the calls to the driver were serialized. Implementing this simple specification manually is difficult and error-prone. By synthesizing synchronization code automatically we fully harvest the benefits of synthesis—unburdening programmers from difficult details of synchronization code, while letting them use generic, reusable correctness specifications. In synthesis for concurrency, we will also focus on performance aspects, mainly by developing techniques that minimize the necessity for locking and other synchronization primitives, but also by using quantitative performance models.

Driver verification Our driver synthesis technique will be designed to generate drivers that are correct-by-construction with respect to the input specifications. Correctness of the synthesized driver is *guaranteed* provided that the input specifications are correct and the synthesis tools have been implemented correctly. We will investigates methods for checking the latter, determining cases where bugs in the implementation of our synthesis algorithm lead to defects in the synthesised device drivers. In addition, we will investigate language- and tool-support for direct construction of high-assurance device drivers: for especially complex and highly parallel devices, such as graphics adapters, direct driver synthesis may be out of scope. In this case we will provide a "high-assurance" subset of C, designed to facilitate formal verification, so that drivers for such devices can be written to high quality standards and checked using rigorous tools.

Verification of input specifications The correctness of drivers generated by our synthesis technique depends on the validity of input specifications: the TLM description of a device, and the description of OS interface specification. In this proposal we do *not* address the problem of checking input specification correctness. There is significant existing technology (especially within Intel) for TLM/RTL cross-checking. The OS interface specification is simpler than the device TLM and is therefore easier to get right; on the other hand it is very hard to verify, as this specification is implemented by hundreds of thousands lines of low-level kernel code, therefore we regard formal validation of the OS interface specification is *outside the scope* the proposed project.

Driver cross-verification Assuming correct input specifications, a bug can only be introduced in the synthesized driver due to a defect in synthesis tools. We will explore two complementary approaches to validating the results of synthesis. The first approach will use *model checking* to automatically check that the generated driver does indeed implement a winning strategy in the game, and hence that it is correct with respect to the input specifications. This is indicated by the *verification engine* process of Figure 1: the engine is used to analyse a driver post-synthesis. The second, more complicated but potentially more reliable approach, consists of extending our synthesis tools to produce a formal *correctness proof* of the synthesized driver. This proof can be validated by a theorem prover such as Isabelle [28] or Coq [6], as indicated by the *proof checker* process in Figure 1. Proof validation provides a strong guarantee that the driver is correct with respect to the device and OS specifications that were used as input to synthesis.

Verification and language support for development of complex drivers We are confident that our novel synthesis techniques will be capable of generating drivers for a wide range of devices. Nevertheless, drivers for highly complex and concurrent devices such as graphics adapters will be difficult to obtain via synthesis alone. To support the development of complex drivers for parallel devices (see Figure 1) we will investigate language support to allow such drivers to be expressed using a subset of C that is "high-assurance" in the sense that it is deliberately restricted for simplicity of coding and ease of formal analysis. The verification engine component of Figure 1 will be extended with novel algorithms to verify drivers written in this high-assurance language subset, with a particular focus on scalable reasoning about concurrency.

2.3 Further background: TLMs

Transaction-level device models (TLMs) [9] are commonly used in modern hardware design. A TLM describes device operation in terms of high-level *transactions* rather than clock cycles. A transaction represents an event such as a bus transfer, a device configuration change, or transmission of a network packet. Being much faster to develop than RTL, TLMs enable rapid architectural exploration early in the design flow. They are also commonly used in building functional system simulators long before the device becomes available in silicon. Finally, TLMs are used at the RTL verification stage, to ensure that the resulting design behaves accord-

ing to the model. Because TLMs operate at the level of abstraction relevant to driver developers, they are ideal for use in driver synthesis. In particular, a TLM focuses on describing *externally visible* device behavior, rather than its internal architecture, and on wall-clock timing of device operations rather than cycle-accurate timing.

3 Detailed technical rationale, approach, and research plan

Our aim is to develop scalable algorithms for driver synthesis and verification. We will implement these algorithms in a practical driver synthesis toolkit and evaluate them by synthesizing and verifying drivers for a variety of I/O devices, ranging from simple integrated sensors and hardware accelerators to high-end devices such as Intel Gigabit Ethernet controllers, Wi-Fi controllers, storage controllers, audio and graphics adapters.

We now present a detailed technical plan for the project as three work packages: *Guided Sequential Synthesis* (WP 1), *Synthesis for Concurrency* (WP 2) and *Formal Verification* (WP 3). The work packages are related and together constitute an ambitious plan of work. To reduce the risk of the project producing "all or nothing" results, the work packages are loosely coupled and have been designed to generate independently useful outputs. Furthermore, as presented in our schedule in Section 4, the work packages have been designed to run concurrently. Thus while we believe the expertise of our high quality investigatory team make the project likely to succeed in its aims, even a partial success will yield techniques that are highly relevant to industry.

WP 1. ***Guided Sequential Synthesis.***
Lead partners: NICTA and University of Toronto.
This work package is devoted to guided synthesis, which is at the core of our novel approach. The work package is supported by WPs 2 and 3 which cater for concurrency and high assurance, respectively.

The guided sequential synthesis tool will automate the tedious and error-prone task of implementing the driver logic. The key insight behind our approach is that this task can be naturally formalized as a *game* that the driver plays against the device. *Goals* in this game are set by the OS via I/O requests to the driver. In order to complete the request, the driver must control the device to perform the requested I/O action. The driver forces state transitions inside the device by reading or writing device registers. However, not all device transitions are directly *controllable* by the driver. Device-internal events, such as receiving of a network packet or occurrence of an error condition are outside the driver's control. The synthesis tool must find a *winning strategy* in the game on behalf of the driver, that guarantees that the driver is able to satisfy all possible sequences of OS requests under any sequence of *uncontrollable* device transitions permitted by the device specification.

By reformulating driver synthesis as a game, we open the way to leveraging the rich theoretical and algorithmic apparatus of reactive game theory. Turning this vision into a practical tool requires addressing several further challenges.

Scalability challenges Real I/O devices exhibit rich internal state and complex behavior. Synthesizing a driver for such a device can, in the worst case, involve a prohibitively expensive *exhaustive exploration* of the state space of the device model. To date, game-based synthesis algorithms have not been applied to problems comparable in size and complexity to those arising in driver synthesis [30]. As a result, existing theory does not offer a satisfactory solution to the state explosion problem.

In this work package we will develop a new technique for combating the state explosion problem based on *abstraction*. Abstraction-based techniques have revolutionized software verification [3], making it applicable to complex real-world systems. We believe that this approach will also prove fruitful in driver synthesis. Abstraction allows eliminating irrelevant details from input specifications, focusing on the information relevant for the task at hand.

Ideally, we would like to find the coarsest possible abstraction that still contains enough information to compute a winning strategy for the driver or to prove that such a strategy does not exist. We will obtain such an abstraction incrementally using a new *abstraction refinement* algorithm. Starting from a very coarse initial abstraction, the algorithm computes progressively more precise abstractions of the game automaton. At every iteration, it attempts to find a winning strategy in the abstract game. This can lead to three possible outcomes: (1) synthesis fails due to imprecise abstraction, (2) nonexistence of a winning strategy is established, or (3) a winning strategy is found. In the first case, the algorithm identifies a set of transitions of the abstract game automaton that must be refined for synthesis to make progress. It then computes a more precise abstraction of these transitions and retries the synthesis step, thus starting a new iteration of the abstraction-refinement loop. In the second and third cases, the abstraction refinement algorithm terminates and returns either a failure (case 2)

or a computed winning strategy (case 3).

Usability challenges The second set of challenges is related to the creation of a practical software development process around driver synthesis. To have serious impact on developer productivity our techniques must be highly automatic, yet they must allow the user to communicate their expert knowledge of device performance characteristics to generate highly optimized code. Similarly, the user should be able to guide synthesis towards the production of structured code that can be easily maintained. When synthesis fails because of a mismatch between input specifications, meaningful feedback must be provided, helping the software developer to identify and rectify the issue.

Our answer to this challenge is *guided synthesis*. Instead of synthesizing a concrete winning strategy, our synthesis algorithm will produce *the most general strategy*, that contains all possible winning driver moves in each state. As the code generator backend unwinds this strategy into a program control flow, the user can optionally control this process. Actions available to the user at this point include choosing one of available winning moves in the current program location, rearranging program statements, and outlining statements into functions. Importantly, the synthesis tool makes sure that any choices made by the user remain within the most general winning strategy; hence *user error cannot lead to bugs in the synthesized driver*.

Our strategy for debugging synthesis failures due to input specification mismatches is based on the use of *counterexample strategies*. Whenever the synthesis algorithm cannot find a winning strategy for the driver, it generates an explanation of the failure in the form of a counterexample strategy. This strategy describes an environment behavior under which the driver is unable to win the game. The user explores the counterexample strategy using an interactive debugging tool. During the debugging session the user plays the expected winning driver strategy and the debugger responds with counterexample transitions on behalf of the environment. Through this interaction the user encounters an unexpected state of the game, from which the driver is unable to complete an OS request, providing insight into the root cause of the mismatch.

Figure 2 shows the proposed architecture of the driver synthesis tool that we will design to address the above challenges. Each component of the tool corresponds to a task in this work package:

T1.1 Compiling HDLs to synthesis specifications. We will design a compiler to translate device TLMs expressed in a hardware description language (HDL) to the *internal specification language* of the synthesis tool.

T1.2 Translation into game automata. We will define a translation of a device model (after conversion to our internal specification language) and an OS specification into a *game automaton*, which compactly represents all game components.

T1.3 Abstraction and synthesis. These are the central components of the synthesis toolkit responsible for computing a winning strategy for the driver or a counterexample strategy for the environment. They implement the abstraction refinement scheme described above.

T1.4 Debugging GUI. To help the user troubleshoot synthesis failures we will design a graphical format for tracing failures back to defects in input specifications. The interface will allow interactive exploration of the counterexample strategy produced by the synthesizer.

T1.5 Code generation. The final stage of synthesis is to convert the winning strategy computed by the synthesizer into C source code under optional interactive user guidance. Starting from the initial state of the system, it unwinds the strategy into a program control flow, translating winning driver actions into C statements. In doing so, it aims to minimize the resulting control flow automaton by merging similar states in order to avoid code bloat.

In *T1.1* we will develop a compiler for a single HDL, chosen based on the availability of device specifications in this language. One candidate is the WindRiver DML [40] modeling language, which ships along with a set of DML models for many Intel devices as part of the Simics simulation platform [39]. Support for additional HDLs (SystemC, C++, SystemVerilog, etc.) can be implemented later via additional frontend compilers.

Extensions Subject to time and resource availability, we will explore two further extensions of the basic synthesis tool described above. First, we will investigate synthesis of *hardened* device drivers [22, 26]. A hardened driver is able to detect unexpected environment behaviors, such as device malfunctions or OS errors, and take appropriate recovery measures, e.g., aborting or retrying the current I/O operation, resetting the device, or logging the faulty input. Second, we will explore the

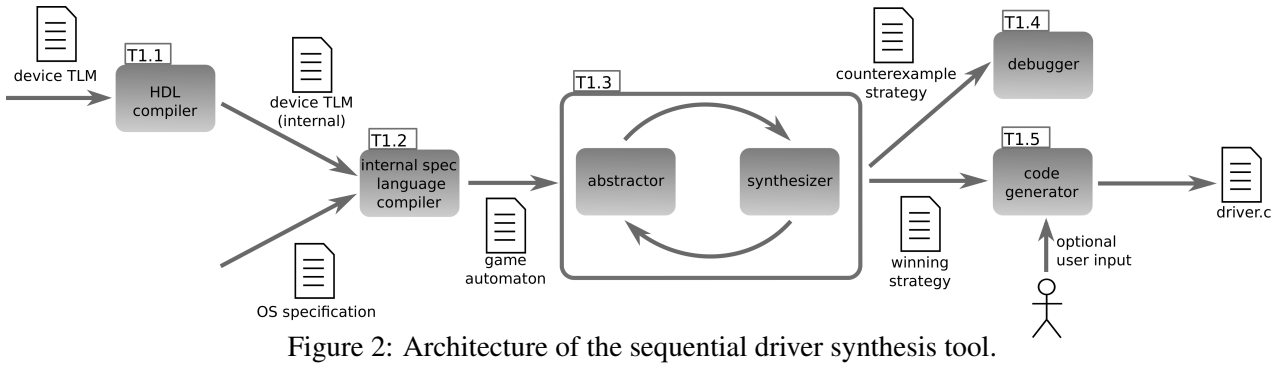

Figure 2: Architecture of the sequential driver synthesis tool.

use of *quantitative synthesis* techniques [8, 10, 11] in device driver synthesis. Quantitative synthesis allows synthesizing code that is not only functionally correct, but is also optimal in terms of execution time, power consumption, memory footprint, or other criteria important to the developer. While we expect strategies synthesized using methods described so far to perform well in practice, further performance tuning can be achieved by assigning costs to various driver and device operations and using quantitative synthesis algorithms to generate optimal or close-to-optimal implementations.

Related work Automatic game-based device driver synthesis was pioneered by two of the investigators (Ryzhyk and Heiser) in their work on the Termite driver synthesis tool [35]. This research laid out the key principles behind the approach and demonstrated its feasibility. However, this proof-of-concept implementation lacked a scalable synthesis algorithm applicable to complex real-world devices and did not support synthesis from existing device specifications. In this work package we will address these limitations, turning automatic driver synthesis into a practical and highly desirable alternative to the conventional manual driver development process.

WP 2. ***Synthesis for Concurrency.***
Lead partner: University of Colorado Boulder.
The *Synthesis for Concurrency* work package will develop techniques and tools to automatically synthesize drivers that can be safely called concurrently from multiple OS kernel threads. The drivers produced in WP 1 assume a single-threaded environment, where each call to a driver is executed in isolation. However, contemporary OS kernels are multi-threaded, and thus several threads can invoke the driver concurrently. In order to prevent program errors stemming from concurrency, synchronization between driver methods is necessary. Therefore, the next step in full functional synthesis of drivers is to transform the driver code produced in WP 1 to make

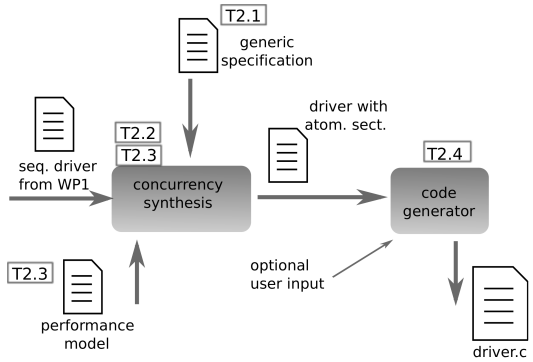

Figure 3: Synthesis for concurrency tool.

it work correctly in a multi-threaded environment. The transformation can be done for instance by adding locks or other synchronization primitives.

Synchronization code is an ideal target for synthesis. This claim is based on two observations. First, synchronization code is notoriously hard to implement and debug, as witnessed by a large number of concurrency-related bugs in existing device drivers [13, 29, 34]. Second, correctness is easy to specify declaratively. The reason is that one can often use *generic* (i.e., application independent) specifications, such as deadlock-freedom or linearizability. A tool that synthesizes synchronization code will thus have a big effect on programmer productivity.

Related work. Synthesis for synchronization code is an area that has attracted considerable attention recently [12, 18, 37, 38]. These pioneering works on synthesis cannot be directly applied to device drivers. The reasons are two-fold. First, code synthesis for device drivers has to take into account specific constraints imposed by the OS on the device driver architecture. For instance, one cannot use common primitives for mutual exclusions, such as semaphores, in interrupt handlers (a core part of device drivers). Second, the previous work does not consider performance objectives in synthesis,

or considers only limited syntactic criteria that might (but are not guaranteed to) influence performance positively. We propose to go beyond the previous work and develop new synthesis methods that do not suffer from these limitations.

The goal of the WP 2 is to produce a tool that takes the sequential driver produced in WP 1 as input, and inserts synchronization primitives to ensure that the driver behaves correctly in the concurrent (multi-core) setting. The research proposed in this work package will focus on the following four tasks.

T2.1 Specifications for concurrency in drivers. We will introduce specifications that are generic and that take into account device driver architecture.

T2.2 Synthesis by code transformations. We will devise a synthesis algorithm that minimizes the number of synchronization primitives used.

T2.3 Performance-aware synthesis. Building on our previous work, we will develop quantitative synthesis methods for device drivers.

T2.4 Synchronization code generation. We will develop code generation techniques for device drivers that respect constraints by the OS on drivers.

The workflow of the proposed tool is in Figure 3. We now describe each part in more detail.

Specifications for Concurrency in Drivers. Program synthesis is most useful when it is much easier to say *what* needs to be done than to say *how* it should be done. That is, program synthesis is most useful when declarative specifications are easier to write than imperative programs. This is the case for concurrency in drivers. Intuitively, the concurrency specification requires simply that the invocations of the driver return the same results as they would if all the calls to the driver were serialized. In *T2.1*, we will develop *generic* specifications that capture this intuition for broad classes of drivers, are simple to write and use, but take into account all sources of concurrency in drivers, e.g., interrupts, top- and bottom-halves, and workqueues. This requires extending standard correctness notions such as linearizability.

Synthesis. We will develop synthesis algorithms for placement of synchronization code, focussing on *correctness* (*T2.2*) and *performance* (*T2.3*). There are two principal inputs to the tool we plan to develop: (i) the driver code produced in WP 1, which assumes a sequential setting, and (ii) the generic specification produced in *T2.1*. The output of *T2.2* and *T2.3* is a driver that works correctly in concurrent setting, but uses *atomic sections*, a synchronization primitive that marks parts of code that is to be executed atomically. Atomic sections are desirable as they are easy to reason about algorithmically, but they are not provided by mainstream OS kernels — the purpose of Task *T2.4* is to implement them using common kernel synchronization primitives.

In *T2.2*, the novel technical contribution is the development of *semantics-preserving transformations* of sequential code that aid in synthesis of synchronization. Synchronization code (such as locks) can often be thought of as having no effect on sequential execution (i.e., it is semantics-preserving), but makes concurrent execution safe. Generalizing this view, we propose to consider a number of other semantics-preserving code transformations. Such transformations will decrease the need for the use of synchronization primitives, thus ensuring better performance. For instance, the tool might obtain a correct driver with fewer locks, if it first rearranges the order of instructions in the driver code (if the rearrangement does not change the sequential behavior of the driver). We will use a counterexample-based approach to synthesis. In a loop, we will first use an off-the-shelf tool to check if there is a concurrent execution leading to a bug (such as assertion violation). If there is no bug, we output the current code. If there is a bug, we will transform the code in order to disallow the counterexample. The transformation can for instance be a rearrangement of the order of instructions (in a way that preserves sequential correctness), or it can be insertion of atomic sections.

In *T2.3*, we will work on synthesizing highly performant code. In prior work we developed the first approach for synthesizing concurrent programs with respect to both performance *and* correctness criteria [10] for *finite-state* programs. We propose to extend this approach using our recent work on abstract interpretation for quantitative objectives [11]. To quickly guide the synthesis engine to well-performing solutions, we will investigate common synchronization patterns used by driver developers. Such patterns are often chosen with performance in mind, so using them in synthesis will improve the quality of the synthesized code.

Synchronization code generation. The output from *T2.2* and *T2.3* will be a correct driver code with atomic sections. The purpose of *T2.4* is to implement atomic sections using synchronization code supported by the kernel, such as spinlocks, mutexes, RCU locks, etc. This goal is not straightforward. For instance, in interrupt handlers, the use of semaphores, or other primitives that

my cause a process to block, is not allowed. Therefore, existing techniques for code synthesis cannot be used. We will develop algorithms for code generation that respect constraints given by device driver software architectures, and are informed by the best practices from current driver development. To help achieve the latter, we will allow user feedback, where a user could flag synchronization code as not suitable, and can suggest other synchronization primitives to use. This is an instance of our guided synthesis approach, the paradigm used throughout the project.

Our techniques for synthesising synchronization code will rely on a technique for finding bugs in, or verifying, concurrent software. Early experiments with off-the-shelf techniques for this purpose [32] show promise, but suffer from scalability limitations. In WP 3 we will investigate technique for efficient analysis of concurrent drivers, which will accelerate the techniques here for synthesising synchronization code.

WP 3. ***Formal verification.***
Lead partner: Imperial College London.
This work package is devoted to achieving strong formal guarantees that device drivers behave correctly. The focus is on analysis of both drivers generated by the synthesis techniques of WPs 1 and 2, which may be incorrect due to bugs in the synthesis algorithm implementation, and complex hand-written drivers that are beyond the scope of our synthesis techniques. Furthermore, the efficient verification and bug-finding techniques for concurrent device drivers developed in this WP will serve to accelerate the techniques for synthesising concurrency-safe device drivers in WP 2.

T3.1 Device driver cross-verification. Apply model checking- and theorem proving-based formal verification techniques to drivers generated by our synthesis tool

T3.2 Automatic proof generation. Extend the synthesis techniques of WPs 1 and 2 to facilitate the generation of device driver correctness proofs that can be automatically checked

T3.3 Verification and language support for complex, concurrent device drivers. Devise scalable techniques for verifying concurrent drivers, to accelerate synthesis for concurrency, and complex drivers that are beyond the reach of our synthesis algorithms

To avoid erroneous drivers resulting from bugs in our synthesis implementation, *T3.1* will investigate model checking- and theorem proving-based techniques for driver validation. We will build on prior work on device driver verification by the Investigators [1, 2] and others [4, 5, 41].

The techniques developed in *T3.1* will be specifically geared towards our synthesis setting, exploiting the guarantee that synthesised drivers will use a restricted, clean subset of C programming language, free from some of the complexity of low-level systems code which often causes formal verification techniques to fail.

Another strategy for validating our synthesis algorithms, discussed in Section 2.2, is a *proof generation* approach. This is the focus of *T3.2*. The idea is that our synthesis tool will generate both a driver implementation and an associated proof of correctness. This proof is produced by capturing decisions made by the synthesis algorithm using formal logic. For example, whenever the algorithm chooses a certain winning action in a given state of the game, it records the fact that all possible device states reachable by taking this action are within the previously computed winning region, and hence the choice of the action is correct.

We will extend the synthesis technique of WPs 1 and 2 so that the proof of a winning strategy (certifying driver correctness) is generated as a proof script for a theorem prover such as Isabelle or Coq. This will allow generated proofs to be automatically checked by Isabelle/Coq, yielding a very high degree of assurance that the synthesized driver is correct.

T3.3 will concentrate on verification techniques for highly complex and concurrent drivers. This has two aims: to produce stand-alone analysis techniques that can be directly applied to complex driver source code, and to serve as an efficient engine for the concurrency-aware synthesis techniques of WP 2. While we are confident that the synthesis techniques proposed in WPs 1 and 2 will be capable of generating drivers for a wide range of devices, drivers for especially complex and concurrent devices, such as graphics adapters, will be out of scope. Furthermore, concurrency-aware synthesis, especially generation of synchronization code (*T2.4*) depends on an efficient engine for finding bugs in, or proving correctness of, concurrent driver code.

Our strategy for delivering scalable analysis techniques for complex and concurrent driver code has three strands. First, for *verification* we will exploit recent advances by Donaldson and collaborators in techniques for source-level verification of highly parallel software [7, 16, 17]. Second, for *bug-finding* we will employ tech-

niques for mitigating state-space explosion due to concurrency in a manner that preserves large classes of bugs, including context-bounded search [27] and concurrent-to-sequential program transformations [23]. Third, we acknowledge that full device driver verification is virtually impossible to achieve for existing drivers written in C, hence we will investigate *language support* for writing *verifiable drivers*. This will involve identifying a restricted subset of C that allows full driver functionality to be expressed, but which features a conservative type system, limited pointer arithmetic (which together guarantee type safety), and specially managed dynamic memory allocation. These restrictions will allow the verifier to make stronger assumptions when reasoning about access to shared data, simplifying the process of reasoning about correct concurrency.

The combination of verification techniques and programming language support developed during this work package will complement the synthesis approach of WPs 1 and 2, yielding high assurance device driver implementations.

Related work. In addition to the state-of-the-art model checking research cited throughout this section, previous work on proof generating compilers [31, 33] is relevant to this work package. Such compilers produce, in addition to the compiled code, a proof that this code is correct with respect to the source program. This proof is checked independently of the compiler by a theorem prover, thus ensuring that compiled code is correct. In task *T3.2* of this work package, we will develop a similar technique applicable to game-based software synthesis.

Programming language support for systems software development has been investigated before, most notably in the context of Microsoft's Singularity OS project [24]. We propose language support for writing verifiable drivers in task *T3.3*. We argue that our approach is more pragmatic than that of Singularity, allowing developers to write drivers using standard C, simply restricting the use of well-known dangerous and difficult to reason about language constructs in order to allow scalable verification.

4 Schedule, milestones, deliverables and evaluation criteria

Figure 4 presents a proposed schedule for the work packages and tasks described in Section 3 over a three year period. Key relationships between tasks are indicated by dashed arrows in the figure. PhD students at NICTA, Boulder and Imperial will work full time on the project, as will Dr Ryzhyk. The tasks for which they will be principally responsible are indicated in the figure. The other investigators will each devote around 10% of their time to the project, devoted to working with the team members on key algorithms and analyses.

Although aspects of WPs 2 and 3 depend on some outputs from WP1, we have designed the tasks so that it is possible for all work packages to commence simultaneously. This reduces the risk associated with strong inter-work package dependencies, and means that each institution will be able to start work on the project as soon as high quality PhD students are hired.

Milestones and deliverables

Milestone 1 (month 6): **State-of-the-art survey.** To enable technology transfer and collaboration between sites with differing expertise, and to quickly get PhD students working on the project up to speed, we will jointly produce a report describing the state-of-the-art in each area of the project. This survey of relevant literature and open source and commercial tools will also give Intel a wider perspective on the landscape of the project.
Associated deliverable: Unified survey document, contributed to by all work packages.

Milestone 2 (month 12): **Initial results for driver and hardware case studies.** During year 1 we will gather a set of *challenge benchmarks*: practical case studies to drive and evaluate our novel research. By the end of year 1 our synthesis tool will be capable of synthesizing sequential and concurrent device drivers, and will be relatively feature-rich. The focus so far will be on *correct* synthesis; synthesis of highly efficient drivers will be the focus during years 2 and 3. We will seek feedback from Intel on these case studies to ensure relevance to their device driver roadmap.
Associated deliverables: Three deliverables, one per work package, each comprised of a report describing relevant case studies and initial techniques, and an alpha distribution of tools. We will also deliver a report describing one or more *global* case studies, used to drive and evaluate our end-to-end solution: open source device descriptions for which synthesis of high assurance drivers will have high impact.

Milestone 3 (month 24): **Prototype synthesis and verification tools.** By the end of year 2, we will deliver open source prototype software tools for full driver synthesis and verifying drivers written in a high-assurance subset of C, with documentation. These tools will be capable of handling a large subset of the case studies associated

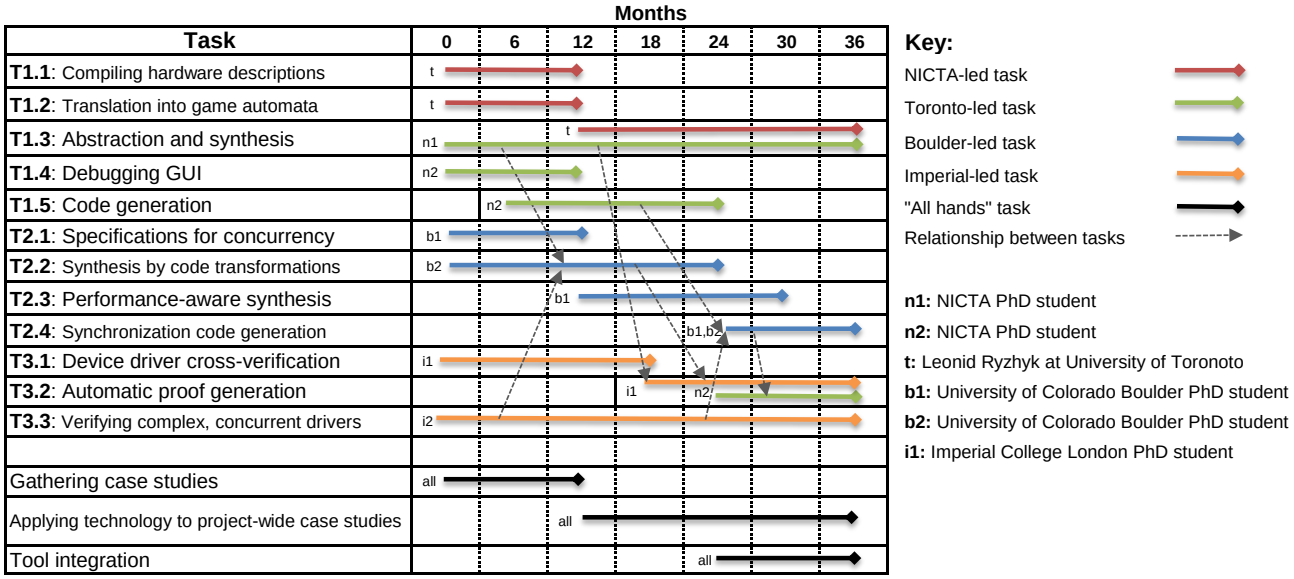

Figure 4: Schedule for the work package tasks described in Section 3.

with Milestone 2. This will allow Intel to evaluate the project outputs and provide feedback and guidance for the final year.

Associated deliverables: Two separate tools: the guided synthesis engine, which is the combined results of WPs 1 and 2, and the driver verification engine developed during WP 3.

Milestone 4 (month 36): **Verification and synthesis tools for high-end drivers.** At the end of the project, we will deliver updated versions of the open source tools that are capable of handling a wide range of drivers and devices. Our synthesis technique will be capable of synthesizing a wide range of complex drivers, and our verification methods will scale to large hardware designs, demonstrating that the techniques can be broadly applied. Due to the level of risk and scientific adventure associated with our approach, aspects of some case studies may be beyond the scope of our final tool set. In these cases we will document a set of open problems which will form the basis of further research.

Associated deliverables: Final versions of the synthesis and verification tools. The synthesis tool will be equipped with proof-generation facilities as described in WP 3. We will also deliver a report detailing our evaluation and documenting remaining open problems.

Collaboration and meetings

Collaboration between teams is essential for the success of the project. We will hold fortnightly teleconferences involving two or more partners, in response to the project requirements. Monthly, the PIs will hold a joint teleconference, where we will invite Intel collaborators, to discuss the overall direction and success of the project. Annually we will hold an informal internal project workshop involving all project members, to which Intel will be invited. The purpose of the workshops will be to evaluate the milestones reached and ensure tight integration of techniques. To reduce travel costs, workshops will be co-located with top conferences in relevant areas. Additionally we will organize extended student visits between partner sites to enable close collaboration.

Evaluation criteria

We now comment on how our proposed research meets the evaluation criteria described in the RFP.

Potential contribution and relevance to Intel. Intel devote significant effort to the design and implementation of device drivers. Our approach to automatic driver synthesis thus has the potential for major impact on Intel's device driver process, significantly reducing the bottleneck of driver development and consequently reducing product time-to-market. The techniques we design for TLM/RTL validation will also have high relevance for Intel: TLMs are used by hardware designers indepen-

dently of device driver development, and correctness of these models with respect to RTL is essential.

Innovation and non-incremental potential. Automatic synthesis has been described as a "holy grail" for device driver development. Our guided synthesis approach will provide *largely* automatic synthesis for complex device drivers with high assurance guarantees. **If we are successful in achieving these aims this will constitute a breakthrough in the field**. Our methodology is highly non-incremental: it differs radically from the current state-of-the-art in device driver design and verification.

Objectives, milestones and success criteria. The goals of our project are ambitious but the objective is simple: *A highly automatic technique for synthesizing **correct**, **efficient** drivers from hardware and OS specifications.*

The main success criterion is whether our approach will ultimately be capable of synthesizing drivers for a wide range of case-studies. In quantitative terms, we require that our tools are able to synthesize and verify drivers for high-end devices, such as Intel Gigabit Ethernet, SCSI, and Wi-Fi controllers, within 20 minutes, with the performance and code size of synthesized drivers being within 10% of manually developed and tuned drivers. We further expect the overall driver development time to decrease from 6-to-12 months, common in current industrial practice, to less than one month, when using our tools.

The plan of Section 3 breaks the project down into logically connected but loosely coupled tasks, scheduled as in Figure 4. The intermediate milestones described at the start of this section have been designed to ensure that prototype tools are built throughout the project; these tools can be directly evaluated by Intel.

Qualification of participating researchers. The Investigatory Team is described in detail in Section 5. Our multi-partner consortium combines world-leading expertise in systems, verificaiton, synthesis and concurrency, providing an excellent environment in which to make breakthrough progress in this important area.

Cost effectiveness and cost realism. Our budget is provided as an attachment to this proposal. Given the potential impact of our research for Intel, we believe that a total cost of $1,320,779 (or $1,124,986 without overheads) is a valuable investment.

Potential for co-funding. This proposal is an excellent fit for the Intel RFP. In addition, we will seek funding from national research councils to pursue research in related areas which will be complementary and supportive to the project. In particular, program synthesis research has recently been strongly supported by the US NSF, and research into system-level verification techniques is a stated growth area for funding by the UK Engineering and Physical Sciences Research Council.

5 Investigator team

Our team has a unique combination of skills and experience covering the full spectrum of theoretical and applied topics required for this project.

Gernot Heiser, Leonid Ryzhyk, and Michael Stumm will lead the effort on WP1 with personnel consisting of Leonid Ryzhyk at the University of Toronto, and two graduate students at NICTA. Pavol Cerny will lead the work on WP2, with two graduate students working with him at the University of Colorado Boulder. Alastair Donaldson will lead the research on WP3, working with two graduate students at Imperial College London.

Our team has a track record of successful relevant collaboration between team members and with Intel. Ryzhyk led a joint project between NICTA and the OS Research Group at Intel Labs, developing a new technique for improving device driver quality based on *hardware/software co-verification* [36]. Donaldson has worked with Heiser and Ryzhyk on static driver verification [1, 2]. Cerny hosted a visit from Ryzhyk and Heiser at IST Austria in 2011 to collaborate on theoretical aspects of driver synthesis, and has since collaborated with them on synthesis for concurrency in OS code.

Prof Gernot Heiser (NICTA) leads the Software Systems Research Group at NICTA and holds UNSWs John Lions Chair for Operating Systems. Since 2011 he is a Scientia Professor (UNSWs term for laureate professors). Heiser has a 20-year track record of research in operating systems. His research group at UNSW and NICTA has built a number of research OSs and kernels, including the L4-embedded microkernel and the seL4 microkernel, which is the first protected OS kernel with a *complete formal proof of functional correctness*. His team's kernels achieved several still unbroken records on inter-process-communication performance.

In 2006, Gernot Heiser founded OK Labs, a Chicago-based startup with a Sydney-based engineering team. OK Labs commercialized L4-embedded, leading to its deployment in over *1.5 billion mobile devices* to date.

Heiser's research group has completed several influential research projects on device driver reliability, including research on user-level device drivers [25], the Dingo reliable device driver framework for Linux [34],

hardware/software co-verification (*in collaboration with Intel*) [36], and the pioneering work on *automatic device driver synthesis* [35]. This work has been published in top research venues and made significant impact in the community. Heiser has been awarded the titles of *Innovation Hero*, the New South Wales *Scientist of the Year* award (2009) for the category Mathematics, Engineering and Computer Science, and is listed as one of Australia's 100 most influential engineers.

Dr Leonid Ryzhyk (University of Toronto) is currently a researcher at NICTA and a conjoint lecturer at UNSW in Sydney, Australia. In 2013 he will join the Department of Electrical and Computer Engineering at the University of Toronto as a postdoctoral researcher. He holds a PhD from UNSW.

Ryzhyk's PhD and postdoctoral research has focused on *device driver reliability* and in particular on the development of formal techniques for driver verification and synthesis. Results of this work are published in top systems research conferences, including SOSP, Eurosys, and ASPLOS. During his PhD Ryzhyk developed the Dingo reliable device driver framework for Linux [34], which is centered around a formal language for specifying communication protocols between device drivers and the OS. During the latter part of his PhD Ryzhyk developed Termite, the *world's first tool for automatic device driver synthesis* [35]. This research identified the key principles behind driver synthesis and demonstrated the feasibility of this approach.

Prof Michael Stumm (University of Toronto) is a Full Professor in the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Toronto, Canada, where he has been a faculty member since 1987. Stumm was Director of Computer Engineering from 1996–1998, and has graduated 18 PhD and 20 Masters students. Before joining the University of Toronto, Stumm pursued post-doctoral studies at Stanford University.

Stumm's primary research interests lie in the area of systems software, primarily for *parallel systems*, focusing on performance issues. He and his students developed the *Tornado* operating system which was licensed to IBM and then jointly with IBM developed the *K42* operating system. More recently, he developed a new operating system call mechanism that, for example, increases the throughput of software servers, such as Apache and MySQL by 100% and 40%, respectively. He has published over 80 papers in leading journals and top-tier computer systems conferences (H-Index: 30). He holds 7 patents, with two additional patents pending.

Stumm has co-founded two startups: he was CTO of SOMA Networks, which at its peak had over 250 employees, and until May 2012 was CEO of OANDA, the first company to bring currency trading to the retail market which now has a daily cash flow of about $10 billion.

Prof Pavol Cerny (University of Colorado Boulder) is an Assistant Professor at the University of Colorado at Boulder. Before joining CU Boulder he was a postdoctoral researcher at IST Austria. He obtained his PhD in 2009 from the University of Pennsylvania. Cerny's research interests currently center on program synthesis. His recent contributions include the techniques for *quantitative synthesis for concurrent programs*. The techniques and the resulting tool was the first to take into account performance objectives, in addition to correctness criteria, for synthesis of concurrent programs. The research was published in top conferences in verification (CAV 2011) and programming languages (POPL 2013).

His other main research contributions include work on synthesis of component interfaces (POPL 2005), on new foundational automata-theoretic model for program verification (POPL 2011), and on verification of concurrent programs (CAV 2010). His paper (TACAS 2007) won a Microsoft Research Cambridge award for best student paper. His paper on synthesis was a finalist (one of three) for a best paper award at EMSOFT 2012. He was a member of a multi-university team that performed security evaluation of voting machines for the state of Ohio prior to 2008 US presidential elections. He led the efforts in static analysis of the back-end system.

Dr Alastair F. Donaldson (Imperial College London) is a Lecturer (Assistant Professor) in the Department of Computing at Imperial where he leads the Multicore Programming Group. He is Scientific Coordinator and PI of *CARP: Correct and Efficient Accelerator Programming*, an eight-partner European Union-funded research project (total value: $ 3.6M, value for Imperial: $780,000), and PI of the UK-funded project *Scalable Automatic Verification of GPU Kernels* (value: $170,000). Before joining Imperial Donaldson was a Research Fellow at the University of Oxford working with Prof Daniel Kroening. During fall 2011 he was a Visiting Researcher at Microsoft Research, Redmond, USA, and subsequently hired as a Consulting Researcher by Microsoft during 2012.

Donaldson's research interests span two main areas: verification of system-level software and programming models for multicore architectures, and has published

more than 40 peer-reviewed research papers in these areas. His main contributions to verification (with collaborators at Oxford, Imperial and Microsoft) include a technique for automatic analysis of DMA races in multicore software [17], the first scalable predicate abstraction technique for concurrent programs [16], and a method for automatically checking that GPU kernels are free from data races [7].

Collaboration with Microsoft Research

Prof Byron Cook is a Principal Researcher at Microsoft Research, Cambridge, UK and joint manager of the Programming Principles and Tools group. Cook was a key contributor to Microsoft's SLAM project on driver verification. SLAM has been incorporated into Microsoft's Static Driver Verifier tool which ships with the Windows Driver Development Kit, and is widely used by driver developers. Cook will provide advice related to driver reliability, and plans to collaborate on the formal verification part of the project. Cook has provided a letter of support for this proposal which is attached to this application.

References

[1] Sidney Amani, Peter Chubb, Alastair Donaldson, Alexander Legg, Leonid Ryzhyk, and Yanjin Zhu. Automatic verification of message-based device drivers. In *SSV*. EPTCS, 2012.

[2] Sidney Amani, Leonid Ryzhyk, Alastair F. Donaldson, Gernot Heiser, Alexander Legg, and Yanjin Zhu. Static analysis of device drivers: we can do better! In *APSys*, page 8. ACM, 2011.

[3] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Eurosys*, pages 73–85, Leuven, Belgium, April 2006.

[4] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *EuroSys*, pages 73–85. ACM, 2006.

[5] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, 2011.

[6] Y. Bertot, P. Castéran, G. Huet, and C. Paulin-Mohring. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[7] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: a verifier for GPU kernels. In *OOPSLA 2012*, 2012.

[8] Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. CAV, pages 140–156, 2009.

[9] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *CODES+ISSS*, pages 19–24, Newport Beach, CA, USA, 2003.

[10] Pavol Cerny, Krishnendu Chatterjee, Thomas Henzinger, Arjun Radhakrishna, and Rohit Singh. Quantitative synthesis for concurrent programs. In *CAV*, pages 243–259, 2011.

[11] Pavol Cerny, Thomas Henzinger, and Arjun Radhakrishna. Quantitative abstraction refinement. In *POPL*, 2013 (to appear).

[12] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI*, pages 304–315, 2008.

[13] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP*, pages 73–88, Lake Louise, Alta, Canada, October 2001.

[14] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[15] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, Ottawa, Ontario, Canada, 2006.

[16] Alastair Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl. Counterexample-guided abstraction refinement for symmetric concurrent programs. *Formal Methods in System Design*, 2012. DOI: 10.1007/s10703-012-0155-3.

[17] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of dma races using model checking and *k*-induction. *Formal Methods in System Design*, 39(1):83–113, 2011.

[18] M. Emmi, J. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL*, pages 291–296, 2007.

[19] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using

system-specific, programmer-written compiler extensions. In *OSDI*, pages 1–16, San Diego, CA, October 2000.

[20] Archana Ganapathi, Viji Ganapathi, and David Patterson. Windows XP kernel crash analysis. In *LISA*, pages 101–111, Washington, DC, USA, 2006.

[21] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *CAV*, pages 526–538, Copenhagen, Denmark, 2002.

[22] Intel Corporation. Device driver hardening and manageability, April 2011.

[23] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[24] James R. Larus and Galen C. Hunt. The singularity system. *Commun. ACM*, 53(8):72–79, 2010.

[25] Ben Leslie, Peter Chubb, Nicholas FitzRoy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting (Rita) Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, September 2005.

[26] Microsoft Corporation. Fault resilient drivers for Longhorn server, 2004.

[27] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 446–455. ACM, 2007.

[28] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[29] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: ten years later. In *ASPLOS*, pages 305–318, Newport Beach, CA, USA, 2011.

[30] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of Reactive(1) designs. In *VMCAI*, pages 364–380, January 2006.

[31] Arnd Poetzsch-Heffter and Marek Gawkowski. Towards proof generating compilers. *ENTCS*, 132(1):37–51, May 2005.

[32] Shaz Qadeer. Poirot - a concurrency sleuth. In Shengchao Qin and Zongyan Qiu, editors, *ICFEM*, volume 6991 of *Lecture Notes in Computer Science*, page 15. Springer, 2011.

[33] Martin Rinard. Credible compilers. Technical report, Cambridge, MA, USA, 1999.

[34] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Eurosys*, Nuremberg, Germany, April 2009.

[35] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with Termite. In *SOSP*, Big Sky, MT, USA, October 2009.

[36] Leonid Ryzhyk, John Keys, Balachandra Mirla, Arun Raghunath, Mona Vij, and Gernot Heiser. Improved device driver reliability through hardware verification reuse. In *ASPLOS*, Newport Beach, CA, USA, March 2011.

[37] A. Solar-Lezama, C. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI*, pages 136–148, 2008.

[38] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, pages 327–338, 2010.

[39] Wind River. Wind River Simics. http://www.windriver.com/products/simics/.

[40] Wind River. Wind River Simics Model Builder reference manual. version 4.4, September 2010.

[41] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. Model checking concurrent linux device drivers. In *ASE*, pages 501–504, 2007.

[42] Raj Yavatkar. Era of SoCs, presentation at the Intel Workshop on Device Driver Reliability, Modeling and Synthesis, March 2012.