Modeling, analysis, and simulation of communication software execution on multicore devices

Øystein Dale
Master’s Thesis Spring 2016
Modeling, analysis, and simulation of communication software execution on multicore devices

Øystein Dale

Supervisors
Stein Kristiansen and Thomas Plagemann

April 29, 2016
Abstract

The effects of software execution are typically not accounted for when performing experiments in network simulators. This leads to discrepancies between the results observed in simulation experiments and the effects that can be observed in real networks. A methodology to model communication software execution exists, with an extension for the ns-3 network simulator to model the effects of software execution in network simulations. However, this approach has only been considered for singlecore devices. In this thesis the methodology, including the tools and the implementation of the singlecore processing model for ns-3, is extended to enable modeling, analysis, and simulation of multicore devices. Extending the methodology and the implementation of the processing model requires extensive knowledge about multicore execution. This topic is investigated using the Linux kernel as an example, including how the kernel supports symmetric multiprocessing and how networked packets are handled by the kernel. The existing methodology is presented, along with the design changes required to facilitate modeling, analysis, and simulation of multicore devices. The applicability of the extended methodology is demonstrated by modeling the execution of communication software on a Galaxy Nexus, an Android-based multicore smartphone. The network driver on this device is analyzed and the methodology is applied to model the driver. Software execution is simulated using a processing model for ns-3 that is extended to account for multicore execution. The result of a simulation experiment is compared against measurements taken on the smartphone under similar conditions in a real network. The results show that the methodology can be applied to capture and simulate behaviors that can be observed on the real device. Further, experiments that reveal how multicore software execution affects network performance are done. These show that thread migration does not significantly impact the latency a packet experiences on devices with small caches, that reducing the amount of queuing in the kernel greatly reduces the latency experienced by a packet, and that small changes to the behavior of software can have a significant impact on network performance.
# Contents

I Introduction & Background 1

1 Introduction 3

1.1 Goals .............................................. 3
1.2 Approach ........................................ 4
1.3 Contributions ................................. 4
1.4 Thesis structure ............................. 5

2 Background 7

2.1 The ns-3 network simulator ................. 7

2.1.1 The ns-3 communication software execution model 7
2.1.2 Discrete event simulation ............... 8
2.1.3 History of ns-3 ............................. 8
2.1.4 ns-3 structure ............................. 9

2.2 A methodology for modeling the execution of software 9

2.2.1 Key concepts ................................ 9
2.2.2 Overview of the modeling workflow .... 12
2.2.3 Instrumentation of code .................. 13
2.2.4 The tracing framework ................. 15
2.2.5 Analysis .................................... 17
2.2.6 Modeling .................................. 18
2.2.7 Simulation ................................ 20

2.3 The Linux kernel ............................. 24

2.3.1 Symmetric Multiprocessing .............. 24
2.3.2 Deferred interrupt processing ........... 25
2.3.3 Multicore scheduling ..................... 26
2.3.4 Kernel concurrency ....................... 26
2.3.5 Synchronization in the Linux kernel ... 27
2.3.6 Networking in the Linux kernel ........ 29

2.4 The Android platform ....................... 32

II Design 35

3 Design of Multicore Tracing, Analysis, and Simulation 37

3.1 Methodology concepts for multicore execution ........ 37
3.2 Tracing events on multicore systems ............ 38

3.2.1 Requirements ............................... 38
3.2.2 Design ........................................ 38
3.3 Enabling simulation of multicore execution .......... 40
  3.3.1 Requirements ................................ 40
  3.3.2 Design ..................................... 41
3.4 Analysis of multicore traces .......................... 44
  3.4.1 Requirements ................................ 45
  3.4.2 Design ..................................... 45

4 Analysis and Instrumentation of the bcmhd Driver 49
  4.1 Background ..................................... 49
  4.2 Working modes .................................. 50
  4.3 Overview of driver thread execution ............... 51
  4.4 Instrumenting and modeling the driver ............. 53
    4.4.1 Model-specific tracepoints .................. 53
    4.4.2 Behavior-specific tracepoints ............... 56
    4.4.3 Measurement-specific tracepoints ............. 56

III Experiments .................................. 59

5 Experiment Design ................................ 61
  5.1 Experiment 1 — Accuracy of processing delay and forwarding capacity .......... 61
    5.1.1 Experiment parameters ...................... 62
    5.1.2 Expected results ............................ 62
    5.1.3 Setup of the real-world experiment ........... 63
    5.1.4 Setup of ns-3 experiment .................... 66
    5.1.5 Sources of errors ............................ 66
  5.2 Experiment 2 — Concurrent forwarding while sending locally generated traffic .......... 68
    5.2.1 Experiment parameters ...................... 68
    5.2.2 Experiment setup ............................ 69
    5.2.3 Expected results ............................. 69
  5.3 Experiment 3 — Effect of driver modification ....... 70
    5.3.1 Experiment parameters ...................... 70
    5.3.2 Experiment setup ............................ 71
    5.3.3 Expected results ............................. 71
  5.4 Experiment 4 — Scaling of the multicore processing model ........... 71
    5.4.1 Experiment parameters ...................... 72
    5.4.2 Experiment setup ............................ 72
    5.4.3 Expected results ............................. 72

IV Conclusion ................................... 75

6 Results and Analysis ............................. 77
  6.1 Experiment 1 .................................. 77
    6.1.1 Overview of figures ......................... 77
# List of Figures

2.1 Workflow of the model development ................. 13
2.2 Diagram of the tracing framework .................. 16
2.3 Analysis stack before and after exiting a service. .... 18
2.4 Original ns-3 singlecore processing model architecture. 20
2.5 Creation of a SEM from signatures .................. 22
2.6 Socket buffer operations. ............................ 31

3.1 Kernel trace probe execution ........................... 39
3.2 Extended ns-3 multicore processing model architecture ... 48

4.1 Logical layout of the network-related hardware in the Galaxy Nexus device. ..................................... 50
4.2 Sequence diagram of driver execution with early tx ....... 51
4.3 Sequence diagram of driver execution without early tx .... 52
4.4 Example of nested services ............................. 55
4.5 Call graph of the modeled driver thread. ................. 58

5.1 Logical layout of the rig used in the experiments ........ 63
5.2 View of the experiment rig ............................. 64
5.3 Pseudo-code of four different signatures ................ 73

6.1 Forwarded packet rates as seen by monitor ............... 78
6.2 Distribution of processing delays from receiving packet to sending packet. ...................................... 79
6.3 Distribution of processing delays between handing a packet to the kernel and sending the packet. .................. 79
6.4 Migration of driver thread vs. forwarding delay .......... 84
6.5 Packet rate of two different traffic flows. ................. 84
6.6 Distribution of processing delays when the early tx code is used 87
6.7 Distribution of processing delays when the early tx code is not used ........................................... 87
6.8 Scaling with number of threads .......................... 89
6.9 Scaling with number of nodes ........................... 89
6.10 Scaling with number of cores ........................... 89
## List of Tables

B.1 Instrumentation of NIC driver .......................... 109  
B.2 Instrumentation of the Linux scheduler .............. 110  
B.3 Instrumentation of platform-specific code .......... 110  
B.4 Instrumentation of networking code in the Linux kernel . . . 110  
B.5 Instrumentation of interrupts and deferred processing . . . 111
Acronyms

adb   Android Debug Bridge.
AOSP  Android Open Source Project.
API   Application Programming Interface.
BKL   Big Kernel Lock.
CPU   Central Processing Unit.
DCE   Direct Code Execution.
DMA   Direct Memory Access.
IBSS  Independent Basic Service Set.
IPI   Inter-processor interrupt.
IRQ   Interrupt Request.
ISR   Interrupt Service Routine.
LEU   Logical Execution Unit.
MAC   Medium Access Control.
NIC   Network Interface Card.
OFDM  Orthogonal Frequency-Division Multiplexing.
OS    Operating System.
PEU   Physical Execution Unit.
PID   Process ID.
PMU   Performance Monitoring Unit.
SDIO  Secure Digital I/O.
SDN   Software Defined Network.
SEM   Software Execution Model.
SMP   Symmetric Multiprocessing.
TLB   Translation Lookaside Buffer.
TTL   Time-to-Live.
Preface

First and foremost I wish to thank my supervisors, Stein Kristiansen and Thomas Plagemann. Their knowledge, guidance, support, and feedback has been of great help in both the thesis work and in writing this report. This thesis is based on work that Kristiansen did for his Ph.D., and would not have been realistically possible without this previous work. His presence has been invaluable.

Jan Anders Bremer served a mentor-like role in the early years of my studies. He is responsible for introducing me to tools that are invaluable to me today. His proofreading and feedback has been highly appreciated.

Hans Petter Taugbøl Kragset also deserves many thanks for reading through my thesis and pointing out areas where clarification was needed.

Thanks to Olav Stanly Kyrvestad and the NANO research group for letting me conduct my experiments in their Faraday cage.

Thanks to all lecturers and teaching staff at the Department of Informatics at the University of Oslo for all the all the courses and lectures I have had the privilege to attend.

Finally, I wish to thank my family and friends for their continued support and understanding during the entire process.

A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.

Antoine de Saint-Exupéry

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke
Part I

Introduction & Background
Chapter 1

Introduction

Performing experiments using real networks is time-consuming, complicated, costly, and it is especially difficult or even impossible to perform repeatable experiments. Network simulators have been created to study the performance of networks and protocols in isolated, repeatable conditions. However, the vast majority of commonly used simulators do not account for the overhead of communication software execution on network nodes and its impact on network performance. The behavior of software has been shown to have a non-negligible impact on the performance of a networked device. This is especially true for devices that have limited resources: low-powered devices with slow processors and limited memory. Not accounting for software execution may lead to differences in measured performance between a simulator experiment and a corresponding experiment with hardware in a real network. Having models in a network simulator to account for software execution not only provides more accurate results, but it also increases our confidence in the results provided by simulation experiments.

A processing model extension for a network simulator exists, which is developed by Stein Kristiansen as part of his Ph.D. thesis on modeling, analyzing, and simulating communication software execution[24]. The methodology, including the simulator extension and the implementation of supporting tools, were made with execution on singlecore devices in mind. This is a shortcoming as more and more devices are equipped with multicore processors today.

The purpose of this thesis is to address this shortcoming by extending the existing methodology to account for parallel processing on multiple cores. This includes to extend the tools and the network simulator extension.

1.1 Goals

This thesis has four goals:

1. Extend the existing processing model,
2. instrument and model a multicore device,
3. verify the model, and
4. gain knowledge about communication software execution.
To achieve the first goal, the existing processing model is extended such that it can be used to simulate software execution on multicore devices in addition to singlecore devices. The second goal requires to instrument and model software execution on a multicore device to demonstrate the use of the methodology when applied to multicore processing. The third goal requires to test the extended processing model by comparing the modeled device in the simulator against the real device in a physical network. Finally, the last goal is to gain knowledge about communication software execution in general, which is achieved through experiments. Modeling and designing communication software execution requires knowledge about how software executes and how execution contributes to, for instance, the latency experienced by a packet in a network. Hopefully some of the results from the experiments in this thesis can be used as guidelines for designing communication software for low-powered, resource-limited devices.

1.2 Approach

Extending the methodology, including the tools and the simulator, requires knowledge about multicore execution and how this is managed in operating system kernels. This knowledge is gained through a review of how multicore execution is handled in the Linux kernel, and will be used to identify the parts of the methodology that requires modifications to enable simulation of multicore devices. This includes changes to the tools and simulator extension to create a network simulator capable of simulating multicore software execution.

A multicore device, the Galaxy Nexus smartphone, is instrumented, analyzed, and modeled using the extended methodology. An experiment is performed on the real device. The same experiment is repeated in a simulator and the results from the two experiments are compared to evaluate the accuracy of the model and new simulator extension. Several experiments are done to investigate the impact of software execution in a resource-limited device on network performance.

1.3 Contributions

The first contribution of this thesis is to review the concepts in the methodology to determine its suitability for application on multicore communication software execution, to create a processing model capable of simulating multicore execution in a network simulator, and to create tools that are capable of supporting the use of the methodology: a tracing framework for the Linux kernel that is able to capture meaningful data from a multicore device, and a tool to analyze the resulting trace files and generate output that can be used to construct a device model. This allows for the study and simulation of software execution on both singlecore and multicore devices, thus increasing the accuracy of simulation experiments.

The second contribution is a model of the communication software on a multicore device which can be used as a reference when modeling new
devices. The insight gained through making this model can be used in the future to construct models for other devices more easily, thus making the methodology more accessible.

The third contribution is an experiment to show that the extended methodology is able to capture the observable behaviors of software execution on a device and the effects of these behaviors on network performance, thus increasing our confidence in the results provided when performing simulation experiments that makes use of the processing model.

The fourth contribution is a set of experiments that demonstrate the impact of communication software execution. This is knowledge that can be used to influence the design of communication software on new devices, and decisions on what behaviors and effects to include in models of devices.

1.4 Thesis structure

The rest of this thesis is structured as follows:

Chapter 2 presents the methodology: The concepts, the existing singlecore processing model for a network simulator, and the accompanying tools. This chapter also gives the reader some insight into the fundamentals of multicore execution in the Linux kernel, as well as background knowledge on how the kernel handles concurrency and synchronization. The network code in the Linux kernel is also introduced.

Chapter 3 outlines the required design changes for the tools and the processing model in the simulator to support modeling, analysis, and simulation of multicore devices. An analysis of the wireless network driver in the multicore Galaxy Nexus smartphone used in the experiments is done in Chapter 4, including how this driver is modeled and instrumented based on the methodology.

The design and setup of the experiments is given in Chapter 5. The results are presented in Chapter 6, along with an analysis of the results and conclusions with background in the analysis. Chapter 7 concludes this thesis, presents some personal thoughts on working with this thesis, and suggests open issues and future work.

The appendices contain information that is useful for anyone that is doing work on similar types of hardware platforms: A short summary of challenges encountered, an overview of the device instrumentation, and instructions on how to cross-compile a Linux kernel for an Android smartphone.
Chapter 2

Background

In this chapter, the background information that is needed to understand the thesis is presented. First, the network simulator ns-3 is introduced along with an explanation of discrete-event simulation, followed by the methodology of software execution modeling. Next, a description of the Linux kernel, how it handles Symmetric Multiprocessing (SMP), and a high-level description of how networking is implemented. Finally, a short introduction to the Android platform is given.

2.1 The ns-3 network simulator

ns-3[1] is an open-source, discrete-event network simulator written in C++, popularly used in research and education on computer networks. It contains several advanced features such as a fully functional model of the Internet protocol stack, models of application layer protocols, mobile communication protocols and link technologies, models for signal propagation in various physical media, models for mobility, and more.

2.1.1 The ns-3 communication software execution model

Kristiansen used ns-3 as a basis for a proof-of-concept implementation of a model for communication software execution[27], something which was not previously available for ns-3 or any other commonly used network simulator. This processing model allows to simulate the impact of software execution on networking performance with increased accuracy in ns-3. The input for this model is a device specification, which is generated from trace files captured on the device itself. The trace files are subjected to an analysis, from which the results can be used to create a device model. The methodology is explained in more detail later in this chapter.

This methodology, including the processing model implementation, the associated trace analysis scripts, and kernel tracing framework forms the foundation for the work done in this thesis. The current implementation of the software execution model in ns-3 assumes that the device has a single Central Processing Unit (CPU) core. The goal of this work is to extend the singlecore execution model implementation to a multicore execution model.
The analysis scripts and tracing framework were made with singlecore devices as the intended targets. Thus, it is likely that these tools require extensions as well to enable the capture of traces on multicore devices and the subsequent analysis of these. The existing methodology will be further explained in Section 2.2, after the remainder of this introduction to ns-3.

2.1.2 Discrete event simulation

Discrete event simulation is based on processing one event at a time. It is assumed that the simulated system does not change between events, which allows the simulation to jump immediately to the next event when having finished processing an event, regardless of what time the next event occurs at. Events are scheduled by placing them in a priority queue. This queue is ordered by the simulation time when the event is scheduled to occur. During simulation the event with the earliest simulation time is dequeued and processed, potentially queuing more events that will happen at a later point in the simulation time. This way, the time in the simulation jumps in discrete steps of arbitrary lengths, as opposed to the simulation time progressing at a fixed rate. Any action must either have an immediate effect or lead to a new action that is scheduled to happen in the future of the simulation time.

2.1.3 History of ns-3

The history, goals, and design guidelines for the network simulator ns-3 can be found in [23]. A short summary is provided here.

The network simulator ns-3 is the successor of the popular ns-2 (1996), which in turn was based on ns (1995), developed by Lawrence Berkeley National Laboratory, and the REAL simulator made by S. Keshaw. The architecture of ns-2 is substantially different from ns. The most significant feature of ns-2 is that the simulation core is written in C++ while the simulation scenarios are written in OTcl. This allowed simulations to be written in a high-level language and permitted changes to the simulation without potentially having to recompile the simulator. The primary goal for the ns-3 simulator is to provide a free, open-source network simulator that is modular, extensible, scalable, realistic, well-documented, testable, and with validated core models.

According to a survey done by Lindeberg et al. in 2011 on the use of network simulators in experiments, 37.5% of the papers using a network simulator identified ns-2 as the network simulator in use. 20.3% mentioned another simulator; the remaining 42.2% did not specify which simulator was used at all[28]. From this, it is clear that ns-2 had a strong position as a popular network simulator for use in research at that time. Given that ns-3 is based on the same principles as ns-2, but improved in several aspects, it is reasonable to assume that ns-3 has a similar position in the field of network simulators today as a successor to ns-2.
2.1.4 ns-3 structure

The ns-3 code is structured in separate modules. A module typically consists of a model of a network protocol, a model of a link technology, or functionality that is used by multiple other modules. The processing extension is implemented as a model, with a minimal amount of changes to the surrounding code. This should, in theory, allow the processing extension to function with the existing models.

Experiments in ns-3 are set up and compiled as C++ programs. ns-3 provides tools to simplify this process. The source code for experiments is placed in a special folder outside of the main ns-3 source code tree, providing good separation between the experiment code and the implementation of the various models found in ns-3. This also allows developers not familiar with the internal structure of ns-3 to write and execute experiments easily.

2.2 A methodology for modeling the execution of software

A methodology for modeling software execution in network simulators is presented in [27]. This methodology provides guidelines to instrument communication software, it supports the subsequent analysis of traces generated by the instrumentation, and modeling the behavior of communication software execution. This makes it possible to use simulation to study and determine the performance characteristics of a network consisting of the modeled devices. Accounting for the effects of software execution gives more accurate results, resulting in a higher confidence that the results of the simulation represent observable effects in a real network.

2.2.1 Key concepts

The methodology consists of a set of key concepts, which are explained below:

- Physical Execution Units (PEUs),
- Logical Execution Units (LEUs),
- services,
- packet and service queues, and
- state variables and contexts.

Physical execution units

Hardware represents the lowest layer of software execution where different physical components perform some execution on behalf of the system. We refer to these components as PEUs. The primary example is the CPU, which executes the instructions of the software. Other examples are the Direct Memory Access (DMA) chip, which performs memory transfers, and the Network Interface Card (NIC), which provides the link interface.
to the physical medium. Neither is programmable directly by the end user, but they can be instructed by the CPU to perform certain tasks such as moving data to and from main memory or transmitting packets. Therefore, these components are of interest regarding the execution of communication software.

Logical execution units

Most of the commonly used multitasked Operating Systems (OSes) support the concept of threads, which can be used by the programmer to divide work into separate execution streams, each representing a single context of execution. Multiple threads can execute concurrently on the same CPU core by the use of time-sharing mechanics, such as preemptive scheduling. However, within a single CPU core, no thread will execute in parallel with another thread. We refer to them as LEUs, because each of these threads represents a logically separate context of execution. An interrupt is also a type of LEU, because an interrupt executes a stream of instructions that is logically separate from the thread that was executing at the time of interruption. As with threads, no interrupt handler will execute in parallel on the same CPU core.

Services

Next, a model for the communication software itself is needed. Software is divided into sections, e.g. functions, that may be used by multiple LEUs. An OS usually provides a unified interface for sending and receiving packets, such as the `ip_send` and `ip_rcv` functions in the Linux kernel. Both functions are used by multiple different LEUs to send and receive packets. Functions such as `ip_send` and `ip_rcv` are conceptualized as services, representing points of execution which are potentially used by several LEUs. More formally, a service is defined as all reachable code between two points marking the start and the end of the service respectively, except the code that belongs to another service.

Services are divided into two groups: The first type of services, *Packet Handling Services*, performs a packet processing task that is defined by a protocol, and may call other services to perform a subtask. `ip_send` may use the transmit function of the NIC driver to send a packet. The other kind of services is *Work Scheduling Services*. These services use a queue of function pointers. Each function is executed in turn within a loop. The called functions become services of their own. These are commonly used in the Linux kernel to perform work in a different LEU than the one that is currently executing.

An example of this is how the processing of interrupts is deferred in the Linux kernel. This is done by activating a function to process the interrupt in the future by placing a pointer to the function in a queue and scheduling a thread that processes the queue. This relates to the concept of service queues, which will be discussed in the next section along with packet queues.
Packet and service queues

Communication software consists of several queues of packets waiting to be handled. These queues typically have FIFO properties and a finite bound either determined by the number of packets allowed in the queue, or by a total number of bytes available for all packets in the queue combined. Packets remain in the queue until a service that processes the queue is enabled by the scheduler. If a queue is full, any subsequent packets that are enqueued are simply discarded. Queuing of packets leads to increased latency and packet loss, which are essential properties to capture in models of communication systems.

From an external point of view, a node in a network may be seen as one queue, where some packets are received by a node, but not yet forwarded. Internally, a node may have several packet queues. For example, a NIC typically has queues for incoming and outgoing packets. Likewise, a driver may have a queue for incoming packets fetched from the NIC and a queue for packets waiting to be transferred to the NIC. The OS may have one or more queues for packets awaiting processing or packets that are waiting to be handed to the associated driver.

The concept of deferred processing is commonly used in modern OSes. Instead of processing a hardware interrupt right away, a service is scheduled to do the actual processing of the interrupt at a later time. In its simplest form, this can take the shape of a kernel thread and a queue of function pointers, akin to work queues in the Linux kernel. A function pointer is enqueued, and the kernel thread is scheduled. Once the kernel thread is dispatched, it will dequeue the function pointer and call the queued function. Delaying the execution of some code is done for instance to reduce the size of critical, uninterruptible regions of code to promote interactivity at the cost of some added latency.

Deferred processing is conceptualized in the methodology as a service queue. As with packet queues, these have FIFO properties and may be infinite or have a finite bound. They are used to model the activation and execution of services that are directly activated by other services, as opposed to services that are activated by calls to the scheduler or through interactions with synchronization primitives.

State variables and context

Separate executions of the same service may differ due to certain variables having different values. These variables are called state variables. Different combinations of values assigned to the state variables of a service yield different contexts. The execution of a service given a particular context results in a specific behavior. For example, a service that processes packets in a queue may behave differently depending on whether the queue is empty or not. This is called a queue condition. A service may also treat a packet differently depending on the packet size, destination address, or other values in the packet. State variables can be used to model the impact of any variable or expression during execution.
The value of state variables is captured during tracing. When executing the simulation, an expression corresponding to the state variable is evaluated. The appropriate signature is selected based on the result of the evaluation of all the different state variables in the service. The evaluation of some state variables, such as queue conditions, is implemented directly in the processing model. More complex state variables may require custom expressions to be created and implemented in the simulator script.

2.2.2 Overview of the modeling workflow

Producing a model of the communication software on a device for simulation is an iterative, time-consuming process. A good overview and understanding of the code are prerequisites. Thus, the first part is often to read the code of the kernel that is to be instrumented and optionally make diagrams, flowcharts or other types of notes. Understanding the code and its structure can often be sufficient to identify the services that may be of interest.

Once an initial instrumentation has been implemented, the tracing may begin. The device is booted and executed with specific workloads, ideally to capture all different behaviors. The resulting trace file gives an overview of the events happening on the device, from the start of tracing until the end of tracing.

The trace file is subsequently subjected to analysis to extract the information of interest. The goal of the analysis is to produce a set of signatures. The signatures describe the behavior of a service during execution. One service may have multiple signatures, describing the service behavior in different conditions. All executions of a service with the same assignments to the state variables and conditions are grouped into one signature. Thus, the number of different signatures from each service depends on the number of different values assigned to the state variables. Some services can have infinite state spaces and can produce a large number of signatures.

Once a set of signatures has been generated, the signatures are used to develop a model of the software on the target device that can be simulated. In this part of the process, the model developer must determine which signatures to include in the model. Many of the signatures may be the result of only a small fraction of the executions of the service. Such signatures may be left out of the final model. Some manual tweaking of the model may also be required. The result of this process is a device file containing a description of the device hardware and a model of the communication software on the device, including the signatures. This is discussed in more detail in Section 2.2.6.

The entire modeling process is highly iterative. At any point in the process, it might be possible that one must go back to a prior step. For example, after analysis, it may be the case that more services must be instrumented, that additional state variables must be captured and so on. This requires additional instrumentation, recompilation of the kernel, tracing, and analysis. Likewise, an issue discovered during testing of the
model in the simulator may require additional instrumentation and tracing, as well as requiring the analysis to be extended. As such, the process is repetitive and very likely to be time-consuming.

![Workflow of the model development](image)

**Figure 2.1: Workflow of the model development**

### 2.2.3 Instrumentation of code

The instrumentation step involves the insertion of tracepoints in the code. These tracepoints record the occurrence of an event at the given code location and collect data about the event. These data is used later for analysis. For each event, a set of common values is stored in addition to event-specific values.

When instrumenting code it is important to capture the event as close to the point of effect as possible. For instance, the signatures from the execution of a service may be similar up to a branching point on a condition. The common processing that is done before the branching point must be distinguished from the processing that is done after the branching point. This separates the processing that is, in this case, dependent on the value of the condition from the processing that is not dependent on the value.

The following list contains the events that are traced:

- Service entry and exit,
- loop start, restart, and stop,
- enqueueing/dequeueing of packets and services,
- context switches,
- kernel synchronization events, and
- reading state variables and conditions.

A service typically consists of one or more nested functions that achieve some kind of goal on behalf of the caller. The service is identified by the name of the function it corresponds to. Examples of services are `ip_rcv` which handles an incoming IP packet, or a driver function that enqueues a packet to the NIC. A service entry tracepoint is most often added to the beginning of the function, and a service exit tracepoint to the end of the function. In the case of kernel threads with infinite loops, these are modeled as services that are repeatedly called. They are traced by having a service entry traced at the beginning of the infinite loop body, and a service exit at the end of the infinite loop body. In many cases, it can be useful to reduce the complexity of one service by also tracing additional services inside another service, even though this is not always required.
Operations on queues are traced by inserting tracepoints after the completion of a queue operation. A unique identifier is assigned to each queue and used to specify which queue is modified by the operation. Dequeueing from a service queue implies an immediate call to the newly dequeued service.

Interrupts are an important part of device drivers. Interrupt Requests (IRQs) are traced as services and are identified with a prefix followed by the IRQ number. This is traced by inserting a tracepoint where the code enters an IRQ context, and a tracepoint where the code leaves an interrupt context.

Loops ranging over queues present a challenge when it comes to modeling. The number of times the loop body is executed depends on the conditions of the queues, being either empty or non-empty. Modeling this by simply tracing the condition of each queue would produce far too many different signatures, especially if the execution of the service that contains the loop is not dependent on any of the queue conditions referenced in the loop. Instead, loops are modeled as separate services. Thus, it will have its own set of signatures. They are instrumented by placing tracepoints at the start of the loop before the loop body has started executing, at the beginning of the loop body where each iteration starts from, and at the end of the loop. The queues that affect the loop are recorded along with the maximum number of iterations, if the loop has a limited number of iterations. During simulation the condition of the queues are evaluated in turn and the loop body is executed once for each non-empty queue and, if specified, until the maximum number of iterations is reached.

State variables and queue conditions can be significantly more difficult to discover. Their presence might not be obvious before an analysis is performed. Typically, some can be found in boolean conditional expressions in or surrounding the service. In some cases, code needs to be rewritten to capture a condition or state variable properly, since the tracepoint itself cannot be part of a conditional expression. Tracepoints that capture the value state variables or queue conditions are inserted at the point where they take effect, for instance immediately preceding a branching condition on a state variable or queue condition.

Context switches must be traced to accurately compute the processing delays in a service and to distinguish between different LEUs time-sharing the same PEU. A context switch does not affect the behavior of a service and they are not part of the final signature. Regardless of this, the context switches must still be traced for use in the analysis. It is important to note that the scheduler itself is not instrumented, but the interactions between the LEUs and the scheduler are. A model of the scheduler is used during simulation to schedule the simulated threads in the processing model. It is also used to handle scheduler interactions in the signatures. These are requests to the scheduler made by the simulated threads, such as operations on synchronization variables, or a call to put the thread to sleep.

While synchronization objects used by services do not directly contribute to define how a service behaves, it influences how the service is scheduled in relation to other services and outside events. These synchronization points must be captured in order to model the same synchronization events.
during simulation. It is not necessary to capture the values or state of the synchronization objects; this is also modeled in the simulator as part of the scheduler model.

Synchronization objects may also appear on the stack, in addition to being statically allocated. These temporary synchronization objects must be dynamically allocated during simulation. Thus, it is necessary to differentiate between statically allocated and stack-allocated synchronization objects during tracing and simulation.

### 2.2.4 The tracing framework

The goal of the tracing step is to provide workloads to the device to trigger all possible behaviors of the services of interest. Signatures from different traces may be combined when creating the device file; not all behaviors need to be triggered during the same tracing session. As such, it can be useful to focus on one service at a time.

The tracing framework consists of four components, seen in Figure 2.2: Tracepoint definitions, a tracing kernel module, trace probes, and a user-space program. The tracepoints are based on an existing tracing mechanism present in the Linux kernel[2]. The tracing mechanism consists of two parts: a tracepoint function and a trace probe. The tracepoint function is the function that is called at the point of instrumentation. The trace probe is a separate function. It is called by the tracepoint function when the tracepoint is executed, but only if the tracepoint has been activated. Preprocessor macros are used to declare and define the trace function, and to declare the trace probes. The trace probes must be defined manually. Based on convention, the tracepoints related to the tracing of software execution modeling have the function names prefixed with `trace_sepext_` in the tracing framework. The corresponding trace probe has the prefix `probe_sepext_`.

The tracing module is responsible for handling the kernel-side of tracing. Upon initialization, it allocates a fixed-size buffer for storing the trace data, as well as enabling the tracepoints so that they call the trace probes when executed. All trace probes call a common function, `trace_entry` in Figure 2.2, which stores the data from the tracepoint. Each execution of a tracepoint is recorded in the buffer as a trace entry. Once the buffer reaches its capacity a tasklet\(^1\) is scheduled, which awakens the user-space program. The user-space program issues commands to start emptying the kernel-space buffer and write the trace entries to a file. No data from tracepoints are recorded as long as data is being written to secondary memory, to avoid interference from the additional I/O-operations in the trace results. The tracing resumes once all entries have been sent to the user-space program and the user-space program notifies the kernel module that the data transfer is complete. The tracing module creates an entry in `/proc`\(^2\), which is used to

---

1\(^{\text{Explained in Section 2.3.2.}}\)

2\(^{/proc} \text{is a pseudo-filesystem in Linux, which is used to call specially prepared kernel functions through read and write operations on specific files. These proc entries can be used to send data to the kernel or to fetch data from the kernel.}}\)
Figure 2.2: Diagram of the tracing framework. The original version uses a tasklet, the modified version uses an IPI, see Chapter 3. Black arrows indicate execution flow, open arrows indicate data transfers, and the dotted arrow indicates the logical flow of data.
issue a command to tell the tracing framework to stop tracing permanently, write any remaining entries to the trace output, and stop the user-space program.

The user-space program communicates with the kernel module via Netlink, a communication link between user-space and the kernel based on the socket Application Programming Interface (API). After issuing a message over the Netlink socket to start the tracing, it attempts to read a message from the socket. This causes the process to block until a message is sent from the kernel. Once the trace buffer is full, the kernel module sends a special wake-up message to the user-space program. The user-space program then issues a message to the kernel module to start sending trace entries. Once all entries have been received, the user-space program blocks again, waiting for another wake-up message. If the user-space program receives a message that a command to stop tracing has been issued by a write to the `/proc` entry, it instructs the kernel module to transfer any remaining trace entries that have not yet been written to the trace file. The user-space program exits when all remaining entries are written.

Each line in the trace file contains one entry for an event that is captured during tracing. A trace entry contains the CPU cycle counter value, the ID of the currently scheduled process, type of event, event-specific data, and the address of the tracepoint in the compiled kernel code.

### 2.2.5 Analysis

The goal of the analysis step of the workflow is to separate the linear trace file into multiple signatures, at least one signature for each service that is encountered in the trace file. The analysis is done with a script that automates the process. The analysis is divided into three stages: The first stage separates the services from each other. The second stage calculates the processing delays between each event and groups common service execution cases based on the sequence of event types encountered. The final stage merges the execution cases in each group into a single signature. In addition to these stages, some initial processing may also be applied. An example of initial processing is to translate the memory addresses found in the trace file into names based on data from the kernel compilation, or other types of simple textual replacement.

Each event encountered during analysis that belongs to a service is stored in a buffer. Between the entry and exit of a service, this buffer is filled with events from one specific execution of the service. When a service exits, the buffer contains a complete history of one service execution, referred to as a service execution case. The buffer containing the service execution case is stored for use in the final stage, and a new buffer is allocated for the next set of events belonging to a new service execution case.

The services are separated from each other by using a stack structure. Once a service is entered, it is pushed onto a stack. Any subsequent events in the trace file belong to the service at the top of the stack, until this service exits or calls another service. When the service exits, the exited service is popped off the stack and the service execution case is stored.
This stack operation can be seen in Figure 2.3. The state right before the analysis reaches the service exit event in Service 2 is illustrated in Figure 2.3a. In the buffer for events in Service 1 are events A, B, and an event denoting a call to Service 2. The buffer for Service 2 contains the events C and D. Figure 2.3b shows the state of the analysis after Service 2 has been popped off the top of the stack. The buffer for Service 1 remains while the buffer for Service 2 is stored. This buffer is used when calculating the execution delays of Service 2, and to generate a signature for this service.

The second stage of the analysis calculates the execution delays. The time spent processing between two events within a service is equal to the difference between the number of cycles elapsed, minus the number of cycles spent in interrupts. Preemptions are handled by adding the remaining time spent executing before the preemption and the time after being dispatched until the next event. The same applies to calling a service. All service execution cases that contain the same order of event descriptors and the same state variable values are grouped together into one signature, with a list of the processing delays between each event.

The final stage creates the signature, where the processing delays are aggregated with statistical functions. These statistical values are used in the simulator to produce a model of the delay software execution imposes on the network performance in the simulator.

2.2.6 Modeling

Once a set of acceptable signatures is produced, it is up to the model developer to select a subset of the available signatures that produces an accurate representation of the communication software executing on the device and its characteristics. An important aspect of the modeling step is to decide which signatures to leave out. As with any type of modeling, there is a trade-off between accuracy and complexity. One approach to deciding which signatures to include is to look at how frequently a signature appears compared to the total number of signatures for the service. When a signature occurs frequently, it has a greater impact on the characteristics of the device compared to a signature that is encountered less frequently. As such, it is
of more importance to include the signatures that occur more frequently compared to those occurring less frequently.

**The device file**

The output from the modeling step is a device file. The device file consists of a header section that describes the different structures in the device model and a section with the different signatures. The header section is typically hand-crafted, the signatures are taken from the output of the analysis.

First in the device file is a list of packet and service queues. The order the queues are listed in matters. The range of queues used by a loop is defined by specifying the first and last queue in the range. This implies that the two specified queues and all queues between these two in the queue list are used by the loop.

The queues are specified on a single line starting with an identifier which is used to identify the queue in the signatures and in the simulation code. The type of queuing discipline is specified, e.g. FIFO. Following that is a number that describes the capacity of the queue, and a keyword that determines the type of objects contained in the queue. `services` is the keyword used for service queue. For packet queues, it may be either `packets` or `bytes`, depending on how the capacity of the queue is specified.

Following the description of the queues is a description of the hardware of the modeled device. This contains a list of all PEUs that execute signatures in the simulation. Each PEU is identified with a name, followed by a specification of clock speed in megahertz, and a reference to the object that schedules the threads on the PEU. Additionally, the identifier for the service handling the interrupts and the identifier for the service queue for interrupts is specified.

The next part is for the synchronization objects: semaphores and completions. This section is for synchronization objects that are statically allocated, i.e. global synchronization variables. These synchronization objects are given identifiers and initial values.

The final section before the signatures is a list of triggers. A trigger is used during simulation to switch between simulating in the existing ns-3 models and the processing model. It serves as a synchronization point between what happens in the ns-3 models and what happens in the processing model. This is how the processing model is able to correlate what happens in the signature with what happens in the ns-3 protocol stack. This will be explained further in Section 2.2.7. A trigger may be placed at a location in a signature, on a dequeue from a specified queue, or on the execution of a service.

After the triggers are the signatures themselves. Unlike the other descriptors, these span multiple lines. In addition to the events and processing delays in the signature, each signature contains a header with metadata such as an identifier, the number of times the signature was encountered in the trace file, as well as what type of PEU the signature was executed on, the type of resources the signature consumes and information about the type of statistical function that was used in the analysis. The
second part of the signature is a description of the execution of the service based on the events that were traced, as well as statistical information on the processing delays between each event.

The last section in the device file is a list of LEUs, i.e., threads. They are given an identifier, and the top-level service is specified along with scheduling information such as the number of executions and the priority. A thread may be marked as infinite. An infinite thread will call the top-level service again when the top-level service exits during simulation.

2.2.7 Simulation

The presentation of the existing singlecore processing model extension for ns-3 is done in four steps: First, an architectural overview of the processing model is given, followed by a brief introduction to how signatures are used by the processing model. Then, the initialization of the processing model is explained. Finally, the execution of the processing model during simulation is outlined.

![Figure 2.4: Original ns-3 singlecore processing model architecture.](image)

**Figure 2.4**: Original ns-3 singlecore processing model architecture.

**Architecture**

The architecture of the singlecore processing model for ns-3 can be seen in Figure 2.4. The Node class is the primary class in ns-3 for representing a networked entity. ExecEnv represents the execution environment as a whole, both in terms of hardware and software. An object of this class is aggregated onto the node object with an ns-3-specific object aggregation mechanism. The ExecEnv class contains a reference to the HWModel class, which represents the hardware of the simulated device. Each physical hardware component in the device model is represented by an object of the PEU class.
The interrupt controller is represented by the InterruptController class. The InterruptRequest class is used to store information about the current interrupt. The same instance is reused by all interrupts. Only one is required for a singlecore system since only one interrupt may be active at any point.

The CPU is treated as a special type of PEU by having the CPU class inherit from the PEU class. It contains additional information that is required for simulating the CPU, such as clock speed and the name of the service that handles interrupts when they occur.

Each PEU is scheduled by its own instance of the TaskScheduler class. The TaskScheduler class is responsible for managing the mapping between the thread references used in the scheduler and the corresponding thread objects in ns-3. This includes executing task switches and preemptions in the ns-3 models when they occur in the scheduler. Additionally, the TaskScheduler class provides an interface for the simulated processes to perform scheduler interactions. A scheduler interaction happens when a thread performs a call to the scheduler, e.g., with a direct call to the scheduler, such as yield to relinquish the CPU, or through a call on a synchronization object which subsequently calls the scheduler.

The Thread class contains the state of one thread of execution in the model. This consists primarily of a stack structure that keeps track of the execution of services. Each entry in the stack refers to a service. When a service is entered, it is pushed onto the stack, and popped off the stack when it exits, in a similar manner as during analysis. The stack element also contains data to keep track of the execution of one service, such as the index of the current event being executed in the service.

**Software Execution Model (SEM)**

A SEM is a merge of the different signatures from a service. Events present in multiple signatures of a service are merged into a single event in the SEM. Branching points occur whenever a condition or state is evaluated. Whenever such a branching point is encountered during the simulation, the condition or state represented by the event is evaluated, and the next appropriate event is selected based on the result of the evaluation. The different branches join on a subsequent common execution event.

The process of creating a SEM is illustrated in Figure 2.5, with two signatures from the same service, as seen in Figure 2.5a. The events $\alpha_0$, $\beta_0$, and $\alpha_3$ are common to both signatures. $\beta_0$ is a branching condition, the rest of the events can be any type of execution event, or even a sequence of execution events. $\alpha_1$ and $\alpha_2$ are events that depend on the condition that $\beta_0$ is branching on. That is, the event to execute depends on the evaluation of $\beta_0$ during simulation. A SEM is constructed from these signatures by merging similar events at the beginning and end of the signature. The resulting SEM can be seen in Figure 2.5b. The three pairs of events that are common to both signatures, $\alpha_0$, $\beta_0$, and $\alpha_3$, are merged into one unique event per pair. Note that the signatures do not have to be of equal length, and that this procedure can be generalized to merge any number of signatures.
Initialization of the processing model before simulation

The processing model is installed on a specific node in the simulation with the use of a helper class, as is the custom in ns-3. This helper class is set up in the simulation script\(^3\), which in turn sets up an instance of the ExecEnv class and connects it to the node object. The simulation script also sets up any other models used during the simulation and schedules any initial events before the simulation in started.

The majority of the setup is done while parsing the device file. During this parsing, the various components of the processing model are set up in a hierarchical order. First the queues, followed by hardware, then synchronization variables, condition variables, and triggers, before the signatures of the services, and finally the threads. The scheduler is set up as part of setting up a PEU, and must be present before synchronization variable objects can be instantiated.

When the scheduler in initialized, it sets up an idle thread, as is required in most systems to ensure that there is at least one runnable thread available at all times. The idle thread is simply modeled as an infinite loop containing a single processing event that processes for a sufficiently long time. The idle thread is dispatched at initialization and is only dispatched by the scheduler again if there is no other runnable thread available.

\(^3\)“script” is the ns-3 terminology for the .cpp file containing the main function and the code necessary to set up the simulation experiment.
Execution of the processing model during simulation

Once the processing model and any other ns-3 models have been set up, the simulation is started using the standard mechanism in ns-3.

The Linux scheduler is simulated using LinSched[18], which uses the scheduler code from the Linux kernel. The events in LinSched are kept separate from the events in ns-3. Using LinSched with ns-3 is achieved by always executing the earliest event from either LinSched or ns-3. Before processing an event in ns-3 a check is done to see if any events in LinSched happen before the next event in ns-3. If the next event in LinSched is scheduled to happen at an earlier time, the LinSched simulation loop is executed until the earliest event next is an ns-3 event. Likewise, if the LinSched simulation code queues an event into ns-3 that will happen before the next LinSched event, then the LinSched simulation loop returns and the next ns-3 event is processed. The event that is processed next by ns-3 may not necessarily be the same as the event that was first in the event queue before the scheduler simulation code was called. By switching back and forth in this manner, it is ensured that the earliest event from either of LinSched or ns-3 is executed, such that time progresses monotonically and synchronously in both LinSched and ns-3.

The simulated threads are dispatched from the scheduler model. When a thread is dispatched by calling the dispatch method in the Thread class, it may either resume processing that was previously interrupted or it may process a new event in the current service. Events in the signature are processed one at a time in a loop until the thread blocks or until it encounters a processing stage. When either of these happen, a condition is set that breaks the thread loop, and the simulation returns to the ns-3 event processing loop where the next ns-3 event is determined and dispatched.

When a thread reaches a processing stage in the SEM, an ns-3 event is scheduled at the time when the processing completes. The time when the processing completes is determined by the speed of the CPU and the number of cycles it takes to complete. This is how the processing delay is introduced in the simulation. The simulation of the service execution will continue when the scheduled ns-3 event is handled.

Preemptions can only occur during processing stages since the other events in the SEM are assumed to be atomic and happen instantly. When a preemption occurs, the remaining processing time is computed and the event that signals the completion of the processing stage is canceled. At this point, the thread relies entirely on the scheduler model since no ns-3 events are scheduled for the thread. Once the thread is dispatched again by the scheduler, a new ns-3 event is scheduled at the time the remaining processing completes.

A trigger describes a point where a corresponding action should be performed in an ns-3 model as a result of an action in the processing model. A trigger may be fired upon reaching a location in a signature, calling a service, or dequeuing a packet. The target of a trigger is stored in each packet.

---

4In this section, event means an event found in the SEM, while ns-3 event means an event in the ns-3 simulator itself.
It is specified during execution of the simulation, prior to the processing model reaching the next trigger point in the signature. When the processing model encounters a trigger, it extracts the target function from the packet currently being handled in the processing model. The target function of the trigger is called, which is typically a function in an ns-3 model. The target function can then specify the target of a subsequent trigger in the packet.

For instance, a call to a service in the protocol stack in a signature may use a trigger to ensure that the corresponding function is called in the ns-3 model for that protocol. This mechanism can be used to call a function for receiving packets in the ns-3 IP protocol stack once a signature performs a call to the corresponding service for receiving IP packets in the device model. The receive function in the ns-3 model would then determine if the destination of the packet is the ns-3 node it is currently on, or if the packet should be forwarded. Depending on which action is taken in the ns-3 model, the next trigger is set accordingly in the packet. The trigger may also be used the other way around, to infer which service to call next in the processing model. Following the previous example, the trigger could be set to either a function to queue the packet in a driver, or to receive the packet in a service on the transport layer in the ns-3 protocol stack. The processing model then uses the mapping specified in the device file to determine which service should be called.

2.3 The Linux kernel

Linux is a general-purpose, multitasked OS, which is used on a wide range of devices, such as servers, desktop computers, laptops, and smartphones. This section contains background information about the Linux kernel that is relevant to the work in this thesis. As the Linux kernel is a dynamic and constantly evolving project, information is easily outdated. As such, this section aims not to provide implementation-specific details, but a general overview of some topics to understand multicore execution and networking in the kernel.

2.3.1 Symmetric Multiprocessing

A multicore CPU contains several processor cores on the same chip. Each core has its own registers and possibly some independent caches. Only one core is enabled at boot time. Once the remaining cores are enabled by the first core, they execute parallel and possibly independent instruction streams.

Multicore systems contain some additional hardware to handle interrupts. On singlecore systems, a single interrupt controller may be used to set up handling of interrupts. To overcome issues with handling interrupts on multicore systems, the interrupt controller is split into two parts. One part is a controller that interfaces with external hardware to handle hardware interrupts, and to handle interrupts between different cores. The other part consists of an interrupt controller local to each core, which handles software
interrupts and timer interrupts.

Interrupts are distributed onto CPU cores by the use of a hardware-based arbitration mechanism. This distribution may be programmable on some platforms. It could be set to always send interrupts to a specific core, or to send interrupts to different cores in a round-robin fashion. The latter mechanism is what is typically used in Linux, with the exception of one optimization: If a core \( c_1 \) receives an interrupt already being serviced by a core \( c_2 \), the core \( c_1 \) will set a flag marking the receipt of an additional interrupt of the same kind, instead of processing it directly. After the core \( c_2 \) completes the handling of the first instance of the interrupt, \( c_2 \) will process the second instance of the interrupt as well. This is done to take advantage of the fact that the instructions for processing the interrupt are already present in the cache of the second core. This approach has the additional benefit of not modifying the cache of the first core.

In a multiprocessor system, there may arise the need to execute code on a different core directly. To achieve this, an Inter-processor interrupt (IPI) may be used. In Linux, they are used to pass a function pointer from one core to another, which allows the other core to execute the function. This is useful to set particular control registers to the same value across all cores, to reschedule another core, or to invalidate the Translation Lookaside Buffer (TLB) using special instructions.

2.3.2 Deferred interrupt processing

Many interrupts only signal the availability of new data and do not need to be serviced immediately. Additionally, some of these interrupts may require expensive copy operations, which should be avoided in Interrupt Service Routine (ISR). To handle these situations more efficiently, the concept of top halves and bottom halves were invented. The top half, the ISR, just acknowledges the interrupt to the hardware and schedules the execution of the bottom half, which does the actual processing of the interrupt. The bottom halves execute in a kernel context instead of an interrupt context, which means that other interrupts may be handled while processing the bottom half, thus improving the responsiveness of the system.

Traditionally, these bottom halves were implemented in such a way that only one bottom half would be executing in the system at any time, even if more than one of them were scheduled for execution. This is adequate for a singlecore system, but it is easy to see that several bottom halves could be processed at the same time on a multicore system. For this reason, softirqs were invented. Any softirq may run on any core at any time, even multiple instances of the same softirq. This allows for instance network drivers to perform copy operations in parallel. Softirqs replaced the traditional bottom halves by implementing the bottom halves as a type of softirq.

One drawback of softirqs is that each softirq must be synchronized with itself because a single softirq can execute concurrently on different CPUs. This is more difficult to program and is excessively complex in many cases. Additionally, softirqs must be specified in a central place in the kernel, making it unsuitable for use directly by drivers. For these reasons, tasklets
were invented, not to replace softirqs, but to be a less complex and more modular alternative. Several different tasklets may execute in the system at one time, but only one tasklet of a given kind executes in the system at any point. This leads to simpler code as the tasklet functions themselves do not need to be synchronized. They are also easier to use in drivers, as they do not require changes to the core kernel code. The tasklets are executed by a softirq, TASKLET_SOFTIRQ, which is enabled whenever a tasklet is enabled. This softirq ensures that at most one instance of a tasklet is active at any time.

2.3.3 Multicore scheduling

The task of scheduling a CPU becomes more complex when dealing with multiple cores. There are two main approaches to scheduling tasks in a multicore system. One is to schedule each core individually, which is done in the mainline Linux kernel. The other approach is to schedule the system globally and assign work to different cores. This approach is present in a kernel patch developed by Con Kolivas.

The operating system must utilize the available hardware resources. The scheduling algorithm should make use of all cores to achieve the highest possible throughput. For this to happen, the amount of work on all cores must be balanced, such that no core is overloaded while another is idle. Linux uses a migration thread for each CPU to offload tasks onto a different core with less load. As such, an execution context may be migrated from one core to another. This includes contexts which are executing in the kernel as well.

2.3.4 Kernel concurrency

The biggest issue with concurrent software is the need for synchronization to prevent undesirable concurrent access to values in memory. In the most general cases, this is characterized by a sequence of read, modify, and write operations, meaning that a value is read from memory, modified, and then written back. This sequence of operations may yield incorrect results if a thread reads the value after another thread has modified it, but before it is written by the first thread. The second thread will then be working on old or inconsistent data, write an incorrect value, or overwrite the result from the first thread.

The easiest and most naïve solution is to avoid having any shared value between threads. In many cases, this is a viable solution. Often, the same variable will exist separately for each core. As long as these variables are not accessed by any other core than the one it belongs to, concurrent read, modify, and write accesses are harmless.\(^5\)

Most hardware platforms support the notion of an atomic operation, which is guaranteed to be safe for use even with concurrent access. However, this only works for scalar values, as concurrent access may still lead to

\(^5\)The at-most-once property may be used to determine if concurrent access is harmless, even for shared variables, but this is outside the scope of this thesis.
inconsistencies on structures with multiple values, even if the operation on each value on its own is atomic. Still, atomic operations are crucial when it comes to implementing more advanced synchronization structures. A lock may, for instance, be implemented using an atomic test-and-set operation.

It should also be noted that many platforms, especially those used for low-powered devices, assume a weak memory model. This means that the effects of memory operations may not be visible to other threads in the same order as they appear in the code. This is due to the reordering of instructions done by the processor during execution in order to avoid stalls where the CPU is waiting for the cache. This holds even for some singlecore, concurrent systems. Memory barrier instructions exist to ensure that the effects of instructions are visible in the correct order between different threads when required, such as for synchronization. The Linux kernel is designed to function on a wide variety of platforms and supplies kernel developers with simple and portable ways of specifying the locations of these memory barriers in the kernel code. Additionally, all the synchronization code, including the helpers for atomic variables, take weak memory models into account to alleviate the programmer of the task of inserting memory barriers manually. It is therefore highly advisable to use the existing synchronization and atomicity constructs in the Linux kernel. This ensures that no further thought is required with regards to the memory model. Reasoning about memory models and the issues brought forward by them is no easy task, and is outside the scope of this thesis.

Originally, the Linux kernel was non-preemptive, and it ran only on machines with singlecore CPUs. There would only be one context in the kernel at any time. When preparing the kernel for use with SMP, what became known as the Big Kernel Lock (BKL) was introduced. Its history and issues are described in [29]. This lock synchronized all access to the kernel, such that there never was more than one context executing in the kernel. It had a re-entrant mechanism, such that the lock could be taken multiple times by the same execution context, and released the same number of times afterward. This provided a working, but highly inefficient way of synchronizing the kernel. This would, for instance, prevent concurrent execution of two completely independent system calls. A less immediate, but perhaps bigger issue, was that it was unclear what the lock was protecting at different times. This made the task of splitting the lock into more fine-grained locks more difficult. The undoubtedly biggest issue with the BKL was the fact that it had a lot of unpredictable effects that led to subtle bugs which were hard to fix. The BKL was removed in version 2.6.39, released in 2011, after having survived for more than ten years.

2.3.5 Synchronization in the Linux kernel

The Linux kernel uses a wide range of synchronization structures, which are discussed in [17], with details of how they are implemented. As this book is old and based on an earlier version of Linux where the kernel itself was non-preemptive, some of the descriptions in the book are inaccurate. Provided here is a brief summary of the most important synchronization
structures, updated to comply with the current state of the Linux kernel, based on code inspection.

Locks are at the forefront of synchronization and are used both to protect data and to implement other synchronization structures. A lock can only be held by one context at any time. Locks are implemented by the use of atomic operations. The most common type of lock is the spinlock, on which the processor will busy wait until it can acquire the lock, using an atomic operation when it becomes available. Only one context can hold the lock; any other context must wait for the context currently holding the lock to release it. Locks protect critical sections, which are parts of code where concurrent access would lead to data corruption or undesirable results.

A semaphore is a more advanced form of synchronization in the sense that it suspends any context waiting for the semaphore until it becomes available again. It is more advanced than a simple lock because it can allow more than one context inside the critical region, determined by a counter in the semaphore. Device drivers often rely on semaphores to block until any number of resources, e.g. packets, arrive and are ready to be processed.

A completion is an alternative to semaphores. It is a simple way to tell a thread when it may resume execution, after some other task has completed. Before its invention, many semaphores were used to immediately block one thread until another thread had finished performing some task. As such, the semaphores were being used for the opposite case from what it is optimized for. Semaphores are optimized for the case where they are open; completions are designed specifically for the case where they will block right away.

To increase the level of concurrency, read and write locks allow multiple contexts to read the same value as long as no other context tries to modify it. If a context wishes to acquire the lock for writing, it must wait until all readers have left the critical region. It will then be granted exclusive access to the critical region. When there is a context waiting for a write lock, no other context may acquire a lock for reading. A special case of this lock is called the Big Reader Lock, which splits the reader lock over each core to optimize for caches. The write lock is still common to all cores.

Another optimization is the seqlock, which is short for sequence lock. It uses a sequence number to check if the data was modified while holding the lock. A reader stores the sequence number upon entering the critical region and checks it again when leaving. If the sequence number is the same, it leaves the critical region. If the sequence number is different, it executes the critical region once more to get the new value. A writer updates the sequence number when modifying the protected data. This leads to a very efficient locking mechanism for readers, but only as long as writes are not frequent enough to cause starvation on the readers.

A very efficient synchronization method for readers is Read-Copy-Update. It consists of a global pointer to the protected structure. When writing, the structure is first copied, and the modifications are done on the copy. Once the update is done, the global pointer is modified atomically to point to the newly updated copy. This has the benefit over seqlocks that it does not force the reader to re-read the data if it has been updated.
2.3.6 Networking in the Linux kernel

The networking subsystem in the Linux kernel is highly modular. It consists of device drivers for different networking devices, a network device interface layer, and protocol implementations. Two softirqs are used for networking, NET_TX_SOFTIRQ and NET_RX_SOFTIRQ. Some drivers may use tasklets in addition. The path of execution can often be difficult to follow, because of the extensive use of function pointers. This section aims to give the reader a basic understanding of how packets are sent through the Linux kernel, with a focus primarily on forwarding.

Socket buffers

The socket buffer structure (struct sk_buff) is one of the most central data structures in Linux related to networking. It contains meta-information about a packet in the kernel, as well as pointers to an associated data buffer containing the packet data. It is allocated by device drivers when receiving packets, or by the kernel code when preparing a packet to be sent. The structure itself is large, but only a few of the fields are used in each layer. Only a brief overview of the data management part of socket buffers is presented here. A more thorough examination of socket buffers can be found at [3].

What makes the socket buffers special is that the data buffer is divided into two sections, one for headers and one for the data. This way, space is reserved for headers to be inserted at lower levels, thus avoiding having to allocate new buffers and copy the data multiple times. When passing through the different layers, only the socket buffer structure is copied; the data buffer is not copied. This simplifies allocation and deallocation.

There are four pointers related to the management of the data buffer: head, data, tail, and end. head and end are fixed and mark the boundaries of the buffer. data and tail denote the section of the data buffer that is currently in use, and may be moved when passing the socket buffer structure between the network layers. Further, the socket buffer structure contains pointers directly to the MAC layer header, the network layer header, and the transport layer header. These are set upon initial processing by the different layers.

There are five basic operations available on the data buffer in socket buffers, and their effects can be seen in Figure 2.6. skb_reserve, seen in Figure 2.6a, is called after allocation of the socket buffer to determine how much space should be reserved for headers, moving both the data and tail pointer such that the requested amount of bytes is present between the head and data pointers. skb_push, seen in Figure 2.6b, is used when adding a new header to the data, and moves the data pointer back towards head. Figure 2.6c shows skb_pull, which is used to remove a header by advancing data towards tail. skb_put, shown in Figure 2.6d, is used when adding data to the end of a packet, and moves tail towards end. Figure 2.6e shows skb_trim, which removes data from the end of the buffer, moving tail towards data. Additionally, it is possible to query the socket buffer to
determine the remaining space before and after the data, named headroom and tailroom respectively. These areas of the data buffer are marked in Figure 2.6f. The functions skb_headroom and skb_tailroom return the size of these two regions respectively.

**Packet flow through the Linux kernel**

When a frame is received on a system it is queued on the NIC. Depending on the driver, it either polls the NIC repeatedly in a kernel thread to check for a new frame, or the NIC issues an IRQ to signal the reception of a frame. Polling is rarely used by itself because it wastes CPU cycles. However, some drivers combine the two approaches. The NIC is in a sleep state, and issues an interrupt whenever at least one frame has arrived. The driver then polls for any additional packets, before putting the NIC back into a sleep state.

Upon receiving an interrupt, the OS executes the ISR. The ISR may wake up a kernel thread through an operation on a synchronization object, such as an up operation on a semaphore, or it may schedule a tasklet. In both cases, a deferred context processes the frame. The interrupt handler then returns. When returning from an interrupt in Linux, any softirqs and tasklets\(^6\) that were activated are executed. Further, the scheduler is called to check if any new thread should execute, such as a thread with higher scheduling priority.

The deferred context then initiates the transfer of the frame from the NIC into main memory. This operation depends heavily on the hardware layout, as the NIC may either be connected to the main memory bus, or to a separate bus with a host controller attached to the main memory bus. In either case, a DMA controller is typically used to perform the transfer in parallel. The kernel thread or tasklet typically waits on a completion until the transfer is finished. On systems with no DMA controller, the CPU must perform the transfer itself, but this is rarely done in practice except on very simplistic systems because of the excessive waste of CPU cycles.

Once the DMA transfer is completed the driver may remove any device headers using skb_pull. The frame is then handed to one of the device-specific ingress queues, or backlog in Linux terminology, by calling netif_receive_skb. This queue is processed by the softirq NET_RX_SOFTIRQ. This softirq calls process_backlog, which hands the packet over to the appropriate protocol stack.

The receipt of IPv4 packets is handled by ip_rcv. This function verifies that the packet has a correct checksum and sends it through the various functions that determine the routing to the destination of the packet. Once the route to the destination has been determined, a function pointer in the socket buffer structure, dst, is set to the function that directs the packet onwards. For packets terminating on the system, dst is set to ip_local_deliver and the packet proceeds up the network stack. Packets that are to be forwarded are sent to ip_forward. ip_forward does some additional checking, such as to determine if the packet has exceeded its\(^6\) Recall that tasklets are executed by a softirq.

---

\(^6\)Recall that tasklets are executed by a softirq.
Figure 2.6: Socket buffer operations.
Based on illustrations from [4].
Time-to-Live (TTL). The packet is then sent to `ip_send` through another function pointer. `ip_send` will fragment the packet if the device reports that the packet is too large, and then call `dev_hard_start_xmit`, which in turn calls a driver-specific function that starts transferring the frame to the NIC. This consists of one or more DMA transfers similar to the handling of incoming packets.

### 2.4 The Android platform

The work in this thesis is not Android specific, but since the implementation has been done on an Android platform, a short introduction is given to readers unfamiliar with the platform.

Android is an operating system and a platform intended for use on mobile devices such as smartphones and tablets. It is based on a modified version of the Linux kernel. It is open-source by nature, intended to be used by multiple independent device manufacturers. When porting Android to a new device, the manufacturer will typically supply the drivers required for the new hardware, a bootloader, a Linux kernel with any required changes to support the hardware platform, any additional code required to support the Android platform, and in some cases add extra functionality to differentiate their product from other Android devices. The remaining software is provided by the Android Open Source Project (AOSP), currently developed by Google. This means that device manufacturers spend less time on making software to support their device and that an application ecosystem is already present. Android allows application developers to create applications targeting a wide range of devices without having to consider all possible hardware configurations by providing a uniform software platform. This includes APIs to make use of cameras, sensors, audio, and other hardware devices in applications.

The device manufacturers supply the Linux kernel for their device. As such, the code for these kernels is typically not readily available. Additionally, most devices have locked bootloaders, meaning that they are not able to boot any other kernels than those provided by the manufacturer itself. However, Google partners with manufacturers to create phones and tablets under the Nexus brand. These devices have bootloaders that can be unlocked, and the kernel code is available from Google. As such, these devices are ideal for performing experiments on mobile devices that require changes to the kernel. The Android platform development kit also contains several useful tools such as the Android Debug Bridge (`adb`), which provides functionality for connecting to a shell on the device, as well as sending and receiving files. Another utility for use with the Android devices is `fastboot`, which can unlock the Nexus device bootloaders, perform one-time boots of boot images, and permanently flash a boot image to the device. A short guide on how to compile and boot a kernel for a Nexus phone can be found in Appendix A.

The Android platform consists of additional abstraction layers for hardware such as cameras, various sensors, audio, as well as a layer for the
applications, which are implemented through a separate APIs. The work in this thesis does not relate to these abstraction layers, so they are not covered here.
Part II

Design
Chapter 3
Design of Multicore Tracing, Analysis, and Simulation

This chapter outlines the changes required to enable simulation of multicore devices. First, an argument is made for why and how the existing concepts in the methodology can support multicore execution. Next, the requirements and design of each component in the modeling and simulation workflow is presented, with an emphasis on how to account for multicore execution. The tracing framework is presented first, followed by the processing model, and finally, the analysis tool. The focus is on the differences between the design of the existing component and the new one, where multicore execution has been taken into consideration.

3.1 Methodology concepts for multicore execution

The concepts in the methodology require no changes to support the notion of multicore execution. This is supported by two key factors:

First, having multiple PEUs is already supported by the existing concepts. PEUs support the concept of execution not only on CPUs, but also on DMA controllers and NICs. Parallel execution on a DMA controller and a CPU is, fundamentally, similar to parallel execution on two different CPU cores.

The second factor is that execution does not change between two equivalent PEUs. That is, a service executing on a core $c_1$ will behave identically if executed on core $c_2$ under the same conditions, assuming that $c_1$ and $c_2$ are equivalent types of PEUs. PEUs are equivalent if they can be used for the same purposes, such as two CPU cores. A CPU core and a DMA controller are not equivalent. A CPU may perform the same operations as a DMA controller, but the opposite is not necessarily true. Two equivalent PEUs may have different clock speeds. Even if $c_2$ is a core with a slower clock speed than $c_1$, this only impacts the time it takes to execute the same number of instructions.

For some exotic cases, execution might differ on two equivalent PEUs, i.e., that executing a service on one PEU has a consistently different behavior compared to the same service executing on another PEU. This may be
modeled in the framework by introducing the ID of the PEU as a state variable in the service.

Based on these observations, the concepts in the methodology is left unchanged from what is described in [27].

3.2 Tracing events on multicore systems

This section focuses on how to trace execution on multicore devices. The first part is a short outline of the requirements that a tracing framework must meet in order to support multicore tracing. The second part describes how these requirements are met, and in what way the new tracing framework differs from the previous implementation.

3.2.1 Requirements

A tracing framework for tracing multicore execution should meet the following requirements: First, it should have a low overhead of trace probe execution. As with any measurement, it is inevitable that the act of measuring will impose some level of interference. However, this should be minimized as much as possible. Second, it should be properly synchronized. This is necessary to avoid corruption of trace data due to the possibility of concurrent execution of trace probes. Third, it must take parallel execution of the same service into consideration so that it is possible to handle this appropriately during analysis. Finally, additional execution events that are specific to SMP systems may require tracing. Tracepoint functions and trace probes for these events must be created.

3.2.2 Design

An important decision early on is whether to keep the existing tracing framework or to make use of something that is already known to work properly with SMP. LTTng[20], Linux Tracing Toolkit next-generation, is a strong candidate, but there are several good reasons to not use this approach. Primarily, little work is required to make the existing tracing framework usable with SMP. Furthermore, the goal of the original tracing framework is to maintain control over the tracing process as much as possible, for instance, to avoid tracing while writing the contents of the trace buffer to the file. To the best of the authors knowledge, LTTng does not support this use case directly at the time of writing. However, it is possible to set up LTTng to trace until the buffer is full, and then flush the buffers by some kind of external input through a snapshot feature. While this is possible in practice, it is an approach that requires knowledge or some calculated guess of when the buffer is full, or close to becoming full. Such an approach does not appear to be viable in the long run.

LTTng employs highly sophisticated lockless algorithms[19] providing segmented buffers for each CPU core, which may be more efficient than the more rudimentary single-buffer approach taken by the original framework.
Given that the simulation is able to account for the overhead of executing the trace probes, this is assumed to not be a major issue. A completely different format on the trace output would also require major changes to the analysis tools. Considering all of this, LTTng does not seem to provide major advantages over the existing framework.

The existing tracing framework is made for an earlier version of the Linux kernel. In the newer version of the Linux kernel used by the Galaxy Nexus, the definition of the kernel trace points has changed slightly, requiring some minor changes to the framework. The existing trace probes require no additional synchronization since they are already properly synchronized to account for execution during interrupts. Proper synchronization is achieved by using atomic variable types implemented in the kernel.

A service may execute in parallel on different PEUs in an SMP kernel. To separate the different executions in the trace file, the ID of the CPU where the tracepoint is executed must be recorded for all execution events captured by the tracing framework. Additionally, since some LEUs may be migrated from one CPU core to another, tracepoints are required to capture this event. The tracing of migrations is required for analysis only. Migrating a task from one CPU core to another is a decision made by the scheduler, which is explicitly modeled in the simulator, as explained in Section 2.2.7.

The modified trace probe function can be seen in Figure 3.1. This outlines the execution of the trace_entry function that can be seen in Figure 2.2. Once tracing is activated, each execution of the trace probe stores once trace entry in the kernel trace buffer. Tracing is deactivated when the buffer is full. Any subsequent execution of the trace probe will simply return without any further action unless tracing is re-activated. The user-space process is awakened by sending a message on the Netlink socket. It starts sending messages back to the tracing framework, requesting a trace entry from the kernel buffer. The tracing framework replies with the entries from the
trace buffer until the buffer is empty. Once empty, the user-space process is blocked on a read operation on the socket, and tracing is subsequently re-activated.

Apart from the additional field in the trace entry containing an identifier for the CPU core that executed the trace probe, the only change from the singlecore tracing framework is how the user space program is notified of the buffer being full. Originally, the trace probe would deactivate tracing and schedule a tasklet that would send the wake-up message on the Netlink socket. In the kernel used by the Galaxy Nexus, this approach does not appear to function in a stable manner due to what appears to be either a race condition or a bug in the kernel. In some cases, the tasklet would be scheduled for execution, but the softirq that handled the tasklet would remain unscheduled. This may be an effect due to a weak memory model. A thorough investigation into this issue has not been done due to time constraints and is left as an open issue.

The approach used in this design is to execute the function which sends the wake-up message to the user space program on CPU0. This is done either directly if the trace probe is executed on CPU0, or through an asynchronous IPI handled by CPU0 if the trace probe is not executed by CPU0. This approach is significantly less elegant than using a tasklet, but avoids the issue that was mentioned above. It may have a slightly negative impact on the performance because CPU0 is interrupted. Since this only happens when the trace buffer is full and no further events are recorded, it should have no negative impact on the data captured during the trace. The function executed by the IPI executes quickly, and given enough time before re-activation of tracing when the buffer is empty, any negative impact from this approach should be negligible.

3.3 Enabling simulation of multicore execution

This section details the changes required to the implementation of the singlecore processing model extension made for ns-3. As with the previous section, a set of requirements is first presented, before explaining the design of the new multicore processing model for ns-3 and how this differs from the existing singlecore processing model.

3.3.1 Requirements

Some guidelines should be established for the design of the multicore processing model. First, the majority of the existing code should be kept intact. Code executing on one CPU should behave identically when executed on a different CPU. If it is designed to run differently on different CPUs, this could be modeled by a state variable as previously explained. Second, a thread should have the ability to be executed on any core. While a scheduler might impose a restriction on a thread, such as processor affinity, such a restriction should be enforced by the scheduler model and not be imposed by the implementation of the multicore processing model. Third, an interrupt
should also have the ability to be processed by any core. A model of the interrupt distribution should be responsible for directing an interrupt to a particular core. As a consequence of these requirements, any core should be able to perform packet processing.

3.3.2 Design

This section details the changes made to the existing singlecore processing model. First, a comparison of the architecture of the singlecore and the multicore processing model is given, followed by the different design challenges are presented. In particular, how to implement multiple CPU-type PEUs, how to simulate multiple LEUs at the same time, why LinSched must be replaced, the design of the scheduler model that is used instead, and how to handle the signatures in a multicore processing model.

Architectural changes

The architectural changes to the ns-3 processing extension can be seen in Figure 3.2. This design incorporates the changes required to model a multicore system. The HWModel class has references to multiple CPU instances, and one TaskScheduler governs multiple CPU objects. The LinSched code is replaced with the RR Scheduler class which implements the new task scheduler. This is explained later in this section. Finally, the InterruptController is extended to handle multiple InterruptRequest instances.

Multiple CPU PEUs

The existing processing model was designed to model software execution on devices with only a single processor, accessed through a pointer in the HWModel class. This pointer needs to be replaced by a list such that all CPUs in the system can be accessed. Additionally, each CPU must have a unique ID. Scheduler interactions must receive information about which core the interaction happened on to identify which task should be switched out if needed. For this reason, all methods implementing scheduler interactions must have an additional parameter to determine the ID of the CPU where the scheduler interaction is executed.

In the existing implementation, each PEU, including the CPU, has its own scheduler, as can be seen in Figure 2.4. In an SMP system, there is most commonly one scheduler governing multiple CPUs. This requires changes to how the PEUs and task schedulers are set up in the processing model. Under the assumption that all CPUs on a given device are governed by a single scheduler instance in the kernel, all CPUs can be assigned the same scheduler as the first CPU that was initialized in the simulation. The remaining non-CPU PEUs should have their own scheduler instances separate from that of the CPUs. The scheduler for non-CPU PEUs are simple schedulers that run a finite number of threads in parallel.
Multiple parallel LEUs

The existing singlecore implementation considers only one process to be running at any time, which is represented by a single variable storing the Process ID (PID) of the currently executing process. This PID is used to implement checks for preemption by comparing the PID of the currently running process before and after the scheduler interaction to see if it has changed. By comparing the old and new PID, the processing model can determine if the currently simulated thread should continue to be simulated, or if it should be preempted. If the thread is preempted, the new thread replacing it is dispatched.

In an SMP system, several processes may be active at any moment. An array of the IDs of threads that are currently executing in the simulation must be maintained. This array is used to check for any preemptions that may happen during a scheduler interaction. Any running process may potentially be preempted, not just the process initiating the scheduler interaction. Checking for preemptions is handled in the multicore processing model by making a copy of the array containing the PIDs that are executing before the scheduler interaction. This array is compared with a new array of the executing PIDs after the scheduler interaction has completed. Different values in the same index in the two arrays signals the preemption of one thread and the dispatch of another. Any process that is no longer executing is preempted, and new tasks are activated. Additionally, this implementation checks if the service that initiated the scheduler interaction is preempted, which tells the processing model to stop simulating the execution of the service.

Execution of services in the singlecore processing model is implemented by handling one execution event at a time. Processing delays are introduced by scheduling events to resume processing of subsequent events in the future of the simulation time, based on statistical data that is calculated during analysis. The discrete-event nature of ns-3 allows parallel execution to be simulated by handling the execution events for one PEU until a processing stage is encountered or the thread is blocked. Then, ns-3 selects the next ns-3 event to handle, which may be a sequence of execution events on another PEU. The time is only advanced if the next set of execution events happen later in the simulation time. This gives the appearance of parallel execution in the timeline of the simulation, even though each PEU is simulated in turn. As such, the multicore processing model itself does not execute in parallel in the simulator. Rather, it simulates parallel execution of LEUs.

An interrupt, also a type of LEU, may be distributed to any CPU core. As such, the model of the interrupt controller must be extended to account for multiple interrupts executing at the same time, at most one per CPU core. Interrupts are initiated by an ns-3 model during simulation, as is done in the singlecore processing model. The model of the interrupt controller is simplified, requiring the code initiating the interrupt to specify the receiving CPU. This could be extended in the future by implementing various interrupt controller policies, such as a round-robin distribution of
interrupts and options to route certain interrupts to a subset of the available CPUs.

Replacement of LinSched

LinSched[18], a scheduling simulator using actual Linux kernel code to simulate scheduler interactions, is used in the existing implementation. This approach has the benefit of giving an extremely precise model of how tasks are scheduled in the device.

Implementation-wise, it has several disadvantages. It is no longer officially maintained, and it makes use of an old kernel version. It is only intended to be executed as a single instance, which means that simulating execution on two or more nodes requires LinSched to be dynamically linked, resulting in several scheduler instances in memory, which makes implementing this approach far more difficult. LinSched is lacking documentation, which means that setting it up is difficult, especially when the goal is to simulate multiple cores. Furthermore, mapping a CPU simulated inside LinSched to a CPU inside the ns-3 processing model is not trivial. This mapping is made possible for a singlecore device model by the existing singlecore processing model[26]. However, it is difficult to apply the same approach to simulate a multicore system. An alternative option is to use one LinSched instance per CPU, but this greatly increases the difficulty of migrating tasks between the different scheduler instances since they maintain their own set of PIDs. This approach is also conceptually wrong. Although a scheduler, such as the one found in recent mainline Linux kernels, maintains separate queues of runnable processes for each logical CPU, it is still organized as a single entity in the kernel. This approach would also require multiple instances of LinSched to be linked dynamically, both increasing the simulation overhead per CPU per node considerably, as well as resulting in the same implementation difficulties as mentioned above.

Ultimately, making LinSched usable for the multicore processing model appears to be too difficult compared to the benefits. There is no known successor or alternative to LinSched to the authors knowledge. This leaves no other option but to implement a simpler scheduling model. By making this new scheduler model use the same superclass as the LinSched wrapper, the system can be extended with more sophisticated scheduler models in the future should it be required.

The new round-robin task scheduler

The new task scheduler that replaces LinSched is a basic round-robin scheduler. It uses a single queue for all threads. Unlike LinSched, which uses actual scheduler code to model the scheduler, the new model emulates the fact that each thread should receive the same amount of time to execute on the CPU. Thus, it uses a very short scheduling quantum to achieve this, i.e., it switches very frequently between processes to ensure that they get
to execute for the same amount of time. The impact of this change on the performance of the processing model is investigated in Experiment 4.

The scheduler is executed by periodic ns-3 events. It reschedules all tasks in the system on every tick, as well as at least one task when a scheduler interaction prevents the currently executing process from continuing, such as being blocked on a synchronization variable. The task that was currently running on a PEU is preempted and put at the end of the queue. Each preempted CPU is assigned a new task to execute by taking a task out from the front of the queue. If not enough tasks are available, idle threads are scheduled for the remaining CPUs. There is a single queue maintained for all CPUs in the system. Although the execution of the scheduler would add to the processing delay in practice, this is not modeled in the simulation. It is assumed that any decently implemented scheduler executes reasonably quickly to maintain a low overhead when executing on an actual device. Such a delay could still be modeled in the future.

Signatures

Each signature in the original processing simulation is tied directly to a specific PEU. This works fine in a singlecore model as there is only one CPU to execute the signatures. For an SMP system, there are two options: To keep the signatures tied to each PEU, or to allow one signature to be executed on PEUs of the same type. Having each signature tied to a specific PEU means that many signatures that are mostly equal will be duplicated for each PEU, adding redundant information to the device files. One of the assumptions made earlier was that software execution behaves identically when executed on different CPU cores. Therefore, it is generally not necessary to distinguish between different CPUs in the signatures. For other types of non-CPU PEUs, the case might not be the same. There is however not any obvious practical case where this might apply, nor is it immediately required to model the most commonly available SMP systems. The existing implementation of the processing model is changed to allow for the signature of a service to be simulated on any of the available CPUs on a particular node in the simulation.

3.4 Analysis of multicore traces

This section is about the analysis of multicore traces. It is presented last, because the design of the analysis tool is influenced by the design of the multicore processing model presented in the previous section. After discussing the requirements for such a tool to handle multicore traces, the major differences between the analysis of traces from singlecore and multicore devices is presented, and the design of how these differences are handled. The overall design of the tool itself is briefly discussed.
3.4.1 Requirements

The analysis tool should be capable of producing signatures from a trace file generated on a multicore system. This may include handling events that are not defined for execution on singlecore devices. Further, it should produce the same output for a singlecore trace file as the existing tool. Also, analysis may be repeated several times during the development of a model, as explained in Section 2.2.2. For this reason, the tool should be reasonably efficient. Finally, the tool should be extensible so that new events may easily be defined in the future and handled appropriately during analysis.

3.4.2 Design

This section presents the design changes from the analysis tool for singlecore traces to the tool for analysis of multicore traces. The analysis tool itself is reimplemented; this is discussed first. Then, two major differences between analysis of a singlecore trace file and multicore trace file are given. That is, how to handle the potential for parallel execution of the same service on different PEUs, and how to handle the separate cycle counters for each CPU.

New implementation is Go

The original trace analyzer is written as a Python script, taking a trace output from a singlecore device and converting it into a set of signatures that may later be assembled into a device file. This script is not intuitive to read or understand, because lists are used instead of classes and objects, making it difficult to grasp how data flows through the analysis. Combined with the fact that the underlying methodology is complex, it is not straightforward to extend it to work for analysis of traces from multicore devices without possibly breaking existing functionality.

The alternative is to write a new analysis program from scratch. While this is undoubtedly a major undertaking, it has several advantages. Since Python is an interpreted language, it can be significantly slower than native code. Further, writing a new analyzer gives valuable insight into the process of analyzing a trace file, which is of great help when extending the analysis to work for SMP systems as well. Rewriting it also has the benefit of introducing properly named objects and fields to replace the lists in the old script, where fields were accessed by the use of indices. Properly named types and structures would ideally also make the analyzer more readable to others in the future, should they need to modify or extend the analyzer.

Thus, the requirement is for a compiled high-level language to achieve both good execution speed and readability. While the use of C or C++ would likely produce the fastest code, these are languages that can be time-consuming and error-prone to develop programs in. Additionally, these languages lack the extensive and useful standard libraries often found in high-level languages. Java is a widely used language, and one possible alternative. Although Java has a well-featured standard library, it has the unfortunate requirement to keep all code within classes, which is unintuitive
for a utility program. Additionally, its heavy focus on exceptions, especially when dealing with I/O, generally makes it unfeasible as a language for smaller utility programs.

The language ultimately chosen for the new analyzer is Go[5], sometimes referred to as Golang. It is chosen due to its high-level syntax, despite compiling to native bytecode. Although not yet as common in use as the aforementioned languages, the syntax is easily readable to anyone familiar with C or Java. Additionally, it has built-in types for dynamic arrays and hashmaps, which are frequently used in the analyzer. The standard library is well-featured, and the language has built-in constructs for concurrent programming. Go is also object-oriented, although it is realized syntactically in a somewhat unconventional manner.

The new analyzer features the same three stages as the previous one. The first stage is concerned with extracting data about the execution of the different services from the trace file. The different types of trace entries have a well-defined effect on the state of the analysis. In the old analyzer, this logic is implemented inside one large loop. This has been split up in the new analyzer into separate functions, one for each type of trace entry. A pointer to the appropriate function to handle the event is determined by looking up the name of the event type in a hashmap. This should make it easier to add new event types to the analyzer if required, such as for the migration events. The second and third stage of the analyzer remain structurally the same as the original analyzer.

**Management of service stacks during analysis**

The original analysis script maintained a stack for each of the currently executing services, which is seen in Figure 2.3. Some services may execute on different CPUs at the same time, such as softirqs. For this reason, there must be a stack per CPU for each service to separate the different executions from each other. As the signatures themselves are not CPU-specific, equal executions from all CPUs are grouped together into a single signature. The addition of one stack for each CPU requires task migrations in the kernel to be traced. When the analyzer encounters such an event, the service is moved from the stack of the CPU that was previously executing the service to the stack of the CPU the task was moved to.

**Multiple cycle counters**

Cycle counters on the CPUs are used to measure the number of cycles spent processing between execution events. The values of these counters are stored in the trace entries during tracing. The cycle counters wrap around periodically in practice, so the values of the cycle counters are used to infer a linear, monotonic time by looking at the difference in the cycle counter value between the previous and current event.

Different CPUs on a device may have independent cycle counters. These cycle counters are not synchronized between CPU cores. As such, events on different CPUs happen on separate timelines. This must be accounted for
when calculating the delay from a previous event during analysis. Further, this must also be taken into consideration when a LEU crosses between the timelines. This occurs during a migration and potentially when receiving a notification of the completion of work offloaded to another PEU, such as a DMA transfer.

The case for migrations requires little consideration, as a migration may only happen after the LEU has been preempted. No additional processing cycles are added to an execution case for a LEU between preemption and dispatch of that LEU. As such, the data related to the LEU may simply be moved to the stack for the CPU that it has been migrated to. There are no issues in crossing to the other timeline in this case.

The time it takes to complete work offloaded to another PEU from the CPU is inferred by subtracting the time the request was initiated from the request completes. If the event signaling the completion of offloaded work is received by a different PEU than the one that initiated the work, it is not possible to infer the number of cycles spent processing on the PEU performing the offloaded work in the same manner. This is solved by only considering the case when the completion event is received by the same PEU that initiated the work, e.g., when the CPU that initiates a DMA transfer also handles the completion event for that transfer. Otherwise, the case is discarded. This discards some valid execution cases, but it produces a correct result. Correctly solving this issue in a more elegant manner is left as an open issue. This can be done by introducing an event that records the cycle counter on all CPUs.
Figure 3.2: Extended ns-3 multicore processing model architecture. Changes from Figure 2.4 highlighted in red.
Chapter 4

Analysis and Instrumentation of the bcmhd Driver

The goal of this chapter is to provide insight into how the methodology can be used to model this driver, and provide guidelines on how to approach and model other drivers as well. Modeling a driver requires a thorough understanding of how the driver works. This chapter details the analysis work done on the bcmhd driver to be able to create the model of the driver, which makes out the majority of the device model. The focus of this chapter is on instrumenting and modeling the bcmhd driver in the Galaxy Nexus smartphone used in the experiments in Part III, and how the bcmhd driver operates when forwarding packets.

WiFi and Bluetooth connectivity for the Galaxy Nexus device is provided by a SWB-B42 chip containing a BCM4330 NIC. This is an integrated circuit designed by Broadcom, who has also made the Linux kernel driver for it.

4.1 Background

The NIC in the Galaxy Nexus device is connected to a Secure Digital I/O (SDIO) bus. SDIO is a generalization of the I/O interface used for SD memory cards. SDIO allows external devices to be connected by using an SD slot or internal devices, as in the case of the NIC in the Galaxy Nexus smartphone, to be connected using the internal SD bus. This hardware layout can be seen in Figure 4.1. The NIC is connected to the SDIO bus which is managed by an SDIO controller. This controller also connects to the main memory bus. The DMA controller is used to transfer data from the SDIO controller to main memory on behalf of the CPU.

The driver named bcmhd (Broadcom dongle host driver) is used to enable wireless networking for the Galaxy Nexus device. It is largely based on an earlier driver used in the Nexus 1 smartphone, BCM4329, which was open-sourced to be included in the Linux kernel for the Nexus 1. The bcmhd driver does not exist in the mainline Linux kernel. However, it does share many similarities and may have been the basis for the bcmfmac driver that is present in the mainline Linux kernel.

Wireless NICs can be separated into two categories: Those that, like
the BCM4330, implement the Medium Access Control (MAC) layer in the firmware, and those that expects this to be implemented in software. Implementing the MAC layer in firmware offloads MAC layer processing from the CPU and reduces the number of DMA transfers considerably. It also has the benefit of making the NIC easier to integrate for new platforms, making it ideal for devices such as smartphones.

The driver code comprises of 26 files in the kernel source, but only two files are of interest in this thesis: dhd_linux.c and dhd_sdio.c. The former deals with the interface to the Linux kernel, the latter communicates with the NIC over the SDIO bus.

### 4.2 Working modes

The bcmdhd driver has two different working modes, threaded and tasklet-based. The mode to use is selected at compile-time. Defining the preprocessor macro `DHD_THREAD` selects the threaded mode, which is the mode used in the Galaxy Nexus smartphone. The tasklet-based mode is used if this macro is not defined. Both modes of operation share the majority of the code. The difference lies in how the execution ends up at the shared code. This shared code roughly corresponds to the call to `dhdsdio_dpc` in Figure 4.2. No interrupts are received from the NIC when the shared code is executed; it will be rescheduled as long as there are more packets to be sent or received. The NIC signals the presence of additional packets over the SDIO bus in this case.

In the tasklet-based mode, the tasklet is scheduled by an ISR upon receiving a new packet. The shared code is called following the subsequent activation of the tasklet. Sending a packet also schedules the tasklet.

The threaded mode uses a kernel thread in place of the tasklet. A semaphore is used to synchronize the execution of the thread. The execution of the driver kernel thread can be seen in Figure 4.2. In the threaded mode, the ISR performs an up-operation on the semaphore instead of scheduling the tasklet. This activates the kernel thread which performs

---

1 Found in the directory `drivers/net/wireless/bcmdhd`.
4.3 Overview of driver thread execution

This section provides an overview of the execution of the driver thread. This is a necessary prerequisite to understand how the driver is modeled and how the different services in the model are connected.

A sequence diagram of the driver loop when forwarding packets can be seen in Figure 4.2. The loop is part of the dhd_thread_dpc function in the file dhd_linux.c. This loop is referred to as the outer loop. The contents of Figure 4.2 corresponds to the body of the outer loop.

The driver thread performs a down-operation on the semaphore. This blocks the driver thread until a packet is available to be received, or a packet to be sent is available from the kernel. The thread is activated immediately once an up-operation on the semaphore is performed by another thread or interrupt. This is due to the aforementioned scheduling class of the driver thread. The thread calls dhd_bus_dpc in dhd_sdio.c which in turn calls dhdsdio_dpc.

Packets are transferred from the NIC to main memory by calling...
Figure 4.3: Sequence diagram of driver thread function dhdt_dpc_thread without early tx.

dhdsdio_readframes, which transfers one or more packets in a loop. This loop is referred to as the receive loop. Inside this loop is first a call to dhdsdio_sendfromq, which transfers one packet from the kernel to the NIC using DMA if there is one or more packets available to be sent. After transferring a packet from main memory to the NIC, the driver sets up and issues the commands for the DMA operations to transfer a packet from the NIC to main memory. Following the completion of the DMA transfers the packet is delivered to the kernel through the dhd_rx_frame function. The receive loop is repeated up to 30 times following a budgeting mechanism in the driver. For each iteration of the outer loop in the driver, the driver may receive up to 30 packets and send up to 20 packets. If there are still packets left to send or receive the driver thread performs an up-operation on the semaphore, thus enabling a new iteration of the driver loop. Otherwise, the driver notifies the NIC that the thread is entering a sleep state, so that the NIC issues an interrupt if any new frames are received.

The aforementioned effect, where one packet is sent before a new packet is received, is referred to as “early tx” in this report. When removing the code that causes the early tx behavior, the driver operates differently. In Experiment 3 the effect of removing the code that causes the early tx behavior is investigated and compared against the behavior when the early tx code is present.

A sequence diagram for the execution of the driver without the early tx code can be seen in Figure 4.3. The outer loop remains unchanged. However, no call is made from dhdsdio_readframes to dhdsdio_sendfromq. Instead, dhdsdio_sendfromq is called from another loop below the receive loop. The reason this second loop is never called when doing forwarding with early
tx is because of the budgeting mechanism in the driver. The driver will always send up to one packet to the NIC before receiving a frame when the early tx code is present. This is by itself not affected by the aforementioned budgeting mechanism, but it counts towards the budget for sending frames. Thus, there is no budget left for any additional packets to be sent when the early tx code is present.

### 4.4 Instrumenting and modeling the driver

This section provides an overview of the process of instrumenting and modeling the driver. A full overview of the instrumentation in both the kernel and the driver can be found in Appendix B. The purpose of this section is to provide insight in how other drivers may be modeled using the same methodology.

The tracepoints for the driver can be divided into three classes:

- Model-specific tracepoints,
- behavior-specific tracepoints, and
- measurement tracepoints.

The model-specific tracepoints represent the static part of the device model. These tracepoints divide the software into the different modeled services, specifies packet queues, and define the logical flow of both execution and the packets through the model. These are the first tracepoints to be inserted since they provide the outline on which the model is built. The behavior-specific tracepoints are added to capture how the different behaviors depend on specific contexts. Queue conditions, state variables, and operations on synchronization objects are captured by tracepoints in this category. Measurement tracepoints are used during the experiments to record the time when the execution of code handling a packet is at a certain point. They are not related to modeling or analysis, and are not present in the signatures. They are only used during real-world experiments for measurements on the device.

A diagram for the final model of the driver can be seen in Figure 4.5. Each ellipsis corresponds to a service in the model. The service `dhd_dpc_thread` corresponds to one iteration of the driver loop. The rectangles correspond to a queue. The diamond corresponds to the synchronization variable in the model, in this case, a semaphore. The dashed ellipses are part of the pseudo-service used the combination of the four functions that forward the packet and sends it back to the driver. The black arrows correspond to intra-LEU service calls. Green arrows are operations on synchronization variables. Red and blue arrows are operations on packet queues, dequeues and enqueues respectively.

#### 4.4.1 Model-specific tracepoints

It is natural to start with the outermost packet-handling service when modeling a network driver. This is the function that is activated by a tasklet
in a tasklet-based driver and is referenced when setting up the tasklet. Thread-based drivers may use kernel_thread to set up a thread.

In the bcmdhd driver, the dh_dpc_thread function is the outermost packet-handling service which, in practice, is an infinite loop. However, modeling this as a loop would make it impossible to execute in the simulator. Instead, it is modeled by using service entry and exit tracepoints at the start and end of the loop body and by marking the thread as infinite in the device model. The service will be called again once it exits during the simulation.

Identifying packet-handling services called directly or indirectly by the outermost service is the next step. Most drivers have a function to read one or more frames from the NIC. This function should be present in the call graph from the outermost service. In the bcmdhd driver, the function dhdsdio_readframes is used to transfer one or more packets from the NIC. The service is not called directly from the outermost service. However, the function calls in-between are insignificant enough to not be modeled as services. This is generally the case for functions that contain no modeled state variables.

The dh_drx_frame function is used by the driver to deliver the received frames to the kernel to schedule a softirq to process it. This function is useful to model as a separate service for two reasons. First, it performs several actions that are not related to the transfer of a frame itself. Second, its behavior is independent to that of the dhdsdio_readframes function, i.e., its behavior is affected by different state variables than dhdsdio_readframes. Having dh_drx_frame as its own service both reduces the size of the dhdsdio_readframes service and the scope of any context or state variables inside the dhdsdio_readframes service.

In general, a nested service should be modeled explicitly if it is affected by a variable that the surrounding service is not affected by. This will greatly reduce the number of contexts for the surrounding service. Figure 4.4 shows two services, \( S_1 \) and \( S_2 \), each affected by variables \( C_1 \) and \( C_2 \) respectively. The variables affect the behavior of only one of the services, not the other. If only service \( S_1 \) is explicitly modeled, the total number of contexts is \( |C_1| \cdot |C_2| \), where \( |C_1| \) and \( |C_2| \) are the number of different values that can be assigned to each variable. When modeling \( S_2 \) explicitly, the total number of contexts is \( |C_1| + |C_2| \).

As for packets being sent to the driver from the kernel, it is useful to look for the driver function called by dev_hard_start_xmit.\(^2\) It is also possible to find a reference to the function in the netdev_ops field of the net_device struct. This struct is used to inform the kernel of the driver functions it needs to call for various operations such as initiating the transmission of a frame. The function that initiates transmission of frames to the NIC in the bcmdhd driver is dh_dstart_xmit. The function could be instrumented directly, but it is better to instrument the function pointer call from dev_hard_start_xmit instead as the service entry and exit because this will work for any driver.

Instrumenting the code that delivers packets to the kernel presents

\(^2\)xmit_one in more recent kernels.
a challenge since the function called via the function pointer in
__netif_receive_skb depends on the packet type and settings defined
by the kernel at runtime. This is modeled by using a dummy service with
the service name 0. The dummy service includes the execution of all the
different functions called from that point. The simulation will use a trigger
to infer which service to call. This has the added benefit of supporting
models with multiple protocols where it is required, which may be done
by setting the function to call at simulation-time according to the type of
the packet. However, it is not required in this model as only IPv4 packets
are used. The dummy service is used only to implicitly model the call
to ip_rcv, ip_forward, dev_hard_start_xmit and dhd_start_xmit in the de-
vice model. These four services have been combined into one service as
none of them contain any state variables.

The ISR in the driver, dhdsdio_isr, interacts with the semaphore that
is used by the driver thread. It is used to wake up the driver thread
whenever new packets arrive on the NIC. Including this service in the
model is optional, as the activation of the driver thread in the simulated
device model may either be done by performing an up-operation on the
semaphore object directly or by issuing an interrupt that calls the modeled
ISR service. Due to the scheduling class used by the driver thread, the
scheduler will dispatch the driver thread immediately after the interrupt,
thus making the delay negligible. This service is included to provide a more
complete model of the driver, as well as to test the interrupt mechanisms in
the extended multicore model.

The final service in the driver is the function dhdsdio_sendfromq. This
function is natural to have as a separate service as it has a distinct purpose of
sending a packet to the NIC. Additionally it helps to reduce the complexity
and scope of the outermost packet-handling service.

Following the instrumentation of services is the instrumentation of
packet queues. A NIC typically has one or two internal queues for received
packets or packets that are ready for transmission. The BCM4330 NIC has
two buffers for incoming and outgoing packets respectively[6]. The bcmhd
driver has one queue for outgoing packets. Additionally, the kernel has a
backlog queue for each protocol stack. This results in four packet queues in
the device model. The tracepoints for the enqueueing or dequeueing of the packets are generally placed immediately after the operation has completed, such as after a call to a function to place the packet in a queue, or when a DMA transfer has completed.

4.4.2 Behavior-specific tracepoints

It may be required to refine the model with some state variables to capture the different behaviors of the services. There are several complex macros in the bcmhd driver that check many conditions on the NIC. As there is no documentation on the semantic meaning of these macros, modeling and representing these macro-based checks as state variables in the simulator was determined to be too complicated and time-consuming. The driver model contains no other state variables apart from queue conditions because these macro-based checks are the only candidates for this particular driver.

Queue conditions may often be in parts of the code where instrumenting the condition is complicated or impossible. It is therefore often necessary to modify the code to capture the condition. The simplest type of rewriting is to extract the condition and store it in a variable. In the bcmhd driver, this is the case for the condition that signifies if the driver should perform an up operation on the semaphore after executing the driver loop. This condition is extracted by storing it in a separate variable, followed by a tracepoint, before executing the branching condition based on the value of the variable.

A more complex case is present in dhdsdio_sendfromq. The behavior of this service depends on a condition which can be captured in a variable in the surrounding service, dhdsdio_readframes. This requires the function signature for dhdsdio_sendfromq to be rewritten to include a parameter for a variable present in dhdsdio_readframes.

The driver uses a semaphore for synchronization. Operations on semaphores are captured by the kernel instrumentation and does not need to be instrumented in the driver. However, it must be specified in the device model and correctly referenced in the signatures.

4.4.3 Measurement-specific tracepoints

The last type of tracepoints is the measurement tracepoints. These are used to record the time of execution when handling a packet. Their location can be seen in Figure 4.2 and Figure 4.3. The first measurement tracepoint is at the start of the loop inside dhdsdio_readframes. This is before the DMA transfers to read a frame are started. When the early tx code is present, this measurement point is also before the transmission of a frame through the call to dhdsdio_sendfromq. The second measurement point is following the DMA transfers and before the call to dhd_rx_frame which hands the packet over to the kernel. The third tracepoint is immediately after the DMA transfer of the packet to the NIC has completed. The first and last tracepoints measure the time it takes to forward one packet through the kernel.
Unlike the other tracepoints, these measurement tracepoints inspect the packet and record a sequence number found in the packet during the experiments along with the timestamp reported by the cycle counter. This way the time it takes to forward a packet can be inferred from the resulting trace file. Corresponding measurement points are inserted into the device model for use during the experiments in the simulator.
Figure 4.5: Call graph of the modeled driver thread.

<table>
<thead>
<tr>
<th>Arrow Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green arrows:</td>
<td>Synchronization operations.</td>
</tr>
<tr>
<td>Red arrows:</td>
<td>Dequeue operations.</td>
</tr>
<tr>
<td>Blue arrows:</td>
<td>Enqueue operations.</td>
</tr>
<tr>
<td>Black arrows:</td>
<td>Service calls.</td>
</tr>
<tr>
<td>Dashed arrows:</td>
<td>Service calls with target extracted from trigger.</td>
</tr>
<tr>
<td>Ellipses:</td>
<td>Services.</td>
</tr>
<tr>
<td>Dashed ellipses:</td>
<td>Implicitly modeled services.</td>
</tr>
<tr>
<td>Diamonds:</td>
<td>Synchronization variables.</td>
</tr>
<tr>
<td>Rectangles:</td>
<td>Packet queues.</td>
</tr>
</tbody>
</table>

58
Part III

Experiments
Chapter 5

Experiment Design

The experiments that will be outlined in this chapter are performed with four goals in mind. The primary objective is to evaluate the accuracy of the model that has been extended to account for multicore execution. Through the development of the extended model, the LinSched scheduler simulator has been replaced with a simpler scheduler model implemented inside the ns-3 processing model. The effect of the switch from a highly detailed model of the scheduler to a less detailed one is another point of investigation. Further, the multicore model contains more objects in the simulation. As such, the scalability of this model is key for its viability in practical experiments. Finally, some experiments are performed to gain insight regarding the effect of software execution on network performance. These insights can be valuable when making decisions on what behaviors to include or exclude when modeling a device.

5.1 Experiment 1 — Accuracy of processing delay and forwarding capacity

The first experiment is done to verify that the model is able to capture the processing delay imposed on packets in a network when the forwarding is done in software. The model should be able to represent the limitations of the device, such as how many packets the device is able to forward at most over a given time interval. The forwarding capacity is here defined as the highest achievable number of packets forwarded over a given interval. How many packets a device is able to forward depends on many different properties such as processor speed, memory speed, scheduling, caches, bus transfer speed, the NIC, and so on. The queues on the device, both those in the hardware components and those implemented in software, have an impact on the forwarding capacity. This metric should give an indication on how well the model performs in conditions where the packet rate of the traffic that is to be forwarded is at or above the forwarding capacity of the device.

The experiment is done in two stages. The first stage is done with the Galaxy Nexus phone in an ad-hoc wireless network to study the behavior of
the device in the real world. The second stage of the experiment is done in the ns-3 simulator, replicating the parameters of the real-world experiment and using a model of the device. The two resulting data sets enable us to compare the behavior of the device with the behavior of the device model in the simulator.

5.1.1 Experiment parameters

The experiment will be done with traffic using a static rate of packets with a fixed size. Limiting the traffic to one static packet size is done to reduce the number of variables in the experiment. The packet size has been chosen to focus the analysis on per-packet processing. A smaller packet size should mean that the amount of per-packet processing becomes more significant, i.e., the processing that is done once per packet, such as looking up in a forwarding table. This is opposed to per-byte processing, i.e., where the cycles used for packet processing increases proportionally with the packet size. Per-byte processing would become more significant with larger packet sizes, for instance with DMA transfers taking more time. Results from preliminary experiments and analysis of the device model showed that the only significant source of per-byte processing in the model is from DMA transfers. Based on these results only one packet size is used in the experiments. Other packet sizes could be accounted for by modeling DMA transfers explicitly. A small but not insignificant packet size of 100 bytes is chosen to focus on measuring packets per second forwarded. The size of the traffic generator header, UDP header, IPv4 header, and a standard Ethernet header is subtracted from this number, leading to a payload of 42 bytes. In effect, each packet leads to 160 bytes transferred over the wireless medium due to additional headers used in the 802.11 protocol.

The experiment is repeated three times for each packet rate to achieve some statistical confidence in the results. The packet rate is not varied during the experiment. The use of a static traffic rate is done to study the device under stable conditions, which makes it easier to determine the limitations of the device. Bursty traffic would likely uncover several other interesting behaviors, but interpreting the results in the context of the experiment might be more difficult, partly due to the more erratic behaviors induced by the bursty traffic.

5.1.2 Expected results

The phone should be able to forward packets up to a certain rate where saturation starts to occur. Under saturation, one or more queues on the device will overflow continuously, causing packets to be dropped. At the point of saturation, the rate of packets forwarded by the phone starts to diverge from the rate of packets sent from the source. This effect should ideally be possible to demonstrate in ns-3 assuming that software execution is the limiting factor.
5.1.3 Setup of the real-world experiment

The testbed used for the experiments is illustrated in Figure 5.1. It consists of four nodes, three communicating nodes and one node used to monitor the communication on the wireless medium. The same testbed is used for all of the real-world experiments in this thesis.

The network is set up as a wireless ad-hoc (infrastructure-less) 802.11n network, meaning that there is no Access Point that facilitates communication between the nodes. The ad-hoc network is communicating on channel 11, i.e. the frequency band located at 2462MHz, using Orthogonal Frequency-Division Multiplexing (OFDM). The source machine (source in Figure 5.1), given the IP address 10.46.47.30, generates the IPv4 flow in the experiment. This flow is sent to the phone (phone in Figure 5.1), which is given the IP address 10.46.47.45. The phone forwards the flow to the sink (sink in Figure 5.1), which is given the IP address 10.46.47.60. The traffic flow is explicitly directed through the phone by setting the destination MAC address of the packets to the MAC address of the phone. Setting the network interfaces on the machines to the proper configuration is done using the tools `ifconfig`, `iwconfig`, `iw` and `ip`. All of these tools are commonly available on most Linux distributions. They are also available on Android 5.1, with the exception of `iw`. Connecting the phone to the ad-hoc network is done automatically by Android when booting the phone. The modifications required to make ad-hoc networking available on the phone are mentioned below.

The experiments are conducted using a Samsung Galaxy Nexus smartphone (model no. I9250). The phone is running Android 5.1, with a Linux kernel of version 3.19 that has been customized in three aspects: First, the tracing framework is used to record the time when a packet arrives in the driver after being transferred from the NIC and when a packet has been transferred back to the NIC after forwarding has been done in the kernel. The time is recorded using the cycle counter in the Performance Monitoring Unit (PMU) on each CPU core. Second, the CPU sleep states have been disabled in the kernel config to make the cycle counters perform correctly. Third, three small patches\cite{7}\cite{8}\cite{9} are applied. The first patch to make the bcmdhd driver handle wireless ad-hoc networks correctly, the second and third to make `wpa_supplicant` list ad-hoc networks, thus allowing the Android system to see and connect to the ad-hoc network automatically.

\footnote{\texttt{wpa_supplicant} is a standard user-space tool in Linux to manage authentication in wireless networks. It is used by Android to list available networks and authenticate to them if necessary.}
Some run-time settings need to be changed in addition to the kernel modifications. The default kernel firewall setup in Android is set to drop all packets that are not destined to be received by the system. This is changed by using the `iptables` tool to remove all rules and construct a new set of firewall rules that allows forwarding. Forwarding also needs to be enabled in the kernel by issuing the command `sysctl net.ipv4.ip_forward=1`. This setting is commonly disabled by default on most Linux distributions as a security measure.

Two laptops of type Hewlett-Packard Pavillion 8540p running Ubuntu 15.10 are used in the experiment as network nodes. One generates the IPv4 flow that is forwarded by the phone, the other to receive the flow after the phone has forwarded it. These two machines are referred to as source and sink respectively. The source machine is used to control the experiment. The phone is connected to the source machine with a USB cable, and controlled using `adb` to send commands to the device.

Traffic generation is done using `pktgen[30]`, which is configured and controlled by writing to files under the directory `/proc/net/pktgen/"
Examples of how to control pktgen from a script can be found in the Linux kernel source, under Documentation/networking/pktgen.txt.

A desktop machine, also running Ubuntu 15.10, is used to monitor and capture the wireless network data during the experiments. The machine is connected to the wireless network using a TP-Link USB Wi-Fi dongle. tshark[10] is used to capture the network traffic during the experiment. It instructed to set the network interface in monitor mode through a command line option before starting the capture. By running in monitor mode the interface delivers all frames it has received to the kernel, including those frames where the MAC address does not equal the MAC address of the interface, which would have been discarded under normal circumstances.

The experiments are performed inside a Faraday cage to reduce the amount of interference, primarily from other networks, during the experiment. The testbed must therefore be automated to avoid having to enter the Faraday cage during any of the experiments. Further, it is desirable to perform multiple experiments in series, each with a different packet rate. This presents a challenge regarding the capture of network traffic as there is no way to discern when one of the experiments starts or stops without manual inspection of the captured traffic data. To solve this, special UDP packets with different port numbers are sent to the sink before and after each experiment. These packets are recorded in the captured traffic, and is used to automate the splitting of one traffic capture file into one traffic capture file for each experiment. Splitting the traffic capture is done by extracting the special UDP packets from the traffic capture and using the timestamps of these packets to split the file using the editcap tool which is a part of tshark.

A shell script executing on the source machine is used to control the experiments. The script boots the correct boot image containing the custom kernel on the phone before starting a series of experiments. Once the phone has finished booting, the source machine checks that the sink and phone is correctly connected to the ad-hoc network and reachable. This is accomplished by issuing a ping on the wireless interface.

The script starts executing the experiments when the phone and network are set up. The UDP packet marking the start of the experiment is transmitted. Following the UDP packet, pktgen is set up. The desired rate of packets per second for the experiment is set, and the packet count is set 60 times the packet rate. The destination IP address is set to that of the sink and the destination MAC address is set to that of the phone to ensure that the phone forwards it. The user-space program for collecting and writing trace entries on the phone is started via adb. Finally, a command is given to pktgen to start sending packets. This blocks the script until pktgen has sent all the packets. A new UDP packet is transmitted to mark the end of the experiment after pktgen has finished sending all packets. A command is sent to the phone, again using adb, to stop tracing and write the remaining trace entries to secondary memory. adb is used to transfer the trace file from the phone and save it on the source machine. The script waits for a few seconds before executing the next experiment with new parameters, or it terminates if all the experiments have been executed.
5.1.4 Setup of ns-3 experiment

The simulation part of the experiment is done in ns-3.19, with the presence of the software execution model that has been extended to account for multicore execution. A single node, the smartphone, is modeled in the simulation. This is done because only the delays inside the modeled device are of interest. Hence, the communication between the different nodes in the equivalent real-world experiment are not modeled.

A device model has been created for the Galaxy Nexus. Within the device model is a model of the bcmhd NIC driver, some of the interrupt handlers related to the NIC driver, a part of the Linux IP stack, a part of the Linux networking subsystem and a softirq. The device model has been prepared by subjecting the smartphone to a workload consisting of forwarding IPv4 packets while using the aforementioned tracing framework to capture the traced events while processing this workload. The resulting trace file is then analyzed with the new analysis tool. The analysis process generates a set of signatures from which a subset of the most is selected for use in the model.

A simulation script is written to set up and execute the ns-3 simulation. This script contains a dummy version of a protocol stack. A full-fledged model of the IPv4 stack is not required since there is only one node in the simulation. A node representing the modeled device is set up, and the multicore execution model is installed on the node. This loads the device model from a file.

The simulation generates packets at a configurable rate. These packets are generated by queueing an initial event in ns-3 to generate a packet. When the event is handled by ns-3, a function in the protocol stack is executed. This function creates a packet, injects it directly into the NIC queue of the device model, and issues an interrupt. The interrupt is handled by a signature in the device model, which in turn wakes up the driver thread to handle the packet. Finally, a new event is scheduled to call the same function, thus generating another packet.

The measurements done on the modeled device in ns-3 correspond to the measurement points seen in Figure 4.2. The current time in the simulation is stored in the packet at these points. Additionally, the time is also measured when a packet is added to the NIC queue.

5.1.5 Sources of errors

As with any experiment in the real world, there is a possibility for errors from different sources. An error is in this context defined as an effect that negatively impacts the accuracy of the experiment. Performing experiments in a simulator provides a significantly more isolated and controllable environment. As such, this section is only relevant for the real-world experiment, not for the experiment done in the simulator.

For the real-world experiment the errors can be divided into four categories:

- Interference,
• measurement errors,
• measurement overhead, and
• system-level background noise.

Interference is a challenge for all experiments using wireless networks. This is both from other wireless networks and devices emitting electromagnetic radiation in the frequencies used by wireless networks. Walls and large objects in the vicinity of the experiment may also affect the propagation of the signals. In order to mitigate the amount of outside interference, the experiment is done in a Faraday cage. This ensures that no outside electromagnetic radiation affects the experiment.

The accuracy of a measurement depends on two factors: How close the measurement is in time and space to the effect that is to be measured and the resolution of the measurement. Measuring execution time is not trivial since it depends on the clock speed of the CPU. The clock speed varies between devices, and may also be adjusted while the CPU is running by a driver commonly referred to as a CPU governor. The cycle counter of the CPU is used to measure the execution time in these experiments because the number of cycles does not depend on the clock speed of the CPU. Regardless of how the execution time is measured, cache stalls may lead to unpredictable delays, and for this reason it is necessary to measure a significant number of executions to achieve a valid result that accurately represents the distribution of execution times.

It is inevitable that a measurement will have some impact on the subject in study. For this experiment, this primarily relates to the trace probes that are used to gather results from the kernel. Each execution of the trace probe results in cycles spent executing code that is normally not present. Once the trace buffer is full, the contents are written to secondary memory, which accounts for additional I/O operations that would not occur without the instrumentation in place. This is mitigated in two ways. The tracing framework is reduced, such that only the trace probes that are strictly necessary for the experiment are activated. This reduces the number of entries written to the trace buffer, in turn reducing the number of times it needs to be written to secondary memory. Additionally, no tracing is done when the buffer is being written to file, in order to avoid the additional I/O operations interfering with the results. The trace probes themselves are as minimal as possible, and thus, the overhead should be low. It should be noted that it is possible to account and correct for the overhead of the execution of trace probes as is done in [25], where the trace probes were shown to have approximately 300 cycles overhead.

The experiment is done on a smartphone with as few modifications as possible in order to perform the experiment. It is to be expected that a smartphone, either through the platform layer (in this case Android) or installed applications, performs more processing in the background than a development board with a minimal amount of installed software. This is however not assumed to be a major source of interference due to how the wireless network driver thread is scheduled in the kernel. It should also be taken into consideration that this is the most common configuration
of the device, and running the experiment while matching the default configuration as much as possible contributes to improve the repeatability of the experiment, in addition to making the results more realistic.

5.2 Experiment 2 — Concurrent forwarding while sending locally generated traffic

The second experiment has two goals. The first is to examine the impact of processing on two independent traffic flows. That is, to see if processing done on a traffic flow that is generated on the phone has an impact on a flow that the phone is forwarding at the same time. The second goal is to look at the impact of migrations on forwarding delay. Migrating a thread from one CPU core to another is done to balance the workload on the different cores. However, this leads to an increase in cache misses as the instructions and data of the newly migrated thread need to be fetched into the cache of the CPU core the thread was migrated to.

In this experiment, the phone will be used to forward an IP flow in a similar manner as in Experiment 1. At the same time, the phone will generate a second IP flow and perform additional processing on the generated traffic before transmitting it.

5.2.1 Experiment parameters

As with Experiment 1 the packet rate of the traffic being forwarded is kept uniform and static during each run. The packet rate of the traffic is varied between different runs. For this experiment the packet rates used are assumed to be within the forwarding capacity of the device. This is done to discover behaviors that are due to the additional processing and not due to approaching or exceeding the capabilities of the device.

The traffic on the device is generated in three stages for each packet: Waiting, processing, and sending. First, the packet will be waiting for some specified time. Then, the device will perform some synthetic processing on the packet. Finally, the packet will be transmitted. The time it takes between each packet is sent is defined here as a packet cycle. The cycle time is the inverse of the packet rate.

The device will generate 1000 packets per second of traffic, i.e., one packet every millisecond. The processing done on each packet is based on this number. The different runs will perform 50 microseconds, 100 microseconds, 250 microseconds, and 500 microseconds. These values are based on the following assumptions: The amount of time spent processing should not exceed the cycle time of a packet, so that pktgen is able to keep its speed. Other system processes may account for an unspecified amount of additional processing. This is assumed to not be greater than half a packet cycle, e.g. 500 microseconds at 1000 packets per second. Also, it is not possible to process for the entire packet cycle since the Linux scheduler explicitly disallows any thread from claiming ownership of the CPU for an
indefinite amount of time. As such, the specified packet processing has been limited to at most half of the packet cycle time.

The effect of migrations is measured by recording the value of the cycle counter when a migration occurs. This is done with a tracepoint in the scheduler when a migration happens, using the tracing framework. The trace file is then post-processed after the experiment. The number of cycles it takes to forward a packet is correlated with the number of cycles since the last migration.

5.2.2 Experiment setup

The experiment setup is similar to that of Experiment 1, with one additional traffic flow from the phone to the sink. This traffic is generated by using pktgen on the device, which has been added to the kernel by enabling pktgen in the kernel configuration. The experiment is divided up into multiple runs with fixed parameters. The amount of processing done is also modified between each run along with the packet rates.

The pktgen code used to generate traffic on the device is modified in two ways. The processor affinity is removed from the kernel thread used by pktgen. This enables the thread to be migrated in a manner similar to a regular thread. Further, the delay implementation in pktgen is changed from a blocking implementation to use busy-waiting. The busy-waiting loop is used to simulate processing for a set amount of time.

A traffic with a rate of 2000 packets per second is chosen for the traffic flow being forwarded by the device. This is based on preliminary results that showed a stable behavior when forwarding this kind of traffic. The packets are generated on the phone at a rate of 1000 packets per second. Both flows use packets 100 bytes in size. In order to distinguish the two flows in the traffic capture, the flow generated on the phone is given a different UDP source port number.

5.2.3 Expected results

The Galaxy Nexus has two CPU cores, and should be able to handle two concurrent tasks at the same time. The traffic-generating thread and the packet forwarding thread in the driver should ideally be distributed to separate cores by the scheduler. The additional processing should not have any noticeable effect on the packets being forwarded.

It is expected that the time it takes to forward a packet should increase after a migration. When the caches are warming up more cycles are spent on reading data and instructions from primary memory. Once the caches have warmed up, less cycles are spent waiting for read operations. The time it takes to forward a packet should then start to decrease once the caches are filled. Foong et al.[22] show that there is a correlation between cache locality and throughput when receiving traffic at high speed. The speed in their experiment is significantly higher. However, a similar effect is still expected, though it might be less pronounced due to the slower speed.
Faulkner et al. [21] show that executing a networking application on the same core as the network processing enables support of high transfer rates, again due to caching. These results are less comparable to this experiment as this experiment consists of two flows of traffic competing for resources, as opposed to one flow of traffic used in the article.

5.3 Experiment 3 — Effect of driver modification

The driver code is set up to always transmit a frame, if one is available, before receiving a frame. A comment in the code claims that this is done in order to improve performance when receiving: “tx more to improve rx performance” [11]. This is an optimization that is made with a focus on receiving traffic where the device is the destination. No investigation is done on whether this claim is truthful or not. However, this optimization has a very interesting effect on how the device behaves when forwarding traffic. Each packet that is received will be processed and sent back to the NIC before attempting to process another packet. This behavior is called early tx, and the execution that leads to this behavior can be seen in Figure 4.2. This behavior is opposed to receiving a number of packets, processing them and subsequently transfer them back to the NIC, which is the behavior that can be seen in Figure 4.3.

The first goal of this experiment is to examine the effect of this optimization on a use case different from what the optimization is intended for. The Galaxy Nexus device is intended as an end-node in terms of data communication, and one may safely assume that most network-related optimizations are done with this in mind. It is therefore interesting to see how this impacts the device’s ability to perform forwarding, something typically not done by end-node devices. The second goal is to study the difference between using and not using early tx. That is, examine how the device behaves differently with and without the early tx optimization when forwarding traffic.

In this experiment, the time it takes to forward a packet is measured. Two different kernels are used. One kernel with the early tx code in place, the other with early tx code removed. Both kernels have instrumentation enabled in the same manner as in Experiment 1. By subjecting two different kernels to the same type of traffic to be forwarded, any differences in behavior should be apparent after examining the traces produced by the instrumentation.

5.3.1 Experiment parameters

The packet size used in this experiment is similar to that of Experiment 1 where a small packet size is used to focus on per-packet processing. A packet rate of 2000 packets per second is chosen as this was shown to have a stable behavior in preliminary experiments where early tx was enabled. As in Experiment 2, the focus in this experiment is also on differences in the stable behavior of the device, in this case, the difference between having
early tx enabled and disabled.

5.3.2 Experiment setup

The experiment is done on a rig similar to that of Experiment 1. Unlike the other experiments, this experiment is done with two different kernels. One kernel is equal to that of Experiment 1 where early tx is enabled. The other is prepared by removing the code that causes the early tx behavior. The first execution of the experiment is done with early tx, the second without early tx.

5.3.3 Expected results

The early tx behavior invokes no queueing mechanisms in the kernel itself. Disabling early tx should, therefore, lead to queueing and therefore a less predictable behavior where the time it takes to forward a packet varies more than when using early tx.

5.4 Experiment 4 — Scaling of the multicore processing model

The final experiment is a comparison between the old and the new processing model, and an investigation of how the new processing model scales when introducing larger inputs. The first goal of this experiment is to examine how well the new scheduler model scales in terms of multiple cores, in terms of multiple nodes, and to compare its performance with the original model. The second goal of this experiment is to demonstrate if the multicore processing model scales well enough to be practically useful.

This experiment is done purely in the simulator. The experiment is performed by measuring the time it takes to execute a synthetic simulation. The old singlecore processing model is compared with the new multicore processing model on executing a singlecore device model. The new processing model is not compared with the old processing model on scalability with multiple nodes due to the aforementioned issues with the singlecore model. The device models used in the experiment are synthetic, and is not related to the device model of the Galaxy Nexus smartphone. All experiments use the same set of signatures. The signatures are designed to test a subset of the events that can be found in a real signature.

ns-3 is a discrete-event simulator. The primary factor that decides the time it takes to run a simulation is the number of events that are processed, and the complexity of handling those events. The number of cycles in a processing stage does not have any significant impact on the time it takes to simulate the processing stage. The effect of simulating a processing stage is that an event is queued in the simulation at a specific time based on the number of cycles that are to be executed before continuing to the next event in the signature. To evaluate scalability, it does not make sense to vary the
duration of the processing stages. All processing stages in the synthetic signatures use the same distribution of cycles to process.

Execution time is not an exact measure of efficiency. As such, each measurement is in itself not of any particular value. The key factor to look at is the relation between the size of the input and how long it takes to execute the simulation. The comparison of execution time for inputs of various sizes gives an indication of how the processing model scales when introducing larger inputs, and it can be used to determine the factors that contribute to the increase in execution time for larger inputs.

5.4.1 Experiment parameters

There are three main parameters in this experiment when examining the scalability of the multicore processing model: The number of threads executing concurrently, the number of CPU cores that are being simulated, and the number of nodes in the simulation. When comparing it to the singlecore model, only the number of threads are relevant.

Each experiment is set up to simulate one minute. 10000 packets will be generated every second during the experiment.

5.4.2 Experiment setup

The experiment is done by setting up a synthetic device model. The model consists of four queues and one synchronization variable.

Four different signatures are used, which can be seen in Figure 5.3. Two of the signatures are for the same service. All signatures execute in infinite loops. service_a is a signature that will fetch a packet from a queue (input) if one is available. If so, the packet will be moved to a different queue (internal) and an up operation is performed on the semaphore thus enabling another thread to dequeue it. Otherwise, only some processing is done. service_b fetches a packet from the internal queue after performing a down operation on a semaphore. The packet is then enqueued onto the output queue after some processing is done. service_c only performs processing. This is comparable to background processing on a system, i.e., processes that do not perform any network communication.

5.4.3 Expected results

Ideally, the multicore processing extension should be efficient enough to be used in further experiments. This will be the case if the overhead grows at most linearly with the number of threads, cores or nodes. The round-robin scheduler model used in the multicore processing model should impose a lower processing overhead compared to the singlecore processing model using the LinSched scheduler model.
service a, input is empty:  service a, input is not empty:
  <process>  <process>
queuecond input empty queuecond input notempty  
  <process>  <process>
  dequeue input dequeue input  
  <process>  <process>
enqueue internal enqueue internal  
  <process>  <process>
semaphore up semaphore up  
  <process>  <process>

service b: service c:
  <process>  <process>
  semaphore down semaphore down  
  <process>  <process>
dequeue internal dequeue internal  
  <process>  <process>
enqueue output enqueue output  
  <process>  <process>

Figure 5.3: Pseudo-code of four different signatures. Two of the signatures belongs to service a, which has a different behavior depending on whether the input queue is empty or not.
Part IV

Conclusion
Chapter 6

Results and Analysis

The results of the four experiments is presented in this chapter. For each experiment, first a short description of the figures used to display the results is given, followed by a summary of the results themselves. The results are analyzed, and a conclusion is drawn.

6.1 Experiment 1

The first experiment is a test to see how well the model is able to capture the behavior and forwarding delays imposed on packets by communication software execution on the device. Measurements are taken from the phone and compared to similar measurements taken in the simulator using the multicore processing model, using a traffic flow with similar characteristics in both cases.

6.1.1 Overview of figures

The rate of packets being transmitted by the device, as measured by the monitor, can be seen in Figure 6.1. The forwarding rate is calculated by counting the number of packets in each one second interval during the experiment. Figure 6.1a to Figure 6.1e show the packet rate forwarded by the phone for each packet rate generated by the source. The average and standard deviation seen in Figure 6.1f is calculated using measurements from 5 to 55 seconds in each experiment to accurately represent the stable behavior.

Figure 6.2 and Figure 6.3 shows the distributions of intra-OS delays measured on the real device and in the device model in ns-3. The x-axis shows the intra-OS delay in nanoseconds. The measured values are grouped together in bins 1000 nanoseconds wide. The y-axis represents the number of packets in each bin. These figures have different scales on the y-axis. This is due to the fact that the distribution of the delays measured on the phone is significantly narrower than the distribution of delays measured in the simulator, even though the number of measurements are the same. The difference in scale between the figures are not of importance for the results of the experiment. The important factors are the location of the peaks on
Figure 6.1: Rate of packets forwarded by the device for different rates of generated packets as seen by the monitor.
Figure 6.2: Distribution of processing delays from receiving packet to sending packet.

(a) Distribution of forwarding delays in ns-3 at 2000pps
(b) Distribution of forwarding delays measured on phone at 2000 packets per second

Figure 6.3: Distribution of processing delays between handing a packet to the kernel and sending the packet.

(a) Forwarding delays within loop in ns-3
(b) Forwarding delays within loop on phone
the x-axis and the relative difference in size to any other peaks in the same figure.

All four measurements are performed when forwarding a traffic flow of 2000 packets per second. The measurements displayed in Figure 6.2a and Figure 6.2b correspond to the measurement points numbered 1 and 3 in Figure 4.2. In Figure 6.3, the difference in time from the delivery of the packet to the kernel at measurement point 2 to the packet is sent back to the NIC at measurement point 3.

6.1.2 Summary of results from real-world experiment

The results from the real-world part of Experiment 1 show that the device is capable of forwarding traffic consisting of up to 3000 packets per second. Additionally, the device shows a negligible amount of deviation in the forwarded packet rate up to 2000 packets per second. A significant drop in the packet rate forwarded by the device is present when the packet rate of the generated traffic flow is between 3000 and 4000 packets per second. A traffic consisting of 4000 packets per second or above does not appear to be sustainable. That is, the device is not able to keep up with the packet rate of the traffic.

The distribution of intra-OS delays show three distinct peaks, one around 140000 nanoseconds and two around 120000 nanoseconds. Following the peak at 140000 is a series of small, repeating peaks that are approximately 25000 nanoseconds apart, each with approximately 12500 nanoseconds of standard deviation.

6.1.3 Analysis of the real world experiment

On closer inspection of the traffic data from the experiment there is strong evidence suggesting that MAC layer contention is a significant contributing factor in the drop in forwarding performance above 3000 packets per second. This conclusion is backed by two factors: First is the number of retransmissions that occur. The number of retransmissions grows as the packet rate increases. These retransmissions are unlikely to occur due to interference from other networks as the experiment is done inside a Faraday cage, hence the source and the device must be transmitting packets at rates that cause the shared medium to be contended. The second factor is that this drop in performance could be revealed in ns-3 only if the medium is part of the ns-3 experiment. ns-3 contains models for shared mediums such as WiFi, but this has not been included in the simulation experiment due to time constraints.

The majority of the packets are forwarded within 140000 nanoseconds from the start of the iteration of the receive loop, as seen in Figure 6.2b. The packets in this peak are characterized by the fact that they are forwarded right after the driver has been awakened by the ISR. The two peaks around 120000 nanoseconds are the packets that are received while the driver thread is active. The remaining smaller peaks past 150000 nanoseconds are packets that are forced to wait for any number of iterations of the driver thread,
either due to the budgeting mechanisms in the driver or due to the driver thread exhausting the time-slice.

6.1.4 Summary of results from ns-3 experiment

The distribution of intra-OS delays measured in ns-3 can be seen in Figure 6.2a. The distribution of measurements in ns-3 shows some similar peaks to that of the real-world experiment, namely the two around 120000 nanoseconds and one around 160000 nanoseconds. Additionally, three peaks are present at 205000 nanoseconds, 225000 nanoseconds and 275000 nanoseconds.

Figure 6.2a shows the distribution of delays measured from the delivery of a packet to the network stack and until the packet is handed back to the NIC. It presents as a single peak around 40000 nanoseconds, with a standard deviation of approximately 10000 nanoseconds.

6.1.5 Analysis of ns-3 experiment

The distribution of intra-OS processing delays measured in the ns-3 experiment, seen in Figure 6.2a, presents six distinct peaks.

The two first peaks, at 120000 and 130000 nanoseconds, are packets that are forwarded almost immediately after being received when the queue in the driver for sending packets is empty. These packets do not have to wait for any DMA transfers of packets that are being sent to the NIC.

The third peak, at 160000 nanoseconds, is characterized by packets that have to wait for the DMA transfers of another packet that is being sent to the NIC. This is the most common case and is the result of the early tx behavior. Once the other packet has been transferred to the NIC, the handling of the next packet proceeds.

The last three peaks in Figure 6.2a correspond to packets that have to wait for additional DMA transfers done on other packets before being serviced themselves. They are received on the NIC at some point after the early tx code inside the receive loop of the device model. They are forced to wait until after the current packet has been received, forwarded by the IP layer, and then transferred back to the NIC before serviced. The different peaks correspond to the number of DMA transfers it has to wait for. Packets received just after the early tx code will end up in the last peak, as they will be delayed by multiple DMA transfers of other packets before being serviced. The second-to-last peak and the peak before that are received after the first and second DMA transfer in the driver loop respectively.

Figure 6.3a show a near-perfect normal distribution around 40000 nanoseconds. This is to be expected when dealing with execution containing no state variables.

6.1.6 Comparison of real-world and simulator experiment

The results from the ns-3 experiment, seen in Figure 6.2a, show a similar set of peaks in the distribution of intra-OS delays with the addition of three
more peaks. The additional peaks in the results from the ns-3 experiment stem from the fact that it is not possible to record the time when a packet is received by the NIC on the device itself. In ns-3, it is possible to record the time when the packet is inserted into the NIC receive queue.

The communication between the NIC and the driver is likely more complex than what is modeled, in part due to the lack of documentation on the driver and the lack of source code for the firmware. Despite this, the model is still able to capture much of the same behavior that can be seen in Figure 6.2b, with regards to the forwarding delays in the three first peaks seen in Figure 6.2a.

The measurements from the delivery of the packet to the kernel until the packet is handed to the NIC, seen in Figure 6.3, suggests that the non-DMA part of the communication software used by the device has been modeled sufficiently well. The difference between the averages is 5000 nanoseconds. Additionally, the peak of values measured in ns-3 is slightly wider than the corresponding peak measured on the phone. This may be explained by the synthetic nature of the device model and differences in workload when tracing execution before creating the model.

### 6.1.7 Conclusions

This experiment demonstrates that the device is capable of forwarding traffic consisting of reasonably high packet rates in a wireless ad-hoc network in practice. The device model shows that the device is theoretically capable of forwarding packet rates above 5000 packets per second. Contention at the MAC layer becomes the limiting factor, not delay or queueing imposed by software execution. The contention at the MAC layer is not captured by the software execution model itself, but may be modeled separately in ns-3. This has not been done due to time constraints, and is left as a possibility for future work.

The device is capable of forwarding traffic up to 3000 packets per second, given that the packet size remains reasonably small. This is useful for generic types of communication flows in an ad-hoc network. A reduced packet rate, 2000 packets per second, has the benefit of reducing the MAC layer contention. This lower rate leads to less variation in the packet rate that is forwarded by the device. This is especially important for real-time applications where variation in delay must be kept to a minimum.

Large packets will experience increased delays from DMA transfers. The increased delay will likely bring down the rate at which the device is able to forward packets. The behavior of the device when forwarding traffic consisting of larger packet sizes has not been investigated in this experiment, and is left as an option for future experiments.

The experiment has shown that the models are capable of capturing most of the intra-OS delays and the queueing behavior of the device, even when only a subset of the execution path that the packet processing is dependent on is modeled. Fully modeling a device is greatly limited by the fact that the source code for NIC firmware in general is not publicly available. Additionally, the NIC on the device used in this experiment is
connected to a separate memory bus with its own controller that is in turn connected to the main memory bus. Even without extensive knowledge of the inner workings of the separate bus and its controller, the DMA controller, or the NIC, it is possible to model the device with reasonable accuracy, as is shown in Figure 6.2. The model is very accurate when disregarding the complex bus and DMA transfers on the device, as can be seen in Figure 6.3.

6.2 Experiment 2

The second experiment tests how processing performed exclusively for one flow affects another flow. One flow is sent from the source, forwarded by the device, and received by the sink. The other flow is generated on the device, synthetic processing is done for each packet. Additionally, the impact of migrating the driver thread is investigated to see if a migration has any effect on the time it takes to forward one packet.

6.2.1 Overview of figures

Figure 6.4 shows the relation between the time since the last migration and the forwarding delay experienced by a packet. The x-axis shows the time since the last migration occurred for any given packet. The y-axis shows the time it took to forward a given packet. Each cross corresponds to one packet. Only 1% of the packets are displayed to make the figure legible. However, the same characteristics are still visible as they would be with a larger sample size.

Figure 6.5 shows the packet rates of two different traffic flows. Both flows have the sink as destination. One traffic flow (red) is traffic that is generated and sent from the source machine and forwarded by the device. No additional processing is done on the device for each packet in this traffic flow apart from the processing required to forward it. The other traffic flow (blue) is traffic that is locally generated on the device. The generated flow is subjected to different amounts of additional processing before being transmitted. The four subfigures, Figure 6.5a to Figure 6.5d, show the packet rates for different amounts of processing done on the generated flow, from no additional processing and up to 500 microseconds of additional processing for each packet.

6.2.2 Summary of results

There are two major peaks of forwarding delays in Figure 6.4: Just below 160000 nanoseconds, and between 120000 and 140000 nanoseconds. The peaks in forwarding delays form two distinct horizontal “bands” in the plot. This is similar to the peaks that can be seen in the distribution in Figure 6.2b.

Both packet flows in Figure 6.4 appear to be generally stable during the entire experiment regardless of the amount of processing that is being done on the generated packets. The forwarded flow is forwarded and transmitted at 2000 packets per second, and the generated flow is transmitted at 1000 packets per second.
Figure 6.4: Migration of driver thread vs. forwarding delay

(a) No additional processing
(b) 100 microseconds processing
(c) 250 microseconds processing
(d) 500 microseconds processing

Figure 6.5: Packet rate of two different traffic flows.
packets per second. Three of the figures show an irregularity around 40 seconds that affects both flows.

6.2.3 Analysis

The author has not been able to determine the cause of the irregularities around 40 seconds in Figure 6.5a, Figure 6.5b, and Figure 6.5c. It is assumed that they may be caused by scheduling a background process. The averaging over 1-second intervals may also contribute as some packets that normally would be sent near the end of an interval may experience a short delay that causes them to end up in the next interval. There are no indications in the network traffic data to suggest contention or retransmissions at these points. The assumption that scheduling a background process is the cause is supported by the fact that both flows appears to be affected. If the flows were competing for a resource only shared between the two flows, one of them would come out as the “winner”, and experience little to no decrease in packet rate. This is concluded to not be the case, as there is no apparent “winner” in Figure 6.5b or Figure 6.5c.

There is no significant correlation between the time since the last migration and the time it takes to forward a packet that can be seen in Figure 6.4. Such a correlation would have presented itself as a decreasing trend in forwarding delay as the time since last migration increases. The two major bands visible in Figure 6.4 do not show any significant decreasing trend, suggesting that migrations do not incur any major penalty on the driver thread when forwarding packets. This is not only contrary to the expected result, but also the existing results in this area. This is likely due to a combination of factors: The cache on this device is smaller than those used in the previous works. A smaller cache is able to hold less data, leading to more frequent evictions of cached data. Thus, it is less likely that previously read data is still present in the cache when the execution would be able to re-use it. This leads to additional stalled cycles that could have been avoided with a larger cache. Also, the rate of the traffic is significantly lower than that of the previous works. A higher traffic rate might cause the kernel to execute the same pieces of code within a timespan that is short enough to re-use the contents of the cache.

6.2.4 Conclusion

In the context of device modeling, this is a result that suggests that modeling the impact of thread migration is of less importance as there is no significant correlation between the time since the migration and the time it takes to forward a packet, given the same circumstances as used in this experiment. That is, packets small in size, a low packet rate, using a device with a small cache, and a shared medium, as opposed to a wired single-access medium.

In general, the result implies that there is no benefit to restricting a driver thread to a specific core, even when the thread is given an explicit priority when scheduling. Further, it also implies that a smartphone such as the Galaxy Nexus is able to both forward and generate traffic without any
significant interference between the two flows of traffic, even when some processing, e.g. video encoding, is done on the generated traffic. When multiple CPU cores are available, these two tasks, forwarding and encoding, are typically scheduled on different cores. These two tasks are largely independent of each other with the exception of handing the generated packets to the driver, which requires only a small amount of synchronization between the two threads.

6.3 Experiment 3

The goal of this experiment is to examine the impact of the early tx optimization. The experiment is done in two stages; first with a kernel where the early tx code is left in place, then with a kernel where the early tx code has been removed. Both kernels are subjected to the same traffic flow to forward. The measurement tracepoints in the driver are used to measure the time it takes to forward a packet.

6.3.1 Overview of figures

The results from Experiment 3 can be seen in Figure 6.6 and Figure 6.7. Figure 6.6 shows the results from forwarding traffic at a rate of 2000 packets per second when using a kernel that has the early tx code in place. Figure 6.7 shows the results when forwarding the same type of traffic, but using a kernel where the early tx code has been removed. Figure 6.6b and Figure 6.7b show the cumulative distribution function of the data in Figure 6.6a and Figure 6.7a, i.e., the percent of packets (y-axis) forwarded within a certain amount of time (x-axis).

As noted in the results from Experiment 1, the y-axes differ in Figure 6.6a and Figure 6.7a. This is due to a narrower distribution of values in Figure 6.6a. Note also that the x-axis of Figure 6.6b and Figure 6.7b differ significantly.

6.3.2 Summary of results

Figure 6.6a shows a similar type of distribution as can be seen in Figure 6.2b, with peaks around 140000 nanoseconds, 117000 nanoseconds, and 113000 nanoseconds.

The majority of the packets are forwarded between 100000 nanoseconds and 170000 nanoseconds with a sharp increase between. The cumulative distribution function in Figure 6.6b displays two intervals of slower increase: between 35% and 40% and after 95%. A sharp increase can be seen in remaining intervals, which suggest that a significant amount of packets experience the same amount of processing delay when being forwarded. 95% of the packets are forwarded within 150000 nanoseconds.

There is still a peak around 140000 nanoseconds when using a kernel without early tx code, as seen in Figure 6.7a. However, it is significantly less pronounced than in Figure 6.6a. The distribution of forwarding delays
Figure 6.6: Distribution of processing delays when the early tx code is used

(a) 2000pps with early tx  
(b) 2000pps with early tx

Figure 6.7: Distribution of processing delays when the early tx code is not used

(a) 2000pps without early tx  
(b) 2000pps without early tx
is considerably wider. The cumulative distribution function shows a near-linear increase in the fraction of packets forwarded up to 90% before rate of growth diminishes. 95% of the packets are forwarded within 500000 nanoseconds.

6.3.3 Analysis

The distribution of the forwarding delays experienced by the packets with early tx code in place has already been explained in the analysis of Experiment 1. The early tx code enables packets to be forwarded immediately at the beginning of the next iteration of the receive loop in the driver code, leading to a predictable behavior due to less queueing inside the OS. In particular, the queuing mechanisms inside the OS are far more susceptible to delays invoked by scheduling. This leads to the wider distribution of processing delays when the early tx behavior is disabled compared to when it is enabled. Additionally, the driver is more likely to exhaust the timeslice given to it by the scheduler when dealing with a higher number of packets in the receive and transmit loops which further contributes to the less predictable behavior.

Repeating this experiment at 3000 packets per second does not show the same behavior when using early tx. This is the same effect that can be seen in Experiment 1 when forwarding 3000 packets per second. The variation in processing delay increases with the packet rate. This effectively cancels out the benefit imposed by the early tx behavior. This is believed to be caused by an increase in the number of interrupts due to DMA transfers when forwarding traffic of higher rate.

6.3.4 Conclusion

This experiment shows that the early tx behavior has a significant effect on the forwarding delay. By finishing processing one packet before handling another, the time a packet spends in kernel queues is significantly reduced.

Although it is not likely that the early tx code likely meant as an attempt to optimize the driver for use with forwarding, it still is a an optimization that can be implemented in NIC drivers for devices where forwarding is a large part of the workload.

This result also proves that effects of software execution can, and does have, a significant impact on network performance. Even small changes to the code, such as removing the code that causes the early tx behavior, can cause a drastic change in measurable delays that directly affects network performance.

6.4 Experiment 4

The final experiment is a test to determine how the multicore processing model scales with regard to increased number of simulated threads, cores, and nodes.
(a) LinSched

(b) Round-robin

Figure 6.8: Scaling with number of threads

Figure 6.9: Scaling with number of nodes

Figure 6.10: Scaling with number of cores
6.4.1 Overview of figures

The result from varying the number of threads can be seen in Figure 6.8. The number of threads is presented on the x-axis, the simulation time in seconds on the y-axis. Figure 6.8a shows the time it took to execute the simulation using the existing singlecore processing model with the LinSched scheduler model. The same kind of measurements for the multicore processing model using the round-robin scheduler model can be seen in Figure 6.8b.

The results of varying the number of nodes can be seen in Figure 6.9. The number of nodes in the simulation can be seen on the x-axis. Each mark on this axis corresponds to the number of nodes in the measurement. The circles mark the average of three executions of the experiment with the same number of nodes.

Figure 6.10 shows the results from varying the number of CPU cores on a single node in the simulation. The marks on the x-axis correspond with the number of cores used in the synthetic device model used in the experiment. The experiment has been performed with a device model containing two cores as well; it has been left out of the figure for clarity. The y-axis shows the time in seconds.

The circular marks in all figures is the average of three measurements. The standard deviation is low enough to be insignificant in all cases and has been omitted.

6.4.2 Summary of results

The LinSched model presents a peak at four threads, taking 270 seconds to simulate. A single thread takes 221 seconds to simulate. After the peak the time it takes to execute the simulation decreases, approaching approximately 230 seconds on average past 6 threads.

The round-robin scheduler model presents a peak at 3 threads, taking 387 seconds to simulate. As with the LinSched model, the time spent simulating declines before approaching an average around 340 seconds past 6 threads. Simulating a single thread takes 355 seconds.

With a single node the experiment takes approximately 400 seconds. The simulation time increases linearly to around 2000 seconds for five nodes.

Executing the simulation with a device model containing a single core takes approximately 400 seconds. A peak at two cores, taking approximately 1600 seconds is observed next. From eight cores and up, the simulation takes around 1200 seconds to execute on average.
6.4.3 Analysis

Both models present a linear growth for the first number of threads before declining. This happens because the ratio of processing events versus the number of other events increases. Processing events are considerably easier to simulate as they do not change the state of the simulation, they simply lead to a new event being queued.

The new round-robin scheduler spends more time than the LinSched model. This is attributed to the fact that the round-robin model is executed far more often than the LinSched model. The round-robin model has a significantly shorter quantum, 0.015 milliseconds, compared to the quantum that is used in LinSched, 100 milliseconds. The short quantum of the round-robin scheduler model generates a significant amount of additional events in ns-3 which contributes to the increase in the time it takes to execute the simulation.

Performing a simulation with multiple nodes shows a linear growth in the time it takes to execute the simulation. This is due to the corresponding linear growth in the number of ns-3 events. There is one round-robin scheduler per node, each generating the same amount of events per time. Thus, increasing the number of nodes leads to more ns-3 events over the same duration of the simulation timeline, making the simulation take longer to execute.

6.4.4 Conclusion

The results of the thread scaling experiment shows that the round-robin model requires refinement in terms of selecting a more ideal quantum. The original time-slice was selected in order to simulate that each thread gets as close to the same amount of CPU time as possible. Other than the shown increase in processing time, this has no adverse effects on the model as the overhead of context switches are not explicitly modeled.

The model is, even with a disfavorable quantum in the scheduler model, able to scale well for multiple cores. Simulating 30 nodes running similar sets of threads for 60 seconds of simulated time takes only 75 seconds longer to execute than simulating five nodes with the same sets of threads for the same amount of simulated time. This shows that while the processing model does add some overhead as opposed to simulating without any processing delay, the processing model is still able to simulate a reasonable amount of nodes without incurring any major penalty in terms of simulation time compared to simulating a smaller number of nodes.

Executing an experiment with multiple nodes show a linear growth due to the increased number of ns-3 events as a result executing the schedulers individually. This could potentially be mitigated by executing the scheduler on all nodes at the same time, thus requiring only one ns-3 event per scheduler tick. This could have some undesirable inaccuracy due to the scheduling being done at the same time on multiple nodes, which may limit the number of observable execution behaviors during the simulation if simultaneous scheduling of two communicating processes on two different
nodes turn out to be highly beneficial for the efficiency of communication.

6.5 Main findings

This section provides a short summary of the most important points of the conclusions from all experiments in this thesis.

Successful capture of device behavior

The experiments have shown that the multicore processing model is able to capture the behaviors of the modeled Galaxy Nexus smartphone, despite only having modeled a subset of the hardware and software on the phone. Access to the firmware of the DMA controller and NIC would have facilitated the creation of a more accurate model. Despite this, the majority of the packets forwarded by the device in the simulation experiences a similar forwarding delay to that of packets forwarded by the real device.

Intra-OS queueing key contributor to latency at low packet rates

Intra-OS queueing leads to a significant increase in the latency experienced by a packet during forwarding in software. The early tx behavior, explained in Chapter 4, leads to a considerable reduction in the time it takes to forward a packet due to no queueing of packets within the kernel. This has the additional benefit of a stable and predictable performance, which may be of importance to traffic flows with real-time requirements.

Impact of migrations at low packet rates

The results in this thesis show that, at lower packet rates, there is no discernible correlation between the time since the last migration of the driver thread occurred and the time it took to forward a packet. This is a different result than those presented in previous works which suggests that affinity improves performance. If migration of the driver thread were to have an impact on the latency experienced by a packet, then there should have been a correlation between the time it takes to forward a packet and the time since the last migration. Such a correlation could not be found in the experiment results, which suggests that the opposite is the case. It is assumed that this is caused by the low packet rate and the limited size of the cache on the device.

Replacement of LinSched

The results of the experiments show that on one hand, the scheduler model contributes to accurate results. On the other, the results also show that there is room for improvement in the implementation of the model. In particular, there are two parts of the implementation that may be improved. The short scheduler quantum contributes to a large number of ns-3 events during execution of the simulation. Increasing the quantum, thus reducing the
number of ns-3 events per scheduler instance will decrease the amount of
time it takes to execute the simulation. Additionally, larger simulations
could be handled more effectively by using one ns-3 event per quantum
to execute the scheduler on all nodes, instead of using one ns-3 event per
scheduler.
Chapter 7

Conclusion

A brief summary of the work done in this thesis is given in this final chapter, as well as a summary of the results. Following that is a personal reflection on working with this thesis. Open issues and future work is presented last.

7.1 Work on the methodology

The methodology, including the underlying concepts, tracing framework, analysis tool, and the implementation of the processing model, has been extended to support the notion of multicore execution. The underlying concepts were found to support multicore execution through analysis. A model with multiple hardware units performing execution was, on a conceptual level, supported through PEUs. This is for instance the case when a DMA controller is performing a transfer operation on behalf of, and in parallel with, execution on the CPU.

The tracing framework required no significant changes to be usable for tracing multicore execution. The framework was already properly synchronized to be usable in the presence of interrupts and preemptions. Further, the weak memory model is implicitly taken care of by using the synchronization code provided by the Linux kernel. As such, no explicit memory barriers are required. An unspecified bug regarding the activation of the tasklet led to the introduction of an IPI to activate the user-space process responsible for writing trace entries to secondary memory. The existing trace event definitions were extended to include an identifier for the CPU core the tracepoint was executed on. An additional trace event was defined for tasks migrating from one CPU core to another.

The process of analyzing traces required significant changes to support traces from multicore execution. Each CPU core has its own cycle counter. This means that events can occur on different timelines which must be handled separately during the analysis. The analysis only crosses between these timelines during migrations and when issuing work to be done on a different PEU. A migration implies a context switch, which is already handled. Thus, supporting migrations in the analysis is only a matter of moving the data related to the current execution of a LEU from one data structure to another because neither a context switch nor a migration depend
on any previous event in the analysis. Inter-PEU interactions, such as a notification that work on a different PEU has completed, may be received by a different PEU than the PEU that initiated the operation. This is currently handled by considering only the cases where the same PEU initiates the work and receives notification of the completion of the offloaded work. The cases where a PEU receives a notification about the completion of work initiated by a different PEU are ignored.

The analysis tool is rewritten in Go to provide an implementation that is easy to read, understand and more easily extendable.

The existing processing model for ns-3 required more extensive changes. The fundamental architecture of the model needed to be changed to support multiple CPU instances. All parts where singlecore processing was assumed were identified and subsequently modified. Alongside this, a minimal device model was constructed and used to test the modifications. Inserting assertions allowed for easy discovery of which parts of the code needed modifications when new types of events were added to the device model.

A new round-robin scheduler model was implemented for the simulation. The existing scheduler model, LinSched, was both difficult to set up such that it maps against multiple CPU objects in the simulation, and it is no longer in active development. Further, using it for multiple nodes required undesirable solutions. Even with a scheduler model that is substantially more high-level than LinSched, the processing model is still able to provide accurate results. However, additional tuning is required for the round-robin scheduler model to be efficient, as the model currently schedules processes far more often than required.

### 7.2 Summary of results and goals

Four goals were established for this thesis, as outlined in the introduction:

1. Extend the existing processing model,
2. instrument and model a multicore device,
3. verify the model, and
4. gain knowledge about communication software execution.

The work in this thesis has demonstrated how a singlecore execution model may be extended to a multicore execution model, thus making it generally applicable to model communication software execution on any type of device, including those devices that use multicore CPUs.

The application of the extended methodology was demonstrated by modeling the execution of IP forwarding on a Galaxy Nexus smartphone. The kernel was instrumented, and the resulting trace files from this instrumentation was analyzed to generate execution signatures. These signatures were then used to construct a model of the device.

An experiment was set up to verify the model against the real device. The experiment showed that it is possible to capture a subset of the behaviors
that can be seen on a multicore device in a real network. This was achieved by applying the methodology to capture the behavior of the communication software execution on the Galaxy Nexus device using instrumentation of the kernel and creating a device model based on data generated by the instrumentation. In particular, when measuring the latency imposed by software processing on packets, a partial correspondence between the results from the simulator and the results from the real-world experiment was found. The correspondence in queuing behavior could not be shown because the impact of MAC layer contention was underestimated in the design of the experiment. More accurate results in the verification experiment and making an accurate model of the forwarding on a Galaxy Nexus device was hindered by lack of access to the NIC firmware.

Several interesting facts about the effects of software execution on network performance has been uncovered through the experiments. Migration of the network driver thread was shown to have no adverse effect on the latency that was experienced by a packet. This is most likely due to the limited size of the cache on the Galaxy Nexus device. Further, an optimization in the network driver on the device was found to minimize the amount of queuing of packets inside the kernel during forwarding. This greatly reduced the latency experienced by packets being forwarded by the device. These results may be taken into consideration when designing network drivers for use in ad-hoc networks, or other types of networks with real-time requirements where forwarding is done in software, such as sensor networks.

7.3 Personal reflections

One of the major challenges initially was grasping and fully understanding the methodology. There is a lot of terms and concepts just within the methodology which must be fully understood before the work could begin. On top of that, understanding how the Linux kernel works is a massive task, but this knowledge was very important to determine which parts of the methodology and tools needed modifications. For instance, the fact that Linux treats each CPU core as a logical CPU of its own highly influenced the decision of making each core a separate PEU in the methodology.

On top of that there is the challenge of working with the extensive code base of the Linux kernel and cross-compiling it for a smartphone, without any prior experience in compiling the Linux kernel or booting a self-compiled kernel successfully on any sort of device. Thankfully, several guides exist on how to compile kernels for the Nexus devices due to the open nature of the Android platform and the Nexus devices. Automating the kernel compilation and boot-image build process has been crucial to productivity. A short guide on this can be found in Appendix A.

There has been plenty of frustrations in the work with this thesis as well. Much time was spent on both all of these issues, which is time that could otherwise have been spent on making real progress.

Getting the cycle counter to work correctly has taken a considerable
The cycle counter is paramount to the whole modeling process because the processing delays are calculated based on values taken from the cycle counter during tracing. No cycle counter means no useful tracing data, no analysis, no model to simulate, no experiments on the device. There was also the issue with the tasklet not getting scheduled properly, leading to the eventual replacement of the tasklet with an IPI. The existing analysis script was unreadable at best, and required a complete rewrite. This was a slow and tedious approach, but a necessary one, because extending the original script to account for multicore execution would have been next to impossible.

The existing implementation of the processing model in ns-3 turned out to be slightly fragile. The result of a change would most often be a segmentation fault or some kind of exception. Identifying all the assumptions related to singlecore execution was painstaking work. Changing each of the identified pieces of code that assumed singlecore execution required a slow, methodical approach to avoid breaking other parts of the code.

Modeling the driver was undoubtedly the most difficult part of the work on this thesis. The driver code is undocumented and very sparsely commented. Understanding how the driver worked required reading thoroughly through the code, followed by adding instrumentation points and investigating trace files to understand how the driver operates and to uncover various behaviors.

The most time-consuming part of the work on this thesis was setting up the real-world experiments due to the number of different components that needs to interact. The major steps involved were to set up the rig, set up and configure the ad-hoc network on all the devices, and writing and testing the shell script to automate the execution of the experiment. Many of the challenges in this work is mentioned in Appendix C. It took approximately two months to set up and put the experiment rig in a working condition after the experiments had been designed.

Once the experiments were set up, performing them took in total less than a week, in large part due to the automation of the experiment execution. Experiments could easily be redone to confirm the previous results or parameters could be tweaked. Having a clear idea of how to process the data generated by the experiments prior to performing them was very useful, although the amount of data generated by the experiments was underestimated. Each experiment was typically one minute in length, and would generate up to 150 MB of data. A session could run up to 100 experiments with various parameters, thus generating a fair amount of raw data.

In the end, I have enjoyed working on this thesis. It has been particularly interesting because of the combination of topics such as OSes, networking, and simulation. I have had the opportunity to work with the Linux kernel, and become familiar with how to navigate the vast amount of code, how it actually works in terms of networking and multicore execution, and how to compile it and boot it on a smartphone. Learning, understanding, and working with the methodology of communication software modeling,
which is currently at the forefront of the simulation field, has been a privilege. Overall, Linux, Android, and ns-3 have been great to work with, and the openness of all three has been an important factor in enabling this thesis. I have always been a fan of automating workflows, and during the work on this thesis, automation has proved to be both valuable and time-saving in the long run.

7.4 Open issues and future work

This final section presents the issues in this thesis that were not completely addressed due to time constraints as well as improvements to the work done in this thesis. Following that is a suggestion of topics that may be considered for future work.

7.4.1 Improvements to the work in this thesis

Improved device file parser

The device file parser in the processing model for ns-3 is highly rudimentary and performs little to no error checking. Currently, the parser is unable to provide an error message when it encounters syntax errors or errors that could be discovered through statical analysis, such as references to undefined objects in the device model. Having a proper parsing algorithm would be beneficial since it would make the processing model easier to use in practice. The grammar used for the device model is not complicated. It is likely to be an LL(1) grammar, which can be parsed by a recursive descent parser.

Further, the parsing of the device file is coupled with the setup and initialization of the objects in the simulation. This requires all references in the device model, such as calls to other services, to be defined prior to the call. Separating the parsing and setup operation provides a more flexible approach.

Static compilation of device model

As an extension of the previous point: having a proper parser for the device model would allow for the device model to be compiled to C++ code, which could then either be statically or dynamically linked in the final simulation executable. This could improve the performance of the processing model, especially for more complicated events in the device model. Compiling the device model to C++ code should be an option alongside the current interpreter, since debugging a statically compiled model might be excessively difficult. Care should be taken to ensure identical operation regardless of which option is used.
More efficient round-robin model quantum

The round-robin model that has been implemented to remove the dependency on LinSched is generating a significant number of ns-3 events due to the short time quantum. This was done to ensure that all simulated threads get as equal amounts of processing time as possible. The quantum might be excessively short, but changing it requires further analysis and verification that was not possible due to time constraints. Ideally the quantum should also be tunable so that it is possible to select different quanta to allow for a tradeoff between the time it takes to execute the simulation and the accuracy.

Improve the analysis tool

The analysis tool that has been written to analyze traces should be easy to understand and extend, but there is always room for improvements. The program could be made more efficient on multicore platforms by introducing concurrency using channels\textsuperscript{1} in Go for operations that are independent, such as the computations of averages and standard deviations. Concurrency using channels could also be added for handling events belonging to different CPU cores on different CPUs during analysis. Most events do not require synchronization when being handled, but updates to shared data structures must naturally be synchronized.

Stacks are somewhat crudely implemented. These could be made easier to understand by adding a stack type with push and pop operations. Go does not support templates by design, which makes this approach slightly cumbersome. A new stack with different element types would have to be implemented for each element type. Alternatively, a cast could be used after popping if using a stack that contains empty interfaces, which can store any type. There are community-made tools that automate the task of generating multiple variants of a type in a template-like manner, but this has not been looked into by the author.

The analysis tool should be extended with a configuration file for each device. These files could contain information such as the names of processes, queues, and other references in the trace file that needs to be translated to names.

Investigate and fix tasklet issue in the tracing framework

The issue with the tasklet in the tracing framework should be investigated further, as using an IPI is not an ideal solution. The symptom is that the framework stops issuing requests to the user-space process. Investigation into the issue showed that the tasklet itself was marked as scheduled by the framework, but the softirq that handles this tasklet was not marked as scheduled for execution. Addition of explicit memory barriers did not resolve the issue, neither did the use of atomic variables. A minimal example separate from the framework that recreates the issue might be required for

\textsuperscript{1}Go terminology for a built-in, synchronized FIFO list
further investigation, something that was not prioritized in the work on this thesis.

7.4.2 Future work

LinSched revival

Having an executable model of the Linux scheduler using the actual scheduler code from the Linux kernel is beneficial for simulation efforts. This does not only enable repeatable experiments on the Linux scheduler itself, but embedding the scheduler model in other simulators, as shown in [26], produces highly accurate results. A scheduler simulation framework that supports any scheduler implementation for Linux would also be useful for investigating and improving other schedulers not intended for inclusion in the mainline Linux kernel, such as BFS, commonly known as the BrainFuck Scheduler.

The author would like to propose the following set of requirements for a new version of a Linux scheduler simulator:

- Self-hosted, standalone simulator: It should be able to serve as a scheduler simulator by itself. This enables simulation of the Linux scheduler and changes to it. As such, the simulator should not require modification of the scheduler source code, unless this is submitted to the kernel as a patch.

- Easily integrated with other simulators: The design and architecture of the scheduler simulator should allow for integration with other simulators so that it can be used to model the impact of task scheduling in larger simulations.

- Multiple scheduler instances: A limitation that made LinSched unsuitable for further use in the multicore processing model was the fact that LinSched is designed to only run a single instance of the scheduler. This could be solved by saving and restoring the state of certain data structures in the scheduler code when switching between instances.

- C++ interface: The scheduler simulator framework should ideally be written in C++, or interface easily with it, as simulators are typically object-oriented and implemented in C++ for efficiency.

- Documentation: Information on the implementation and how to use, integrate and extend the scheduler simulator is lacking in LinSched, making it difficult to use in practice. Having well-written documentation with working code samples would have made LinSched easier to use.

More complete device models

More complete driver models would enable more accurate simulations of the device. This would be a model that includes execution on, e.g., the NIC or the DMA controller. There are two complicating factors in developing such
a model. The first is that the source for the firmware for these controllers is typically not available. Second, gathering traces from these controllers is not straightforward. Some NICs, including the one in the Galaxy Nexus, are not connected directly to main memory. If the controller has sufficient amounts of memory available, some trace data may be temporarily stored there. Trace data may also be periodically piggy-backed on transferred data.

Modeling a device at this level requires a platform which is sufficiently open and modifiable. This is typically not the case with the hardware platforms used for smartphones. These devices are meant as end-user devices, and it is often difficult or impossible to flash new kernels successfully on them. Controller chips from manufacturers that open-source their firmware is also a requirement for this kind of modeling. For this reason, such modeling should initially be done with a development board. As for NICs, the author is of the impression that Atheros is generally inclined to open-source their code, even firmware, as much as possible, so a development board with a NICs from Atheros would be a good lead to start from.

**Synthetic packet generation on NIC**

The results of Experiment 1 show that the wireless medium had an undesirable impact on the experiment. Ruling out the medium in future experiments can be done in two ways: Either add a model of the wireless medium to the ns-3 experiment or remove the medium from the real-world experiment. The latter approach has the benefit of reducing the number of variables. This would require the NIC to generate and discard packets, as opposed to receiving packets over the medium respectively. Such an approach is only possible with full access to the NIC firmware. The NIC could be modified to generate packets, transfer them to the kernel, and then send the packet over the wireless network to be stored on a second device. This requires being able to sample a cycle counter or equivalent from the NIC. The counter value could then be embedded in the packet after generating it and upon receiving it from the kernel after forwarding has been done.

**Coupling with DCE in ns-3**

Direct Code Execution (DCE) is a framework for ns-3 to execute existing implementations of network protocols. This includes for instance the IPv4 stack in the Linux kernel. Investigating how to efficiently couple or migrate features from the processing model into the DCE framework and implementing this is basis for a possible thesis in the future.

**Models and simulations of SDNs**

A Software Defined Network (SDN) is a network that exists purely in software. It may be transparently superimposed on a physical network. This is an emerging field of research. These networks are implemented
and managed in software, and as such they are affected by additional software execution on top of the execution required for physical networking. Modeling and simulating SDN implementations using the processing model might uncover challenges or bottlenecks that arise due to the added processing.

**More efficient transfers of forwarded packets**

Most NICs are not optimized to support forwarding. The complete packet, including payload, is transferred using DMA from the NIC to main memory, and then transferred back to the NIC to be transmitted after forwarding has been done in the kernel.

An alternative to this would be to send the headers up to and including L3. The entire packet and payload would remain on the NIC. If the packet is destined to be received by the device, the kernel will initiate transfer of the remaining payload once it is needed. If the packet is not destined for the device, the kernel will, if required, construct any new headers, transfer them to the NIC and then issue a command to the NIC to initiate the transmission of the packet.

Such a design might be useful for certain ad-hoc networks where a large portion of the traffic is intended to be forwarded. This could also be used in a software router. Implementing this may require substantial changes in the kernel and to existing NIC firmware.

A thesis on this topic should provide an implementation of this scheme, and investigate the overhead of using this approach, and determine if there are any scenarios or traffic patterns where this approach is more efficient than always transferring the entire packet from the NIC and back.
Appendix A

Compiling a Linux kernel for Galaxy Nexus

An Android boot image consists of four sections: A boot header which contains information about the size and load address of each section, a kernel image containing the kernel code, and an initial ramdisk containing files related to the boot process. Following the ramdisk is an optional second stage bootloader.

There are two methods of constructing a boot image. The first approach is to compile the kernel and then use the Android Platform SDK to generate the boot image directly from the sources. The second approach is to inject a new kernel into an existing boot image. The latter has the benefit of being less complicated, easier to set up, and requires less initial setup since it does not require downloading parts of the Android Platform SDK. Injecting a new kernel into an existing boot image, the approach used by the author, is the one that will be described in this section.

A.1 Cross-compiling the kernel

The kernel source for the Galaxy Nexus smartphone may be found at AOSP[12]. For other Nexus devices, a list is provided by AOSP[13]. This page also outlines the process of compiling a kernel. Note however that this page uses an Android reference board that runs on ARM64 hardware as example. The Galaxy Nexus and many other Nexus devices does not, so care should be taken to specify the correct platform type if using these examples as a reference.

Cross-compiling is the act of compiling software intended to run on a different hardware platform. The Galaxy Nexus smartphone runs on ARM-based hardware which is commonly used for devices requiring low energy usage. GCC, the compiler typically used for the Linux kernel, is designed to be compiled for one target platform only, which typically is the same type of platform as the host. Cross-compiling for a different platform requires compiling GCC such that it is configured to emit instructions for

---

1 Not to be confused with the Android SDK which is used for application development
the target platform while executing on a host platform that differs from the target platform. Fortunately a precompiled GCC toolchain is available from AOSP[14] which alleviates this task.

Compiling the kernel may begin after the kernel source and GCC toolchain have been acquired. The following environment variables must be set accordingly:

- `ARCH=arm`
- `SUBARCH=arm`
- `CROSS_COMPILE=arm-eabi-`
- `PATH=$PWD/arm-eabi-4.6/bin/arm-eabi-4.6/bin:$PATH`

The next commands are executed inside the root directory of the kernel source. For the Galaxy Nexus kernel this is the folder named omap which was downloaded earlier. The appropriate kernel configuration is installed by issuing the `make tuna_defconfig` command, which copies it to a file named `.config` in the root directory of the kernel source. Compiling the kernel is then done by issuing the `make` command. One may use the `-j` option to compile multiple files in parallel. Specifying a number after `-j` allows you to specify the number of concurrent build processes, e.g. `-j4` will compile up to four files concurrently. Some other useful commands are `make clean` to clean up the directory, and `make mrproper` to clean even more thoroughly. Note that the latter command also removes `.config`. Once the compilation finishes successfully the kernel image will be written to the file `omap/arch/arm/boot/zImage`.

Following the compilation of the kernel the next task is to build the boot image. The author has made use of `mkbootimg` and `unmkbootimg` which can be found on GitHub[15]. The factory boot images can be obtained from Google[16].

The downloaded archive contains a folder with the file `boot.img`. The initial ramdisk is extracted from this boot image by issuing the command `unmkbootimg -kernel /dev/null -i boot.img`, assuming `unmkbootimg` is found on PATH. This will produce the file `ramdisk.cpio.gz` containing the compressed initial ramdisk from the image. Additionally a command to rebuild the boot image is printed to stdout. This command can be used to create a boot image once `/dev/null` is replaced with the path to the kernel image we build, i.e. `omap/arch/arm/boot/zImage`. The command to build a boot image with our own kernel looks like:

```
 mkbootimg --base 0 --pagesize 2048 \
 --kernel_offset 0x80008000 \
 --ramdisk_offset 0x81000000 \
 --second_offset 0x80f00000 \
 --tags_offset 0x80000100 \
 --kernel omap/arch/arm/boot/zImage \
 --ramdisk ramdisk.cpio.gz \
 -o mybootimage.img
```

The tool `ccache` may be used to greatly speed up the compile times of the kernel. See `man ccache` for more information. Keep in mind that `ccache` must be ahead of the compiler on the PATH and that additional symlinks
are required to make ccache work with cross-compilers since they typically have non-standard names.

A.2 Booting the kernel

Booting the kernel is done with the fastboot\(^2\) tool which is available from the Android SDK. This tool allows you to boot an image without permanently flashing it. This is done by first rebooting the device into bootloader mode and then issuing a command to boot the kernel:

```
adb reboot bootloader
cativo boot mybootimage.img
```

The phone should restart and proceed to boot the newly built kernel. Any permission issues may possibly be resolved by running the commands as root or by setting up udev rules to grant ownership of the device to non-administrator users. Once fully booted the kernel version can be confirmed under Settings / Phone details / Kernel Version.

A.3 Automating the boot image build process

Automation of the build process is highly recommended to avoid errors and to increase efficiency. A makefile is provided here as a reference. Executing `make public_sources boot.img` will download the external tools and the kernel for the Galaxy Nexus, compile the kernel and build the boot image.

```bash
BUILD_ROOT := $(PWD)
BUILD_DIR := $(BUILD_ROOT)/build
PUBLIC_DIR := $(PWD)/public
export HOST_ARCH=$(ARCH)
export CC=$(CC)
export ARCH := arm
export SUBARCH := arm
export CROSS_COMPILE := arm-linux-
# Non-ccache path
export PATH := $(PWD)/public/arm-linux-4.6/bin:$(PWD)/public/bootimg-tools/mkbootimg/:$(PATH)
# Example path for use with ccache
# export PATH := /usr/lib/ccache/bin:$(PWD)/bin:$(PWD)/public/arm-eabi-4.6/bin:$(PATH)
.PHONY: run $(BUILD_DIR)/kernel all public_sources
all: $(BUILD_DIR) $(PUBLIC_DIR) boot.img
$(PUBLIC_DIR):
skdir $(PUBLIC_DIR)
$(BUILD_DIR):
skdir $(BUILD_DIR)
$(BUILD_DIR)/yakju-jwr66y. $(PUBLIC_DIR)
tar -xzf $(PUBLIC_DIR)/yakju-jwr66y-factory-09207065.tgz -C $(PUBLIC_DIR)
$(BUILD_DIR)/bootimg-tools:
$(BUILD_DIR)/bootimg-tools: $(PUBLIC_DIR)
git clone https://github.com/pbatard/bootimg-tools.git $@
cd $(BUILD_DIR)/bootimg-tools; \ CC="gcc" ARCH="" make mkbooting/mkbooting mkbooting/unmkbooting; \ cd $(BUILD_ROOT);
$(PUBLIC_DIR)/arm-eabi-4.6: $(PUBLIC_DIR)
git clone "https://android.googlesource.com/"
```

\(^2\)Fastboot is merely the name of the protocol, it has nothing to with the speed of booting.
platform/prebuilts/gcc/linux-x86/arm/arm-eabi-4.6

omap:
  git clone --branch androidomap-tuna-3.0-jb-mr2
  "https://android.googlesource.com/kernel/omap.git" $@

public_sources: $(PUBLIC_DIR)/yakju-jwr66y
  $(PUBLIC_DIR)/bootimg-tools
  $(PUBLIC_DIR)/arm-eabi-4.6

omap

boot.img: $(BUILD_DIR)/kernel $(BUILD_DIR)/ramdisk.cpio.gz
  $(BUILD_DIR)/image_rebuild_cmd
  $(shell cat $(BUILD_DIR)/image_rebuild_cmd) --kernel $(BUILD_DIR)/kernel
  --ramdisk $(BUILD_DIR)/ramdisk.cpio.gz -o $0

$(BUILD_DIR)/original.img: $(PUBLIC_DIR)/yakju-jwr66y/boot.img
  cp $^ $@

$(BUILD_DIR)/ramdisk.cpio.gz: $(BUILD_DIR)/original.img
  unmkbootimg --kernel /dev/null -i $^ | tail -1 | \
    sed 's/^ */\012/' | cut -d ' ' -f 1-13 > $(BUILD_DIR)/image_rebuild_cmd
  mv $(PWD)/ramdisk.cpio.gz $@

$(BUILD_DIR)/image_rebuild_cmd: $(BUILD_DIR)/ramdisk.cpio.gz
  # Depends on the rule above

$(BUILD_DIR)/kernel:
  cd $(KERNEL_DIR); \
  stat .config >/dev/null || make tuna_defconfig; \n  make -j8; cd $(BUILD_ROOT)
  cp $(KERNEL_DIR)/arch/arm/boot/zImage $@

run: boot.img
  adb reboot bootloader
  fastboot boot boot.img

# Do not replace ./build/ with $(BUILD_DIR)
clean:
  rm -rf ./build/
  rm -f boot.img

kclean: clean
  cd $(KERNEL_DIR); make clean;

mrproper:
  cd $(KERNEL_DIR); make mrproper;
## Appendix B

### Overview of Instrumentation in the Galaxy Nexus Kernel

#### B.1 Driver

<table>
<thead>
<tr>
<th>Function</th>
<th>Event type</th>
<th>Location</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>dhd_dpc_thread</td>
<td>Service entry</td>
<td>Start of infinite loop body</td>
<td>Captures the top-level service execution of the driver thread. Infinite loops should be modelled as services that enter and exit at the beginning and end of the loop body respectively.</td>
</tr>
<tr>
<td>dhd_dpc_thread</td>
<td>Queue condition (NIC)</td>
<td>Before call to dhd_bus_dpc</td>
<td>Rewritten to capture condition.</td>
</tr>
<tr>
<td>dhd_dpc_thread</td>
<td>Service exit</td>
<td>End of infinite loop body</td>
<td>Captures execution of the service reading frames from the NIC.</td>
</tr>
<tr>
<td>dhdsdio_readframes</td>
<td>Packet dequeue</td>
<td>After label deliver</td>
<td>Capture a packet being removed from the NIC.</td>
</tr>
<tr>
<td>dhdsdio_readframes</td>
<td>Packet extraction</td>
<td>Before loop</td>
<td>Capture the cycle counter value before first receive loop iteration in this execution of the driver loop. No actual packet data captured.</td>
</tr>
<tr>
<td>dhdsdio_readframes</td>
<td>Loop start</td>
<td>Before loop, after proceeding instrumentation point</td>
<td>Start of loop body</td>
</tr>
<tr>
<td>dhdsdio_readframes</td>
<td>Packet extraction</td>
<td>After loop</td>
<td>Capture the cycle counter value at the start of an iteration in the receive loop. No actual packet data captured.</td>
</tr>
<tr>
<td>dhdsdio_readframes</td>
<td>Loop restart</td>
<td>After preceding instrumentation point</td>
<td>After preceding instrumentation point</td>
</tr>
<tr>
<td>dhdsdio_readframes</td>
<td>Packet dequeue</td>
<td>After label deliver</td>
<td>Capture the successful dequeueing of a packet from the NIC.</td>
</tr>
<tr>
<td>dhdsdio_readframes</td>
<td>Service exit</td>
<td>At function end</td>
<td>Captures execution of service for final processing and delivery of a frame to the kernel.</td>
</tr>
<tr>
<td>dhdb_rx_frame</td>
<td>Service entry</td>
<td>At function start</td>
<td>Captures the contents of the delivered packet.</td>
</tr>
<tr>
<td>dhdb_rx_frame</td>
<td>Packet extraction</td>
<td>At the end of loop body</td>
<td>Captures execution of driver interrupt service routine. Used to wake up driver when thread is sleeping.</td>
</tr>
<tr>
<td>dhdb_isr</td>
<td>Service entry</td>
<td>At function start</td>
<td>Instrumented before all return statements.</td>
</tr>
<tr>
<td>dhdb_isr</td>
<td>Packet enqueue</td>
<td>At function start</td>
<td>Capture queuuing a packet in the driver. Event is captured at the start of function to capture the attempt to queue a packet, regardless of the queuening being successful or not.</td>
</tr>
<tr>
<td>dhdsdio_tskpkt</td>
<td>Packet enqueue</td>
<td>At function start</td>
<td>Capture the contents of a packet that has been transmitted.</td>
</tr>
<tr>
<td>dhdsdio_tskpkt</td>
<td>Packet enqueue</td>
<td>After call to tskptdh_d_tskcomplete</td>
<td>Capture a packet being enqued on the NIC.</td>
</tr>
<tr>
<td>dhdsdio_sendfromq</td>
<td>Queue condition (NIC)</td>
<td>At function start</td>
<td>Function prototype was modified to include state from calling function. The loop inside this function behaves differently depending on whether there are remaining frames to receive or not. It is assumed that the NIC queue will be empty if all frames have been received.</td>
</tr>
<tr>
<td>dhdsdio_sendfromq</td>
<td>Loop start</td>
<td>Before loop</td>
<td>Capture dequeueing packet from driver tx queue.</td>
</tr>
<tr>
<td>dhdsdio_sendfromq</td>
<td>Loop restart</td>
<td>At start of loop body, after lock acquire</td>
<td></td>
</tr>
<tr>
<td>dhdsdio_sendfromq</td>
<td>Packet dequeue</td>
<td>After first if-statement in loop body</td>
<td></td>
</tr>
<tr>
<td>dhdsdio_sendfromq</td>
<td>Loop stop</td>
<td>After loop body</td>
<td></td>
</tr>
</tbody>
</table>

Table B.1: Overview of the instrumentation of the network driver of the Galaxy Nexus smartphone.
B.2 Scheduler

<table>
<thead>
<tr>
<th>Function</th>
<th>Event type</th>
<th>Location</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>pull_task</td>
<td>Migration</td>
<td>End of function</td>
<td>Captures the migration of a thread (LEU) from one core to another.</td>
</tr>
<tr>
<td>down, down_*</td>
<td>Semaphore down</td>
<td>At function start</td>
<td>Captures the attempt to call down on a semaphore. Placed at the beginning of the various variants of down in order to not interfere with the semaphore code. Event includes the address of the semaphore.</td>
</tr>
<tr>
<td>up</td>
<td>Semaphore up</td>
<td>At function start</td>
<td>Capture calling up on a semaphore. Includes semaphore address.</td>
</tr>
</tbody>
</table>

Table B.2: Overview of instrumentation points in the Linux scheduler.

B.3 Platform-specific code

<table>
<thead>
<tr>
<th>Function</th>
<th>Event type</th>
<th>Location</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>omap_start_dma</td>
<td>PEU start</td>
<td>Before p-&gt;dma_write</td>
<td>Captures the start of a DMA transfer. The end of a DMA transfer is inferred during analysis by looking at temporary synchronization variables used by the LEU.</td>
</tr>
<tr>
<td>omap2_dma_handle_ch</td>
<td>Service entry</td>
<td>Before dma_chan[ch].callback</td>
<td>Captures the entry to a callback from the platform-specific DMA driver. Used by the HIMMC driver for the SDIO bus that the NIC is connected to.</td>
</tr>
<tr>
<td>omap2_dma_handle_ch</td>
<td>Service exit</td>
<td>After dma_chan[ch].callback</td>
<td>Captures the exit from the callback.</td>
</tr>
</tbody>
</table>

Table B.3: Overview of instrumentation points in the platform-specific code.

B.4 Networking subsystem

<table>
<thead>
<tr>
<th>Function</th>
<th>Event type</th>
<th>Location</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev_hard_start_xmit</td>
<td>Service entry</td>
<td>Before ops-&gt;ndo_start_xmit</td>
<td>Captures the entry to the driver function that receives a packet from the kernel. The address of ops-&gt;ndo_start_xmit is used for the event address in the tracepoint, so that it looks like the tracepoint is placed at the beginning of the driver function. This tracepoint can be used to determine which driver is in use, and which driver function receives packets from the kernel.</td>
</tr>
<tr>
<td>dev_hard_start_xmit</td>
<td>Service exit</td>
<td>After ops-&gt;ndo_start_xmit</td>
<td>Captures the exit from the driver-specific function.</td>
</tr>
<tr>
<td>___napi_schedule</td>
<td>Service enqueue</td>
<td>Before __raise_softirq_off</td>
<td>Captures enqueueing of the NET_RX_SOFTIRQ into the conceptual workqueue of softirqs. See note on instrumenting functions in header files.</td>
</tr>
<tr>
<td>enqueue_to_backlog</td>
<td>Packet enqueue</td>
<td>Start of function</td>
<td>Captures the attempt to enqueue a packet into the IP backlog queue.</td>
</tr>
<tr>
<td>enqueue_to_backlog</td>
<td>Queue condition</td>
<td>After preceding instrumentation point</td>
<td>The backlog service will be scheduled if the backlog queue is empty before queueing a packet.</td>
</tr>
<tr>
<td>netif_rx_ni</td>
<td>Queue condition</td>
<td>Before local_softirq_pending</td>
<td>netif_rx_ni is called if any of the softirqs have been scheduled.</td>
</tr>
<tr>
<td>process_backlog</td>
<td>Loop start</td>
<td>Before loop body</td>
<td>The entire loop required rewriting to instrument it.</td>
</tr>
<tr>
<td>process_backlog</td>
<td>Loop restart</td>
<td>Start of loop body</td>
<td></td>
</tr>
<tr>
<td>process_backlog</td>
<td>Packet dequeue</td>
<td>Before __skb_dequeue</td>
<td>Capture dequeueing of a packet from the IP backlog queue.</td>
</tr>
<tr>
<td>process_backlog</td>
<td>Loop stop</td>
<td>After loop body</td>
<td></td>
</tr>
</tbody>
</table>

Table B.4: Overview of the instrumentation of the networking subsystem of the kernel.
## B.5 Interrupts and deferred processing

<table>
<thead>
<tr>
<th>Function</th>
<th>Event type</th>
<th>Location</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle_irq_event_percpu</td>
<td>Service dequeue</td>
<td>Before action-&gt;handler</td>
<td>Captures the dequeue and immediate execution of a service from the conceptual queue of IRQs.</td>
</tr>
<tr>
<td>handle_irq_event_percpu</td>
<td>IRQ start</td>
<td>Before action-&gt;handler</td>
<td>Captures the start of an IRQ handler service.</td>
</tr>
<tr>
<td>handle_irq_event_percpu</td>
<td>Service exit</td>
<td>After action-&gt;handler</td>
<td>Captures the exit from the dequeued service.</td>
</tr>
<tr>
<td>__do_softirq</td>
<td>Loop start</td>
<td>Before label restart</td>
<td>Captures start of outer loop.</td>
</tr>
<tr>
<td>__do_softirq</td>
<td>Loop restart</td>
<td>After label restart</td>
<td>Captures start of inner loop.</td>
</tr>
<tr>
<td>__do_softirq</td>
<td>Service dequeue</td>
<td>Before h-action</td>
<td>Captures dequeueing and immediately executing a softirq. net_rx_action is called from here.</td>
</tr>
<tr>
<td>__do_softirq</td>
<td>Loop stop</td>
<td>After inner loop body</td>
<td>Captures the end of the inner loop execution.</td>
</tr>
<tr>
<td>__do_softirq</td>
<td>Loop stop</td>
<td>Before label restart</td>
<td>Captures the end of the outer loop execution.</td>
</tr>
<tr>
<td>irq_enter</td>
<td>Service entry</td>
<td>At function start</td>
<td>Captures the entry to an interrupt context. irq_enter is called by platform-specific code. Used to model the service that handles IRQ. The function does not actually handle IRQs.</td>
</tr>
<tr>
<td>irq_exit</td>
<td>Service exit</td>
<td>At function end</td>
<td>Captures the exit from an interrupt context. irq_exit specified as the service address in the event.</td>
</tr>
</tbody>
</table>

Table B.5: Overview of instrumentation points related to interrupts in the Linux kernel.

Tracepoints cannot easily be placed in code residing in header files. For this reason most activations of softirqs through __raise_softirq_irqoff are preceded with a tracepoint that signifies the enqueueing of the given softirq, instead of instrumenting the function __raise_softirq_irqoff directly. An example of this is given in Table B.4.

Tasklets may or may not require instrumentation depending on the driver. They are modeled and instrumented analogously to a softirq by using a service queue. Likewise, the run_ksoftirq function may or may not need instrumentation, depending on the driver.
Appendix C

Significant Obstacles Encountered and Their Solutions

Cycle counter issues
The cycle counter has been the cause of many headaches during the work on this thesis. Initially, a sequence of well-documented instructions must be executed to tell the PMU to enable the cycle counter. This would work fine for a while until the counters stop counting, making the resulting trace files unusable. Additionally, it would only appear to be actually counting at some point on one of the cores. These issues are caused by two different facts: The PMU is separate for each core, and the PMU control registers are reset when the CPU core enters or wakes up from a sleep state.

These issues were solved by creating a kernel module to control the PMU and cycle counter. The module would enable the PMU on both cores using the `on_each_cpu` function provided in the Linux kernel. This would enable the PMU and cycle counter on both cores. Making sure the cycle counter remains enabled at all times is trickier, and the only viable solution that was found was to disable CPU sleep states, thus keeping the CPU on at all times. This was accomplished by disabling CONFIG_CPU_IDLE in the kernel config. The OMAP platform used in the Galaxy Nexus device is highly optimized to reduce power consumption. There are many registers that control power in various ways. Some research was done into finding a register that would keep the PMU control registers powered even when the CPU itself was sleeping, but this did not produce any successful solutions.

No official support for ad-hoc mode in Android

Android does not officially support ad-hoc mode, as previously outlined in the report. In the case of the Galaxy Nexus smartphone, patches are available to enable support for ad-hoc networking in the driver and in the user-space network management tools. Generally, to modifications must be done: First, the driver must be able to recognize and report an ad-hoc network, also known as an Independent Basic Service Set (IBSS) network. Second,
wpa_supplicant must be modified to not filter IBSS networks. The patch provided for the Android version used by Galaxy Nexus could possibly be used for other devices as well. This patch prefixes ad-hoc networks with (*) to differentiate this type of network from networks with infrastructure.

**Unpredictable hardware**

Hardware should ideally perform predictably, but this has often not been the case during the work on this thesis.

The source machine used in the real-world experiment would not always send packets from pktgen with the specified rates, instead it would reduce the rate to approximately 500 packets per second without warning. Reloading the NIC driver using `rmmod` and `modprobe` fixed this issue.

Connecting to the Galaxy Nexus smartphone over `adb` was only successful on the laptop that is used as source in the experiments. On the sink machine it would fail unconditionally. `udev` rules are typically required to grant the proper access to or ownership of the device. Most Linux distributions provide `udev` files to facilitate this.

The wireless NIC initially used for the monitor machine, an old Ralink device, refused to work with a new kernel. This was fixed by buying a new wireless NIC. A NIC suitable for use in experiments would have to support monitor mode, in order to capture all packets it encounters on the medium. This is commonly used in experiments on wireless security. The NIC ultimately used for the monitor machine in the experiments was chosen based on lists of good chipsets for network monitoring on various wireless security blogs.

Different drivers will react differently to the same sequence of `ifconfig`, `iwconfig` and `iw` commands. Some drivers require to be in the `DOWN` state before entering ad-hoc mode, other do not have this requirement. Otherwise, the driver would refuse to enable ad-hoc mode with only a generic warning message. As such, there is no unified way of entering ad-hoc mode on Linux. Thus, having a simple script on each machine to consistently enter ad-hoc mode greatly eased this issue.
Bibliography


