DEVICE DRIVER SYNTHESIS

Contributors

Mona Vij
Intel Labs, Intel Corporation

John Keys
Intel Labs, Intel Corporation

Arun Raghunath
Intel Labs, Intel Corporation

Scott Hahn
Intel Labs, Intel Corporation

Vincent Zimmer
Software and Solutions Group, Intel Corporation

Leonid Ryzhyk
University of Toronto

Adam Walker
NICTA

Alexander Legg
NICTA

"Device drivers are the major cause of operating system failures"

Automatic Device Driver Synthesis is a research collaboration project between Intel and National Information Communications Technology Australia (NICTA) that aims to synthesize device drivers automatically using formal OS and device specifications. We have built a tool chain that uses Simics* DML Device model sources as an input to the driver synthesis tool chain. The tool chain has a frontend compiler that extracts the device behavior from the Device Modeling Language (DML) model and outputs a formal representation of the device behavior that we refer to as a device specification. The driver synthesis tool combines this specification with a similar O/S specification and applies the principles of game theory to compute a winning strategy on behalf of the driver and eventually converts it into driver C code. This approach aims to use the existing device models for producing device drivers resulting in highly reliable drivers and faster time to market. We have synthesized a number of drivers using our tool chain. Some examples include legacy IDE controller, UART, SDHCI controller, and a minimal Ethernet adapter.

Introduction

A device driver is the part of the operating system (OS) that is responsible for controlling an input/output (I/O) device. There is wealth of research[1][2] showing that drivers are a primary source of bugs, and driver development is a major bottleneck for platform validation and time to market. Figure 1 shows the conventional driver development process, where a driver writer uses two informal documents, OS and device specifications, to convert a series of OS requests to device commands. The process of device driver creation can be error prone and tedious. One of the main reasons is that the driver writer uses informal documents that are susceptible to misinterpretation. In addition, the driver writer has to have domain knowledge of both the OS and the device. In many cases driver writers also reuse existing driver code to write a new driver, inheriting any existing bugs in the process.

We propose to improve the driver development process by automatically synthesizing drivers from formal OS and device specifications, as shown in Figure 2. This is based on the fact that all the information needed to control a device from software is available during the design of the device. The idea is to represent this knowledge, so as to enable synthesizing driver automatically.

For device formal specification, we plan to leverage the high-level device models either written by hardware designers or for software simulation for virtual platforms. We are building a tool chain that applies the principles of

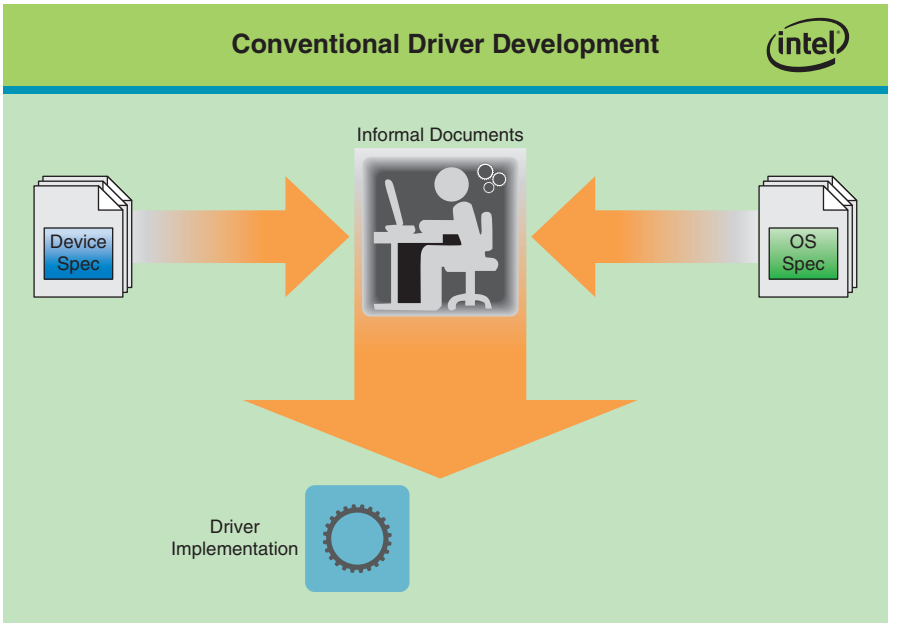

Figure 1: Conventional driver development
(Source: Intel Corporation, 2011)

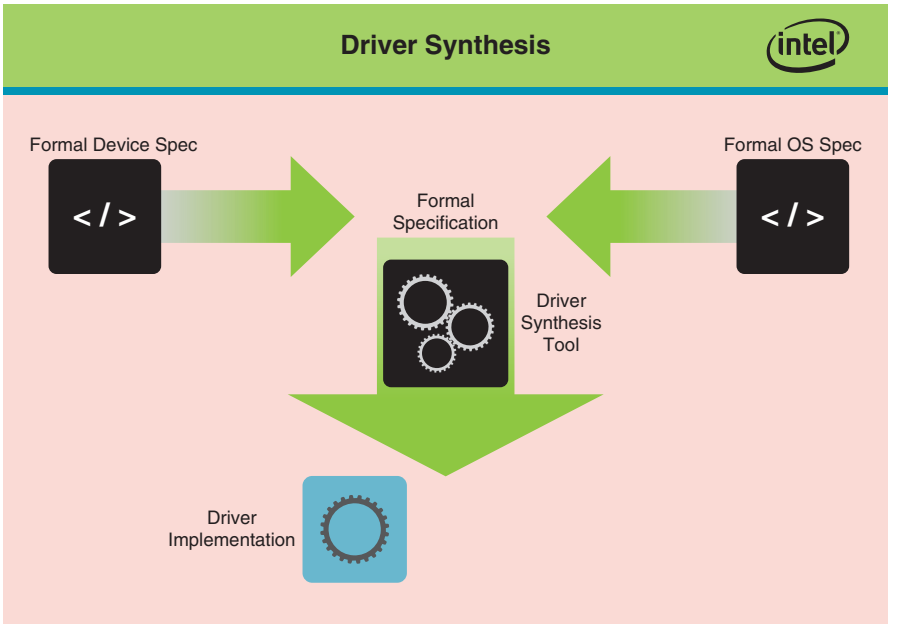

Figure 2: Driver synthesis
(Source: Intel Corporation, 2011)

"Driver synthesis from formal specifications"

"Game theory in driver development"

game theory and synthesizes the driver code from formal specifications. This approach improves driver reliability by reducing manual intervention, avoiding misinterpretation of device documents by driver writers. Moreover, given a device specification, drivers can be generated automatically for all supported operating systems, thereby eliminating the costs associated with porting drivers. With this approach of driver development, DML device models are used not only for simulation, but for driver generation as well. The driver synthesis tool chain also provides some additional capabilities like a state space explorer that aids in DML device model debugging. Overall this approach results in correct drivers and improves time to market by moving development earlier in design cycle, leading to cost reduction.

In the long run we plan to support large classes of devices with this tool, from very simple to complex devices, as long as their behavior can be represented as a state machine. We can't synthesize drivers that perform complex computation and are difficult to represent as a state machine. In addition, we don't plan to support drivers for devices that are based on programmable cores, such as high-end graphics or network processors.

High Level Architecture

Device driver synthesis aims to create device driver code automatically from hardware specifications of a device. Figure 3 shows various components in the driver synthesis tool chain that begins with formal specifications and converts it to various intermediate forms before finally emitting the device driver code. We formalize the driver synthesis problem as a game between the driver and its environment, which consists of the device, additional device interfaces (for example, network) and the operating system. The formal specification of the device and OS interface, together, define the "rules" of a two-player zero-sum

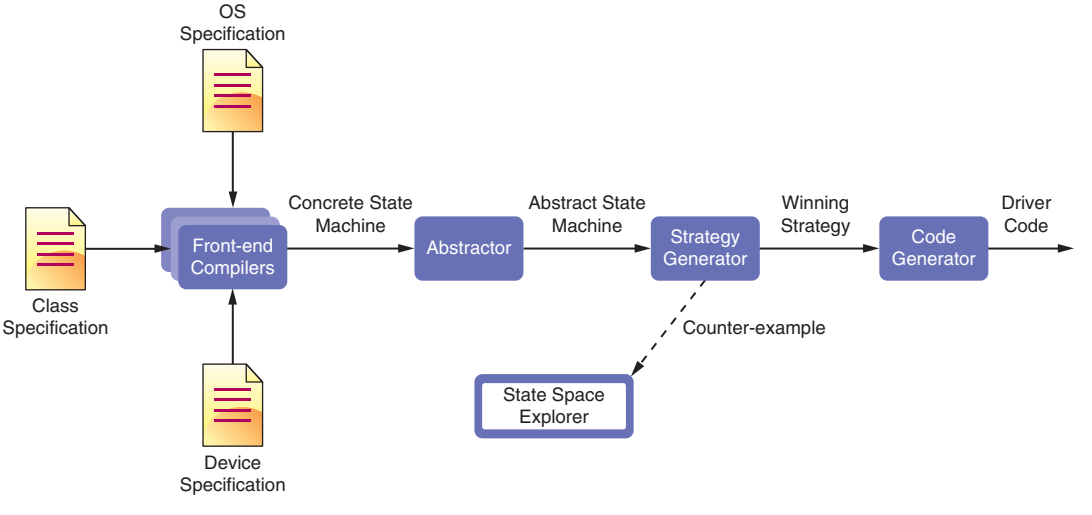

Figure 3: Driver synthesis tool chain
(Source: Intel Corporation, 2013)

game. The driver assumes the role of the first player and the environment (OS, media, and so on) describe the moves of the "opponent." In the context of the game, modeling the environment as an "opponent" puts more emphasis on the environmental events that lead to failure than those that are benign. The environment begins all games with moves that represent OS-to-driver requests. In response to these moves, the driver must try and make "moves" (that is, send commands to the device) to push the device to a winning state, corresponding to a correct device response for the given OS request. The moves chosen by the driver should be such that no matter what external event occurs, the device and driver can either correctly service the OS request or fail gracefully and continue to operate correctly in the future. Effectively the tool constructs a driver algorithm that guarantees that the driver is able to correctly satisfy all OS requests given any feasible driver behavior. We call such an algorithm a *winning strategy* on behalf of the driver.

Tool Inputs

The tool takes multiple formal specifications as input, as described in the following subsections.

Device Class Specification

The Device Class specification models states, events, and functions common to all devices of a given class in an OS-independent and device-implementation–independent manner. The specification describes events that represent interactions between the device and its environment (that is, connected media, external devices, and so on). Events may also represent completion of individual device requests such as setting a configuration. The states describe logical device states applicable to devices of class, such as configured states, initial state, and so on. In addition, the specification may describe sub-states that a device is expected to transition through in order to complete a device function. In addition, the specification also defines all constant values given to or received from devices of class, such as baud rates, configuration values, and I/O signals.

"Device class specification models states, events and functions common to all devices in a class"

Device Class specifications need only be written once per device class and can be used with different OS specifications and devices of the same class from different vendors. We believe a model similar to USB's Device Working Group (DWG) would work best for establishing industry-wide device class specifications. In this model, classes of devices are identified and a working group (WG) is established for each class, drawing WG membership from interested parties who tend to be the leaders and experts in a specific device class. The WG then develops a class specification by consensus, with the result typically being subject to approval of the parent organization.

OS Interface Specification

The OS Interface specification describes legal sequences of interactions between the driver and the OS as well as the expected device response on completion of each OS request. It models when events defined in the device-class specification must be raised in response to OS requests. This specification does not specify how the events in the device-class specification are generated, since that should

be part of the device specification. It is up to the synthesis algorithm to derive the necessary steps for generating these events in response to OS requests.

Ideally, the OS specification for a specific OS will be produced by the entity that produces the OS. This specification needs to be written once per OS per device class and when a new OS release occurs; minimal change should be required to adapt the specification.

Device Specification

Device specifications are device-specific instantiations of device class specifications. They model the device behavior and the externally visible artifacts of the device. In particular, they model externally visible registers and device operations that result from the reading or writing of said registers. The device response depends on the register values and device internal state, such as, for example, whether the device is initialized or waiting for a request to complete. These responses include but are not limited to updating register values, generating interrupts, triggering one or more external events, and interactions with other subsystems. These specifications are written at a high level of abstraction and ignore detailed internal architecture and timing.

Individual device specifications must be produced by the device vendor. In the case of industry-standard devices such as EHCI and XHCI (USB) and SDHCI (MMC/SDIO), a single device specification can be produced by the entity responsible for the standard and used for any device that meets the standard. In the case where a device is industry standard but also contains vendor-specific extensions, the device vendor becomes the responsible party. The vendor can import the industry-standard specification to specify device core functionality, but still remains responsible for specifying the vendor extensions.

Tool Outputs

The tool processes the input specifications and applies the principles of game theory to produce driver code.

"The tool produces "C" code as output if it finds a successful strategy"

Driver Code

The tool produces C code when it finds a successful strategy. In some cases driver writers will need to develop manual wrappers to integrate the code with the OS.

No single entity can be identified as the entity responsible for producing device driver binaries. Industry history suggests three potential sources: OS vendor, hardware vendor, and platform integrator. OS vendors generate large numbers of device drivers, tied to OS release cycles. Hardware vendors produce drivers when 1) the target OS vendor does not support the device (in particular for new hardware), and 2) when the need for the driver falls between OS release cycles. Platform integrators generate device drivers when the driver is not provided by the OS vendor or the device vendor, or they built the device themselves.

Table 1 illustrates the interdependence between the three entities

Entity	Produces	Consumes
Device Class WG	Device Class Specification	n/a
OS Vendor	OS Specification	Device Class Specification
Device Vendor	Device Specification	Device Class Specification
Platform Integrator	n/a	Device Class Specification OS Specification Device Specification

Table 1: Specification Producers and Consumers
(Source: Intel Corporation, 2013)

DML Models for Driver Synthesis

Device Driver synthesis aims to synthesize drivers automatically from formal specifications, so availability of a device specification is a key to success of the tool. If a device specification has to be created specifically for synthesis, then we've only accomplished the shifting of efforts from driver development to specification development, rather than solving the problem. In addition there is no way to validate the manually developed model to make sure that it models the device operation properly.

"Availability of a device model is key to the success of the tool"

There are many high-level device specification languages that are currently used by hardware manufacturers including SystemC, System Verilog, and Simics DML. To ensure that the driver synthesis tools are widely applicable, the architecture provides for multiple frontend compilers that convert specifications written in a given language into an intermediate language Termite Specification Language (TSL) developed by us. TSL provides a means for concise description of FSM states and transitions and is used as the FSM external representation by all other tool-chain components.

Wind River Simics* is becoming the platform of choice for virtual platforms at Intel. Many DML models already exist and are being used successfully in virtual platforms. If a particular DML model doesn't exist, then writing the model contributes to synthesis as well as virtual platforms. We have developed a frontend compiler for DML for using DML models with our tool chain.

DML to TSL Compiler

DML has been designed to facilitate fast model development by software engineers. It is a very forgiving language in general, allowing forward referencing, type casting, and automatic C-style type promotion. TSL, on the other hand, is very restrictive. For example, it does not provide type promotion or casting or allow forward references.

"DML compiler extracts the relevant device behavior from DML device models."

One of the goals of the project is to not modify the actual device models, since we do not want our use of the models to impact their original use in virtual platforms and we do not want to force a fork of the models, which might lead to issues with bug-fix propagation. We have built a DML compiler that tries to deal with the DML to TSL conversion automatically, but in some cases we do need to modify the model. Currently we do modify the model directly, but all of the modifications we currently make to the actual model could instead be kept in a separate annotations file, thereby leaving the model pristine. This support will be added in the future versions of the tool.

Extracting Device Behavior from DML Models

Conceptually, DML architecture is very similar to event-driven GUI architectures. A DML model can be thought of as a collection of responses, where each response corresponds to a message or a set of inputs. Responses execute instantaneously; that is, simulation time does not advance while an individual response is executing, and blocking in a handler is prohibited. Response execution always begins with an external call of an interface method and completes with the return to the external caller.

TSL models express device behaviors as a collection of variables that represent device state and a collection of transitions to these state variables. Given a set of input state changes, each individual transition describes the cascade of changes to other state variables in response to the input changes. In addition, each transition may have guarding constraints that allow it to be enabled or disabled depending on current device state. Similarly to DML, TSL transitions are also instantaneous. While they resemble code, a TSL transition can also be thought of as a formula that computes next state S' given current state S and inputs I: $S' = f\text{Trans}(S, I)$.

Conceptually, DML model structure closely corresponds to the TSL structure. A single TSL transition maps directly to an execution trace of a DML interface method and its called methods. The TSL state variables map directly to the collection of DML registers, fields, attribute objects, and data objects.

Before we can begin extraction, we build out an in-memory representation of the model. This involves application of templates to DML objects, evaluation of parameters, expansion of *select* and *foreach* keywords, and evaluation and pruning/expansion of *if object* statements. Each of these steps can result in significant model changes so evaluation of the model really cannot be performed without these steps.

We begin the extraction process by collecting the model variables that will become the TSL state variables. All data objects and attributes are added to the collection as they are encountered. Fields are added only if their *alloc* parameter is *true* (that is, model space is allocated for its contents). Registers are added only if they do not contain fields and their *alloc* parameter is *true*.

We identify the individual transitions to be extracted (transition entry points) by identifying and collecting all exported interface methods contained in the models. As well as explicit interfaces, this set also contains the read/write bank

access methods for all register banks present. We also add transitions for each DML *event* object and *after* keyword encountered in the model, along with a 1-bit guard variable for each event or after transition.

After identifying the entry points, we can begin extraction of the transitions. This is done by first copying the method containing the entry point, then replacing each *call* or *inline* statement with the body of the target method. This is repeated recursively until no *call* or *inline* statements remain and we are left with a full code trace through all branches of the call. As an optimization, we concurrently evaluate *if* statement conditions to prune branches that will never be taken because they will always be false.

Besides state variables, TSL allows for temporary variables. These are global in scope but do not retain values across transitions. TSL has no notion of transition-local variables. As part of the transition extraction, we must convert all local variables found in DML methods to TSL temporary variables. Because of TSL's global scoping, some amount of variable name mangling is required to ensure unique variable names.

TSL restricts transitions from modifying a variable more than once per transition. This requires us to analyze each extracted transition and introduce new temporary variables and assignments when violations are identified.

TSL also requires that any single transition must update all state variables. To meet this requirement, we analyze each branch in the transition for assignment statements. For each variable assigned, we add an identity assignment (state' = state;) to the corresponding branch. We complete this requirement by adding identity assignments to the end of the transition for all remaining unassigned state variables.

The following subsections describe how our frontend DML compiler deals with the conversion from DML to TSL.

DML Templates

Development of the compiler caused us to study several of the import files in great detail, specifically dml-builtins.dml and utility.dml, leading us to realize the power of well-planned template and parameter use. This in turn allowed us to write "extensions" in DML itself, rather than extending the language.

"The tool converts models into an intermediate representation called TSL that is amenable for analysis and synthesis"

The file dml-builtins.dml provides the glue that ties banks, registers, and fields together, as well as providing default methods and parameters for most types of DML objects. Unfortunately, it is so closely tied to the Simics DML compiler, dmlc, that we could not use it without porting it. Our first porting task was to create our own versions of the methods that are "intercepted by the DML compiler." These methods are involved in the read/write access fan-out from bank objects to registers and fields.

For Simics device I/O, the bank method access() serves as the primary entry point for the I/O-memory interface (register read/write operations). Instead of a single method that takes direction and size as parameters, TSL uses a set

"Built-in templates for register access provide a software interface to the device internals with appropriate constraints"

of entry points: read8(), write8(), read16(), write16(), read32(), and write32(). To accomplish this change, we modified the behavior of our bank objects to create parameters containing lists of mapped registers of specific sizes: mapped_regs8, mapped_regs16, and mapped_regs32. We also defined an iioregion interface with methods corresponding to the TSL requirements and modified the default "bank" template in our dml-builtins.dml file to implement the iioregion interface and instantiate the individual access methods as applicable. In addition, we added the ability to turn off access for banks we were not interested in. For instance, we may be working with a PCI-based UART where we are interested in the UART register banks but not the PCI configuration space register banks. This control allows us to extract UART register-related transitions while ignoring PCI-configuration related ones.

Early on, we discovered that our game-playing solver did not always follow the rules that driver writers do. Specifically, it would attempt device register access before the driver's probe() routine had been called. To solve this issue, we added a guarding constraint to the access methods, blocking them until probe() had been called. The following is a portion of our dml-builtins file illustrating these changes:

```
// io_waits_for_probe – define to block IOs before probe() is called
parameter io_waits_for_probe  default undefined;
// conditionally create a variable to track if probe() has been called
if (defined $dev.io_waits_for_probe) {
  data uint1 probe_called;
}

template bank {
        .
        .
        .
        // extensions for tsl
        parameter mapped_regs32          default undefined;
        parameter mapped_regs16          default undefined;
        parameter mapped_regs8           default undefined;

        // controls if bank-related transitions will be emitted
        parameter emit_accessors         default true;

        if ($this.emit_accessors == true) {

            // not emitted if bank has no visible registers
            if (defined $this.mapped_registers)  {

                // The TSL access interface
                implement iioregion {
                    // Does bank contain mapped 8-bit registers?
                    if (defined $parent.mapped_regs8)  {
```

```
// emit guard if we need to wait for probe
if (defined $dev.io_waits_for_probe) {
  parameter guard_read8 = ($dev.probe_called == 1);
}
// and emit the read access method
method read8(uint32 roffs8 ) -> (uint1 rstatus, uint8 rval8)  {
```

"Event handling is challenging in TSL, as there are no queues. "

Event objects presented another challenge. In Simics, execution of an event object's event() method is constrained by its posted state. It can only be called if it has previously been posted to an event queue. In TSL, no such queues exist. This is further compounded by the almost 100-percent rate of models overloading the default event() method. We needed to constrain the event() method to only run when posted, and we needed to retain control of the event's entry point so we could apply the constraint and perform constraint housekeeping. Again, we were able to perform the bulk of this work by modifying the default event template:

```
template event {
    .
    .
    .
    // variable to track posted state
    data uint1 _posted_;

    // methods to manipulate posted state
    method post(when, data) {$this._posted_ = 1;}
    method remove(data) {$this._posted_ = 0;}
    method _cancel_all() {$this._posted_ = 0;}

    // instantiate an event "wrapper" entrypoint
    implement event_entry  {
        //entry point only enabled when event is posted
        parameter guard_pre_event = ($_posted_ != 0);

        method pre_event(void *param)  {
            // housekeeping – reset posted state
            $_posted_ = 0; //Clear posted flag and call event
            // call control to real handler
            inline $parent.event(param);
        }
```

Unused Code

There is some code in DML device models that is for DML infrastructure and not for device operations. Our tool has no need for such code and we needed a way to eliminate such code from models without modifying the models. We have defined a few annotations for use in the models. They all begin with the sequence *//@* and so are transparent to the Simics DML compiler. We use the pair *//@ignore* and *//@resume* to hide portions of DML from our DML tool.

"The TSL compiler performs strict type checking requiring the DML compiler to coalesce types by rewriting expressions in the emitted TSL"

We have used these to some extent in the models but mostly use them in our copies of the system import files, the DML equivalent of user/include/*.h.

Width Conversion

TSL does not support type promotion or casting, so our DML compiler performs a significant amount of expression rewriting in order to provide explicit width conversions. Width conversion to a wider type requires the original assignment be converted to a conjunction of two assignments, the original assignment and a second assignment to the extra bits. For example, assuming a 32-bit variable named *foo* and a 16-bit variable name *bar*, the statement:

```
foo = bar;
```

becomes:

```
((foo[15:0] = bar) && (foo[31:16] = 0))
```

In some cases, the format of a DML expression may prevent our tool from being able to make this modification. For instance, the DML expression:

```
foo = (somevar == 0) ? bar :0;
```

cannot be modified because the conversion is only needed conditionally but can only be expressed in terms of the global *foo*, not the conditional *bar*. In these cases, we rewrite the DML in a form that allows for the conversion:

```
if (somevar == 0)
        foo =  bar;
else
        foo = 0;
```

This rewriting provides separate conditional assignments to *foo*, allowing each to be converted as needed.

Arithmetic Operations

Current version of TSL does not support arithmetic operations (such as +, −, ×, ÷, or modulo) or magnitude comparison operations (such as <, <=, >, or > =). At this point this is just a limitation of our tool and we plan to add this support in our tool soon. For dealing with this issue for now, our tool detects cases where power-of-2 techniques can be used instead and performs automatic conversion. The detection depends on one of the operands being a constant power-of-2 value. In cases where this is not obvious, we have to modify the model by hand.

Some models contain complex arithmetic expressions that calculate some binning value based on one or more inputs. In these cases, we have replaced the

arithmetic expressions with if-else trees or switch statements coded to achieve the same result without arithmetic.

Driver Verification Using DML models

We use the same Simics model that is used to synthesize the driver in the Simics framework to execute and test the synthesized driver.

For some of the devices for which the hardware is available, we also tested the driver on actual hardware.

Tool Chain Capabilities

The synthesis tool chain has some additional capabilities that can be useful to a DML model writer. In the following sections we describe these capabilities and how a DML model writer can use it to their advantage.

State Space Explorer

The driver synthesis tool chain includes a utility that allows a user to visually inspect the combined device and OS state machine. The utility is a state space explorer, a graphical user interface that allows the user to perform various operations on the state machine, like analyzing available driver actions in a given device state, applying an action from the current state and inspecting the changes to the device state, and viewing the effect of external environment events.

"State space explorer allows the model developer to visualize the device model"

While the state space explorer is a critical component of a tool chain that synthesizes driver code, it also offers capabilities that can be quite useful to a DML model developer.

Visual Model Debugger

As illustrated in Figure 4, the state space explorer GUI allows a DML model developer to visualize the device model as a directed graph where each node in the graph represents a state (or a set of states) and each arc in the graph represents a transition from one state to another.

The GUI allows a model user to inspect the values of any device internal variable in a given state by simply clicking on the node in the graph representing the state. A pane on the left lists all the device internal variables, and clicking on a particular state node causes this list to be updated with the values of each variable in that state.

Further, from a given state, the GUI allows a user to pick the next transition which would move the device state machine to another state. While this feature is somewhat similar to the *step* or *next* operation in a traditional software debugger, the event-driven nature of a DML model requires the tool to provide more flexibility. The events triggering state transitions are broadly classified into events that can be controlled by software and those that depend on the environment (like platform hardware interrupt, line unplugged, and so on) and therefore cannot be controlled by the device or software. The tool allows a user to choose which event occurs next in a

given device state. The choice includes both controllable and uncontrollable events. In the case of software-controlled actions, the user can also specify the parameters of the action.

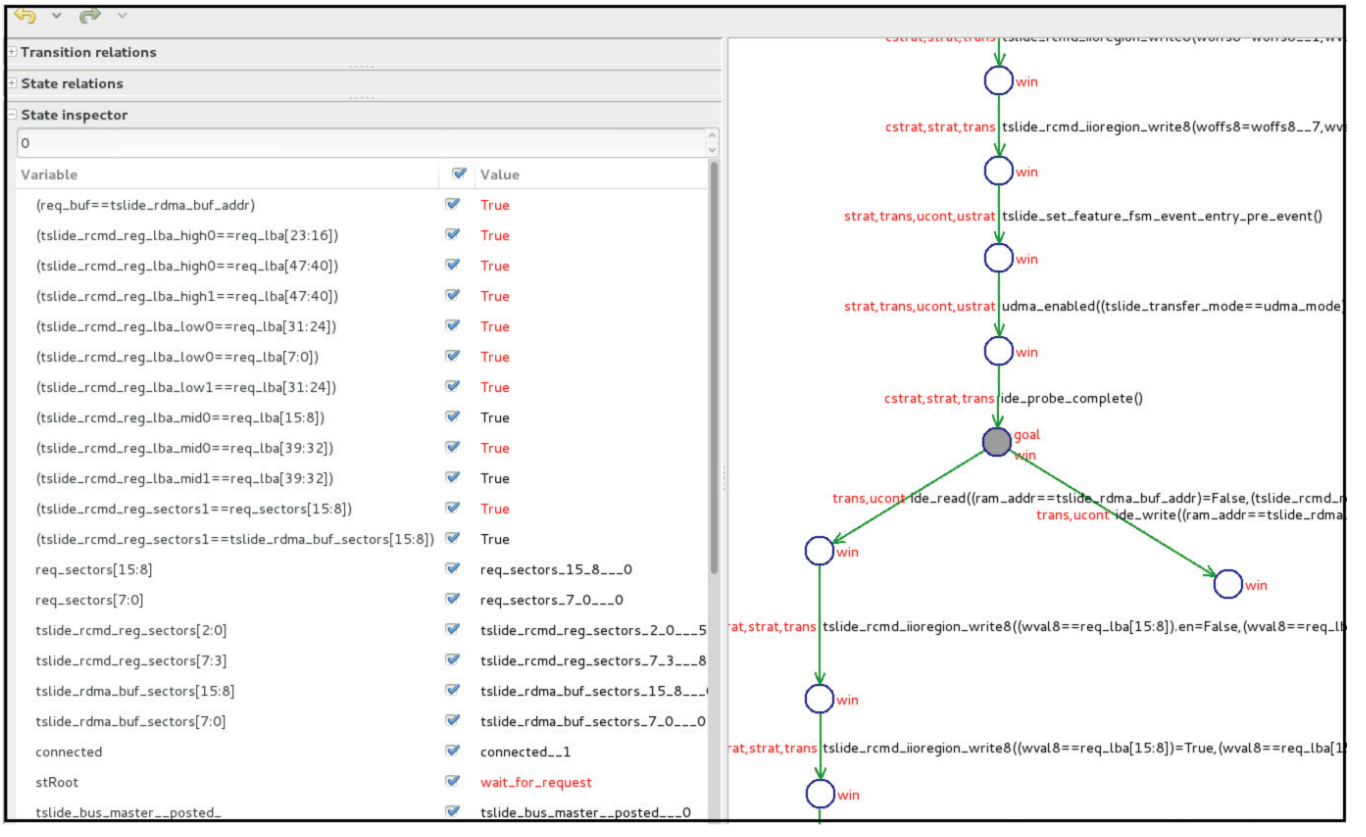

Figure 4: State space explorer GUI. The right pane shows the device model as a directed graph. The left pane shows device internal variable values.
(Source: Intel Corporation, 2013)

"State space explorer provides a counter example when no winning driver strategy exists"

The capabilities described above (inspecting device variable values and directing the state machine by choosing the next transition via the GUI) allow the model writer to use the state space explorer as a debugging aid, examining the effect of (a chain of) events on the device.

Counterexample Generation

The primary challenge in exploring the state space of a hardware device model is its huge size, which would quickly make visualization incomprehensible and state management cumbersome. The GUI explorer utility in the synthesis tool chain employs numerous techniques, built on a foundation of formal methods and symbolic execution to address this issue. These techniques include:

- aggregating states with the same properties with respect to the DML mode code into a set of states and displaying the entire set as one node
- symbolic representation of the model code, which allows abstracting the model variables (which can have a massive number of values)

into Boolean predicates that distinguish specific paths through the code

- showing only relevant subset of actions and parameter values when adding a state transition
- automatically "running" (tracing out a path in the device state machine) till a specified "way-point condition" (a predicate expressed over device model variables) is true

One of the most useful capabilities from a model developer's perspective is the tool's ability to generate counterexamples. The normal operating mode is to develop a successful strategy for the driver, but when the model is buggy such that it is impossible to generate a successful driver strategy, the tool generates a counterexample, that is, a set of actions on the state machine demonstrating how the driver can be prevented from moving the state machine into a desired goal state. This is possible since the tool is built on top of a formal representation of the model.

Providing counterexamples is very useful to a model developer as they can be presented with a specific sequence of actions on the device model that would lead the model into an undesirable state.

Scenario Replication

"Visual tool allows for scenario replication by supporting save and restore feature"

Device programming sequences typically involve massaging of OS input parameters, a long series of register reads/writes, and require specific environment conditions (such as network connectivity for a successful packet transmission) to hold. In order to assist the tool user in efficiently exploring the device state space and quickly repeating long repetitive action sequences, the GUI allows saving traces of action sequences, also known as state transitions, from any given state. In any subsequent run of the tool, as long as the model remains unmodified, the same scenario can be replicated by bringing the model to the same start state and then loading the trace saved.

This capability can be very useful for software-hardware co-development allowing device-driver and device-model developers to work together closely. The driver developer can initiate some OS-based scenario and capture its effect on the device model internals for the model developer to replicate. Typically such errors (for example, race conditions, synchronization errors, or deadlocks) involve very specific interactions of the software, device, environment, and OS actions, making it hard for model developers to replicate the exact error conditions being encountered in a complete system. While Simics does allow easily simulating the complete system state to replicate errors, the model developer would still need to instrument the DML model code with appropriate debug logic (typically log messages, to determine the root cause of the problem). The distinction is similar to classic software debugging done by adding code to print debug messages versus using a debugger to find problems.

The combination of the capabilities to explore model state space, counterexample generation and scenario replication allows a DML model writer to quickly narrow the search for bugs in DML device models as they are directly able to examine the device-internal state in the discovered failure paths.

Prototype Device Drivers

We have successfully synthesized device drivers for multiple nontrivial devices using DML device models. We used some existing models and developed some from scratch. For all the drivers the synthesized code was limited to driver code that handles device specific operations like initialization, configuration, and data transfer. We embedded this synthesized driver code in manually developed wrappers for code that involves OS and bus resource allocation and any data transformation. Resource allocation includes allocating IRQ lines, setting up DMA descriptor rings, creating mappings for memory-mapped device regions, and so on. Data transformations performed by drivers include preprocessing data buffers sent to the device, such as, for example, changing their alignment or padding, and postprocessing data received from the device, such as extracting checksum from a network packet. While many of these operations can in principle be formalized and synthesized using the game-based approach, we believe that a different formalism is needed to automate synthesis of this functionality. We successfully synthesized low-level drivers for the following devices:

- Legacy IDE Controller –Linux driver from manually developed DML model from device datasheet
- W5100 Embedded Ethernet Controller – Native firmware driver from manually developed DML model from device datasheet
- Intel PRO/1000 Ethernet Controller – Linux driver from manually developed DML model from device datasheet
- UART NS16450 – Linux driver from existing DML device model
- SD Host controller – EFI driver from existing DML model

"The synthesis tool has been used to successfully generate device drivers for several non-trivial devices"

SD Host Controller Case Study

This section describes the steps involved in synthesizing a UEFI SD Host controller driver from scratch. This case study is considered in detail here because it is based on using a preexisting device model. As such, it is the most representative of the intended use of this technology.

Input Specifications

Driver synthesis requires three input specifications for the device. This section describes the steps involved in acquiring/developing three input specifications.

Device Specification

We used an existing SD host controller DML device model from Simics team as our device specification. As we began to examine the model to determine where the device-class related annotations should be placed, we noticed that unlike the other DML models we had worked with, this model did not account for in-flight data transfer times. All data transfers to or from the card model happened instantaneously. Our past experience led

us to believe that we would not be able to successfully synthesize a driver from a model in this condition. The problem is that the instantaneous completion leads the synthesis algorithm to assume that any operation started in cycle *x* will be complete in cycle (*x* + 1), eliminating the need to poll status registers for an indication of completion, and so on. Therefore, our first step became a rewrite step.

We rewrote the model to account for the in-flight times and validated the changes using a stock Linux image with the Linux SD Host driver, running on the Simics Framework. We submitted patches for these changes to Simics.

We then began the task of annotating the model with Device Class events and attempting synthesis. As this model was the most complex model we had tried to date, we immediately ran into problems. The complexity of the model resulted in an output TSL file with 6.8 Kb of state space (global variables), another 12.3 Kb for temporary variables, and 45 separate transitions. This extreme size resulted in tool-chain execution times in excess of 4 hours. As we were still trying to determine the correct locations for annotations, the extreme execution time was a significant hindrance to forward progress.

Since the model is a full model, it contains transfer modes and registers that would not be used in our project. In an attempt to reduce the overall size and complexity, we tweaked the model to hide the unused transfer modes and registers. This reduced model has 2.5 Kb of global variable space, 1.5 Kb of temporary variable space, and 14 separate transitions. This reduced tool-chain execution time to tens of minutes.

We also had to make a few changes to the model for TSL compatibility issues. These changes included rewriting arrayed register definitions without arrays, statement adjustments to allow width conversions, and elimination of arithmetic operations.

Class Specification

We needed to define this specification from scratch as it does not exist today. Normally we expect it to be published with the device industry standard specification. This specification defines all the interfaces supported by the SD Host controller device that are expected to be supported by all the drivers. We started with SD host controller standard specification[6] and defined the class interfaces. This is defined as a Word document. The class interfaces are the points of synchronization between OS and device specifications. We will use these interfaces to annotate both the OS and device specifications.

OS Specification

We chose to synthesize the SD host controller driver for UEFI (Unified Extended Firmware Interface). We used UEFI documentation[5] to define this specification. The SD host controller driver is the lowest level driver in the layered driver stack. The OS specification for this driver was motivated by the interfaces expected by the media layer driver.

"The SD host controller DML model was annotated to work with the tool"

UEFI defines a stylized model of system booting that includes interfaces between several different executable entities, including UEFI drivers, as shown in Figure 5.

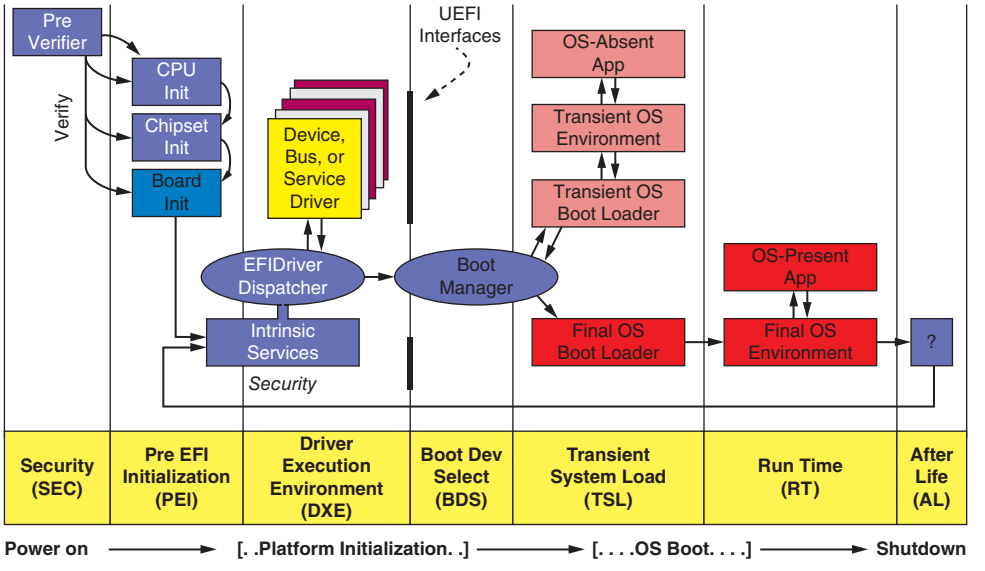

Figure 5: UEFI boot sequence
(Source: Intel Corporation, 2013)

These interfaces are codified by the main UEFI specification and expose abstractions such as block device access, such as the EFI_BLOCK_IO_PROTOCOL. The generic services in the EFI_BLOCK_IO_PROTOCOL, such as ReadBlocks(), WriteBlocks(), and Reset(), need to be refined to an implementation that meets the requirements of the underlying hardware controllers. Today the requirements of the UEFI specification and its associated driver model, along with the semantics of the hardware, are all managed by the developer as part of the code creation process. This process is error fraught, and most developers typically take an existing driver source and "port" it to the requirements of the new hardware. As such, there is no guarantee of correctness, with flawed "existing sources" being evolved via this porting process.

"Strong assurance guarantees needed for firmware along with the extensive specifications available in UEFI make EFI drivers an ideal target for synthesis"

Instead, with the driver synthesis, a single instance of an OS specification for a class of devices can be married to a specific device specification, such as the DML for the hardware, to derive the source. This removes the errant human interpretation of the UEFI specification and the hardware host controller interface definition.

This is an important issue in that the UEFI firmware on the system board is considered hardware by many end users of the platform. And with the trust guarantees around platforms based upon UEFI Secure Boot[7], assurance considerations, such as correctness of the implementation, gain even more importance as all of the UEFI drivers and components are in the same trusted computing base.

Specification Synchronization

We used the class specification as synchronization between the OS and device specification. This involved using the class interfaces in the OS specification at the synchronization points. Finding the correct synchronization points involved studying the DML device model. Finding the correct place to annotate the device model depends on the way the model is written. It was a fairly simple process to annotate the SD host controller and EFI OS specifications.

Integration

Once we had the three inputs ready, it was an iterative process to input them through our tool chain to synthesize the driver. We did not synthesize the configuration interfaces for this device, but synthesized the main function to send a command to the card. At the end of this step we were able to synthesize the device driver strategy for this driver.

Code Generation

Code generation proved much mode tedious than anticipated. At the time of writing, our synthesis tool does not support fully automatic code generation. Instead, it allows the user to interactively construct driver source code by selecting one of several possible actions proposed by the winning strategy in each state. Ongoing research on this problem is focusing on techniques for fully automatic code generation as well as on improved methods for interactive user-guided code generation (see the section "User-Guided Synthesis").

"The synthesized code generated by the tool was tested in the Simics simulator with an Intel® Core™ i7 based platform model"

Testing and Validation

We used the Simics simulator of a target platform based on the Intel® Core™ i7 processor for testing this EFI driver. This model does not contain an integrated SD host controller so our first step involved adding our SDHCI device model to the platform. We created a Python wrapper to instantiate our SDHCI model and Simics MMC Card model and integrated the wrapper into platform model startup script. The startup script was modified to connect the host controller to the platform model through an unused South Bridge PCI bus slot.

With the platform model extended, the next step was to validate the extended model. This was done using the Linux image supplied with the platform model. We booted the image in Simics and recompiled the kernel to create a loadable Linux SDHCI driver. We updated the Linux image to retain the new driver modules. We were then able to load the SDHCI driver and validate our SDHCI-MMC card model combination using Linux file-system commands targeted to the MMC card.

Our next step was to establish an EFI baseline image. To achieve this goal, we built an EFI image with an existing SD host controller driver and tested that simulation environment. We then integrated our driver with the EFI code base, replacing the existing driver. We needed to develop some wrapper code to integrate in EFI environment. We then built and tested this driver on the Simics simulator and successfully brought up the SD host controller and performed read/write operations to the SD card.

"User guided synthesis allows a driver writer to have fine grained control over the driver synthesis process"

User-Guided Synthesis

Our initial approach with this project was complete automatic synthesis, where once the specifications are available, a push-button approach will result in a driver. In practice we realized that users want much more control over the structure of the driver code. In addition, in some cases synthesis gets stuck, and having users provide some simple hints can make the job of the synthesis tool much easier. Given these findings we decided to make a shift toward user-guided synthesis, as illustrated in Figure 6.

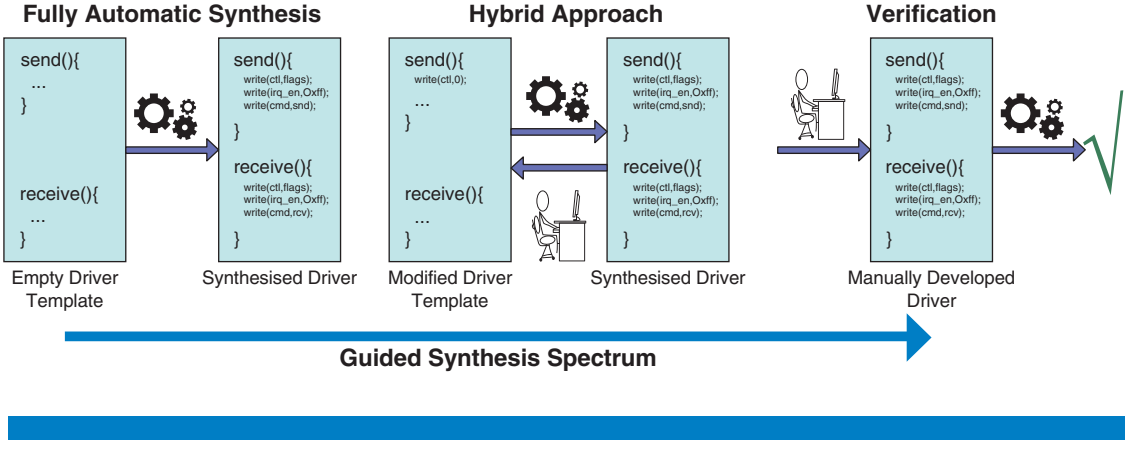

Figure 6: Guided synthesis spectrum (Source: NICTA, 2013)

To this end we plan on using driver templates that specify the driver structure. The user can add additional constraints on the synthesized driver by defining a device-specific driver template that can include some hints, or anything that is specific to a device. We plan on supporting a complete spectrum from fully automatic synthesis, where the device-specific template is empty, to the other extreme, where the user manually writes the complete driver in device-specific template and our tool can then act as a verifier to verify the driver against input specifications. We think the sweet spot is somewhere in the middle, where the user specifies some code structure and constraints in the device-specific template and generates more usable and readable code (see Figure 7).

We are also working on an interactive code generation GUI that gives user the flexibility to add any code at code generation steps. Any code manually added this way using the code generation GUI is saved by adding it back to the template and will automatically be available at the next iteration. Using a combination of templates and code generation GUI, our tool chain will provide user control over generated code at all stages of synthesis. Even though the user gets complete control, our tool chain will validate that the user has added correct code. Any errors caused by the user will result in synthesis failure and not an incorrect driver.

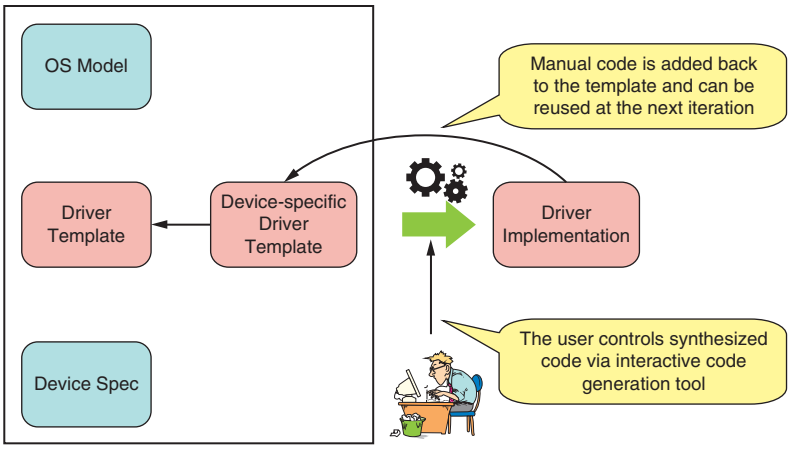

Figure 7: User-Guided synthesis with templates (Source: NICTA, 2013)

Future and Summary

Using existing device models for driver synthesis is a great start, but in practice we realized that we had to modify and annotate the models extensively in order to make them suitable for synthesis. In the future we hope to work with model writers to lay down requirements for writing device models with synthesis in mind, so as to reduce manual intervention to annotate or modify the models.

"Synthesis with user guidance has a potential to achieve the holy grail of fine grained user control with formal guarantees of correctness for generated device drivers"

Complete References

[1] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *18th SOSP*, pages 73–88, Lake Louise, Alta, Canada, Oct 2001.

[2] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *20th LISA*, pages 101–111, Washington, DC, USA, 2006.

[3] N. Piterman and A. Pnueli, "Synthesis of reactive designs," in *Proceedings of Verification, Model Checking, and Abstract Implementation (VMCAI)*, 2006.

[4] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers,* Vols. C-35, no. 8, pp. 667–691, August 1986.

[5] Unified EFI Forum, [Online]. Available: www.uefi.org/home.

[6] SD Association, "SD Host Controller Standard Specification Version 4.00," SD Association, 2012.

[7] Magnus Nystrom, Martin Nicholes, Vincent Zimmer, "UEFI Networking and Pre-OS Security," in *Intel Technology Journal - UEFI Today: Boostrapping the Continuum,* Volume 15, Issue 1, pp. 80–101, October 2011

Author Biographies

Mona Vij is a researcher in Intel Labs. She has been a security and operating systems researcher for over 20 years. She has a Masters in Computer Science from the University of Delhi, India and a Bachelor of Science in Mathematics from St Stephen's College, Delhi.

John Keys is a Staff Engineer in Intel Labs. He has been developing low-level software for over 25 years, for both PCs and embedded platforms. He has experience with a wide range of hardware devices, CPUs, operating systems, processor architectures, and platforms from bare-metal to PC to satellites and tunnel boring machines. He has made significant contributions to the development of PCMCIA and USB technologies and standards. Through this leading edge work, he also became an expert in "hacking" an existing platform to add new capabilities, beginning with plug-and-play support for MS-DOS3.2. John has been with Intel for 14 years in a variety of positions. Prior to joining Intel, he was the VP of Software for MCCI in Ithaca, NY.

Arun Raghunath is a Senior Software Engineer in Intel Labs. He has a Masters in Computer Science from University of Southern California, and a Bachelors in Computer Science & Engineering from Pune University, India.

He has been a Systems software researcher at Intel for the last 14 years. He has authored 5 conference papers, 1 book chapter and holds 8 patents in the areas of High performance computer networking, Operating Systems, Compilers and multi-core parallelization.

Scott Hahn is a Principal Engineer in the Systems Architecture Lab within Intel Labs where he leads the Operating Systems Research team. His team primarily focuses on the interaction of system SW and HW. Their projects cover multiple areas including storage, scheduling, memory and device drivers. Scott has been with Intel since 1994 and joined Intel Labs in 2006. Prior to joining Intel Labs, he was an architect in the LAN Access Division (LAD) where he worked on a number of network technologies and was the lead architect of Intel's Active Management Technology (Intel® AMT). Scott also worked in Intel's Supercomputer Systems Division where he was responsible for developing Intel's IP over ATM solution for the world's first TeraFLOP super computer. Scott has published over 15 technical papers, holds 13 patents, and has received an Intel Achievement Award for his work on Intel® AMT.

Vincent Zimmer is a principal engineer in the Software and Services Group at Intel. He has been firmware developer for over 20 years. He has a Bachelor of Science in electrical engineering from Cornell University, Ithaca, NY, and a Master of Science in computer science and engineering from the University of

Washington, Seattle, WA. He has published three books, two book chapters, one IETF RFC, ten publications and over 270 US patents.

Leonid Ryzhyk is a Postdoctoral Fellow at the University of Toronto and Researcher at NICTA. He obtained a PhD in Computer Science from the University of New South Wales, Sydney, Australia in 2010. He received his Bachelor's and Master's degrees in Computer Science from the National Technical University of Ukraine in 2000 and 2002.

Adam Walker is a PhD student at the University of New South Wales, Sydney, Australia. He obtained his Bachelor's degree from the University of Auckland, New Zealand in 2008.

Alexander Legg is a PhD student at the University of New South Wales, working with NICTA in Sydney, Australia. He received a Bachelor of Information Technology (Hons) from the University of Sydney in 2011.