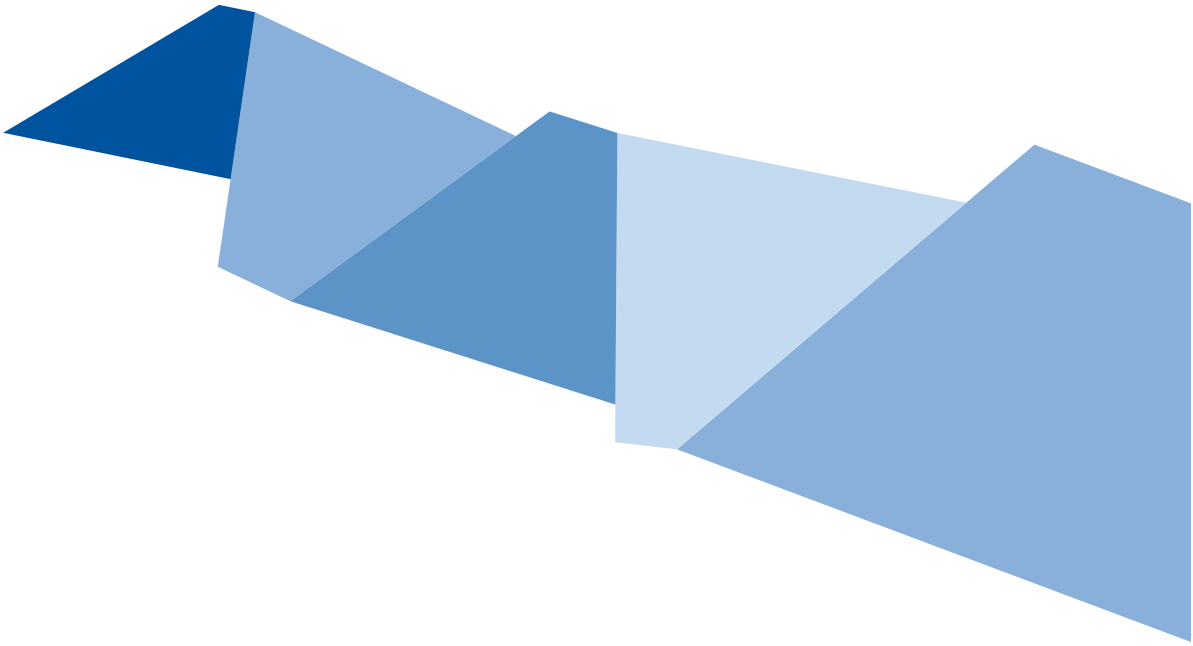


A Real-time Capable, Open-Source-based Platform for Rapid Control Prototyping

Igor Kalkov

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2018-03
RWTH Aachen University | Department of Computer Science | March 2018

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Igor Kalkov, M.Sc. RWTH

aus

Bender, Republik Moldau

Berichter: Universitätsprofessor Dr.-Ing. Stefan Kowalewski
Universitätsprofessor Dr. Uwe Aßmann

Tag der mündlichen Prüfung: 10. August 2017

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

Igor Kalkov
Lehrstuhl Informatik 11
kalkov@embedded.rwth-aachen.de

Aachener Informatik Bericht AIB-2018-03

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

Nowadays, software plays a crucial role in the development of embedded products. The product's design and development process can be significantly simplified by relying on reusable software components and modern general-purpose hardware. However, such off-the-shelf components are typically not able to provide real-time support required in the industry. To address this limitation, different approaches were presented in the past to enable the use of high-level software while retaining precise timings during the interaction with the physical environment. For example, real-time capable Linux running on general-purpose hardware is widely used in industrial automation today. Furthermore, Linux-based platforms like Android are continuously gaining popularity in all sectors of the market. Due to its intuitive user interface and a wide range of supported hardware, Android is already broadly deployed in different industrial use cases. However, Android devices used in embedded products are still restricted to serving the purpose of pure data visualization and handling of the user input. To extend Android's field of application to time-critical domains, this work presents a holistic approach for combining the performance of modern general-purpose off-the-shelf hardware with predictability and determinism of a real-time capable operating system. Instead of adopting common methods for limiting the real-time support to the Linux kernel and native applications, it provides a global perspective on Android's architecture including the Linux kernel, main high-level components and their interaction in time-critical scenarios.

The first part of this dissertation covers enhancements for minimizing the process scheduling latency using the `PREEMPT_RT` patch. Since Android is built upon Linux, introducing a fully preemptible kernel and a high-resolution timer enables a more precise process management. This approach allows Android to achieve bounded scheduling deviations in the same order of magnitude as industrial Linux-based systems. In the second part, it is shown that Android's original memory management may cause unpredictable suspensions of running applications or even automatically terminate long-term background processes. To address this issue, the platform is extended with a real-time capable garbage collector based on reference counting. The new collector operates concurrently to real-time threads in a non-blocking incremental manner, avoiding undesired interferences. Furthermore, the proposed modifications provide additional protection for persistent background services. Finally, this thesis implements enhanced methods for data exchange between separate Android applications. Being seamlessly integrated into the platform, new mechanisms allow predictable communication and bounded delays for delivering arbitrary messages across process boundaries.

A detailed evaluation of the introduced platform changes highlights the effectiveness and scalability of the presented approach. The resulting system performs better in terms of responsiveness and determinism, while staying fully compatible with standard Android components and third-party applications. By combining powerful general-purpose hardware and high-level programming paradigms, Android applications are now able to additionally fulfill strict timing requirements. This allows utilizing the advantages of Android in industrial use cases, which facilitates the development of easily extendable and less complex embedded products with intuitive user interfaces.

Zusammenfassung

Heutzutage spielt Software in der Entwicklung eingebetteter Systeme eine besonders wichtige Rolle. Obwohl der Einsatz von Standardhardware und wiederverwendbaren Software-Komponenten die Produktentwicklung bedeutend beschleunigen kann, erfüllen diese in der Regel nicht die industriellen Anforderungen an Echtzeitfähigkeit und Robustheit. Aus diesem Grund wurden in der Vergangenheit mehrere Ansätze vorgestellt, um auch mit High-Level-Software präzise Timings im Umgang mit der physischen Umgebung zu erreichen. So sind echtzeitfähige Linux-Systeme auf Basis von Standardhardware in der Industrie bereits weit verbreitet. Auch Linux-basierte Plattformen wie Android gewinnen kontinuierlich an Popularität in unterschiedlichsten Marktsegmenten. Dank der intuitiven Benutzeroberfläche und der zahlreichen kompatiblen Hardware-Plattformen findet Android heute zunehmend Anwendung in der Industrie. Bisher beschränken sich solche Anwendungen jedoch auf die reine Visualisierung von Programmdaten und Nutzerinteraktion. Die vorliegende Dissertation präsentiert einen ganzheitlichen Ansatz, um das Einsatzgebiet von Android auf zeitkritische Systeme zu erweitern, indem die Leistungsfähigkeit moderner Standardhardware mit der Vorhersagbarkeit eines echtzeitfähigen Betriebssystems kombiniert wird. Anstatt die Echtzeitfähigkeit wie üblich auf Linux zu beschränken, wird ein komponentenübergreifendes Lösungskonzept vorgestellt. Dieses umfasst sowohl den Linux-Kern als auch die wichtigsten Komponenten von Android und deren Interaktion in zeitkritischen Szenarien.

Im ersten Teil der Arbeit wird der `PREEMPT_RT` Patch eingesetzt, um die durch den Prozess-Scheduler verursachte Latenzen zu minimieren. Verbesserte Unterbrechbarkeit des Linux-Kernels sowie ein hochauflösender Timer ermöglichen auch unter Android ein präziseres Prozessmanagement. Damit können die maximalen Scheduling-Abweichungen für Android-Apps auf die Größenordnung von industriellen Linux-basierten Systemen beschränkt werden. Im nächsten Schritt werden die unvorhersehbaren Unterbrechungen von laufenden Anwendungen durch die automatische Speicherverwaltung untersucht. Dieses Problem wird durch die Einführung eines echtzeitfähigen Garbage Collectors basierend auf Reference Counting gelöst. Dieser arbeitet parallel und nicht-blockierend, sodass die Ausführung von Anwendungen mit Echtzeitanforderungen nicht mehr beeinträchtigt wird. Die eingeführten Änderungen bieten zusätzlichen Schutz für persistente Hintergrunddienste. Weiterhin präsentiert diese Arbeit erweiterte Methoden für einen zuverlässigen Datenaustausch zwischen laufenden Anwendungen. Diese Erweiterungen werden nahtlos in die Plattform integriert und ermöglichen eine vorhersagbare Kommunikation zwischen separaten Android-Prozessen mit beschränkten Verzögerungen.

Eine detaillierte Auswertung der vorgeschlagenen Modifikationen belegt die Effektivität und Skalierbarkeit des gewählten Ansatzes. Das neue System zeigt eine bessere Reaktionsfähigkeit und bleibt vollständig kompatibel mit Standard-Android-Komponenten und existierenden Anwendungen. Dadurch können Android-Anwendungen strikte zeitliche Anforderungen erfüllen und gleichzeitig von leistungsfähiger Standardhardware und höheren Programmiersprachen profitieren. Der Einsatz einer solchen echtzeitfähigen Android-Plattform ermöglicht die Entwicklung von weniger komplexen eingebetteten Produkten in neuen industriellen Szenarien.

Acknowledgments

I hereby express my sincere appreciation to Prof. Dr.-Ing. Stefan Kowalewski for giving me the opportunity to join his group, for supervising my thesis and for providing me the needed freedom to pursue and implement my ideas in research. I also thank Prof. Dr. Uwe Aßmann for serving as the second supervisor and his valuable remarks and suggestions. Furthermore, my thanks go to Prof. Dr. Matthias Müller and Prof. Dr. Jürgen Giesl for taking their time and participating in my examination committee.

I have also received help and support from many others during my time as a research assistant. First of all, I would like to thank my former colleagues at the *Chair of Computer Science 11 – Embedded Software* for creating a friendly environment and for inspiring discussions. In particular, I thank Thomas Gerlitz, David Thönnessen, Mathias Obster, Dominik Franke and John F. Schommer for their critical opinions and great collaboration during the development of new concepts in the RTAndroid project.

This dissertation owes many of its results to my talented students. A particular mention deserve Stefan Schake, Alexandru Gurghian, Stefan Klug, Florian Sehl, Andreas Wüstenberg, Stefan Rakel and Johannes Lipp, who wrote excellent theses, shared their ideas, implemented new approaches, performed evaluation studies and contributed as co-authors of publications.

I had a chance to extend my knowledge and make progress in writing my thesis at IHMC, *The Institute for Human & Machine Cognition* in Pensacola, Florida. I express my great appreciation to Jerry Pratt and Peter Neuhaus for inviting me to the Robotics Lab and providing me a high degree of freedom during my research. Thank you for helpful insights in the state-of-the-art robotics and the unforgettable time at the DARPA Robotics Challenge. My warmest thanks go to Doug Stephen, Jesper Smith, Tobias Meier, Elena Galbally, Stephen McCrory, Chris Schmidt-Wetekam and the whole IHMC for their support and amazing team spirit.

Last, but not least I would like to thank my family and friends for their unconditional support throughout all these years. Special thanks go to my better half Julia Streitz for her endless patience and continuous encouragement through the entire time of my research and especially during its final stages. Thank you.

Igor Kalkov
February 2018, Aachen

Contents

1 Introduction **1**
- 1.1 Objectives and Contributions . . . 2
- 1.2 Thesis Outline . . . 3
- 1.3 Bibliographic Notes . . . 4

2 Preliminaries **5**
- 2.1 Development of Embedded Systems . . . 5
  - 2.1.1 Design Methodology for Embedded Products . . . 6
  - 2.1.2 General-Purpose Embedded Hardware . . . 8
- 2.2 Linux-based Operating Systems . . . 8
  - 2.2.1 Linux Kernel . . . 9
  - 2.2.2 Android Platform . . . 10
- 2.3 Designing Real-time Systems . . . 15
  - 2.3.1 Real-time System Characteristics . . . 16
  - 2.3.2 Hardware for Real-time Computing . . . 17
  - 2.3.3 Real-time Operating Systems . . . 17

3 Concept for an Industrial Android **21**
- 3.1 Related Work . . . 22
- 3.2 Discussion of Possible Approaches . . . 25
- 3.3 Adding Real-Time Support to Android . . . 26

4 Real-time Linux Kernel **29**
- 4.1 Related Work . . . 30
- 4.2 Improving Kernel's Preemptibility . . . 32
- 4.3 CPU Frequency Lock . . . 33
- 4.4 Experiments . . . 35
  - 4.4.1 Configuration of the Linux Kernel . . . 35
  - 4.4.2 Native Scheduling Latency . . . 37
  - 4.4.3 Dynamic CPU Frequency Scaling . . . 38
- 4.5 Summary and Discussion . . . 41

5 Non-blocking Memory Management **43**
- 5.1 Related Work . . . 44
  - 5.1.1 Tracing Garbage Collection . . . 44
  - 5.1.2 Reference-Counting Garbage Collection . . . 45
  - 5.1.3 Real-Time Garbage Collection . . . 46
- 5.2 Real-time Garbage Collector for Android . . . 47
  - 5.2.1 Extended Metadata Management . . . 48

5.2.2 Integration of the Write Barrier . . . 49
5.2.3 Implementation of the Garbage Collector . . . 50
5.3 Memory Adjustments for Real-time Processes . . . 52
5.4 Experiments . . . 54
5.4.1 OOM Adjustments for Real-time Processes . . . 54
5.4.2 Latency Caused by the Garbage Collection . . . 56
5.4.3 Latency Evaluation in a Long-Term Execution . . . 58
5.4.4 Analysis of the Memory Management Overhead . . . 59
5.5 Summary and Discussion . . . 60

6 Bounded Remote Procedure Calls 61
6.1 Related Work . . . 61
6.2 Extended Binder Driver . . . 62
6.2.1 Overview of Binder's Architecture . . . 63
6.2.2 Analysis of Binder's Real-time Support . . . 66
6.2.3 Integration of the Priority Inheritance . . . 67
6.3 Experiments . . . 68
6.3.1 Performance Evaluation under Load . . . 69
6.3.2 Multiple Real-Time Threads . . . 71
6.4 Summary and Discussion . . . 72

7 Prioritized Intent Broadcasting 73
7.1 Related Work . . . 73
7.2 Extended Broadcasting Architecture . . . 74
7.2.1 Analysis of Intent Broadcasting . . . 74
7.2.2 Intent Prioritization Mechanism . . . 78
7.2.3 Handling of Prioritized Global Broadcasts . . . 79
7.2.4 Handling of Prioritized Local Broadcasts . . . 81
7.3 Experiments . . . 83
7.3.1 Performance Evaluation of Global Broadcasts . . . 83
7.3.2 Impact of the Foreground Priority Flag . . . 84
7.3.3 Handling of Global Broadcasts under Load . . . 85
7.3.4 Evaluation of the Explicit Prioritization . . . 86
7.3.5 Performance Evaluation of Local Broadcasts . . . 87
7.4 Summary and Discussion . . . 89

8 Evaluation 91
8.1 Impact Analysis of the Introduced Extensions . . . 92
8.2 Summary and Discussion . . . 96

9 Conclusion 99
9.1 Summary . . . 99
9.2 Future Work . . . 100

List of Figures

1.1 Embedded market trends between 2013 and 2015 [122]. 2

2.1 Popular general-purpose hardware for embedded projects. 6
2.2 Basic structure of the Linux kernel. 9
2.3 Android architecture overview. 11
2.4 Process importance hierarchy in Android. 15
2.5 Simplified life cycle of Android applications. 15

3.1 Most popular embedded operating systems in 2015 [122]. 21
3.2 Top selection factors for embedded operating systems in 2015 [122]. . . . 22
3.3 Strategies for the integration of real-time support into Android [74]. . . . 23
3.4 Android subsystems extended for integration of the real-time support. . . 26

4.1 Evolution of `PREEMPT_RT`'s size between 2006 and 2017. 30
4.2 Patching with `PREEMPT_RT` (illustration derived from [79, Fig. 2]). 33
4.3 Simplified mechanism for dynamic frequency locking. 34
4.4 Worst-case latency analysis for different kernel configurations. 36
4.5 Evaluation of the scheduling latency for extended period times. 38
4.6 Evaluation of the calculation time for extended period times. 39
4.7 Impact of the frequency on the calculation time. 40

5.1 Schematic representation of objects within the DVM [38]. 48
5.2 Evaluation of the latencies introduced by the garbage collection. 57
5.3 Number of scheduling latencies recorded in different classes [38]. 58
5.4 Distribution of the scheduling latency in the long-term test [38]. 59

6.1 Schematic illustration of Android's Binder architecture. 63
6.2 Binder steps executed during a remote procedure call [62]. 64
6.3 Measuring time for a single RPC invocation in Android. 69
6.4 Analysis of Binder delays for $k_{nrt} = 20$ regular threads [62]. 70
6.5 Analysis of Binder delays for $k_{nrt} = 100$ regular threads [62]. 70
6.6 Invocation delays for rising number of real-time threads [62]. 71

7.1 Schematic delivery of global Intents using the Activity Manager Service. . 75
7.2 Internal broadcast processing in the Activity Manager Service [61]. 75
7.3 Internal architecture of the Local Broadcast Manager [61]. 77
7.4 Broadcast handling in the Local Broadcast Manager [61]. 77
7.5 Encapsulation of the critical section in the Activity Manager Service [61]. 80
7.6 Processing delays for global real-time broadcasts with $k_{nrt} = 10$. 84
7.7 Distribution of T_{PROC} for real-time broadcasts during a long-term test [61]. 84

7.8 Worst-case processing delay for real-time Intents with rising k_{nrt} [61]. . . 85
7.9 Case 4 violates the processing order of prioritized Intents [61]. 86
7.10 Processing delay for local broadcasts with $k_{nrt} = 10$ regular threads [61]. 88
7.11 Statistics for local broadcast processing [61]. 89

List of Code Listings

2.1 Calculating scheduling priority from nice value. 10
2.2 Translation of thread priorities to nice values in Android. 15

5.1 Loading memory and adjustment threshold values during Android boot. . 54
5.2 Termination of the normal activity in the OOM situation. 55
5.3 Protection of the real-time activity in the OOM situation. 56

7.1 Sending prioritized broadcasts in Android [61]. 78

List of Tables

2.1 Property comparison between general-purpose and embedded systems. . . 7
2.2 Description of the main application components in Android. 13

4.1 Kernel configuration options relevant for real-time support. 35
4.2 Test specification for a detailed evaluation of the scheduling latency. . . . 37
4.3 Latency evaluation results. 37

5.1 Opcodes for updating object references on the heap. 50
5.2 Recorded latencies for different types of object load [µs]. 58
5.3 Time for object allocation. 59
5.4 Time for object operation. 59

7.1 Analysis of the broadcast handling order [63]. 87

8.1 Impact analysis for all introduced platform extensions. 94

List of Acronyms

ADC	Analog-to-Digital Converter
AM	Activity Manager
AMS	Activity Manager Service
AOT	Ahead-Of-Time
ART	Android Runtime
ASHMEM	Anonymous Shared Memory
BSP	Board Support Package
CFS	Completely Fair Scheduler
CPU	Central Processing Unit
DVM	Dalvik Virtual Machine
FIFO	First In – First Out
FPGA	Field Programmable Gate Array
GC	Garbage Collection
GPIO	General-Purpose I/O
GPOS	General-Purpose Operating System
GUI	Graphical User Interface
HMI	Human-Machine Interface
I^2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IPC	Inter-process Communication
ISR	Interrupt Service Routine
JIT	Just-in-Time
LBM	Local Broadcast Manager
OOM	Out-Of-memory
OS	Operating System
PID	Process ID
PWM	Pulse Width Modulation
RAM	Random-Access Memory
RPC	Remote Procedure Call
RR	Round Robin
RTOS	Real-time Operating System
RTSJ	Real-time Specification for Java
SBC	Single Board Computer
SoC	System on a Chip
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter
WCET	Worst-Case Execution Time

1 Introduction

Technological trends emerging in the market of consumer electronics can have a major impact on the industrial domain. For instance, the importance of embedded off-the-shelf products based on Android, with intuitive interfaces and extended mobility, has continuously increased in all market segments during the last decade. Other examples of such trends include reusable open-source components, flexible general-purpose hardware and agile project management techniques, which became popular in the development of high-level software. Their combination can significantly simplify the development process and improve the quality of the resulting product. However, embedded systems are typically characterized by a strong coupling between hardware and software, which limits the integration of modern approaches. Thus, the development process for embedded systems still largely relies on expert knowledge. Due to strict timing requirements, embedded software is commonly implemented using low-level or specialized programming languages, while its testing heavily depends on proprietary simulators.

A recent study of the worldwide market for embedded systems shows that general-purpose Linux-based platforms like Raspberry Pi and Beagle Board belong to the most popular hardware platforms for proprietary embedded projects in 2015 [122]. Additionally, operating systems in general – and real-time operating systems in particular – are expected to be the greatest technological challenge in the future (see Figure 1.1). Although only 12% of respondents identified this as a challenge in 2013, this value grew during 2014 (17%) and reached 26% in 2015. At the same time, the percentage of all projects using an open-source OS/RTOS increased from 31% in 2012 to 39% in 2015. This study also illustrates that new embedded products are more likely to include a graphical user interface.

Some of these challenges can be addressed by introducing a general-purpose operating system able to fulfill real-time requirements. Off-the-shelf operating systems like Linux can be easily combined with modern concepts of software development, resolving the currently existing trade-off between flexibility and development efficiency [27]. Furthermore, Linux is also compatible with common techniques for process and system virtualization, which are designed to improve the system's predictability and determinism. Combining real-time capable virtual machines with a real-time Linux kernel is a widely accepted practice in the industry, not least because it enables the usage of high-level programming languages like Java [115]. Based on Linux, the Android platform inherits its flexibility and additionally provides a rich application framework for various use cases. As it is optimized for hardware with limited resources, Android can be deployed on a wide range of different embedded devices, including low-end general-purpose hardware. Although current trends indicate a rising popularity of Android in the embedded field, there exists no holistic approach for extending the platform with real-time support.

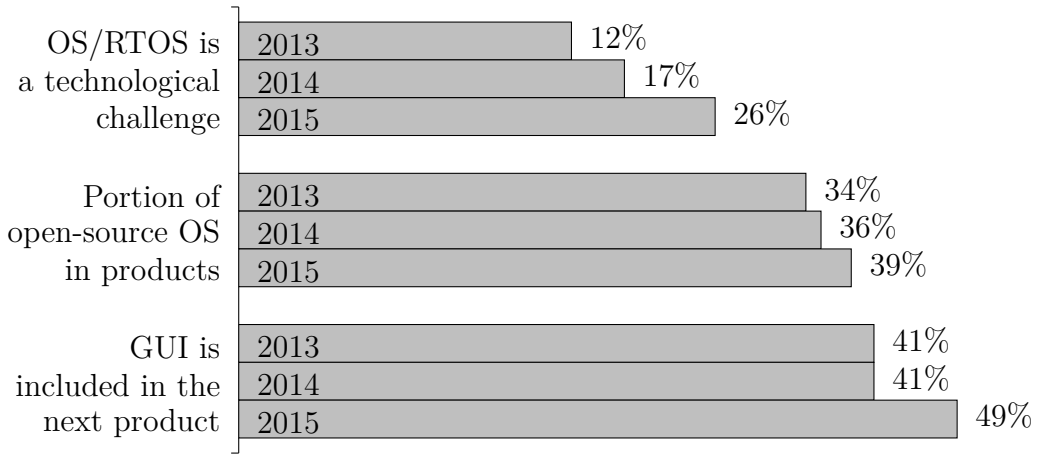


Figure 1.1: Embedded market trends between 2013 and 2015 [122].

1.1 Objectives and Contributions

This thesis implements a conceptual approach for extending the general-purpose Android platform with soft real-time support. Since there is no comprehensive documentation about the internals of Android's architecture, the platform functionality shall be analyzed in order to identify the main sources of unpredictable system behavior. The determined bottlenecks shall be resolved to ensure bounded execution overhead caused by the respective system component. The primary goal is the reliable execution of Android applications on off-the-shelf embedded devices with minimal system latencies. The secondary goal is preserving Android's original application model and allowing conventional applications written in the Java programming language to benefit from the new real-time capable execution environment. The key contributions supporting the objectives of this work are the following.

- Evaluation of techniques to fulfill real-time requirements on the Android platform: This includes the identification of high-level components responsible for unreliable process behavior in Android and the discussion of possible extension strategies.
- Integration of the preemptible kernel from industrial real-time Linux into Android: This is accomplished with a new simplified approach of applying the `RT_PREEMPT` patch to Android's Linux kernel. It allows the usage of priority-based scheduling for Android applications, leading to significantly reduced scheduling latencies.
- Design and validation of the CPU frequency locking mechanism for real-time processes: The presented approach allows reducing non-deterministic scheduling latencies caused by Android's dynamic CPU frequency scaling.
- Development of a non-blocking garbage collector to avoid unexpected suspensions caused by the memory management in Android: It allows concurrent reclaiming of memory and guarantees an undisturbed execution of real-time threads.

- Analysis of the memory adjustment values and extension of the internal process management: This avoids the termination of background real-time processes by the Low-Memory Killer, which is used by Android for memory usage optimization.
- Implementation of the priority inheritance during remote procedure calls using the Binder driver: This effectively bounds the invocation delays between separate processes and allows real-time applications to incorporate functionality implemented as part of other applications in a controlled manner.
- Investigation of Android's internal architecture for IPC messaging and integration of an explicit prioritization mechanism for broadcasted parallel Intents: This ensures the correct ordering of messages transmitted across process boundaries and significantly reduces blocking times during the internal data delivery.
- Evaluation of the practical applicability of the presented approach in the industrial domain: It analyzes the effectiveness of the proposed changes and discusses the effort required to make a general-purpose platform from the domain of consumer electronics eligible for industrial use.

1.2 Thesis Outline

The remainder of this dissertation is structured as summarized in the following. First, Chapter 2 presents an overview of the relevant background information, including the state-of-the-art design methodology for embedded systems and currently available software and hardware platforms. Furthermore, it covers the most important components of the Linux kernel and the Android platform, as well as the main aspects of real-time systems. Then, Chapter 3 summarizes current industrial trends and discusses the importance of open-source-based high-level platforms. It includes the analysis of possible approaches for augmenting Android with real-time support and the integration strategy pursued in this thesis. Chapter 4 focuses on the optimization of the underlying Linux kernel. In addition to improving the kernel's preemptibility with `PREEMPT_RT`, a new method for CPU frequency locking is integrated in order to minimize scheduling latencies. The following Chapter 5 discusses the elimination the undesired process suspensions caused by the automatic memory management. Chapter 6 and Chapter 7 extend the soft real-time support of the new platform by predictable methods for inter- and intraprocess communication. Each of the mentioned chapters also presents the corresponding related work and experimental testing results. A general evaluation is presented in Chapter 8. It addresses the question of the integration overhead required for the introduction of a more predictable system behavior and provides a discussion of advantages and limitations of the resulting platform. Finally, Chapter 9 concludes this thesis and sketches possible directions for future work.

1.3 Bibliographic Notes

This dissertation is based on bibliography of three distinct types: own papers published beforehand, graduation theses supervised by Igor Kalkov and other literature. The original idea was elaborated during the author's master's thesis [59] and the corresponding publication [60]. Both works cover the basic conceptual approach behind this dissertation, including the preemptible Linux kernel, which is explained in Chapter 4, and the preliminary analysis of memory adjustment values, which is part of Chapter 5. However, Section 4.3 of this dissertation additionally presents a novel approach for further latency reduction based on CPU frequency locking.

Ideas behind the new memory management in Chapter 5 were derived from the work of Thomas Gerlitz, implemented in his master's thesis [37] and published in a workshop paper [38]. In addition to his concept of a reference-counting garbage collector for Android, this dissertation also provides an extended evaluation of memory adjustment values [59] in Section 5.3, which was not included in other publications.

Both approaches for the improved interprocess communication were initially published in corresponding papers. The concept of extending the Binder driver, which is presented in detail in Chapter 6, was published in a workshop paper [62] beforehand. The analysis of Android's architecture for Intent broadcasting was originally described in a publication focused on predictable data exchange between multiple Android processes [61]. However, the original technique relied only on implicit prioritization, whereas this thesis presents a more general approach in Chapter 7. This extended approach was implemented and evaluated in the context of a recent journal paper [63].

Further ideas out of the direct scope of this thesis were developed over the course of the dissertation project. The utilization of a real-time capable Android in industrial safety-critical embedded systems [7] was elaborated together with Ashraf Armoush and Dominik Franke. Graduation theses by Jonas Fortmann [34], Mathias Obster [88, 89] and Stefan Rakel [102] have covered the design of an integrated development environment and a Soft-PLC for industrial programming languages standardized in the IEC 61131-3. Further theses by Alexandru Gurghian [45], Philipp Haller [46], Stefan Schake [105] and David Thönnessen [119, 120] focused on the implementation and evaluation of reliable communication based on USB, Bluetooth, PROFINET and Wi-Fi protocols in Android.

Author's Contribution to Included Publications

In the paper [60], the author has contributed the fundamental design choices, the experimental results and the general discussion, which was refined in collaboration with co-authors. Publications [38] and [89] base on graduation theses in which the author has supervised the development of the main approach, contributed to the experimental part and evaluation analysis as well as participated actively in writing the paper. The research for publications [61], [62] and [63] was initiated by the author, where he contributed the key ideas, planning of the empirical evaluation and interpretation of the experimental results. In the paper [7], the author of this dissertation was actively involved into the development of the main idea and preparation of the paper.

2 Preliminaries

This chapter summarizes the basic terminology and fundamental principles used in the thesis. Section 2.1 provides a brief introduction to the development of embedded systems, including the specific aspects of software design methodology and an overview of embedded general-purpose hardware. After that, Section 2.2 covers fundamentals of Linux-based operating systems beginning with the Linux kernel itself in Section 2.2.1. The second part in Section 2.2.2 focuses on the general information about the Android platform, its global architecture and introduces important implementation details. Finally, requirements and solutions for real-time systems are presented Section 2.3.

2.1 Development of Embedded Systems

Embedded systems are computer systems *embedded* into complex cyber-physical environments. They are used to perform a set of predefined functions, running on hardware with constrained resources [110, 117]. Embedded systems are one of the most important yet hidden parts of the modern world. In 2008, for example, the number of microprocessors per person in developed countries passed 30. More than 98% of all manufactured microprocessors are used for embedded systems and not for classical desktop computers [28]. Today, embedded systems can be found in most electronic products: microwave ovens, TVs, mobile devices, industrial machinery and automotive applications. Regardless of whether the embedded component was designed to be autonomous or interactive, its main purpose is linking physical processes with computational logic and data processing. Since the correct reaction to external and internal events determines the behavior of the embedding system, close cooperation between software and hardware components is an essential factor. Depending on the system requirements, embedded systems are typically implemented with application specific integrated circuits (ASIC) or single board computers (SBCs) by using a microcontroller or a system on a chip (SoC).

Recent technological improvements allow modern embedded systems to be more powerful and interconnected than ever before, making it difficult to distinguish between embedded and general-purpose devices. Cheap SBCs like Raspberry Pi[1] or ODROID[2] (see Figure 2.1), which used to be the embodiment of embedded hardware platforms, are getting powerful enough to run general-purpose operating systems like Linux or Android. The most prominent example for such a rapid improvement can be found on the market for modern mobile devices. They emerged from originally serving a dedicated purpose of voice transmission to full-fledged computer systems, being introduced to a big

[1]Raspberry Pi homepage: `https://www.raspberrypi.org/`
[2]ODROID homepage: `https://www.hardkernel.com/`

(a) Raspberry Pi 3 Model B (b) ODROID XU4

Figure 2.1: Popular general-purpose hardware for embedded projects.

range of different use cases. Nevertheless, smartphones and tablets are still submitted to constraints like limited battery lifetime and memory limitations, when compared to general-purpose systems as desktop computers. Main differences between the properties of general-purpose systems and embedded systems are summarized in Table 2.1.

Despite of various use cases available for mobile devices today, their form factor and the limited power supply are still highly important constraints. Because of these limitations – and considering the historical origination of smartphones as single-purpose devices – they will be classified as embedded devices in context of this thesis. Common technological challenges and rising complexity of cyber-physical products are further blurring the borders between these two classes of devices. A comprehensive survey[3] on recent trends in the development of embedded systems has shown a high increase in networking capabilities and the number of common interfaces for data exchange in the past years. About 62% of all embedded devices are already connected to the internet or planned to be connected in the near future. This implies the need for comprehensive methodology and standard software components, which can be easily reused in new applications without jeopardizing security or increasing the production time.

2.1.1 Design Methodology for Embedded Products

As described in the last section, software and hardware components are tightly coupled in embedded systems. Generally accepted methods like Scrum [111] from software development are typically not effective for embedded software-hardware projects. The same applies to approaches arising from the domain of pure hardware design. Naive processes for product development typically begin with detailed hardware planning to meet the project requirements. The software part comes in only after the hardware (prototype) is manufactured and tested. This leads to high costs of changes [44], if incompatibilities between hardware and software are discovered in later phases. Ronkainen and Abrahamsson [103] have introduced the term *hardware-related software* for embedded systems, in which the functionality is split between hardware and software implementation during the development process.

[3]Survey results: `http://www.barrgroup.com/Embedded-Systems/Market-Surveys/2016-Safety-Security/`

Property	General-Purpose Systems	Embedded Systems
Size	The dimensions of general-purpose workstations can be freely selected to fit the hardware which able to provide the required performance.	Embedded systems have to seamlessly fit into the form factor of the specific product. Mobility aspects make smaller sizes even more important.
Power supply	Devices with a stationary installation can be easily connected to a powerful energy source using a power supply cable. They can also make use of active cooling.	Mobile embedded systems can only rely on a small battery. Even with permanent power supply active cooling is often not possible because of the limited space.
Performance	Powerful CPUs and high amount of RAM is common for modern general-purpose systems, allowing them to be used for a variety of different tasks.	Only limited resources are available for data processing and connectivity. Higher performance on mobile devices leads to a shorter battery time.
Connectivity	High-speed communication setups utilizing optical cables or multiple antennas are possible.	Limited resources make the integration of full-fledged communication stacks difficult.

Table 2.1: Property comparison between general-purpose and embedded systems.

Greene has identified the lack of a systematic design approach in projects developing embedded systems [42]. Separate processes for design and development of hardware and software may result in a less optimal final product [36]. As embedded systems have a strong coupling with the physical world, a holistic design process is required to enable a coherent consideration of different perspectives. To overcome this challenge, a number of different approaches have been introduced in the past. Functional decomposition [72] was one of the first methods proposed specifically for designing real-time systems. Henzinger and Sifakis [52] postulate that such approaches have to consistently integrate techniques from multiple worlds including software and hardware development. This requirement is fulfilled by *hardware-software-co-design* [50, 118], which ensures the simultaneous consideration of both hardware and software [126] for each separate feature.

Growing performance of modern SoCs enables the use of supportive tools and advanced programming paradigms, which typically require more resources at runtime. Besides of low-level and specialized languages, object-oriented programming languages like Java are increasingly used in embedded programming [32]. However, this can be explained by the popularity of the language itself, rather than its suitability for this particular purpose. Nevertheless, this trend did shift the focus of embedded projects from hardware development to software development. Using object orientation and design patterns in embedded products based on powerful general-purpose hardware can decrease the product development time [26, 127], while the overall product quality increases.

2.1.2 General-Purpose Embedded Hardware

As described in previous sections, the nature of embedded systems is interfacing and interacting with its physical environment. Changing requirements during any development stage often not only affect the software part, but also lead to adaptations of the hardware layer. Especially if working with custom designed printed circuit boards (PCBs), this may increase both the production cost and the overall product development time.

The modern market for embedded platforms offers a number of different FPGA- and SoC-based solutions. Advanced applications like signal processing or hardware emulation are typically implemented using an FPGA. Although FPGAs provide a reliable solution for industrial applications, the high complexity of the corresponding hardware description languages and proprietary development tools prevent the platform from gaining widespread acceptance in other embedded domains.

Arduino[4] was one of the first affordable embedded boards for general public presented in 2006. While being rather a small-scale device based on the AVR family of 8-bit microcontrollers, its simplicity and high usability caused a revolution in do-it-yourself electronics [68]. Depending on the board variant, Arduino boards provide a rich set of common periphery and hardware interfaces like general-purpose I/O (GPIO), which are easily accessible from user-space applications.

As mentioned earlier, another very popular embedded device for *physical computing* [101] is Raspberry Pi. Similarly to Arduino, Raspberry Pi aims at replacing custom made PCBs for proof-of-concept implementations and small-scale projects. But in contrast to Arduino, Raspberry Pi uses an embedded microprocessor and a more sophisticated system architecture, which provides higher performance and allows the usage of a full-fledged operating systems (OS) based on Linux.

2.2 Linux-based Operating Systems

Linux was created as a free and open-source operating system, compatible to the portable operating system interface (POSIX) standard. Today, it is widely available, highly modular and ported to a number of different hardware platforms. With standard features like multitasking and networking available out-of-the box, Linux has become popular not only for server solutions and for desktop PCs in the private sector, but also in commercial products like mobile devices, internet appliances and automotive applications. It is installed on millions of general-purpose computer systems and used for a wide range of different tasks [78, p. 1]. Recent survey results[3] show that 21% of all embedded devices rely on Linux. Its open-source nature boosts the general innovation process, allowing commercial companies to collaborate on Linux development and to benefit from contributions [49]. In the last decade, several new OSs emerged on basis of the *Linux Kernel*. For example, to the most popular Linux-based platforms for mobile devices belong Ubuntu Touch, Chromium OS, Firefox OS and Android.

[4] Arduino project homepage: `https://www.arduino.cc/`

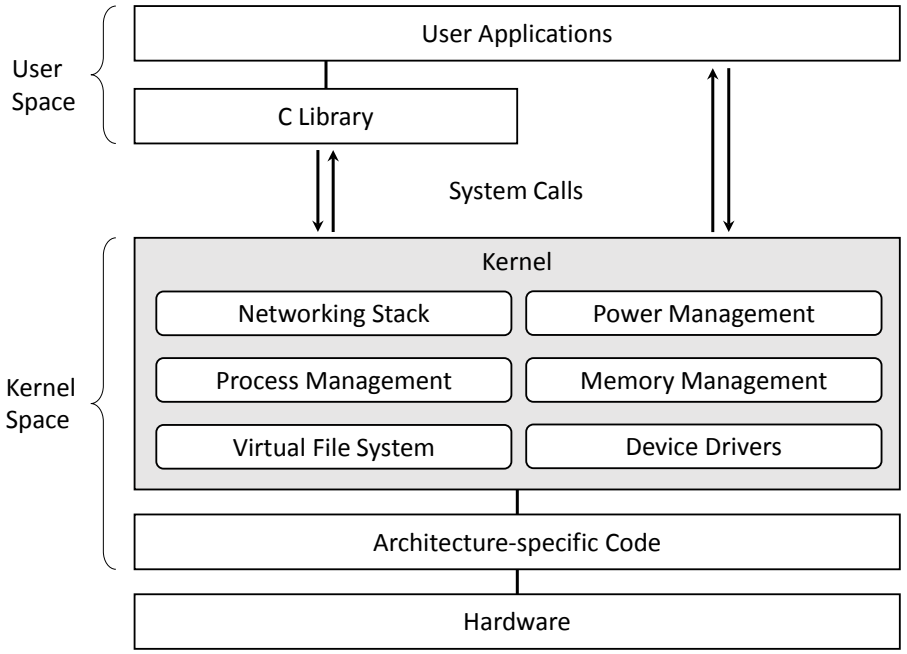


Figure 2.2: Basic structure of the Linux kernel.

2.2.1 Linux Kernel

The kernel is the core component of an operating system, which provides an abstraction layer between user-level applications and the actual hardware. The following description of the kernel architecture and its main elements is derived from Mauerer [78, pp. 2-6].

Linux utilizes a monolithic kernel that can be dynamically extended with additional functionality using loadable modules. This architecture partially compensates the limited flexibility in comparison to microkernels, since separate device drivers for all types of subsystems (memory management, filesystem and others) can be loaded at runtime. A general overview of the kernel architecture gives Figure 2.2.

The architecture-specific code for the target hardware platform, also referred to as board support package (BSP), is used to perform the initialization of the main hardware components like processor, memory or bus subsystem. This layer is accessed by the kernel in order to translate the platform-independent software instructions into the hardware control commands. The kernel itself is divided into multiple elements for essential system tasks like process and power management.

Although there is no explicit distinction between processes (also called *heavy-weight processes*) and threads (also called *light-weight processes*) in Linux, following definitions will be used in this thesis:

- A *program* is a list of instructions required to perform a specific task.
- A *thread* is an independent execution instance of a program or a program part.
- A *process* consists of one or more threads, representing a running program.

One of the most important differences between processes and threads is the resource sharing. Each process has its own independent execution environment with separate context, address space and data structures. Threads are only executed as part of a process. They typically share the memory space and other resources of the corresponding process, introducing almost no overhead at context switches.

Linux utilizes *preemptive priority scheduling*, which enforces active processes to be temporarily suspended for a context switch. At start, each process p is assigned a specific scheduling priority value $s \in [0..139]$, where higher values denote lower process priority [78, p. 93]. This way a running process p_i is preempted every time another process p_j with $s_j < s_i$ is started or woken up. User-space processes are scheduled with the `SCHED_NORMAL` policy based on *nice values* denoted as $n_i \in [19, ..., -20]$. The simplified equation for the calculation of the actual priority $s_i = 120 + n_i$ can be constructed based on the `NICE_TO_PRIO` define [78, p. 94] as shown in Listing 2.1.

```
#define MAX_USER_RT_PRIO 100
#define MAX_RT_PRIO MAX_USER_RT_PRIO
#define NICE_TO_PRIO(nice) (MAX_RT_PRIO + (nice) + 20)
```

Listing 2.1: Calculating scheduling priority from nice value.

Besides of process and memory management, the Linux kernel also provides built-in support for networking, various file systems and peripheral devices. User applications reside on the top, communicating with the kernel space using system calls. This separation creates a beneficial ecosystem and makes it easy for platforms like Android to integrate the Linux kernel for main system tasks.

2.2.2 Android Platform

The Android platform is part of the Android Open Source Project (AOSP), which is developed and maintained by Google and Open Handset Alliance (OHA). The term *platform* indicates that the Android package contains more than just an operating system. Figure 2.3 shows the high-level system architecture[5] of Android.

For easier development and better portability, Android applications are written in Java programming language. All platform applications and the installed third-party software reside on the topmost architectural level. They rely on the Application Framework, which provides an abstraction of the underlying functionality and makes system calls available in Java programming language. This layer also contains the main facilities and services for the management of the user interface and Android-related entities like packages and activities.

Android's core is a set of native libraries written in C++ programming language, highly optimized to be used on devices with limited resources. Besides of Android's own version of the libc library referred to as *bionic* [20, p. 156], media and networking libraries were also optimized regarding energy and memory consumption, as well as for the usage with low powered CPUs. In order to perform the translation of the API calls into the

[5] Android documentation: `https://developer.android.com/guide/platform/index.html`

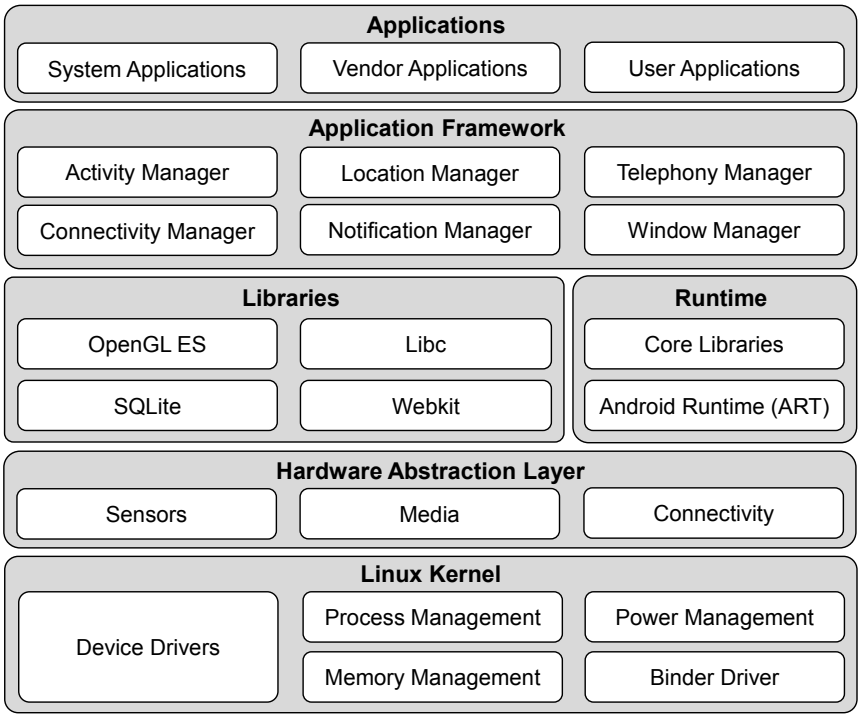


Figure 2.3: Android architecture overview.

native code base, an additional set of Java libraries is also included in Android Runtime, residing in the third layer. It allows the usage of Java API in Android applications and forms the basis for the application framework [82, p. 15]. It also provides the *Dalvik Virtual Machine* (DVM), which is the most essential part of the Android platform. The DVM is a sandboxing mechanism and a byte-code interpreter, which encapsulates instances of running programs and executes the optimized Java code. Every Android application is executed in a separate instance of the DVM with its own memory segment and separate resources. The DVM was designed with focus on efficient parallel execution of multiple VM-instances on a single mobile device [129, p. 60]. Starting with Android 5.0, DVM was replaced[6] by Android Runtime (ART). It preserves the original byte-code format, but instead of Just-In-Time (JIT) compilation used in DVM, ART introduces the Ahead-Of-Time (AOT) compilation approach, which reduces power consumption and improves execution efficiency [56, pp. 65-66].

As already mentioned in the last section, Android is built upon a Linux kernel. It resides on the lowest architectural level and is used for the basic services like process and memory management. Additionally, this layer contains the Linux networking daemons like *bluez* and *wpa supplicant*, which are required for providing the Bluetooth support and the Wi-Fi encryption. In contrast to the mainline Linux, the kernel in Android is also optimized for mobile devices and contains additional Android-specific components[7].

[6]Android documentation: `https://source.android.com/devices/tech/dalvik/gc-debug.html`
[7]Based on "Clarification on the Android Kernel": `http://lwn.net/Articles/373374/`

Android Extensions for Linux

Several hundred of additional patches were merged into the Linux kernel used in Android. The kernel contains a number of additional system components, sometimes referred to as *Androidisms* [129, pp. 34-45]. The following description briefly presents extensions which are most relevant in context of this thesis.

- **Low-Memory Killer** is included in Android to improve the handling of out-of-memory (OOM) situations on devices with limited resources. It is based on the native Linux OOM adjustments and specifies the order of process termination if the available system memory falls below a predefined threshold. Multiple policies define distinct priority classes for active processes that have not been used in a long time. These priorities describe which process is a better candidate for being killed, which mainly depends on the process category (e.g. system component or user application) and the degree of involvement into the user interaction (e.g. visible element or hidden service).

- **Binder Driver** is part of Android's architecture for remote procedure calls (RPCs) which allows efficient data exchange across process boundaries. This kind of inter-process communication (IPC) makes use of object invocation capabilities, which allows system services running in separate processes to be handled as standard Java objects in user applications. The actual remote service can be implemented in any supported programming language and executed in a separate process or be bundled with other services in a single process and share its address space.

- **Anonymous Shared Memory** (ASHMEM) is another Android-specific IPC mechanism. The anonymous shared memory is used instead of the Linux's SysV IPC in order to avoid resource leakage and improve system security. Most significant differences to POSIX SHM are the usage of reference counting for the detection of unused memory regions and automatic shrinking in OOM situations. Although the ASHMEM is widely used by the core components of the Android runtime, its interface is not exposed through the public API.

Further Linux extensions like the new alarm and logging subsystems are essential parts of the overall system architecture. Altogether, they form several Android-specific concepts for the application development.

Android Concepts

Developers have to incorporate a set of unique Android APIs, when designing new system-level or user-level components [129, pp. 26-30].

Android application are created using four main types of loosely coupled building blocks as described in Table 2.2. To implement the desired functionality, one or multiple of the required components can be combined in a single application. The platform was designed to be modular and extendable, such that components of one application can

Component	Description
Activities	Provide the graphical interface with control elements like buttons and text labels. Although Activities can contain program logic, it is only triggered while the Activity is visible.
Services	Allow execution of the program logic even if the corresponding application is currently in the background. As services do not contain UI elements, they are used for long-term operations like monitoring and synchronization tasks.
Broadcast Receivers	Serve as handlers for specific system- or user-defined events. They are implemented using the Binder driver and get triggered by the system as soon as the respective event occurs.
Content Providers	Allow persistent storage and sharing of structured data between different applications or application components.

Table 2.2: Description of the main application components in Android.

call or reuse components of other applications. The interaction between distinct applications or internal components relies on *Intents*. They are mainly used to trigger specific events in other applications like starting Activities or Services. Furthermore, Intents and Broadcast Receivers provide a simple mechanism for the exchange of structured data between different components of the same process or across process boundaries. It is the recommended way of interprocess communication due to Android's security model, which includes strict sandboxing and fine-grained access control at the process level. All application components (e.g. multiple Activities or Services) are bundled into a single process by default[8], taking advantage of the Linux user-based protection. Unique Linux user identifier (UID) and group identifier (GID) values are assigned to every application, preventing unauthorized access to private data.

In addition to existing concepts for process control and interprocess communication included in Linux by default, Android also includes similar functionality on a higher architectural level. Some of these instruments – for example the message passing between different processes or the internal management of application life cycles – are relevant in the context of this dissertation and will be briefly presented next.

Intent Broadcasting

Utilizing Intents and Broadcast Receivers allows the exchange of structured data across process boundaries without jeopardizing the strict process isolation. For this purpose, Android provides dedicated system components responsible for delivering of global and local broadcasts. In case multiple receivers were registered for the same event and its processing order is important, *ordered* broadcasts can be used. After the Intent object is delivered, the receiving application can abort further processing, ignoring the remaining receivers. *Sticky* broadcasts are persistently stored in the system. They are delivered

[8]Android documentation: `http://developer.android.com/guide/topics/fundamentals.html`

automatically as soon as a new Broadcast Receiver with matching filter is registered. The most common type is represented by *parallel* broadcasts, which are delivered to the corresponding receivers without any particular ordering.

In Android, parallel broadcasts can be transmitted either globally or locally, depending on whether the target Broadcast Receiver is part of the same application. The vast majority of Intents are broadcasted globally by using the Activity Manager Service (AMS), which is part of Android's application framework. It is invoked automatically each time an application passes an Intent object to the method `sendBroadcast()`. The AMS manages a list of all known receivers, which is used to extract applications interested in this specific Intent. Finally, AMS delivers the Intent object to the filtered Broadcast Receivers. This approach allows Android to enable flexible communication between separate applications while preserving the sandboxing mechanism.

For applications with direct user interaction, broadcasted Intents may activate the `FLAG_RECEIVER_FOREGROUND`. It is used to mark data of high importance and request the system to handle the corresponding Intent with higher priority.

The application framework of Android provides a Local Broadcast Manager (LBM), which is designed to process broadcasts in the context of the same application. If both sender and receiver are created inside the same process, the LMB significantly increases the broadcasting performance, since no interaction with other processes is required. Similar to the global approach, sending local broadcasts is performed by using the method `sendBroadcast()`. However, it has to be invoked using a singleton instance of the class `LocalBroadcastManager`. This class also provides the method `sendBroadcastSync()`, which can be used for an immediate delivery of the Intent object by the sending thread.

Process Management

Each Android process is managed by its own main thread. This thread – commonly referred to as *UI thread* [81, p. 345] – is responsible for the application's life cycle management and the handling of user input. Other threads inside the same process will be initially assigned a lower priority than the UI thread. Nevertheless, additional threads for controlling UI elements are considered more important and thus will have a slightly higher priority value than a thread for pure background computations.

Android relies on the Linux kernel's priority-based scheduling mechanism for thread management. Application threads are scheduled using the `SCHED_NORMAL` policy with predefined[9] nice values as shown in Listing 2.2. Android also deploys *cgroups* to prevent all background threads from using more than 5% of the combined CPU time[10].

As described in Section 2.2.2, Android may kill active processes if the system is running out of memory. In this case, the process to be killed is selected based on Android's importance hierarchy[11] as depicted in Figure 2.4. In order to preserve the correct behavior of the foreground application the user may currently be working with, cached and background processes are killed first. The importance class can also be estimated

[9] From the source code analysis of Android 5.1.1 in `system/core/include/system/thread_defs.h`
[10] From the source code analysis of Android 5.1.1 in `system/core/rootdir/init.rc`
[11] Android documentation: `http://developer.android.com/guide/topics/processes/process-lifecycle.html`

```
/* use for background threads */
ANDROID_PRIORITY_BACKGROUND = 10,
/* most threads run at normal priority */
ANDROID_PRIORITY_NORMAL     =  0,
/* threads currently running a UI that the user is interacting with
   */
ANDROID_PRIORITY_FOREGROUND = -2,
/* the main UI thread has a slightly more favorable priority */
ANDROID_PRIORITY_DISPLAY    = -4,
```

Listing 2.2: Translation of thread priorities to nice values in Android.

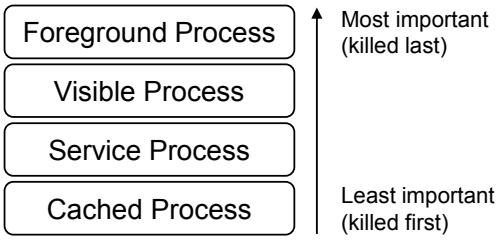


Figure 2.4: Process importance hierarchy in Android.

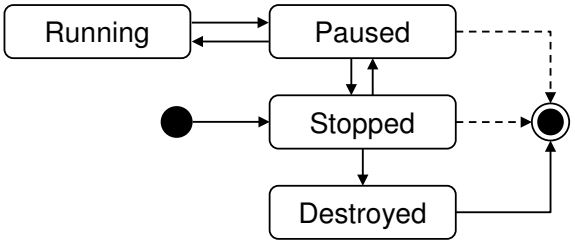


Figure 2.5: Simplified life cycle of Android applications.

for applications with multiple different components bundled into a single process. In this case, the overall importance class is set to the class of the bundle's most important component. This allow non-interactive background services to be executed in a high-prioritized process, if the given application also handles UI elements.

The activity diagram in Figure 2.5 shows a simplified version of the life cycle model for Android Activities, based on the observed system behavior [35, p. 21]. The application framework has to notify the running process in case of an enforced termination and switch it to the *destroyed* state. This allows a graceful shutdown before the process is actually killed by the operating system. However, experiment results show that running applications can also be terminated from states *paused* and *stopped* without a prior notification (denoted in the diagram by dashed transitions). Such life cycle model is sufficient for non-demanding use cases in consumer electronics, but to be suitable for applications with real-time requirements, the system has to guarantee a deterministic and predictable behavior.

2.3 Designing Real-time Systems

Embedded systems are typically deployed in safety-critical environments where they have to react to external stimuli within a predefined period of time. Missing a deadline while waiting for a calculation result might yield severe consequences. According to Stankovic, considering only the computational result is not sufficient for real-time systems [116]:

> "[...] the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced."

Unmodified general-purpose OS like Linux and Android cannot guarantee a maximum reaction time for an external event in complex dynamic processes. The main challenge in augmenting a GPOS with real-time capabilities is to provide low-level control over the most important system components, which commonly conflicts with high-level concepts like encapsulation and inheritance.

The next section gives an overview of the formal definitions for real-time characteristics used in this work. Sections 2.3.2 and 2.3.3 briefly introduce already available hardware and software solutions for the development of real-time systems.

2.3.1 Real-time System Characteristics

In the Linux kernel, real-time processes use the scheduling policies `SCHED_FIFO` and `SCHED_RR` with priority values in the range of 0 to 99 (higher priority), while normal processes always have the priority value between 100 and 139 (lower priority) [78, pp. 93-94]. A task is called *periodic* if it has to be activated by the system scheduler at a fixed rate. The opposite type is represented by *sporadic* tasks, which are typically triggered by an irregular internal or external event.

Real-time operating systems (RTOS) must ensure predictable timings for all kinds of active processes. In reality, the scheduler cannot immediately switch to another process after it becomes eligible for execution at its *activation time*. The processor may be busy executing another with higher priority, leading to a delay. Even if no other process with a higher priority is blocking the CPU currently, a context switch to the new process produces additional overhead which may vary depending on the current system state. The time between the activation time and the actual execution of the corresponding process is called scheduling *latency* [1]. As it is essential in real-time systems to account for the worst-case scenario, knowing the possible maximum latency value is crucial. For a predictable and deterministic system behavior, a RTOS has to limit the maximum latency for high-prioritized processes. Furthermore, a fixed upper bound for maximal scheduling latency is required for the calculation of the maximum *reaction time*, which a process needs to respond to internal and external events.

Two main classes of real-time systems can be distinguished depending on the consequences of missing a firm deadline. In *soft real-time* systems, missed deadlines may have unwanted, but not destructive effects. Calculation results in *hard real-time* systems must be delivered in time even under worst-case conditions. Otherwise, they are considered useless and the system state can lead to a serious hazard for human and machinery [71]. Examples for soft real-time systems can be found in vending machines and video streaming applications. Hard real-time systems are typically used in automotive and medical applications. To guarantee the required timing behavior, hard real-time systems are commonly implemented using low-level or specialized hardware.

2.3.2 Hardware for Real-time Computing

Various kinds of digital controllers designed for dedicated performance ranges can be used to implement and deploy industrial real-time systems [83]:

- **Application-specific integrated circuits** (ASICs) are tailored to the intended purpose, such that their logic is hard-wired into an integrated circuit. This allows ASICs to provide high performance in their specific applications, but to gain only a low score in terms of feature flexibility and development ease.
- **Digital signal processors** (DSPs) and **FPGAs** also provide high performance, but the application logic can be modified by re-programming the same device. The use of proprietary solutions based on DSPs like "dSpace" is common for rapid control prototyping (RCP) and hardware-in-the-loop (HiL) purposes. Besides of high investment costs, the application-specific configuration of such products requires expert knowledge of specialized modeling (e.g. Matlab/Simulink) or hardware description (e.g. VHDL) languages.
- **SoC- and microcontroller-based boards** for general-purpose use were already briefly presented in Section 2.1.2. Besides of simple re-programmability using high-level programming languages, such devices contain digital and analog I/O as well as additional peripherals like communication interfaces, timers and event counters. Solutions based on such flexible platforms are suitable for a wide range of data processing and control applications.
- **Programmable logic controllers** (PLCs) are microcontroller-based devices for industrial purposes, providing rugged design for an in-field deployment. Their modular architecture allows simple components upgrades and industry-ready I/O connections. In contrast to general-purpose embedded boards, PLCs are programmed by using standardized industrial programming languages [55].
- **Industrial PCs** with general-purpose hardware have been shown suitable for soft real-time applications. They can be used for industrial monitoring and control when combined with real-time OS extensions or virtualization methods.

2.3.3 Real-time Operating Systems

As described in Section 2.3.1, RTOS must provide a deterministic system behavior and minimize scheduling latencies. Nowadays, there are multiple commercial and non-commercial real-time capable OS available on the market for both specialized and general-purpose hardware. The most prominent RTOS products are QNX Neutrino[12], VxWorks[13] and RTEMS[14]. Furthermore, hard real-time requirements can also be fulfilled

[12]QNX Neutrino RTOS by QNX: `http://www.qnx.com/`
[13]VxWorks by Wind River Systems: `http://www.windriver.com/`
[14]RTEMS project homepage: `http://www.rtems.com/`

with a conventional desktop operating system by using a *hypervisor* or *OS virtualization*. In this case, the guest OS is completely encapsulated by the hypervisor and executed as a preemptible low-prioritized process. Examples for hypervisor-based solutions are SIGMA [64] and INTEGRITY Multivisor[15].

In addition to resource reservation and system virtualization, preemption enhancement methods are the most important techniques to enable real-time support in the Linux kernel, as described in the following [114], [130, pp. 360-364].

Resource Reservation

In order to ensure that real-time processes receive sufficient CPU time, even if non-real-time processes allocate major system resources, processor reservation techniques were presented in the past [2, 84, 112, 113]. Processes with assigned CPU reservation are guaranteed to be executed for a predefined time slice in each activation period, enforcing a preemption of other user-space activities. The selected time slice is applied in a strict manner, such that the CPU is released after this time slice expires. This may suspend the active real-time process even if the calculation was not finished yet.

Interrupt Layer Virtualization

Interrupt abstraction introduces an additional separation layer between the Linux kernel and the hardware to gain control over the timers and interrupts. This allows a selective prioritization of interrupt sources and forwarding of real-time interrupts into a dedicated subsystem for immediate execution. Such architecture reduces scheduling latencies for real-time handlers and creates a lightweight virtual machine where the Linux kernel is executed as a preemptible process as long as no real-time processes are active. Approaches for the virtualization of the interrupt controller were successfully implemented in RTLinux [134] and RTAI [77]. Xenomai [39] is an adaptation of RTAI with even deeper virtualization and explicit domain separation between the real-time subsystem and the Linux scheduler. Additionally, Xenomai can be combined with other solutions like `PREEMPT_RT` in order to achieve further latency improvements.

Preemptible Kernel

An alternative approach to reduce scheduling latencies is proposed by an enhanced preemption model of the Linux kernel itself. The `PREEMPT_RT` patch by Ingo Molnar makes in-kernel spinlocks preemptible, allowing the scheduler to switch to a high priority process almost immediately after its activation time, even if a low priority process currently executes a critical section in kernel space. In order to avoid the arising problem of *priority inversion*, new spinlocks implement a *priority inheritance* protocol. Furthermore, the `PREEMPT_RT` patch additionally enables the use of a high-resolution timer and converts all blocking interrupt service routines (ISRs) to preemptible kernel threads.

[15] INTEGRITY Multivisor by Green Hills: `http://www.ghs.com/`

These features have to be explicitly activated at the compilation time by using the new kernel configuration options[16]:

- `CONFIG_PREEMPT_NONE`: The default configuration in Linux. The preemption model is optimized for throughput and does not provide any timing guarantees.
- `CONFIG_PREEMPT_VOLUNTARY`: Introduces explicit preemption points such that low priority processes can voluntarily preempt themselves during system calls.
- `CONFIG_PREEMPT_LL`: Makes nearly all kernel code outside of explicit critical sections preemptible and offers immediate event handling.
- `PREEMPT_RTB`: The basic preemption mode enables the necessary minimum of the introduced enhancements for systems with soft real-time requirements.
- `PREEMPT_RT_FULL`: Enables the full kernel preemption. This configuration option should be activated for guaranteed latency bounds in time-critical systems.

As a number of changes introduced by the `PREEMPT_RT` is also useful for non-real-time systems, parts of the patch are gradually merged by the community into the Linux code base. Except for the full real-time preemption mode, almost all other features are already included into the mainline Linux for CPU architectures `x86/x64` and `ARM`[17].

Several research groups have evaluated the effect and the impact of `PREEMPT_RT` on the system behavior in terms of latency reduction and determinism. Different evaluation studies [67] and performance assessments [97, 8] have shown that general-purpose hardware with `PREEMPT_RT` can fulfill soft and even hard [5, 87] real-time requirements. Although the statistical results seem conclusive, the suitability of patched Linux in safety-critical systems is still a highly controversial topic. Yaghmour et al. question its general applicability in setups where a deadline miss or a system fault may endanger human or machinery [130, p. 363]:

> "For most applications that need real-time determinism, the RT-patched Linux kernel provides adequate service. But for those real-time applications that need more than low latencies and actually have a system that can be vigorously audited against bugs, the Linux kernel, with or without the RT patch, is not sufficient."

Nevertheless, `PREEMPT_RT` was shown effective and flexible, leading to its wide use particularly in the field of industrial automation[18]. For this reason, it will be considered in addition to other methods as an adequate solution for augmenting Android with real-time support.

[16] Based on WindRiver's Linux guide: `https://knowledge.windriver.com/en-us/000_Products/000/010/040/020/000_Wind_River_Linux_Kernel_and_BSP_Developer's_Guide,_8.0/020/040/000`

[17] Feature overview: `http://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html`

[18] Industrial use of `PREEMPT_RT` by OSADL e.V.: `http://www.osadl.org/`

3 Concept for an Industrial Android

A common way of implementing embedded control systems is to split regular tasks (e.g. graphical user interface – GUI) from the actual control logic into independent subsystems or even separate physical devices [83]. In such configuration, a real-time capable embedded controller can benefit from an undisturbed program execution, while non-critical data visualization and user interaction is handled by other hardware. A major disadvantage of this separation is the additional overhead for communication and synchronization of data between multiple devices. To ensure a high grade of reliability, off-the-shelf RTOS typically focus on computational predictability and small memory footprint, offering only rudimentary tools for GUI design. On the contrary, GPOS offer rich functionality and ready-to-use components for building intuitive user interfaces, but provide no real-time support.

Despite the apparent unsuitability of modern GPOS platforms for industrial use, recent survey results show that Android and desktop OS like Ubuntu are increasingly used in embedded products for monitoring and controlling purposes [122]. Although Android is mainly targeting the private user segment, it already represents the second most used OS for embedded systems as shown in Figure 3.1. The biggest challenges for industrial-wide application of Android remain its current real-time limitations and possible security vulnerabilities[1].

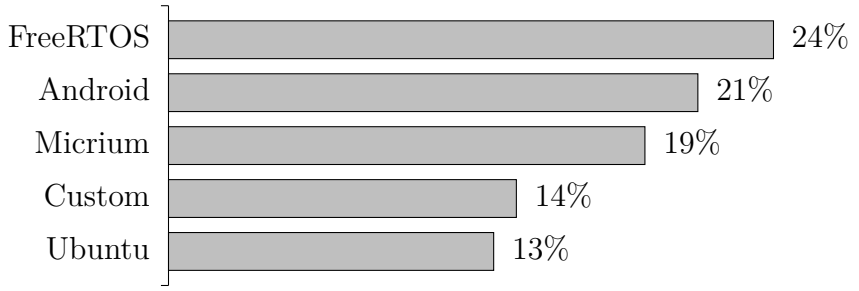


Figure 3.1: Most popular embedded operating systems in 2015 [122].

In addition to the desired functional (e.g. support for real-time computing) and non-functional (e.g. intuitive UI of the resulting applications) requirements mentioned above, other selection factors may also play an important role when choosing an OS for embedded products. According to survey results shown in Figure 3.2, the most relevant factor is the availability of the OS source code, which highlights the industry's interest in open-source software. Further decisive criteria include broad hardware compatibility, availability of integrated development tools, personal familiarity and ease of use.

[1]From online article: `https://www.linux.com/news/android-follows-linux-wide-world-embedded`

Figure 3.2: Top selection factors for embedded operating systems in 2015 [122].

Being fully compliant with the vast majority of customer and developer expectations mentioned in the survey results, it is not surprising that Android continuously increases its popularity on the embedded market. This can also be observed in the rising number of scientific publications and various industrial use cases related to the Android platform. Android-based devices are deployed for patient monitoring [128] and data analysis [41] in the health-care sector, as well as for automotive infotainment [14] and cyber-physical systems [100]. The platform has also been evaluated in more complex automotive setups like ABS control [66, 123], system diagnostics [125] and energy management [16, 23]. Further industrial use cases can be found in the field of robotics [92, 95], navigation control [47, 94] and artificial satellites[2].

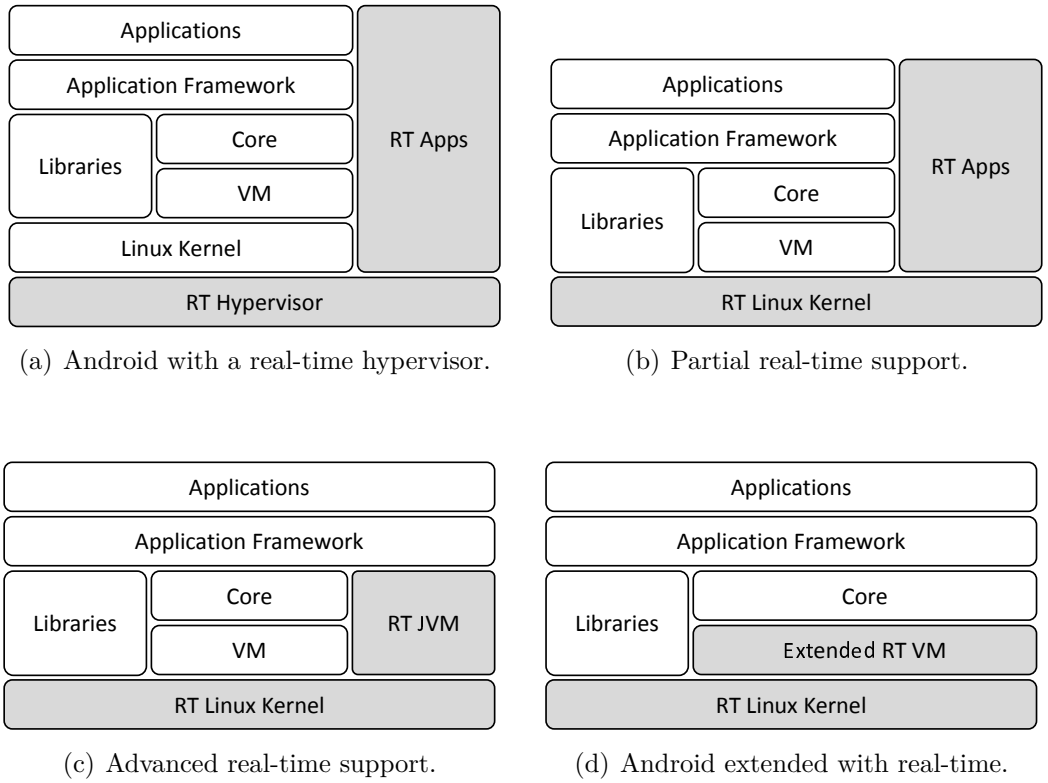
Several of the mentioned applications have real-time requirements with strict timing constraints that must be met by the operating system for a faultless operation. The question of Android's general reliability and suitability for real-time processing was evaluated in different scientific studies as presented in the following.

3.1 Related Work

One of the first evaluations of Android's real-time capabilities was performed by Maia et al. After a detailed analysis of the original platform, Android was concluded to be incapable of providing real-time guarantees due to the internal system architecture [74]. This statement was confirmed by results of experimental latency measurements by Mongia and Madisetti, who conducted responsiveness tests in different system load scenarios: with no load, with normal load and with heavy load [86]. Corresponding results have shown reproducible violations of predefined scheduling deadlines reaching from 1 millisecond up to ½ second, demonstrating the inability of Android to provide reliable process scheduling. Other research groups made similar observations at later stages [79, 95, 96, 133].

In addition to the architecture evaluation, Maia et al. have identified the most critical high-level components – for example process and memory management – which need to be modified or replaced for achieving a more deterministic system behavior. Addition-

[2]PhoneSat project homepage: `http://www.phonesat.org/`



(a) Android with a real-time hypervisor.

(b) Partial real-time support.

(c) Advanced real-time support.

(d) Android extended with real-time.

Figure 3.3: Strategies for the integration of real-time support into Android [74].

ally, the authors have proposed different approaches to extend Android with real-time support [74], as illustrated in Figures 3.3(a) to 3.3(d).

The first approach shown in Figure 3.3(a) utilizes a real-time hypervisor in order to split the real-time domain and the actual Android platform into distinct priority categories, which can be executed side by side. As it was shown in Section 2.3.3, system virtualization strategies and Linux-based real-time solutions like RTLinux or RTAI are commonly used in embedded systems. Similar system designs for hypervisor-based coexistence of Android and RTOS on mobile devices were also proposed by Mentor Graphics[3] and WindRiver[4]. On true parallel multi-core hardware this approach may provide a fully concurrent execution, where Android and real-time application are executed simultaneously without negatively affecting each other. On the other hand, such separation can lead to a limited system functionality available to real-time tasks [74], as they are running detached from the Android framework directly on the top of the hypervisor and only have access to low-level interfaces.

[3]MentorGraphics webinar presents hypervisor solutions for Android: `https://www.mentor.com/embedded-software/events/android-linux-real-time-webinar`

[4]WindRiver solutions for Android: `https://www.windriver.com/announces/android-solutions/`

The replacement of the underlying Linux kernel is an alternative approach chosen by strategies 3.3(b) to 3.3(d). In this setup, the Android platform is modified to run on the top of a real-time capable Linux and to utilize its API for CPU and process control.

Given the real-time support from the kernel, native applications can be executed directly on the top of Linux, as shown in Figure 3.3(b). This approach is comparable to a hypervisor-based setup, as the process scheduling is done by a RTOS for example based on process priorities. As shown by Mauerer et al. [79], it can provide significant improvement of execution determinism and process behavior in terms of predictability and timeliness. Still, this approach is affected by the same limitation – real-time applications cannot access the functionality provided by the Android framework.

The approach in Figure 3.3(c) addresses the disadvantage of missing high-level API in real-time applications by introducing a real-time capable Java VM (JVM) in addition to ART or DVM (see Section 2.2.2). Integration of an off-the-shelf RT-JVM into the Android Runtime allows execution of Java bytecode in a controlled and deterministic manner, providing the necessary timing guarantees. This idea goes in accordance with the lasting interest of scientific community to deploy Java-based controllers in time- and security-critical environments [15, 22, 40]. For this purpose, different advanced real-time capable VMs like KVM [73], Jamaica VM [53, 54] and Fiji VM [98, 99] were presented in the literature. Yan et al. have demonstrated the *RTDroid* project [132, 133] as a proof-of-concept integration of the Fiji VM into Android. While more details of their work will be presented over the course of this thesis, the main drawback of such implementation is the missing native connection to the Android framework. Similar to the previously introduced approaches, the plain Fiji VM does not provide any interfaces for utilization of Android APIs in real-time applications. As stated by its authors, "RTDroid provides a faithful illusion to an existing Android app running on our platform that it is executing on Android" [133]. It can be assumed that such approach – described in their work as a "clean-slate design" – requires a significant amount of implementation and integration overhead for re-architecting the platform, creating an adequate runtime environment for the introduced RT-JVM and extending the latter to translate calls between real-time applications and the official Android API.

The last method presented by Maia et al. for providing real-time support in Android is depicted in Figure 3.3(d). In contrast to previous approaches, it aims at extending the original platform components with predictability and determinism. While this method entails considerable effort for analysis and modification of the underlying architecture, different projects have demonstrated its suitability for augmenting standard Linux and Android architectures with real-time support. This was achieved by implementing additional techniques of resource reservation [75] and advanced dynamic process scheduling [43] on the top of the platform architecture. Both techniques have shown that the evaluated system was able to provide reliable process behavior and guaranteed responses within a predefined time frame [74].

3.2 Discussion of Possible Approaches

While the performance of embedded devices continuously increases, limited predictability and unbounded scheduling latencies of mobile software platforms prevent their usage in the domain of time- and safety-critical systems. As shown in previous sections, current research is focused on combining Android's rich functionality in terms of reusable high-level components with reliability and determinism of a real-time OS. A separated real-time ecosystem implemented solely for native Linux applications on the top of a real-time hypervisor (Figure 3.3(a)) or on the top of a real-time Linux kernel (Figure 3.3(b)) will suppress the advantages of the Android platform. This kind of architecture can be justified if the Android API is only necessary for non-real-time applications running on the same device. In case high-level frameworks or advanced programming paradigms need to be available in the real-time domain, this approach is not sufficient.

A platform with a seamlessly integrated RT-JVM does not suffer from this disadvantage (Figure 3.3(c)). Besides of creating new opportunities by the usage of Java API, modern solutions compliant with RTSJ [40] like Jamaica VM or Fiji VM are capable of precise timings and hard real-time processing. As such off-the-shelf products commonly provide own components for thread and memory management, a redesign of the platform architecture may be required during the integration. This approach provides the best real-time performance in terms of deterministic behavior and minimal scheduling latencies in comparison to other methods, as the interaction between the introduced real-time capable components can be designed from scratch. Nevertheless, the integration process itself is not trivial. Besides of the actual implementation overhead, introduced modifications have to be maintained and updated with every new Android version. This can become challenging, as Android updates are released every six months and often include major changes in virtualization mechanisms, memory management, system libraries and internal APIs. If calls to the high-level Android API are encapsulated by the RT-JVM, changes in platform components may pose a particular obstacle if they contradict or cancel out the behavior improvements gained from the additional VM. This effect can also be caused at runtime by forwarding method calls from real-time applications into the Android framework. As RT-JVM and Android's internal framework provide competitive execution environments, leaving the boundaries of a real-time process and executing unmodified Android code may introduce unbounded delays and have negative impact on the process behavior. It is therefore inevitable to analyze and improve Android's components even if an off-the-shelf RT-JVM is additionally integrated into the platform for full real-time support.

As the integration of Android APIs into a separate real-time domain leads to major architectural changes regardless of the selected approach, the method of extending Android's native components (Figure 3.3(d)) can be assumed to allow the highest implementation flexibility. In this case, essential real-time features are not provided by a dedicated RT-JVM, but must be implemented separately, significantly increasing the implementation overhead. Nevertheless, the required changes can be tailored specifically to Android's original architecture. The possibility to integrate the optimal combination of techniques and methods for real-time support allows reaching the expected perfor-

mance and maximizing portability to new Android versions at the same time. Instead of creating competitive real-time subsystems, the concept proposed by this thesis enables a holistic view on all central parts of Android's architecture.

3.3 Adding Real-Time Support to Android

The real-time integration approach pursued in this thesis is based on the extension of Android's native components. It will be shown that modifications in multiple parts of Android architecture are required for achieving the real-time performance on the application level. Figure 3.4 summarizes these main system components including their most important features, which will be analyzed and modified over the course of this work. The structure of the implementation chapters follows a bottom-up design to resolve architectural dependencies.

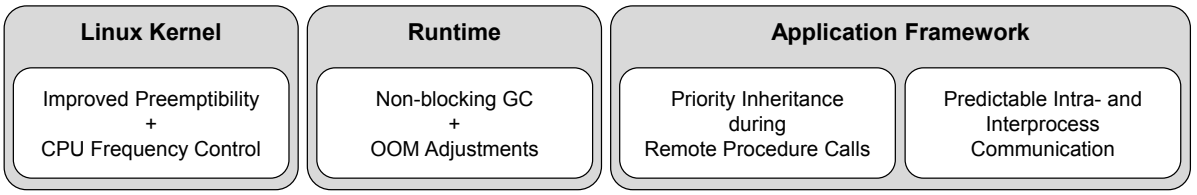


Figure 3.4: Android subsystems extended for integration of the real-time support.

Android Linux Kernel

The Linux kernel is responsible for process management in Android, providing the foundation to other system layers. Although the Android framework includes own mechanisms for taking influence on running processes, the kernel plays a crucial role in the life of active applications and services. Reliable process management and bounded scheduling latencies are essential for deterministic behavior of the whole platform. Since these requirements are not fulfilled by the original kernel, this thesis presents an approach based on the `PREEMPT_RT` patch to improve the kernel's preemptibility and extend it with a real-time capable, priority-based scheduler. It is also analyzed how the specific kernel configuration affects the real-time behavior of native applications. Furthermore, Android's kernel provides advanced low-level features that can have severe impact on the timeliness of running tasks. In addition to the kernel extension, the presented approach introduces full control over dynamic CPU frequency scaling.

Android Runtime

As described in Section 2.2.2, Android Runtime allows the usage of high-level APIs by providing a virtual machine for optimized Java code. The Dalvik VM, which was used in versions prior to Android 5.0, and the new Android Runtime (ART) both provide own mechanisms for automatic memory management. The included garbage collector

(GC) is automatically invoked each time an application is running out of memory. Both VMs implement a stop-the-world garbage collection paradigm in order to ensure data consistency in the target application. However, this method introduces non-deterministic interruptions as all application threads have to be suspended until the GC invocation is finished. Instead of executing the application logic, one or multiple time slots may be spent for garbage collection. This effectively prevents the application from gaining benefit out of the deterministic scheduling provided by the real-time Linux kernel, since the execution path of the application's logic can be interrupted at any time. Applications with real-time requirements need a more predictable solution in order to avoid high latencies caused by GC invocations [86]. This thesis presents an approach to improve the responsiveness real-time processes by introducing a non-blocking memory management. Instead of relying on the native stop-the-world algorithm, the proposed GC does not affect real-time applications. Unused memory objects are collected concurrently to real-time threads instead of suspending them. This avoids interference with the application logic and facilitates predictable process behavior.

In addition to the process scheduling controlled by the Linux kernel, Android's framework provides advanced mechanisms for system resource saving. Evaluations show that these mechanisms can negatively affect long-term background processes and applications with high memory demands. This work presents a detailed analysis of Android's internal OOM adjustment values and evaluates a possible protection mechanism.

Android Application Framework

Native Android APIs can be leveraged in real-time applications in the standard way by incorporating the Binder driver for remote procedure calls. The Binder represents the central mechanism for application linking (see Section 2.2.2). It has a strong impact on the effectiveness and timeliness of inter-process communication, as it passes the execution control across process boundaries. A deterministic behavior of the Binder driver is especially required when real-time applications interact with other platform components. The presented approach includes an in-depth architecture analysis of the Binder subsystem and proposes a priority-aware extension to the corresponding kernel module and its middleware wrapper. Evaluation of the modified system indicates the increased performance during the RPC execution and reduced processing latencies.

The Binder driver is particularly important for Android's generic message passing framework based on Intent broadcasting. It will be shown that even the extended Binder cannot prevent non-deterministic transmission delays when Intents are used for exchanging application data between separate processes. As bounded IPC is crucial for a real-time capable operating system, the proposed solution incorporates non-blocking operations for priority-based Intent handling.

4 Real-time Linux Kernel

Integration of real-time capabilities into the Android platform cannot be fulfilled without extending the underlying Linux kernel. The most prominent implementations and commercially available products for real-time Linux were already presented in Section 2.3.3. While all of them are suitable for reliable process scheduling, different approaches vary greatly in terms of overhead required for their integration in Android. Since many of the platform workflows internally rely on the kernel, the selected solution must provide sufficient compatibility to main Android components.

A full replacement of the original kernel by off-the-shelf products like RTAI or Xenomai requires a major redesign of the low-level architecture and the hardware abstraction layer. Such products do not include the required Androidisms (see Section 2.2.2) in their distributions, being not capable of running the Android platform by default. Android extensions like Wakelocks and the Binder driver have to be implemented and integrated separately, while taking into account requirements of the specific kernel. A particular challenge arises from the internal architecture design in products like RTLinux. Virtualization of the interrupt layer introduces a major obstacle for the communication between regular and real-time processes. The inability of both RTLinux and RTAI to use mainline Linux device drivers in the respective real-time subsystem leads to code duplication and additional implementation overhead [114]. One example is the network protocol stack, which was implemented in both products for the second time specifically for real-time processes. It can be reasonably assumed that this limitation also applies to the Andoidisms, which would have to be integrated separately into the standard and real-time application domains.

In contrast to the deployment of a commercial solution with a micro kernel or interrupt virtualization, the application of the `PREEMPT_RT` patch to Android's standard Linux kernel does not require any changes on the architectural level. This patch is designed explicitly as an extension to the mainline Linux and it has already proven its effectiveness in the industry (see Section 2.3.3). The main goal of `PREEMPT_RT` is the integration of real-time capabilities into the standard Linux by making it fully preemptible. Since the proposed modifications are also useful in a big number of other application scenarios, the patch is being gradually merged into the main kernel repository. Figure 4.1 illustrates the evolution[1] of the patch's size between version 2.6.15-rt21 released in 2006 and the newest[2] available version 4.9.13-rt10. It shows the number of files affected by the respective patch with a recognizable downward trend. After a major part was incorporated in Linux 3.0, noticeable portions were also merged into the mainline repository with Linux 3.2

[1] Generated based on raw patches from `https://www.kernel.org/pub/linux/kernel/projects/rt/`

[2] Last accessed on 02.03.2017.

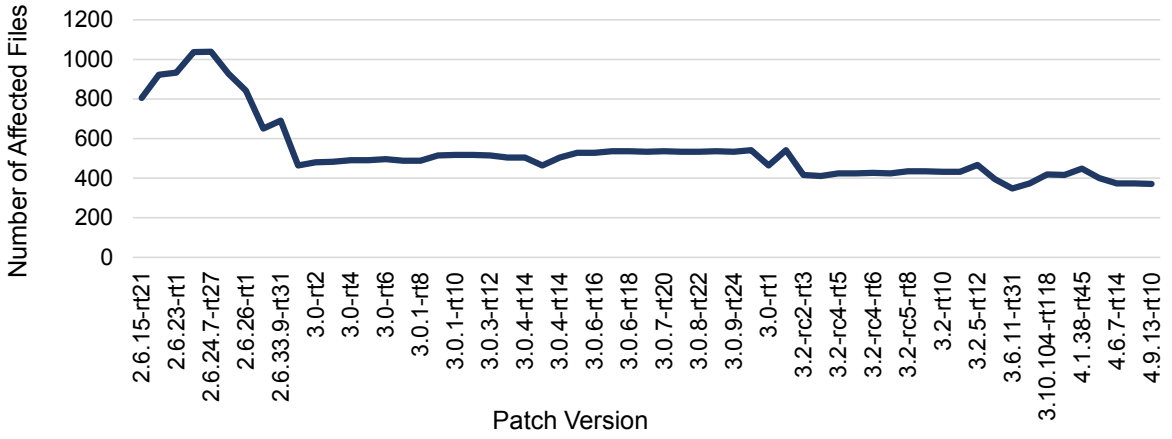


Figure 4.1: Evolution of PREEMPT_RT's size between 2006 and 2017.

and Linux 3.6. It can therefore be expected that the overhead for the application of PREEMPT_RT will continue to decrease in future. Thus, the main benefit of this approach is the reduced complexity, which is achieved by lower integration overhead and consistent API for both real-time and non-real-time domains. This makes it an adequate alternative to setups with a real-time hypervisor and additional off-the-shelf kernels.

The remainder of this chapter is structured as followed. A short summary of the related work is presented in Section 4.1. The chosen implementation approach is covered by Section 4.2 with improvement of kernel's preemptibility and Section 4.3 showing the integration of a frequency locking mechanism. The introduced changes are evaluated by measuring the worst-case scheduling latency for different system configurations in Section 4.4. Finally, a concluding discussion is presented in Section 4.5.

4.1 Related Work

Android was originally designed for mobile devices with ARM processors. Despite the *Android x86* project[3], which started in 2009 and aims at porting Android to the x86 platform, the vast majority of produced Android devices are still built based on ARM and ARM64 architectures like the Cortex A72 series. According to Altenberg [5], this architecture is especially interesting in context of real-time Linux. Cortex A series can be seen as a successor of the ARM11 architecture family, which was widely used in the industry because of the beneficial multicore design, ability to upscale the CPU frequency and highly efficient execution of Java code. Altenberg enumerates the available real-time features – which are already part of the vanilla Linux kernel – and presents a detailed evaluation of a generic ARM system patched with PREEMPT_RT. Testing is performed under system load and includes measurements of the context switching time in reaction to external events and accuracy evaluation of the internal timer. Recorded worst-case latencies on the ARM11 platform range between 68 µs and 110 µs. Since the evaluation

[3]Android x86 project website: http://www.android-x86.org/

results show a strictly deterministic system behavior in all test scenarios, the patching approach is concluded sufficient to enable hard real-time support on ARM CPUs [5].

In 2013, the suitability of PREEMPT_RT specifically for the Android kernel was questioned by Zores [135, p. 7]. In addition to the apparent impracticability of the patch application, the author lists severe design flows in the Android platform. The most important ones that are preventing Android from gaining real-time capabilities are Dalvik's automatic memory management and inefficient audio architecture.

Nevertheless, two successful proof-of-concept realizations of the patching approach were published several months earlier. A conference paper [60] presenting preliminary research results in the context of this dissertation and a detailed evaluation by Mauerer et al. [79] pursue similar ideas for kernel modifications. Mauerer et al. consider the scenario of a partial real-time support (see Figure 3.3(b)) and provide an in-depth effort analysis for the application of PREEMPT_RT to the kernel of a commercial off-the-shelf Android tablet. Despite the large number of actual patch files contained in the set of PREEMPT_RT, the actual patching process was accomplished significantly faster than expected. Finally, concluding performance tests of the patched kernel showed acceptable worst-case latencies between 65 µs to 100 µs, which corresponds with the measurement results presented by Altenberg.

Yan et al. suggest off-the-shelf products like RTEMS and RTLinux [131, 132, 133] as replacement for Android's Linux kernel in their research project *RTDroid*. Leveraging the execution model of RTEMS and compiling the real-time process directly into the kernel provides real-time applications the full control of all available hardware resources. This approach is referred to as the *single-app deployment* [133]. But the authors also point out the incompatibility of this setup with original Android. Since Android Runtime is built upon separate system services, all of them have to be integrated into the single-app in form of threads. This approach yields a significantly more complex build process and includes basically the whole Android platform into each real-time application. Additionally, the main statement of Yan et al. regarding the recommended integration of the real-time Linux seems inconsistent. While generally promoting the full kernel replacement in favor of off-the-shelf products, their publications use *RTLinux* as an equivalent to the PREEMPT_RT patched Linux, instead of the equally named standalone solution for virtualization of the interrupt layer (compare Section 2.3.3). At the same time, the authors presumably adopt the position of Zores, stating that the patching approach is not suitable for consumer devices based on ARM CPUs [131]:

> "The x86 and the LEON3 environments do not require any more than replacing the non-real-time kernel with either real-time Linux kernel (by applying an RT-Preempt patch, i.e., RTLinux) or the real-time RTEMS kernel. The same strategy, however, does not work for the smartphone environment because Android has introduced extensive changes in the kernel that are not compatible with RTLinux patches."

The evaluation of RTDroid shows predictable and deterministic platform behavior with both approaches, using either PREEMPT_RT or RTEMS. Unfortunately, Yan et al. do

not conduct latency measurements for native Linux processes, as all their tests involve interaction with high-level Android components. Nevertheless, the authors also identify low-level Linux components which may have negative effects on the kernel's real-time capabilities. The first issue is caused by the low-memory killer, which is used in Android for improving the handling of OOM situations (see Section 2.2.2). This feature is disabled in RTDroid, since it seems meaningless in setups with only one active process. Disabling the low-memory killer in standard Android would allow non-real-time applications with excessive memory usage to jeopardize the system stability. On the other hand, active low-memory killer can silently terminate real-time processes if they allocate big memory chunks. A general solution proposed in a preliminary paper published during this dissertation project [60] will be presented later. Another issue arises from dynamic CPU frequency scaling which is used in Linux to create a trade-off between CPU performance and energy saving. However, changes in the current CPU frequency directly affect the execution time of all active processes, possibly introducing additional latencies. While Yan et al. leave this as future work [131], this dissertation handles the aspect of dynamic frequency scaling in Section 4.3.

Finally, Ruiz et al. evaluate the resource reservation using *cpusets* for native real-time processes running on the top of Android' Linux kernel [104]. This is a known approach of partitioned scheduling [6, 17, 30, 69], which creates a static assignment of active processes to specific processor cores. While it is a common strategy to improve the behavior of real-time processes on Linux, Ruiz et al. were the first to publish corresponding test results for the Android platform. Notably, the authors do not incorporate any other methods like `PREEMPT_RT` and perform their measurements with an unmodified kernel. Their results show worst-case latencies for a real-time task isolated on one of the CPU cores to be bounded by about 47 µs. Although this is a notable improvement in comparison to the standard setup, the duration of the test was limited to $10^8 \times 5\,\mu s = 500$ seconds, which is not sufficient for a representative sampling set.

4.2 Improving Kernel's Preemptibility

This section explains the chosen approach of the application of `PREEMPT_RT` to Android's Linux kernel. The following example uses the kernel version `android-jfltexx-3.4.107` for Samsung Galaxy S4 and the patch version `preempt-3.4.107-rt133`. The source code for the standard kernel release, which can be pulled from the official repository, is referred to as *mainline*. Since `PREEMPT_RT` does not contain or depend on binary data, patching can be considered as line-based joining of text files. Figure 4.2 illustrates separate merge events across the respective source trees as explained in the following.

Multiple organizations are typically involved in the development of a Linux kernel for Android. Besides of the official kernel developers and the Android team, hardware vendors like Samsung have to integrate their device-specific extensions and drivers into the source code. For this reason, the final source tree for the shipped device can diverge from the original mainline source. As `PREEMPT_RT` is designed to be applied on the top of the mainline kernel, additional synchronization of the device's kernel towards the

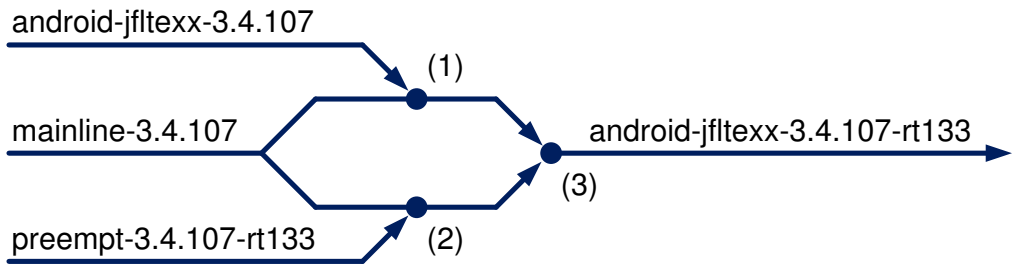


Figure 4.2: Patching with `PREEMPT_RT` (illustration derived from [79, Fig. 2]).

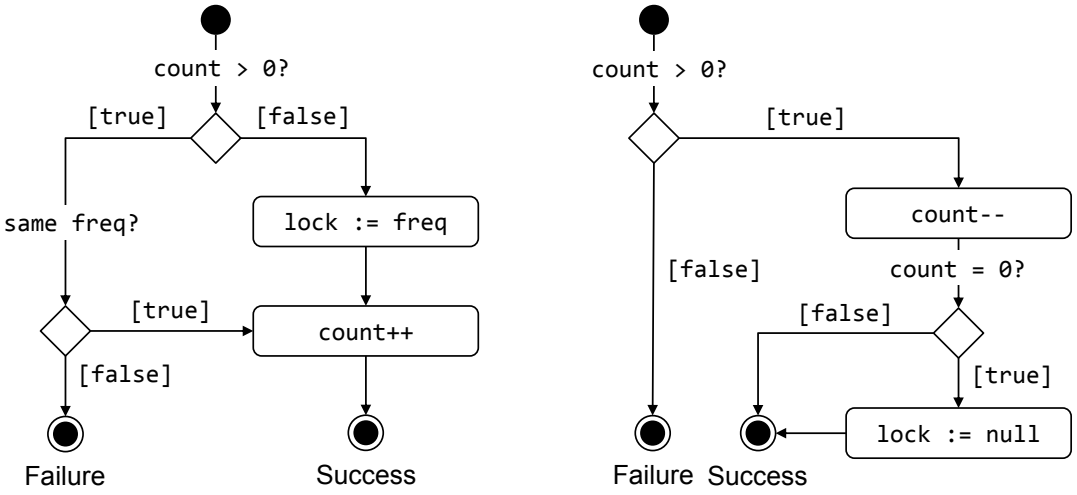
mainline kernel is required to avoid conflicts. This step is denoted (1) in Figure 4.2. After the mainline source is patched in step (2), real-time support for the Android kernel is enabled by merging both branches in step (3). This approach is preferred over the patch application directly on the top of the `android-jfltexx-3.4.107`, as the intermediate patching results can be validated separately in the respective branch. A successful patch application makes the kernel fully preemptible and introduces advanced functionality as it was explained in Section 2.3.3. However, this functionality has to be explicitly activated in the kernel configuration. In contrast to real-time configuration options like `PREEMPT_RT_FULL`, which have to be enabled for providing advanced real-time support, other options (e.g. debugging or function tracing) may increase the system overhead and therefore need to be disabled.

The impact caused by various configuration options on the process behavior is analyzed in more detail in Section 4.4. The following section discusses global effects of dynamic CPU frequency scaling and proposes an approach for locking the frequency in order to reduce scheduling latencies in real-time applications.

4.3 CPU Frequency Lock

As described previously, the Linux kernel utilizes dynamic CPU frequency scaling to adapt the performance to current system needs and to save energy. Since the frequency value directly determines the execution time for CPU instructions, changing it at runtime has global consequences. With a lower CPU frequency, all processes will require more time to complete their execution, possibly causing deadline misses. The dynamic frequency scaling can be completely disabled in the kernel configuration, which causes the system to run at the maximum frequency supported by the CPU. If the corresponding hardware is protected from overheating, this approach may be suitable for stationary devices with constant power supply. Mobile devices however will suffer from the excessive energy consumption, leading to a significant reduction of the battery lifetime. In order to improve the process predictability, but prevent the mobile device from draining its battery, this work implements a frequency locking mechanism based on *governors*[4]. Two

[4]Linux documentation: `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`



(a) Acquiring a frequency lock.

(b) Releasing a frequency lock.

Figure 4.3: Simplified mechanism for dynamic frequency locking.

simplified activity diagrams in Figure 4.3 illustrate the logic of acquiring and releasing the lock.

The CPU frequency can be dynamically locked to a fixed value at runtime. By default, no lock is set and the system may adapt the CPU frequency to the current workload. In order to keep a constant CPU performance, a starting real-time process should try to acquire the lock with the desired frequency value. Every locking request is handled by the system as depicted in Figure 4.3(a). The transition to the right side is taken if no other process is currently holding the lock. In this case, the system governor freezes the system frequency to the provided value. Before the function returns, the internal counter variable is incremented to store the number of currently active locks. If another process has already locked the frequency earlier, the execution path takes the transition to the left side. To avoid conflicting states, the process is only allowed to proceed if the desired frequency matches the currently active value. In this case, only the internal counter has to be increased, as the lock is already active. This avoids the release of the frequency lock if the first locking process terminates earlier than the second does.

Figure 4.3(b) shows the process of releasing a frequency lock, which is done analogously but in the reverse order. The internal counter is decremented on every call until no other process is holding the lock. In this case, the system governor enables the dynamic frequency scaling and the system returns to its normal state.

A thread-safe implementation is achieved by protecting both mechanisms with a real-time mutex included in `PREEMPT_RT`. Additional API opens real-time processes the possibility to make use of the lock in order to keep the CPU frequency at a constant level. It can be used by multiple processes simultaneously to freeze the frequency value until all processes have released their locks again. This eliminates the risk of deadline misses in time-critical applications due to unexpected system transition to a lower CPU performance.

Option Name	Value	Option Name	Value	Option Name	Value
Scheduling options					
PREEMPT	Y	PREEMPT_RTB	N	PREEMPT_VOLUNTARY	N
PREEMPT_NONE	N	PREEMPT_LL	N	PREEMPT_RT_FULL	Y
Debugging options					
SCHED_DEBUG	N	DEBUG_PREEMPT	N	DEBUG_IRQ_FLAGS	N
DEBUG_RT_MUTEXES	N	DEBUG_KERNEL	N	...	N
Other options					
HIGH_RES_TIMERS	Y	CPU_IDLE	?	RCU_BOOST	?
TREE_PREEMPT_RCU	?	AUDITSYSCALL	?	STACKTRACE	?

Table 4.1: Kernel configuration options relevant for real-time support.

4.4 Experiments

This section presents the results of a scheduling latency evaluation for native Linux processes. All tests were performed directly on top of the patched Linux kernel, while high-level Android components will be in focus of the analysis in following chapters. A Samsung Galaxy S4 (codename `jfltexx`) smartphone with an Exynos 5410 Octa (4x 1.6 GHz Cortex-A15 & 4x 1.2 GHz Cortex-A7) chipset and 2 GB of RAM was used to record the measurements. This device runs the Linux kernel `android-jfltexx-3.4.107-rt133` as part of the modified Android 5.1.1.

The evaluation is divided in three parts. The first part evaluates the impact of various configuration options on the system behavior. Thus, an optimized configuration can be used in the second part of this section for benchmarking the scheduling latency under different conditions. Finally, the last part evaluates the process behavior when the introduced locking mechanism is used to prevent the dynamic CPU frequency scaling.

4.4.1 Configuration of the Linux Kernel

A number of different sources provide recommendations for the configuration of a kernel patched with `PREEMPT_RT`. In addition to mandatory options (partially presented in Section 2.3.3) introduced by the patch, other options have to be activated or deactivated in a specific pattern to avoid negative side effects on the system performance. However, publicly available publications and tutorials in this field do not provide a complete evaluation, covering only a subset of available configuration options at once. Table 4.1 summarizes relevant options from online sources[5,6,7] and scientific work [9, 31, 121] with the corresponding recommended setting.

Based on these sources, it can be assumed that the highest possible preemption level and the least possible debugging overhead enable the best real-time performance. Fur-

[5] A realtime preemption overview: `https://lwn.net/Articles/146861/`
[6] RT-Preempt homepage: `https://rt.wiki.kernel.org/`
[7] OSADL recommendations: `https://www.osadl.org/Realtime-Preempt-Kernel.kernel-rt.0.html`

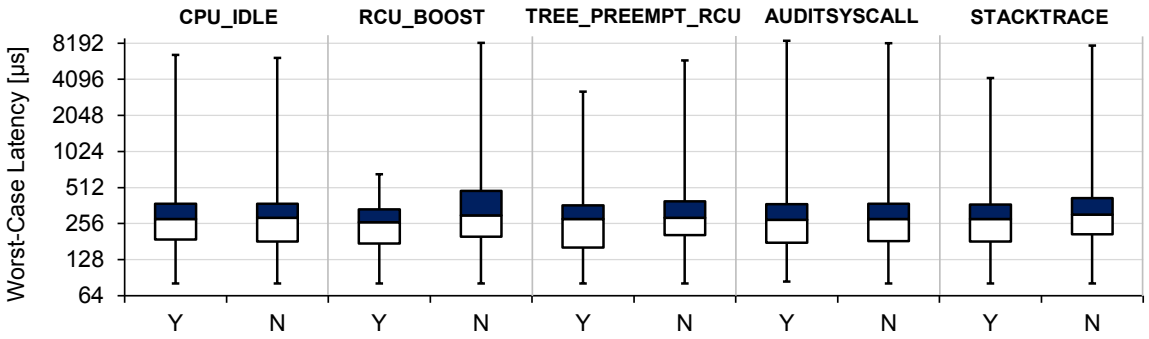


Figure 4.4: Worst-case latency analysis for different kernel configurations.

thermore, the high-resolution timer must be activated for full real-time support, as it is required for precise scheduling. Therefore, options `PREEMPT`, `PREEMPT_RT_FULL` and `HIGH_RES_TIMERS` are set to "`Y`", while all debugging options are set to "`N`".

The remaining options lack a consistent recommendation value and need to be evaluated separately. For this purpose, a dedicated kernel version was compiled for each of the $2^5 = 32$ possible configurations. Actual testing was performed using *cyclictest*[8], which acts as a de facto standard for benchmarking kernels patched with `PREEMPT_RT`. It was ported[9] and compiled for Android in the context of this thesis. The evaluation was done with standard parameters[10] for the duration of 30 minutes per configuration. Although the benchmark prints minimum, maximum and average values for each run, only the maximum value is used in the following analysis, as it represents the worst-case scheduling latency during the execution. For realistic test conditions, the evaluation was performed under additional CPU load. This was achieved using the *hackbench* binary included in the benchmark. With standard parameters[11], it creates 400 thread pairs which continuously pass 100 byte messages via sockets.

This testing process yields 32 measurements, which represent the worst-case latency recorded in the respective kernel configuration. These values can be divided into two distinct sets for each configuration option. Each set contains 16 latency values describing the system behavior when the option is set either to "`Y`" or to "`N`". Box plots with a logarithmic scale are used to illustrate these sets in Figure 4.4.

The option `RCU_BOOST` has clearly the biggest impact on the preemptibility of the kernel and thus on measured scheduling deviations. Activating this options reduced the maximum worst-case latency from 8257 µs to only 661 µs. Similarly, setting the options `TREE_PREEMPT_RCU` and `STACKTRACE` to "`Y`" also has positive effects on the scheduling, reducing the maximum of all measured values by almost 3000 µs in each case. It must be stated that all presented evaluation results for other options contain 8 measurements with `RCU_BOOST` set to "`N`". For this reason, it is presumable that high latencies are caused by this specific setting, hiding the impact of the option that is actually being analyzed.

[8]Original `cyclictest` documentation: `https://rt.wiki.kernel.org/index.php/Cyclictest`
[9]Ported `cyclictest` for Android: `https://github.com/RTAndroid/android_external_cyclictest`
[10]Benchmarking parameters: `cyclictest -n -p 80 -D 30m`
[11]Parameters for stressing the CPU: `hackbench -loops 10000000`

Test	Kernel	Freq. Lock	CPU Load
1	Standard	–	–
2	PREEMPT_RT	–	–
3	PREEMPT_RT	–	+
4	PREEMPT_RT	+	–
5	PREEMPT_RT	+	+

Table 4.2: Test specification for a detailed evaluation of the scheduling latency.

Scheduling Latency [µs]					
Test	1	2	3	4	5
Min	37	27	23	17	16
Q_1	161	76	91	29	32
Q_2	197	79	115	34	36
Q_3	199	79	118	35	36
Max	7407	118	219	58	63
Avg	186	77	108	31	34
σ	48	4	17	6	6

Table 4.3: Latency evaluation results.

However, additional testing of the remaining four options with activated RCU_BOOST has shown a latency distribution similar to Figure 4.4. Enabling TREE_PREEMPT_RCU and STACKTRACE was confirmed to reduce scheduling latencies, while options CPU_IDLE and AUDITSYSCALL made no significant difference regardless of their setting. This optimized configuration will be used for all further tests in the context of this work.

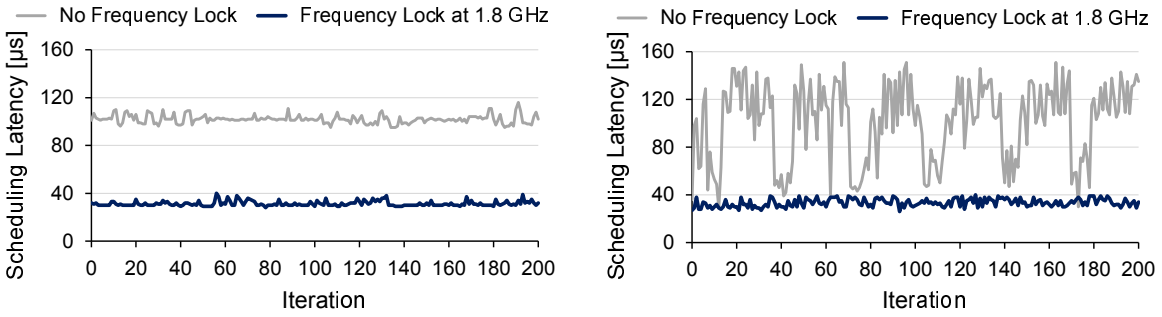
4.4.2 Native Scheduling Latency

The next test evaluates the scheduling latency of native processes in different scenarios. Besides a comparison to the original non-patched kernel, latency values are also recorded during an active frequency lock. To validate the scalability of the resulting approach, some tests are performed under additional CPU load. The hackbench binary was used with default configuration as described in the previous section. A summary of all tests including the corresponding parameters is given in Table 4.2.

A real-time process with priority 80 is scheduled every $T = 500$ µs for the total duration time of one hour. For a more detailed evaluation of the system behavior, the differences between the scheduled and the actual execution of this process (latencies) are recorded on every iteration. Since cyclictest provides only summarized information, this test relies on a new implementation derived from the original benchmark. Table 4.3 presents the most relevant statistical values calculated from the recorded measurements.

Test 1 shows the unsuitability of the original kernel for reliable, deterministic scheduling. Executing a process with real-time priority does not prevent the system from causing significant latencies. The recorded maximum value of 7407 µs means that the process has missed its activation deadline 14 times in a row. Since this test is executed without additional CPU load, it can be assumed that the real-time process will experience even higher deviations from its schedule if other applications will run concurrently.

The system behavior is greatly improved on the PREEMPT_RT patched kernel as shown in Test 2. In this case, all measured values up to the 3rd quartile remain below 80 µs and the standard deviation is reduced from 48 µs in the original kernel to only 4 µs. This is a clear indication for a more deterministic scheduling resulting from the preemptible kernel. Nevertheless, additional CPU load in Text 3 has a noticeable negative effect on



(a) Measurements for $T = 5$ ms.

(b) Measurements for $T = 500$ ms.

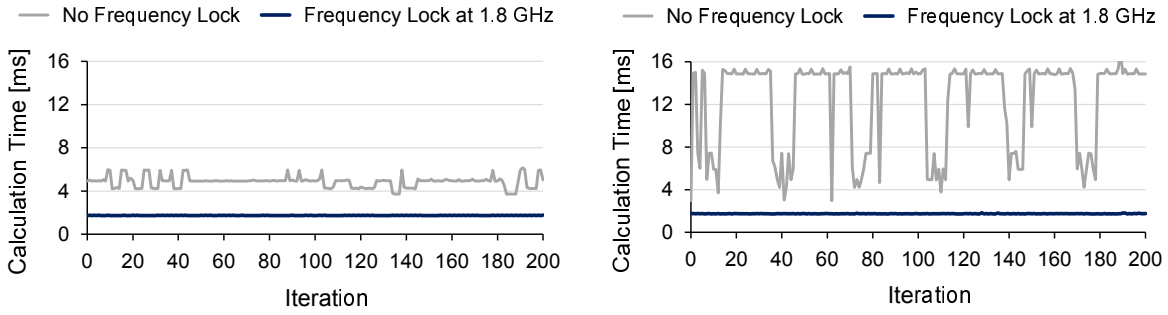
Figure 4.5: Evaluation of the scheduling latency for extended period times.

the latency distribution. The high priority of the real-time process does not completely protect it from increased scheduling delays. All measured values in Test 3 are increased by about 30–50% in comparison to Test 2, while the worst-case latency rises to 219 µs when the CPU is under stress. This may be sufficient for soft real-time systems designed to react in the range of milliseconds or higher, but not for applications with period times below 300 µs.

The remaining tests reuse the configuration of Tests 2 and 3 while additionally locking the CPU frequency to the highest supported value of 1.8 GHz by making use of the new API. Results recorded during Tests 4 and 5 illustrate that the introduced mechanism for preserving a constant CPU speed reduces the scheduling latency regardless of the current load. Disabling automatic frequency scaling leads to a constant execution time of CPU instructions and a more deterministic timing on the system level. This provides a higher system performance allowing the real-time process to be scheduled more precisely, with the maximum deviation from its planned activation time of 63 µs. The other measured values do not exceed 40 µs even when challenging the CPU with additional tasks.

4.4.3 Dynamic CPU Frequency Scaling

Previous test has shown the increased predictability of process scheduling when the CPU is locked to the highest available frequency. This was made possible by preventing the frequency value from being automatically adjusted depending on the system load. Without the lock, all running processes and system activities are affected by dynamic changes of the CPU speed. Measuring these changes is only possible with sufficiently long execution periods without additional CPU load, since the dynamic scaling module requires time to release the frequency between two subsequent iterations. For this reason, the following tests are performed by scheduling the real-time process with different period times T between 1 millisecond and 2 seconds. The benchmark application is extended to calculate a product of two 100×100 matrices in each iteration. The time required for this operation is continuously recorded in addition to the scheduling latency. Evaluation results for 200 iterations with $T = 5$ ms and $T = 500$ ms are illustrated in Figure 4.5.



(a) Calculation delay for $T = 5$ ms.

(b) Calculation delay for $T = 500$ ms.

Figure 4.6: Evaluation of the calculation time for extended period times.

Without a frequency lock, the observed scheduling latency for $T = 5$ ms in Figure 4.5(a) is slightly worse than the one recorded during Test 2 in the previous section (compare Table 4.3 for $T = 500$ µs). Although worst-case values are similar in both tests, extending the period time from 500 µs to 5 ms increases the average latency by about 20 µs. While this value stays nearly constant during the test, effects of the dynamic frequency scaling become apparent in Figure 4.5(b). With even further extension of the period time to $T = 500$ ms, the overall CPU load level falls under a certain threshold and energy saving mode is activated. Without an explicit lock, the Linux kernel automatically reduces the CPU speed by choosing a lower frequency value. As a result, the whole system is slowed down leading to significantly higher latencies up to 162 µs. Figure 4.5(b) shows a recurrent pattern where the CPU frequency is repeatedly increased for a short period of time every 25 iterations of the benchmark process. This pattern might be caused by another active process which generates a substantial system load for an approximate duration of 10×500 ms $= 5$ seconds with a period of 25×500 ms $= 12.5$ seconds. In contrast to dynamic frequency scaling, enforcing a CPU frequency lock at the maximum supported level ensures a bounded latency regardless of the period time. Both diagrams show worst-case measurements in the range of 25–38 ms, which corresponds with the observations from Tests 4 and 5 in Section 4.4.2.

A similar behavior can be recognized in the values for the calculation phase as depicted in Figure 4.6. Both diagrams plot the time required for the matrix multiplication in the respective iteration of the same two runs presented in Figure 4.5. For a disabled frequency lock, process period time of $T = 5$ ms results in an average calculation time of about 4.9 ms with relatively low scattering (see Figure 4.6(a)). With the period time of $T = 500$ ms the same multiplication takes up to 15.3 ms due to the energy saving mode activated by the dynamic frequency scaling (see Figure 4.6(b)). In both cases, the proposed frequency locking mechanism is able to guarantee a deterministic time frame of 1.7 ms in which the result is calculated.

Figure 4.6(b) also confirms the global effect of the dynamic frequency adjustments. The low value recorded in the 63rd iteration of this test indicates that the processing was made with a higher CPU speed. This allowed the result to be calculated in only 2.9 ms, instead of 14.8 ms required in both the previous and the subsequent iteration.

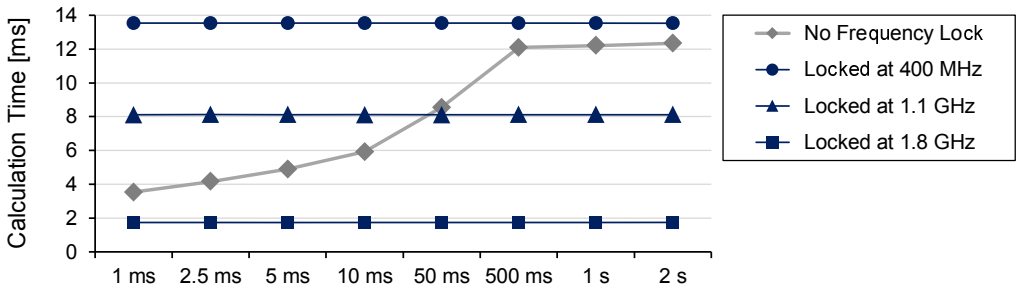


Figure 4.7: Impact of the frequency on the calculation time.

Notably, the outlier in the 63rd iteration can also be identified in Figure 4.5(b), where the real-time process was scheduled with significantly lower latency in comparison to neighboring iterations. Thus, the process was able to benefit from another application or system activity that temporarily caused a transition to a higher CPU frequency.

In order to quantify the processing speedup gained from the lock at different frequency levels, the last test is repeated for $T \in \{1\text{ ms}, 2.5\text{ ms}, 5\text{ ms}, 10\text{ ms}, 50\text{ ms}, 500\text{ ms}, 1\text{ s}, 2\text{ s}\}$. Additionally, measurements for each T are recorded for different settings of the frequency lock: no locking, acquired for 400 MHz (minimum value), acquired for 1.1 GHz and acquired for 1.8 GHz (maximum value). Figure 4.7 shows the average calculation time computed based on 200 iterations for each T and each setting. Data points recorded with the same lock level have been connected for better comparability.

As it was mentioned earlier, short period times keep the CPU busy and prevent the system from reducing the frequency. Correspondingly, rising values for T allow the dynamic frequency scaling mechanism to save the battery by lowering the current CPU speed. This effect is visualized by the test case with no frequency lock. It illustrates that the average calculation time gradually increases from 3.5 ms to 12.3 ms when T is extended up to 2 seconds. These values confirm the previous measurements for $T = 5$ ms (see Figure 4.6(a)) and $T = 500$ ms (see Figure 4.6(b)). However, the performance of the dynamic frequency scaling never reaches either of the limits provided by locking the minimum or the maximum frequency. It was already shown that acquiring the introduced frequency lock for 1.8 GHz leads to a shorter computation time. Notably, acquiring a lock for 400 MHz provides even lower performance than the one observed during the automatic adjustment. This may be caused either by internal limits of the module for dynamic frequency scaling or by other applications performing their work in the background. Thus, activating the frequency lock is confirmed to guarantee a deterministic timing regardless of the chosen CPU frequency value.

4.5 Summary and Discussion

This chapter presented an approach for the application of the PREEMPT_RT patch to Android's Linux kernel. A synchronization of the device-specific kernel before merging it with the patched mainline source tree simplifies the patching process while preserving vendor extensions and included Androidisms. The required overhead is expected to decrease in future, since PREEMPT_RT is being continuously merged into the mainline kernel. Furthermore, the system configuration was optimized for real-time processing based on detailed evaluation of the relevant kernel options. Evaluation results of the scheduling latency for native Linux processes have shown bounded response times and worst-case latencies of 63 µs even under additional CPU load. This was possible by using the introduced frequency locking mechanism based on Linux governors.

This approach protects real-time applications from being undesirably affected by transitions to different frequency values caused by temporary CPU load. The lock can be allocated dynamically at execution time, providing a better trade-off between predictability and energy saving on mobile devices. Additionally, the overhead introduced by the implemented methods for acquiring and releasing of the frequency lock is bounded and predictable. However, the current implementation is not aware of the process that had requested a lock. An acquired frequency lock is preserved permanently if the responsible process terminates without releasing it. Keeping track of the mapping between active locks and the corresponding processes would also introduce additional security and prevent malicious processes from releasing locks they did not acquire.

Overall evaluation results are comparable to those presented in the literature for ARM CPUs (see Section 4.1). It may therefore be concluded that the proposed approach leverages the entire potential of PREEMPT_RT while preserving Android-specific extensions. Nevertheless, higher architectural levels must also be modified in order to make Android applications benefit from the obtained determinism and predictability of the kernel.

5 Non-blocking Memory Management

As explained in Section 2.2.2, each Android application is encapsulated by a dedicated instance of the virtual machine running in a separate Linux process. This way, real-time Android applications can immediately benefit from the fine-grained CPU control and reduced scheduling latencies, as presented in the last chapter. However, the virtual machine has a major influence on the internal process behavior, as it continuously monitors and manages the virtual environment. For example, if the amount of free memory falls below a predefined threshold, the VM automatically invokes the collection of unused memory objects. Regardless of the Android version, both Dalvik VM and Android Runtime provide advanced algorithms for object disposal. Invocations of the GC do not affect other processes, since the garbage is collected only in the memory region of a single process. However, all activities of the target application are suspended by the VM in order to avoid data inconsistencies. The GC is activated depending not only on the process' own memory usage, but also on the global system state and internal thresholds. As a result, all active processes experience unpredictable delays at runtime due to enforced interruptions of all threads inside the corresponding VM instance.

While automatic memory management mechanisms provide a higher flexibility for application developers, they typically cause high runtime overhead. Suspending all active threads is the most obvious way of preserving the consistency of the memory state during the identification of unused objects. Hence, creating a non-blocking GG is of major importance especially in time-critical domains [33]. As pointed out by Schoeberl, automatic memory management in real-time applications is possible, if the garbage collection is performed incrementally with a known maximal blocking time [107, 108]. Based on these ideas, this chapter proposes a concept of a real-time capable garbage collection and its integration into the Dalvik VM. Its non-blocking operation mode minimizes interferences between the garbage collector thread and the thread that executes the application logic. Furthermore, it avoids OOM situations by keeping a sufficient amount of memory available at any time, improving the overall system predictability.

The remainder of this chapter is structured as follows. Related work and known algorithms for automatic memory management are summarized in Section 5.1. Section 5.2 illustrates the conceptual design of the real-time capable collector and its integration in Android. An analysis of how memory consumption affects the behavior of long-term background applications is presented in Section 5.3. Section 5.4 provides a detailed evaluation of the introduced changes. Finally, Section 5.5 concludes this chapter.

5.1 Related Work

The first approach to extend Android's memory management to improve the user experience was presented by He et al. [48]. In addition to a detailed analysis of the heap structure used by the DVM, the authors propose to change the default heap layout. Instead of allocating objects of different types next to each other, the heap is divided into two distinct regions for static and dynamic content. The *class* region stores preloaded class objects, which do not change during their lifetime on the heap. The *user* region offers memory for dynamic allocations of program data at runtime. The authors propose a regional garbage collection, which is primarily focused on the user region. Still, their approach additionally includes a mechanism to scan both regions, which performs similarly to the native stop-the-world algorithm of the DVM and can also free objects in the class region. Presented evaluation results indicate a better performance of this approach in comparison to the original implementation. However, since this algorithm cannot guarantee an undisturbed program execution over a long period of time, it is not suitable for deployment in time-critical domains.

A solution designed specifically to meet real-time requirements was presented as part of the RTDroid project [133]. Instead of modifying the original GC implementation, RTDroid integrates the Fiji VM – an additional JVM with its own mechanisms for memory management. It relies on a parallel GC based on Immix [13], which allocates and reclaims memory in contiguous regions. This allows real-time applications executed inside the Fiji VM to benefit from automatic memory management without negative effects on the application's runtime behavior. However, this protection only applies to threads controlled by the Fiji VM. Android system services and third-party applications will still depend on the original GC implementation, as they are executed by their own DVM instances. This approach can be justified only in scenarios where the real-time process does not use any of Android's internal API, since accessing functionality outside of the process boundaries may still result in unpredictable delays.

In addition to the presented approaches based on regional GC, other state-of-the-art methods for automatic memory management include tracing GC and reference-counting GC. The following sections provide a short summary of the related work on these topics and present further techniques for using automatic GC in the context of real-time computing. Application threads that can allocate and mutate memory will be referred to as *mutators*, while a *collector* is the thread performing the actual garbage collection.

5.1.1 Tracing Garbage Collection

The original GC in Android's DVM is based on a tracing collector. Similar to the vast majority of all tracing-based GC implementation, DVM uses a *mark-and-sweep* algorithm [80] for the detection of unused objects in the memory. This algorithm is divided in two phases. In the *mark* phase, the collector performs a traversing through the graph of all allocated objects and marks all reachable ones. The next phase is called *sweep* phase, since the collector scans all objects on the heap and frees the memory behind all unmarked objects. This algorithm guarantees to correctly identify and reclaim

garbage objects even if they contain circular references. However, its main disadvantage is that all mutator threads have to be suspended during the graph traversion in order to avoid new allocations before the mark phase is finished.

An advanced version of mark-and-sweep called tri-color-marking [25] introduces three different sets to additionally keep track of whether an object can potentially be reclaimed. This modification allows the collector to be executed incrementally, processing only a small number of objects each time. This way, the system does not have to suspend the mutator threads for the duration of the whole collection, splitting the execution into multiple short steps. Other adaptations of the mark-and-sweep algorithms additionally improve the performance of the original method by enabling generational [58] or compacting [93] garbage collection.

5.1.2 Reference-Counting Garbage Collection

Another method for detecting whether an object is still used by the running program is reference counting. It has been originally introduced by Collins [21] and relies on the number of active references to each allocated object. Each object stores how many other objects have a reference to it, allowing the system to reclaim the memory as soon as this number reaches zero. A *write barrier* – which is executed on every object assignment – is used to detect changes and allows the original algorithm to free an object immediately after its counter was set to zero. However, this operation may have negative impact on the process behavior, for example if the freed object references a big number of other objects, which now also have to be reclaimed. This effect is called *cascading deallocation* [124]. It causes non-deterministic delays to the program execution, if a deallocation of one object causes a deallocation of another object by recursively decreasing their reference counts. This issue can be addressed by creating a set of objects waiting to be freed. Instead of reclaiming the memory immediately, the write barrier only inserts the detected garbage object into the set, leading to a bounded execution time. An additional collecting thread can concurrently remove objects from the set and free them, without interfering with the program logic.

A performance gain can be achieved by limiting the memory scope of the write barrier as proposed by the *Deutsch-Bobrow method* [24]. It excludes objects allocated on the stack from the consideration, since they are assumed to be mutated frequently. This leads to a significant reduction of calls to the write barrier, as it is only executed for objects stored on the heap. As a disadvantage, additional checks are required before an object can be freed, since it can still be referenced from other objects on the stack.

The original algorithm for garbage collection can be further improved by implementing *deferred* [24] and *coalescing* [70] reference counting. Both approaches present conditions under which effects of intermediate mutations may be either calculated at a later stage or even completely ignored, reducing the execution overhead of the write barrier.

Nevertheless, reference-counting GC has a major disadvantage, when compared to algorithms for tracing GC: If two or more objects have a circular reference to each other, their respective reference counts will never reach zero, even if they are not referenced from any other objects on the heap. Such structures – commonly referred to as *cyclic*

garbage – cannot be identified by a reference counting method. To address this issue, Bacon and Rajan have presented a cycle detection mechanism [12]. It can be deployed in addition to the write barrier in order to detect and reclaim cyclic garbage.

5.1.3 Real-Time Garbage Collection

Deterministic behavior of algorithms for automatic memory management is crucial in real-time applications. If the garbage collection has to be deployed in real-time scenarios with firm deadlines, it must provide more than just a bounded blocking time. As mentioned above, Schoeberl has identified three requirements a real-time capable garbage collection has to fulfill [107, 108]:

1. The collector has to keep up with the maximum allocation rate of mutators.
2. The worst-case blocking time introduced by the collector has to be known a priori.
3. The time for GC operations like tracing and write barrier has to be predictable.

The first characteristic describes the cooperation between the collector and the mutator threads, rather than a requirement for the GC algorithm itself. Nonetheless, this condition has to be satisfied in order to avoid dangerous OOM situations.

Properties (2) and (3) have a major influence on the design of the GC and how it needs to be scheduled. Schoeberl points out that a real-time capable collector must be executed concurrently to the mutator threads and perform incremental GC with bounded overhead in terms of space and time. The concurrent execution can be achieved by using either *periodic* scheduling [10] or *slack-based* scheduling [51]. In the first case, the collector is executed with the highest possible priority at a fixed schedule. It automatically preempts all the mutator threads for a bounded amount of time during the collection phase. This fulfills the requirement (2), since both the invocation time and the blocking time can be calculated a priori. On the contrary, a slack-based collector is executed with a lower priority and it can be preempted by the real-time thread at any time. This also guarantees the predictability of the GC behavior and fulfills the requirement (2). However, in this case the collector thread requires a sufficient amount of the CPU time to keep up with the allocation rate of the mutator. In both cases, the GC has to rely on an incremental approach, since the time slice of a single iteration might not be sufficient for a full memory scan. Regardless of which approach is chosen for the scheduling of the collector thread, all mutator threads are indirectly slowed down by the introduction of the write barrier. Although the time overhead for barrier's execution is bounded (see Section 5.1.2) as required by the requirement (3), the overall system performance is decreased as the write barrier is executed on every object assignment.

Both tracing and reference-counting collectors begin their execution by scanning the context (e.g. registers and stack – referred to as *local root sets*) of all mutator threads. This is a necessary step, since the optimized write barrier is only executed for objects on the heap, which can still be referenced from the root set. The most straightforward method to obtain a consistent snapshot of the root set is to suspend the target thread.

In real-time applications, these suspensions can be avoided if the snapshot is created by the real-time thread itself. This requires additional synchronization between the collector and the mutator thread, like in the *delegated local root set scanning* proposed by Schoeberl and Puffitsch [109]. It passes the scanning of the respective context to the mutators themselves by setting a predefined flag. A mutator thread periodically checks this flag at its implementation-specific safe points and performs the scan of its own registers and stacks. Thus, no thread suspension has to be enforced and the collector thread can start its execution after all mutators have finished the scanning of their own root sets.

5.2 Real-time Garbage Collector for Android

As presented above, garbage collectors using the mark-and-sweep approach are typically not suitable for applications in real-time environments. They rely on the stop-the-world paradigm, causing non-deterministic suspensions of mutator threads and increasing the risk of deadline misses. To address this issue, this section presents an approach for the design and integration of a real-time capable, reference-counting garbage collector for the Dalvik VM. In order to minimize the integration effort for upcoming platform updates, the new GC is implemented in addition to the original algorithm. This allows the real-time processes to benefit from a more deterministic memory management, while other applications and unrelated system components can be adopted unchanged.

A real-time GC has to fulfill the requirements presented in Section 5.1.3. Thus, the collector implemented in the context of this thesis is designed to avoid interferences with mutator threads running inside the same DVM instance. This is achieved with a concurrent and incremental GC incorporating the following techniques:

- Reference-counting GC [21]: The reference count of each object is stored in the extended meta header of the object itself.
- Deutsch-Bobrow method [24]: A write barrier updates the reference count of each object on the heap by handling assignment operations.
- Delegated local root set scanning [109]: Additional synchronization mechanism delegates the scanning of the root sets to the mutator threads.
- Non-cascading deallocation [124]: Garbage objects are not freed immediately, but rather stored in a *zero count table* for later processing by a dedicated thread.
- Slack-based scheduling [51]: A concurrent collector thread reclaims objects in the zero count table based on snapshots of local root sets.
- Cycle detection [12]: The collector thread is able to detect cyclic garbage.

Following sections present these extensions in detail, beginning with the extended metadata and the integration of the write barrier. Finally, the actual garbage collector is implemented by incorporating the remaining techniques.

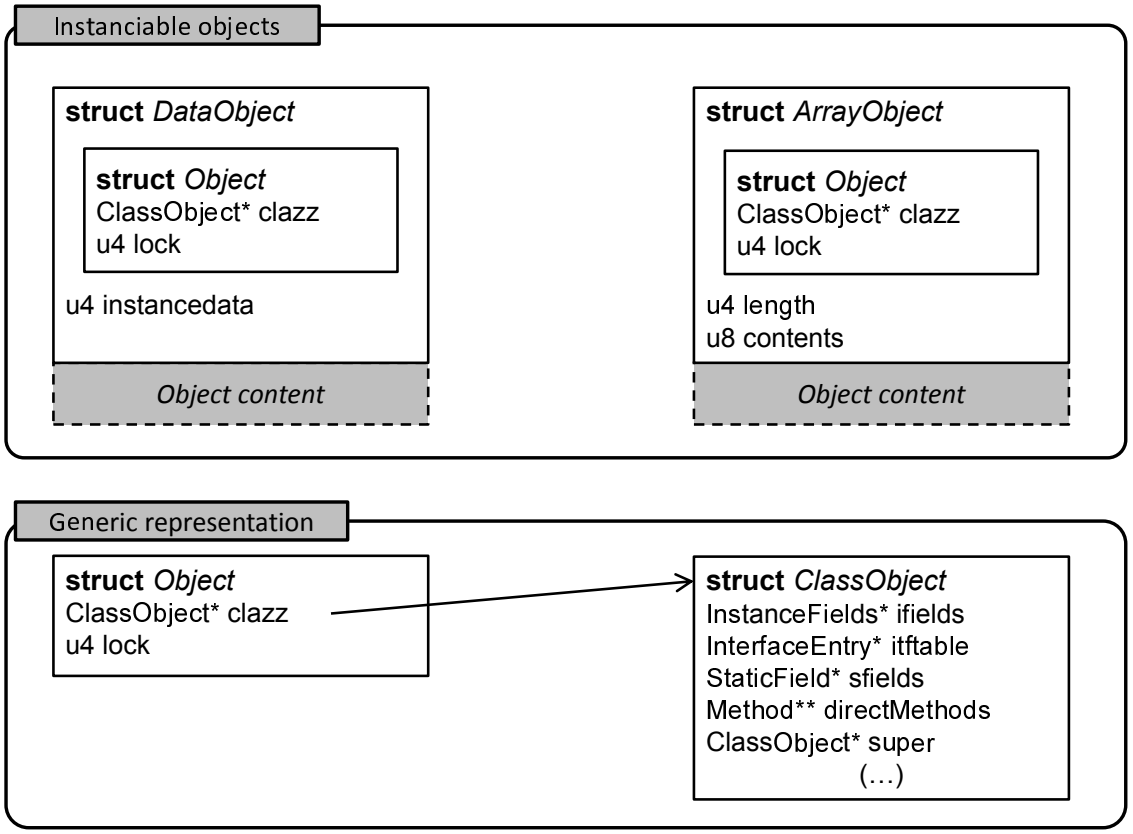


Figure 5.1: Schematic representation of objects within the DVM [38].

5.2.1 Extended Metadata Management

The Dalvik VM provides the function `dvmMalloc` as part of the `dlmalloc` module to access memory on the heap. This function is invoked from `dvmAllocObject` to allocate a memory block of an appropriate size which is used for an object of the desired class. Java objects are internally represented by the DVM as an instance of either a `DataObject` or an `ArrayObject`. As illustrated in Figure 5.1, besides of the actual runtime content, both types also store a generic structure `Object` with meta information. This generic data provides a reference to `ClassObject`, which represents the class this object belongs to, including its fields, methods, interfaces and other relevant information. It is particularly important, since the properties described by this structure are the same for all objects instantiated from this class. This enables a mapping between the specified instance fields of a class and the actual runtime data of an object, which is stored separately inside the content area. While the content of a `DataObject` can have as complex layout as the class it is instantiated from, the content area of an `ArrayObject` can only store primitive data types or references to other objects. Although there might be a big number of different objects of the same given class, the VM optimizes its memory usage by allocating only one instance of the corresponding `ClassObject` structure and saving the same reference to all created objects.

Reference-counting GC implemented in the context of this thesis requires additional information to be stored in all object instances managed by the DVM. Since the native implementation does not provide corresponding data fields, the definition of the `Object` structure has been extended accordingly. Additionally, a zero count table was included into the global runtime context of the DVM stored in the `DvmGlobals` structure[1]. This table is used to keep track of potential garbage objects, which either have just reached the reference count of zero or have been newly created in the `dvmAllocObject` function. Its definition is based on the `ReferenceTable`, which is already part of Android's source code[2] and can contain up to $2^{16} = 65,536$ object references.

5.2.2 Integration of the Write Barrier

As Android 2.2 does not require a write barrier for its original memory management, it was additionally integrated into the DVM. The barrier updates the corresponding reference count each time an object reference is modified. It is implemented to receive two arguments. The first argument is the new object, whose reference is supposed to be stored in the respective field. If valid (the value is not `null`), its reference count is increased since it is about to receive a new reference. The second argument to the write barrier is the old field value. After a validity check (`null` or uninitialized memory), the reference count of the old object is decreased since the corresponding field will not point to this object anymore. Both counter operations are executed using the atomic functions `android_atomic_inc` and `android_atomic_dec` provided by the Android platform. Such thread-safe implementation protects the system from race conditions in case the object is updated from multiple threads simultaneously.

If an object's reference count reaches zero during the execution of the write barrier, this object can be treated as a candidate for garbage collection. It is inserted in the zero count table, if not already stored there, and an additional flag in the object's metadata is set. This flag is used to achieve constant overhead for checking the membership of a specific object in the zero count table. To avoid searching overhead, the corresponding flag is set when the object is inserted into the table and cleared when it is removed. For the same reason, an object is not removed from the zero count table, if its reference count increases again. Instead, the collector thread validates that the reference count of each object equals zero before reclaiming it, or removes it from the zero count table otherwise. This approach allows the introduced write barrier to be executed in $\mathcal{O}(1)$ time and to adds only a constant overhead to the runtime behavior of the DVM.

Besides of internal functions for object manipulation, the DVM also defines interpreter opcodes[3] for updating the object's content region. Six opcodes presented in Table 5.1 have been identified to modify references stored in arrays and instance fields. All of them were updated to execute the introduced write barrier each time the interpreter invokes the corresponding command.

[1] Definition of `dvmGlobals` from Android 2.2 in `dalvik/vm/Globals.h`
[2] Definition of `ReferenceTable` from Android 2.2 in `dalvik/vm/ReferenceTable.h`
[3] Opcodes for the Dalvik VM are defined in Android 2.2 in `dalvik/vm/mterp/c/`

Opcode Name	
OP_APUT_OBJECT	OP_IPUT_OBJECT
OP_FILLED_NEW_ARRAY	OP_IPUT_OBJECT_QUICK
OP_FILLED_NEW_ARRAY_RANGE	OP_SPUT_OBJECT

Table 5.1: Opcodes for updating object references on the heap.

Listing 5.1: Determining the duration of the next sleep phase for the collector.

```
1:  function GETSLEEPDURATION()
2:      t_sleep ← 2 sec                          ▷ Init with maximum sleep duration
3:      for all res ∈ {mem, zct} do
4:          r_diff ← r_cur − r̂_cur                ▷ Allocated in the last iteration
5:          r_rem ← r_lim − r_cur                 ▷ Remaining amount of this resource
6:          t_lim ← r_rem × t̂_sleep / r_diff      ▷ Time until the threshold is reached
7:          t_con ← t_lim / 2                     ▷ Conservative time for this resource
8:          t_sleep ← min(t_con, t_sleep)         ▷ Global sleep time for all resources
9:      end for
10:     return t_sleep
11: end function
```

5.2.3 Implementation of the Garbage Collector

The new garbage collection is performed by an additional concurrent thread. Its priority is automatically adjusted to be lower than the minimum priority of all real-time mutators in the same DVM instance. As explained in Section 5.1.3, this slack-based scheduling approach ensures that the collector thread can be preempted by mutators to avoid undesired delays. The GC itself is not triggered as long as the DVM instance has sufficient resources for an undisturbed operation. More precisely, the collector thread monitors the amount of available memory r_{cur}^{mem} and the filling degree of the zero count table r_{cur}^{zct}. If both values stay above the respective predefined threshold r_{lim}^{res} with $res \in \{mem, zct\}$, there is no need to invoke the garbage collection and the thread is put asleep again. The duration of the sleep phase t_{sleep} is estimated using a heuristic. Listing 5.1 presents this calculation based on the values $\hat{t}_{sleep}$ and $\hat{r}_{cur}^{res}$ from the previous iteration.

The maximum sleep duration is initialized with 2 seconds. This value was chosen as an upper bound to prevent the collector from missing the memory shortage, if the allocation rate of the mutators suddenly increases. The current allocation rate in line 5 is calculated based on changes during the last iteration. This rate is used for the approximation of t_{lim}, which denotes the time until the threshold of the respective resource will be exceeded. However, the value of r_{diff} is subject of application-specific fluctuations and it may change at any time depending on the mutator's state. For this reason, the resource-specific time is additionally divided by two in order to provide a more conservative estimation t_{con}. This approach provides a higher reliability, since the allocation rate of the mutators can be re-calculated earlier in order to react to recent changes.

As soon as one of the values r_{cur}^{res} exceeds the corresponding threshold, the garbage collection is started. Listing 5.2 summarizes the initialization phase (Lines 2-7) and the sweeping phase (Lines 8-25), which are explained in the following.

Listing 5.2: Reclaiming the garbage with a reference-counting collector.

```
 1: procedure performSweeping()
 2:     atomic
 3:         zct_local ← copy zct_global                ▷ Minimize blocking times
 4:         cleanup zct_global
 5:     end atomic
 6:     invoke root sets scan                          ▷ Blocking delegation to mutators
 7:     invoke cycle detection                         ▷ Identify cyclic garbage
 8:     for all obj ∈ zct_local do
 9:         remove obj from zct_local
10:         if obj.refcount ≠ 0 then
11:             continue                               ▷ Drop objects with references
12:         end if
13:         if obj.marked = true then
14:             add obj to zct_global
15:             continue                               ▷ Reachable from the root set
16:         end if
17:         for all child ∈ obj.refs do
18:             child.refcount ← child.refcount − 1    ▷ Its parent will be reclaimed
19:             if child.refcount = 0 then
20:                 add child to zct_local             ▷ Process cascading deallocations
21:             end if
22:         end for
23:         free obj                                   ▷ Reclaim the garbage object
24:     end for
25:     cleanup zct_local
26: end procedure
```

Initialization Phase

Access to the zero count table is protected by a mutex to avoid race conditions. Performing the full garbage collection by iterating through the global table would block the execution of all mutator threads in the same DVM instance. This is avoided by atomically creating a local copy of the zero count table and cleaning up the global one.

The next step invokes the scan of the root sets. This functionality is already available in Android's source tree[4] as a part of the original mark-and-sweep collector. In contrast to mutators with standard priority – which can be suspended during the scan – the

[4]See method `dvmHeapMarkRootSet` in dalvik/vm/alloc/MarkSweep.c

schedule of real-time threads should not be affected. For this reason, the concept of delegated root set scanning (see Section 5.1.3) is used to notify real-time threads about the pending garbage collection. By periodically checking the corresponding flag, a real-time thread can determine at which point in time the scan of the local root set should be performed. The execution of the collector thread is blocked until all threads of the affected DVM instance have finished their scans. After this step, all referenced objects on the heap have been marked without blocking the real-time threads.

Since the original method for reference counting cannot reclaim cyclic garbage, the cycle detection mechanism proposed by Bacon and Rajan [12] is executed prior to the sweeping phase. The write barrier overapproximates the set of candidates for cyclic garbage. All objects whose reference count was decreased, but which still have at least one reference pointing to them, are examined by the cycle detection algorithm. In case they are confirmed to form a cycle, the reference count of the cycle root is decreased, allowing the garbage collector to reclaim its memory in the same iteration.

Sweeping Phase

The initialization phase prepares the local zero count table and ensures that all objects referenced from local root sets are marked accordingly. Before reclaiming objects contained in the local zero table, the collector thread validates they are referenced neither from the heap nor from local root sets. The first check (Lines 10-12) is necessary since a mutator thread could have stored a reference to this object in an instance field on the heap after it was inserted into the global zero count table. Such objects can be dropped, as they will be automatically re-inserted into the zero count table as soon as their reference count reaches zero again. The second check (Lines 13-16) filters all objects that are reachable from local root sets. They are still in use and cannot be reclaimed yet, but they cannot be dropped from the zero count table either, since they have no references on the heap. In order to keep track of such objects, they are re-inserted into the global zero count table and checked again in the next GC iteration.

Remaining objects are identified as unused (Lines 17-23). Before the memory behind a garbage object is reclaimed, the reference counts of all child objects are decremented. If a child object reaches a reference count of zero, it is immediately inserted into the local zero count table to be reclaimed during the same iteration. Finally, the garbage object is deleted using the API already provided by the DVM[5]. Then, the local zero count table is destroyed and the collector thread enters the next sleep phase.

5.3 Memory Adjustments for Real-time Processes

As described earlier, only applications executed in foreground processes can be considered protected from the enforced termination by the system (see Section 2.2.2). However, this does not apply to the vast majority of all activities and services present in the sys-

[5]See method `dvmHeapSourceFree` in `dalvik/vm/alloc/HeapSource.c`

tem at a certain point in time. Hidden activities and background services are destroyed as soon as foreground processes require more memory[6]:

> "Though these processes are not directly visible to the user, [...] the system will always keep such processes running unless there is not enough memory to retain all foreground and visible process. Services that have been running for a long time (such as 30 minutes or more) may be demoted in importance [...]. This helps avoid situations where very long running services with memory leaks or other problems consume so much RAM that they prevent the system from making effective use of cached processes."

In case a real-time application with high memory demands has to persistently run in the background for a long time, using a real-time capable garbage collection is not sufficient to protect it from unexpected suspension or termination events. For this reason, the low-memory killer has to be extended to respect the scheduling of long-term processes with real-time requirements.

Android calculates out-of-memory adjustment (`OOM_ADJ`) values for each active process in order to reliably distinguish between separate processes of the same importance class. Adjustment values are initialized based on the application's importance class, but can be dynamically modified at runtime depending on the process' behavior. This allows considering relevant changes if a visible activity is temporarily moved in the background or if a hidden service creates a visible notification. Both cases can be automatically handled by increasing or decreasing the respective adjustment values accordingly.

The six different importance classes and corresponding memory thresholds are set by Android during the boot sequence[7] as shown in Listing 5.1. The `minfree` parameter denotes memory thresholds to trigger the low-memory killer as soon as the amount of free memory falls below the given value. These parameters are presented in memory pages of size 4096 bytes. For example, the low-memory killer will start terminating empty processes with `OOM_ADJ` > 15 as soon as there is less than 24 MB RAM available[8]. In case the memory situation gets even worse and the system has less than 8 MB RAM available[9], Android will destroy all processes with `OOM_ADJ` > 1, which affects everything except for the current foreground application and system services.

This analysis illustrates that real-time processes have to be used with appropriate adjustment values in order to avoid being killed by the system in OOM situations. The internal design of the low-memory killer allows this approach to be applied to both activities and background services designed for long-term execution. The following experiments will show that choosing an appropriate adjustment value protects real-time processes from being terminated if other applications allocate high amounts of memory.

[6] Developers documentation for processes and application life cycle in Android: `https://developer.android.com/guide/topics/processes/process-lifecycle.html`

[7] From the source code analysis of Android 2.2 in `system/core/rootdir/init.rc`

[8] Conversion from pages to bytes: 6144 pages $\times$ 4096 bytes/page = 25165824 bytes

[9] Conversion from pages to bytes: 2048 pages $\times$ 4096 bytes/page = 8388608 bytes

```
on boot
# Define the oom_adj values for the classes of processes that can be
# killed by the kernel. These are used in ActivityManagerService.
    setprop ro.FOREGROUND_APP_ADJ 0
    setprop ro.VISIBLE_APP_ADJ 1
    setprop ro.SECONDARY_SERVER_ADJ 2
    setprop ro.BACKUP_APP_ADJ 2
    setprop ro.HOME_APP_ADJ 4
    setprop ro.HIDDEN_APP_MIN_ADJ 7
    setprop ro.CONTENT_PROVIDER_ADJ 14
    setprop ro.EMPTY_APP_ADJ 15

# Note that the driver only supports 6 slots, so we have
# HOME_APP at the same memory level as services.
    write /[...]/lowmemorykiller/parameters/adj 0,1,2,7,14,15
    write /[...]/lowmemorykiller/parameters/minfree
       1536,2048,4096,5120,5632,6144
```

Listing 5.1: Loading memory and adjustment threshold values during Android boot.

5.4 Experiments

This section evaluates the changes to the Dalvik VM integrated over the course of this chapter. The following tests are executed on the HTC Dream smartphone with the Qualcomm Snapdragon (MSM7201A ARM11) 528 MHz CPU and 192 MB of RAM. This device runs Linux kernel `android-msm-2.6.29-rt24` as part of the modified Android 2.2. Although the kernel was patched with `PREEMPT_RT` and compiled with the recommended configuration, kernel optimization and frequency locking presented in Chapter 4 were integrated at a later stage and could not be applied in this evaluation. All tests were performed using a conventional Android application, which was executed in a dedicated Linux process with the real-time priority of 80.

The section begins with the evaluation of OOM adjustment values in context of real-time applications. Remaining tests cover the analysis of the reference-counting garbage collection. The first part presents effects of the automatic memory management on the scheduling latencies of periodic real-time tasks with constant allocation rate. In the second part, the same setup is reused for latency measurements during a long-term test. Finally, the performance impact introduced by the additional overhead of new reference GC is quantified and compared to the original implementation.

5.4.1 OOM Adjustments for Real-time Processes

This test compares the behavior of application components depending on their respective `OOM_ADJ` values. In order to identify the differences, the testing environment must allocate significant amount of memory such that the low-memory killer is triggered. HTC Dream

```
Displayed com.evaluation.memory.OomNormal
Evaluation(1378): Allocating 30 MB of RAM...
Evaluation(1378): Allocated 30 MB of RAM.

Displayed com.evaluation.memory.OomRealtime
Evaluation(1491): Allocating 30 MB of RAM...
    Process com.android.keychain has died.
    Process com.android.voicedialer has died.
    Process com.android.gallery3d has died.
    Process com.android.quicksearchbox has died.
    [...]
    Process com.evaluation.memory:oomNormal has died.
Evaluation(1491): Allocated 30 MB of RAM.
```

Listing 5.2: Termination of the normal activity in the OOM situation.

provides about 100 MB of RAM to the Android platform[10], whereby only about 40 MB are available for allocation in user applications[11].

The test is implemented using two activities, executed in separate processes. When started, both activities allocate 30 integer arrays of size 1 MB[12]. The memory overhead of the array object itself and the size of the corresponding references are not taken into account. In contrast to the activity `OomNormal`, which begins with the allocation immediately after start, the activity `OomRealtime` first changes its OOM value to `-1` using an additional interface in the new memory management module. Since the device does not provide sufficient memory to fulfill the demands of both activities at the same time, the system will react to this OOM situation by terminating the less important of both processes.

In the first test, the `OomNormal` activity is started before the `OomRealtime` activity. As shown in Listing 5.2, `OomNormal` was able to successfully allocate the required amount of memory. Starting `OomRealtime` in the next step leads to the lack of free memory, which triggers the low-memory killer and terminates a number of background applications. Since the `OomNormal` activity is not visible to the user anymore, it is classified as being less important and it is killed by the system, reclaiming 30 MB of allocated memory. Notably, this is the desired system behavior in Android and occurs too if the `OomRealtime` activity would keep the default `OOM_ADJ` of `0`. After this test, the device is rebooted to avoid side effects of keeping the real-time application in the memory.

The second test is performed analogously, but with the reversed starting order of activities. As presented in Listing 5.3, it begins with the real-time activity successfully allocating the required memory. After the normal activity is executed, it starts its own allocation, leading to the termination of different applications in the background. Since `OomNormal` is currently in the foreground, it has the highest `OOM_ADJ` value available to

[10]Taken from: `https://groups.google.com/forum/#!topic/android-platform/rbL6mUYnW3M`
[11]Determined at runtime using `MemoryInfo` instance retrieved from `ActivityManager` system service
[12]Each array contains 262144 primitive elements: 262144 integers ×4 byte/integer = 1048576 bytes

```
Displayed com.evaluation.memory.OomRealtime
Evaluation(1289): Allocating 30 MB of RAM...
Evaluation(1289): Allocated 30 MB of RAM

Displayed com.evaluation.memory.OomNormal
Evaluation(1552): Allocating 30 MB of RAM...
    Process com.android.gallery3d has died.
    Process com.android.voicedialer has died.
    Process com.android.quicksearchbox has died.
    Process com.android.keychain has died.
    Process com.android.location.fused has died.
    Process com.android.mms has died.
    [...]
    Process com.evaluation.memory:oomNormal has died.
```

Listing 5.3: Protection of the real-time activity in the OOM situation.

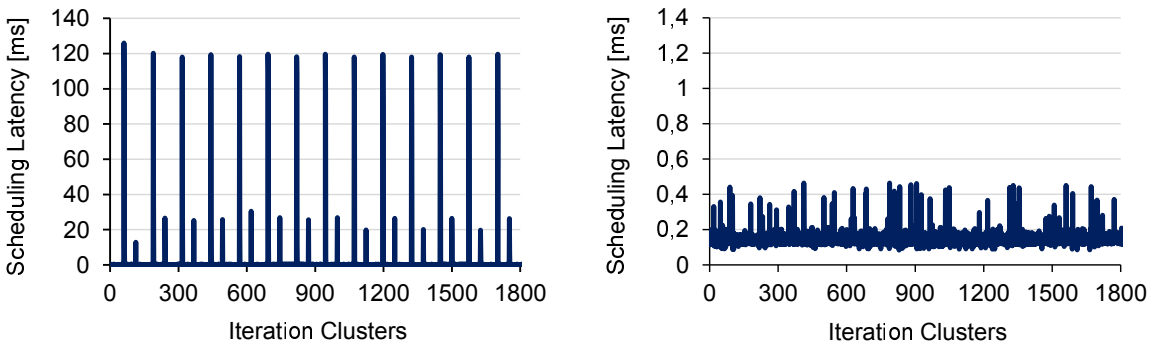
user applications. This causes less important system services like *location* and *mms* to be killed. However, the reclaimed memory still does not suffice for the requested allocation. Both activities are the in direct conflict for the memory, which causes the low-memory killer to take the next step and start destroying visible applications. Although the `OomRealtime` activity is still in the background, it has a higher importance class determined by `OOM_ADJ` set to `-1`. Since the normal activity was assigned the default value of `0`, it cannot justify the termination of `OomRealtime`. For this reason, the allocation of the current array object will fail. However, even if the started activity might be able to continue its execution with a smaller number of objects, the low-memory killer was triggered because the overall amount of free memory has fallen below the lowest threshold of 6 MB. In this case, the foreground activity is also terminated in order to release memory and avoid global system crash.

Both tests have shown that Android kills background processes in order to provide sufficient memory to applications that are classified more important. This behavior can negatively affect long-term real-time applications, if they are not additionally protected and if the foreground application has high memory demands.

5.4.2 Latency Caused by the Garbage Collection

This test is performed using a real-time thread with period time $T = 10$ ms and the total duration of 30 min[13]. In every iteration, the thread allocates 8 acyclic objects (with 16 bytes / object) on the DVM heap. Corresponding references are not saved, such that the objects can be reclaimed by the GC. Latency values recorded during both tests with the native GC and with the reference-counting GC were additionally clustered to provide a better overview. Each value in Figure 5.2 represents the worst-case latency out of the cluster of 100 subsequent measurements during the respective test.

[13]Leading to 30 min $\times$ 60 sec/min $\times$ 100 iterations/sec = 180000 iterations.



(a) Latencies with native collector.

(b) Latencies with reference-counting collector.

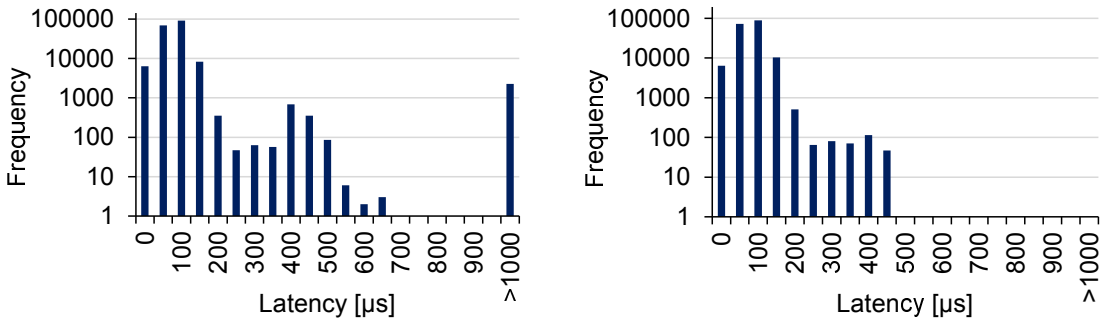
Figure 5.2: Evaluation of the latencies introduced by the garbage collection.

Figure 5.2(a) illustrates the behavior of the native mark-and-sweep collector included in Dalvik VM. Its stop-the-world paradigm causes scheduling latencies up to 126 ms since the mutator thread has to be suspended. A full garbage collection is performed every 12000 iterations, which corresponds to an allocation size of approximately 1.5 MB. This value seems reasonable, considering that less than 40 MB of total memory is shared by all Android applications on the HTC Dream. The original collector is shown to be unsuitable for applications with real-time requirements. During this test, a single invocation of the GC leads to the miss of up to twelve scheduled deadlines.

As shown in Figure 5.2(b), using the proposed reference-counting GC significantly decreases suspension times. In comparison to the original approach, the new collector does not introduce latencies higher than 500 µs, leading to better predictability and higher responsiveness of the system.

For a better visualization, latency values recorded in both tests are clustered into distinct classes and presented in histograms with a logarithmic scale in Figure 5.3. The vast majority of scheduling latencies recorded while using the original collector stay in the range of 50-150 µs (see Figure 5.3(a)). Nevertheless, its mark-and-sweep algorithm has enforced the suspension of the mutator thread for more than 1 ms 2264 times. Figure 5.3(b) confirms the earlier observation that no latencies higher than 500 µs were detected using the reference-counting GC. Furthermore, this approach has also reduced the total number of latencies in the range between 400 µs and 650 µs.

In order to validate the performance of the new collector on different types of objects, additional tests were performed using memory load with acyclic, cyclic and mixed objects. The summarized results of latency measurements are presented in Table 5.2. Although the test with no object load has performed the best in terms of more precise scheduling, the additional overhead of about 15 µs introduced by the garbage collector is not significant. No negative impact of different object types on maximum and average scheduling latencies can be recognized. However, worst-case latencies higher than 500 µs have been observed in every test case. For this reason, another test is performed in the next section in order to validate the upper bound for scheduling latencies when the new collector is used during a long-term execution.



(a) Latency classes with original collector.

(b) Latency classes with rc collector.

Figure 5.3: Number of scheduling latencies recorded in different classes [38].

No Load		Acyclic Load		Cyclic Load		Mixed Load	
Max	Avg	Max	Avg	Max	Avg	Max	Avg
518	91.67	524	103.88	522	104.96	544	104.03

Table 5.2: Recorded latencies for different types of object load [µs].

5.4.3 Latency Evaluation in a Long-Term Execution

The long-term test is conducted for the total duration of 48 hours and memory load of 12 acyclic objects (16 bytes / object) allocated in each iteration. Other settings are set as described in Section 5.4.2. Similar to the previous test, recorded values were splitted into multiple latency classes in order to improve the readability. The resulting diagram with logarithmically scaled y-axis can be found in Figure 5.4.

Although the latency values illustrate a deterministic behavior of the new garbage collector in general, several outliers can be detected. Values in the range of 550 µs to 600 µs were observed only twice, indicating that 600 µs is a reasonable upper bound for delays for real-time applications with a constant memory throughput. Nevertheless, 8 major outliers above 1 ms were detected among more than 17 million of total measurements. Two biggest deviations from the schedule were caused by latencies of about 45.5 ms. Remaining six latencies were measured in the range between 1.1 ms and 1.5 ms. Since the test application would crash if there is not enough memory for the next allocation, these delays cannot be attributed to the garbage collection not being able to keep up with the allocation rate. Since the collector is executed with a lower priority than the mutator, the negative interference between corresponding threads can be excluded too.

As described in Section 5.4, the evaluation of the automatic memory management with HTC Dream was performed without the optimized kernel configuration and frequency locking mechanism. It was shown in the previous chapter that in this situation real-time applications may be negatively influenced by other system components, if corresponding processes are periodically scheduled with $T > 1$ ms.

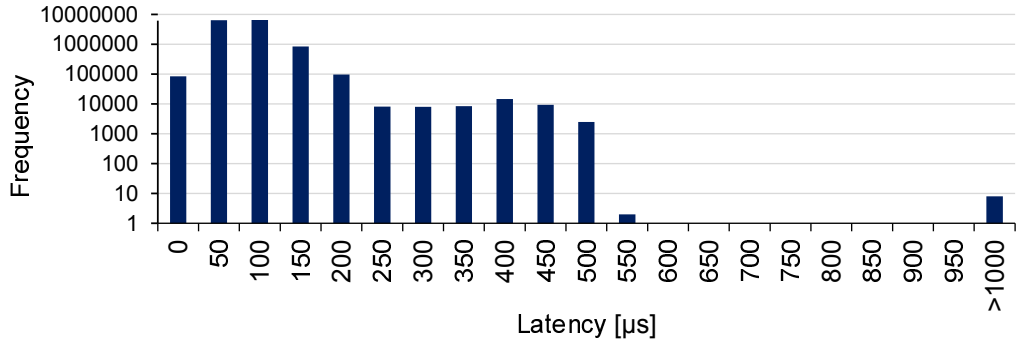


Figure 5.4: Distribution of the scheduling latency in the long-term test [38].

Android	Latency [µs]		
	Max	Min	Avg
Original	14.34	12.06	12.74
Modified	29.35	20.29	28.11

Table 5.3: Time for object allocation.

Android	Latency [µs]		
	Max	Min	Avg
Original	3.24	0.38	0.59
Modified	3.24	0.95	2.20

Table 5.4: Time for object operation.

5.4.4 Analysis of the Memory Management Overhead

The final test analyzes the additional runtime overhead introduced by the integrated facilities for the new garbage collection. Particularly, the execution time of running applications is affected by the proposed write barrier, zero count table and cycle detection algorithm, possibly leading to a decreased system performance. This impact can be measured by recording the time required for object allocation (using the `new` operator) and object operation (storing an object reference in a member field of another object) and comparing it between the original and the new implementation. Since the integrated write barrier may negatively affect the system performance even if the mark-and-sweep collector is used, the tests are executed separately for the original DVM and for the new one with modifications for reference-counting GC.

The test is implemented as a conventional Android application running with real-time priority of 80 and $T = 10$ ms. It consists of three phases, which are repeated in each of 1000 iterations. First, the mutator thread allocates 200 instances of given class C_A. This class contains a single member field – a reference to an object of class C_B – initialized with `null`. Second, existing instance of class C_B is saved in the respective member field of previously allocated objects. The assignment is performed 20 times on four different instances of class C_A (80 assignment operations in total). The time for both allocations and assignments is determined using the high-resolution timer. Finally, all object references are removed and the mutator thread explicitly invokes the garbage collection to ensure that enough memory is available for the next iteration. Values presented in the following were normalized to provide the time for a single operation.

As shown in Table 5.3, integration of the write barrier and overapproximation of cycle roots introduce a notable overhead to object allocations. The worst-case time increases

from 14.34 µs on the unmodified system to 29.35 µs when the extended DVM is used. All recorded values indicate that the proposed approach requires about twice as much time for the allocation of a single object as the original implementation.

Similar behavior can be observed based on the performance evaluation of object operations as presented in Table 5.4. Although the worst-case time of 3.24 µs remains the same on both systems, minimum and average values differ significantly. While the minimum operation time more than doubles in its value, the average time required for an assignment increases from 0.59 µs to 2.20 µs. This means that the new memory management introduces more than 370% of additional overhead, which has to be handled by the system at runtime.

5.5 Summary and Discussion

This chapter has presented the design and integration of a real-time capable reference-counting garbage collector into the Dalvik VM for Android. In contrast to the native mark-and-sweep approach, the new collector is executed concurrently and does not cause suspensions of real-time applications at runtime. Adjustable parameters for the invocation threshold and the used thread priority allow the new collector to be further tailored to the requirements of a specific real-time application.

The effect of the introduced changes was evaluated in multiple tests. It was shown that real-time applications can be protected from terminations caused by the low-memory killer by choosing an appropriate `OOM_ADJ` value. The overall system behavior is improved in terms of determinism and responsiveness, since real-time mutator threads can continue their execution regardless of other applications' memory usage and even during an ongoing garbage collection. Reference counting effectively avoids deadline misses caused by worst-case latencies of up to 120 ms, which were introduced by the original mark-and-sweep algorithm. Instead, no GC-related latencies higher than 600 µs were observed during the tests even if the application has a constant memory allocation rate. Additionally, the runtime overhead of the integrated changes was measured for object allocations and assignment operations. Test results illustrate the increased amount of time required for object processing, when comparing to the original system. However, this was expected, since the implementation of reference-counting algorithm requires additional steps to be integrated into the DVM. The overhead was shown to be bounded and it can be reduced in future by making use of deferred and coalescing garbage collection. Appropriate compaction mechanisms can also address possible heap fragmentation issues, which were not evaluated in this work.

6 Bounded Remote Procedure Calls

The extended Linux kernel and the concurrent garbage collection presented in previous chapters create a solid foundation for real-time support for Java application in Android. However, until now real-time Android applications were considered running isolated, without access to the original Android API. In order to make use of the functionality provided by the highly modular Android framework, developers have to rely on an RPC-based interprocess communication using Binder (see Section 2.2.2). It is the only IPC mechanism in Android compatible with the strict process sandboxing model, which was introduced to improve security and protect the system from undesired interference with other processes. In fact, all methods for interprocess communication recommended by the official Android documentation[1] directly or indirectly rely on Binder. Thus, the Binder driver is the core of Android's functionality linking mechanism, being responsible for all kinds of interaction between the platform and user applications.

Since the overhead for interprocess communication must be bounded in context of real-time applications, this chapter presents the analysis of Binder's internal architecture and an extension approach for priority-based RPC handling. It preserves the priority of the execution thread across process borders and enables the preemption of low-priority threads on the callee side.

This chapter begins with a summary of related work in Section 6.1. Section 6.2 covers the design of the new Binder architecture in two parts. Since there is no public documentation of the Binder framework, a brief overview is presented in Section 6.2.1. After the discussion in Section 6.2.2, the integration of the high-level components and an overview of the required changes is provided in Section 6.2.3. Section 6.3 evaluates the effects of the introduced modifications with focus on delays during remote procedure calls. Finally, the chapter is concluded in Section 6.4.

6.1 Related Work

The evaluation of Binder-related functionality found in the literature mainly focuses on high-level components like Intents and Broadcast Receivers, which will be covered in detail in the next chapter. There is only little effort invested in the analysis of Binder's low-level architecture. Until now, the research was limited to security enhancements and improved privacy protection mechanisms. Tracking of Binder transactions was proposed in RiskMon [57]. It inserts tracing hooks into the Binder driver layer for access monitoring and risk assessment. Binder customization is also used in TaintDroid [29] for

[1] IPC in Android: `http://developer.android.com/training/articles/security-tips.html#IPC`

privacy improvements. Similarly, the Scippa project deploys an extended Binder driver for a more effective prevention of security attacks [11].

The preliminary publication in context of this thesis was the first work to evaluate Binder-based RPC mechanism in context of real-time applications [62]. Although the approach presented by Mauerer et al. allows interprocess communication in real-time processes on Android, it is based on shared memory regions and does not incorporate the high-level Android framework. The RTDroid project has proposed a new design for Android's subsystem for delivery of sensor data [132]. It incorporates modifications of the Handler and Looper classes, which are responsible for internal event handling. However, the actual data passing across process boundaries managed by Binder was not considered in their publication. As it will be shown in the following sections, the original implementation of the Binder framework is not capable of reliable message delivery and has to be extended for usage in real-time domains.

6.2 Extended Binder Driver

The Binder driver included in Android is an adapted version of the OpenBinder[2]. It consists of three layers:

- The actual driver is written in the C programming language as part of the Linux kernel. Its low-level command interface is based on `ioctl` system calls.
- A C++ wrapper encapsulates the kernel access and provides methods to upper layers. This wrapper handles the data serialization and controls Binder worker threads inside running applications.
- The application framework simplifies the usage of the C++ wrapper in Java by offering another level of abstraction. It manages the control flow between Java and native code through autogenerated objects `Proxy` and `Stub`.

Figure 6.1 summarizes the main components involved in the Binder-controlled IPC. Each running application in Android has its own Binder node inside the kernel driver and several local threads – referred to as Binder or *Looper* threads – to manage incoming connections. In order to execute remote procedure calls in the context of another process, Android generates a local *Proxy* object with the identical interface. The desired method can be invoked on this proxy object in the same way, as if the corresponding component was available in the current process. However, since the proxy does not contain the actual method logic, it only serializes the supplied arguments and transfers the execution to the Linux kernel. The serialized data is then passed to the Binder driver, which resolves the target application and copies the data to the callee process. After the respective method was executed on the remote side, the computed result is returned to the caller in the same way. All interactions with the local proxy object are blocking, such that the calling thread is not able to distinguish between method calls on true local objects

[2]OpenBinder project homepage: `http://www.angryredplanet.com/~hackbod/openbinder`

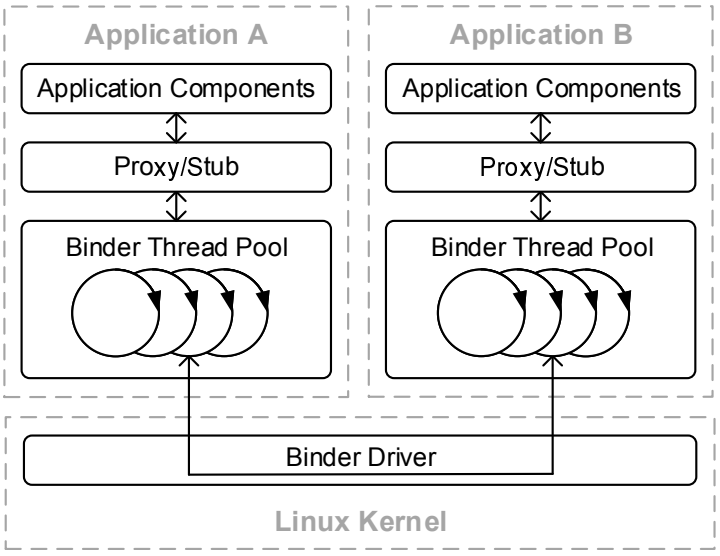


Figure 6.1: Schematic illustration of Android's Binder architecture.

and remote invocations. On the one hand, this encapsulation provides an effective and flexible way for linking applications components running in separate processes. On the other hand, blocking operations across multiple processes may cause random delays at runtime. In a real-time system, all latencies caused by internal operations have to be bounded and predictable.

In its original implementation, neither the low-level Binder driver nor its high-level wrapper provide measures to ensure a deterministic execution of pending RPC requests. The driver contains several critical sections protected by a mutex, which does not consider the priority of waiting threads. This poses a high risk for real-time threads, as the execution of their RPCs may be blocked by regular threads. Mercer and Tokuda have identified similar threats in the context of real-time applications, if the implementation relies on non-preemptible critical sections or interaction between components with different priorities [85]. This issue can be addressed by extending the Binder driver and the corresponding wrapper implementation to support the priority inheritance protocol across multiple processes. Due to the lack of the official documentation, the next section presents a brief overview of Binder's original implementation[3] and discusses its disadvantages if used in real-time applications. Afterwards, modifications to improve Binder's behavior are presented in detail.

6.2.1 Overview of Binder's Architecture

This section illustrates the interaction between different Binder layers in a simplified scenario where a client application invokes a remote procedure call in the context of a separate service process. The sequence of steps, which is performed during this RPC, is summarized in Figure 6.2.

[3]Based on the source code analysis of Android 4.2.2 in `kernel/drivers/staging/android/binder.c`

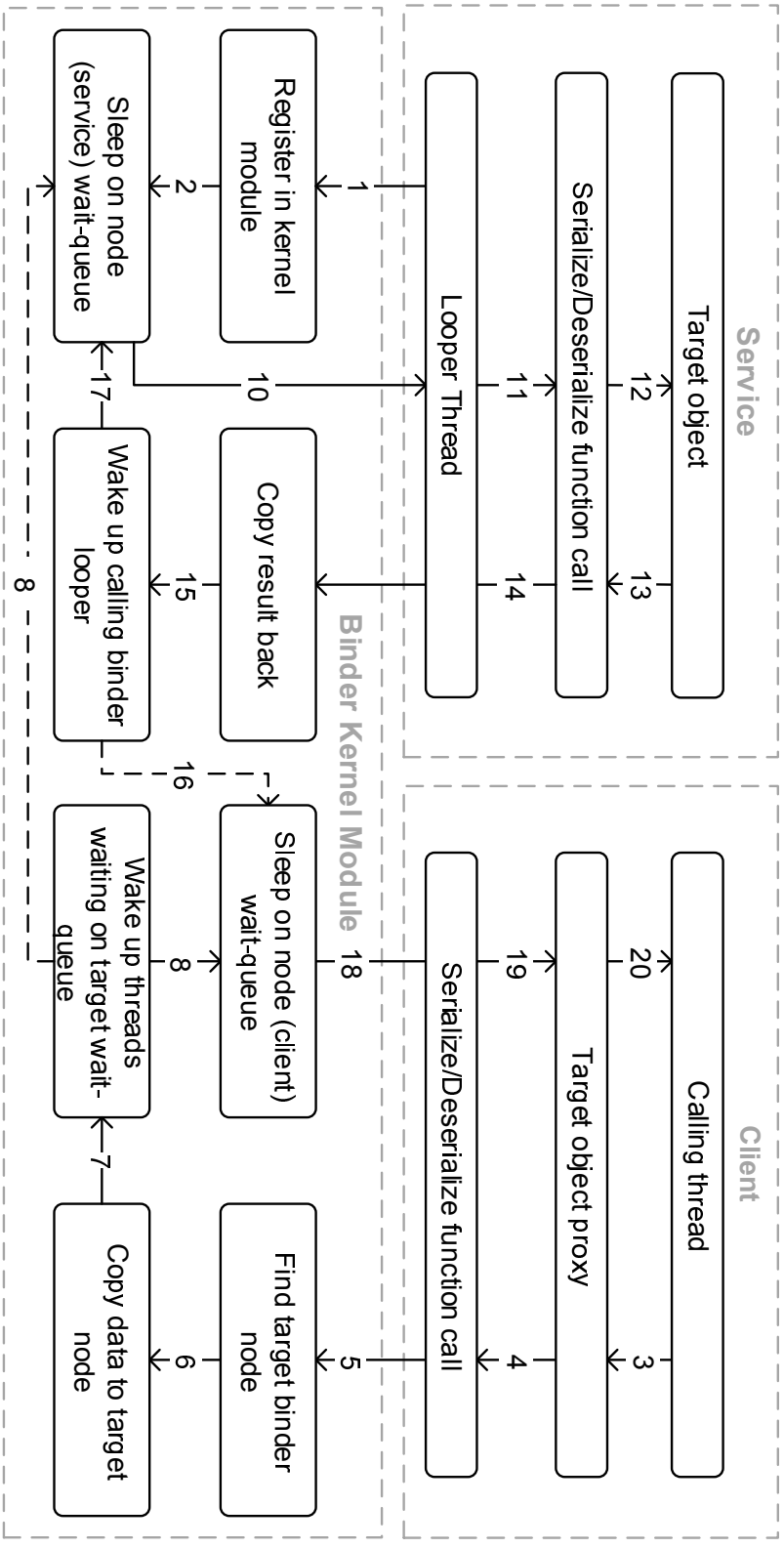


Figure 6.2: Binder steps executed during a remote procedure call [62].

After the service process is started, the Binder driver automatically spawns one or multiple Looper threads to handle incoming connection requests. These threads register themselves in the kernel module (1) and sleep on the corresponding process-specific node as long as there are no pending RPCs. Inactive Looper threads remain in the waiting state by default until another Binder node performs a wake-up call (8) on their node.

The client application stores a proxy object, which implements the same interface as the target object stored in the service. By calling the desired method directly on the proxy object (3), the client implicitly initiates a new RPC. In the first step, the meta information about the call as well as the passed arguments are serialized by the proxy (4) and redirected into the Linux kernel. Since all running processes are registered in the low-level driver, the transmitted meta information (e.g. the package name) is used to find the node of the target application (5). Each node contains a FIFO queue, which stores all pending events waiting to be handled by the corresponding process. After finding the target node, the serialized RPC is inserted into the node's FIFO queue (6). Since each node resides in the memory region of the corresponding process, this operation can only be performed by the kernel driver and not by the client application itself. For this reason, this step finishes the data transmission and passes the execution to the service process (7) by performing a wake-up call on its node (8). To ensure a blocking operation mode, the client thread is put asleep (9) until the execution result of this request is delivered.

The service side responds to the incoming request by waking up all waiting threads and randomly choosing one of them to process the RPC. If the FIFO queue contains only one element, remaining Looper threads will be sent back to sleep until the next event. The thread processing the RPC transaction will return to the user-space (10) and deserialize the original meta information (11). This information is used to invoke the desired method on the actual object (12) and receive the calculation result (13). After the execution is complete, the result is returned to the client in a similar way. It is serialized and forwarded to the low-level Binder driver (14) in order to be copied to client's memory. To finish the RPC processing on the service side (15), the Looper thread notifies the client about the pending result (16) and enters a new sleep phase until the next event occurs (17).

The control flow returns to the original thread on the client side, which was put asleep in step (9). After leaving the kernel-space (18), this thread deserializes the result value (19) and returns from the original method call on the proxy object, delivering the remote result as if it was calculated locally. In this simple example, the processing thread on the client side is not a Looper thread, but a regular user-space thread. Complex Android applications typically allow RPCs calls from the service to the client too, utilizing the Binder-based communication in both directions.

The original implementation of the low-level Binder driver makes use of different components during the RPC handling. One of the most important parts for the node management is encapsulated by the `binder_node` struct for storing the relevant information about running processes. It includes the `binder_proc` struct, which is shared by all Looper threads of the same Binder node. This struct is essential for RPC processing, as it contains both the wait queue `wait`, used to synchronize and wake up waiting Looper

threads, and the FIFO transaction queue `todo`, used to store pending Binder events. Multiple critical sections protected by corresponding mutexes ensure the consistency of the global data. Furthermore, the original implementation defines various internal commands to take influence on the current system state at runtime. For example, the driver reacts to an excessive use of a specific Binder node by spawning additional Looper threads. A new thread with the default priority is created in the given application process in order to handle the rising count of incoming RPC events. This architecture encapsulates the actual process of data passing across process boundaries and enables a flexible IPC mechanism, which can be used in high-level Android applications. However, this flexibility leads to several disadvantages, when Binder-based RPC are used in the real-time domain.

6.2.2 Analysis of Binder's Real-time Support

Since the Binder driver is one of the most extensively used components in Android, a high amount of low-level transactions are generated on the device every second. For this reason, the Binder driver was designed with focus on performance, but not on predictable blocking times. The first drawback arises from using regular Looper threads to process events created by applications with other priorities in the user-space. An RPC invoked by a thread with real-time priority will be handled by a regular thread on the service side. This facilitates priority inversion scenarios, as the processing Looper thread may be interrupted on the remote side by any other thread with higher priority.

Automatically spawning additional threads in high-load situations may decrease the system predictability even further. It leads to additional processing overhead and causes significantly longer waiting periods until the critical section can be entered. Due to the increased number of the competing threads waiting to acquire the mutex, threads processing regular RPCs may block threads with RPCs from real-time applications. In addition, the used mutex does not distinguish between threads with different priorities. Even though the original implementation use the same priority for all Looper threads, temporary increasing the priority of the processing Looper thread will have no effect on the execution. If multiple threads are waiting to acquire the mutex in the moment of its release, the ownership will be transferred non-deterministically to one of the waiting threads regardless of its priority.

Another negative effect is caused by the deployed FIFO queue, which stores pending RPC transactions. Since an event from a privileged thread may be inserted at any time, its processing will be delayed, until the execution of all other transactions inserted earlier have been finished. Furthermore, there is no guarantee that a free Looper thread will be available in the exact moment of the RPC arrival. In the worst case, additional thread has to be started and prepared first, leading to a significant processing overhead.

Identified design flaws can prevent the system from deterministic data delivery and expose real-time applications to unbounded processing delays. In order to improve the process behavior, this dissertation proposes a new approach for priority inheritance during remote procedure calls. Required modifications of the low-level kernel driver and of the corresponding wrapper implementation are summarized in the next section.

6.2.3 Integration of the Priority Inheritance

In real-time systems, the worst-case delay for a remote procedure call has to stay predictable regardless of the current system load. This delay combines the time to find and access the remote process, to execute the desired method and to the return the calculated result back to the client. However, since the actual method execution time is part of the target application and cannot be influenced by the system, it is excluded from further consideration. Thus, following modifications address only the original behavior of the Binder driver itself.

- As shown in the previous section, the low-level driver randomly chooses the next idle Looper thread to process an RPC request, even if it was generated by a real-time thread. This issue is solved by an additional Looper thread, implemented to only handle RPCs from real-time threads. As soon as the Binder node is accessed by a user-space thread with real-time priority, the additional Looper thread is created or woken up and set to the same priority as the caller. In the current implementation, this thread is reused for all privileged transactions. This approach guarantees that a real-time Looper thread will be available in all applications by default. It is used to immediately handle requests from real-time clients on the remote side with the same real-time priority. In addition to the original wait queue `wait`, the struct `binder_node` is extended with a new wait queue `rt_wait`. It is required for a selective wake up operation (see Step 8 in Figure 6.2) depending on whether a regular or a real-time Looper thread has to be activated.

- In order to avoid further blocking by low-priority RPC transactions stored in the `todo` FIFO queue, the struct `binder_node` is extended with an additional high-priority transaction queue `rt_todo`. It is dedicated to store privileged RPCs only and is processed separately by the new real-time Looper thread.

- The last major source of non-determinism are critical sections in the low-level driver. Since the ownership of the original mutexes is transferred randomly, the real-time Looper thread may suffer from priority inversion while one of the regular Looper threads is holding the mutex. To ensure that the access to the critical section is granted strictly depending on the priority of the blocked threads, original mutex declarations are replaced by equivalent `rt_mutex` defines from `RT_PREEMPT`.

Proposed modifications introduce a deterministic processing of incoming transactions and allow priority-based RPC handling. In addition to better system responsiveness, privileged calls can now be executed with minimal blocking time. The Binder RPC sequence presented in Figure 6.2 has to be adjusted to benefit from the new architecture. The following list shows only the modified steps of the full sequence in case a real-time thread on the client side generates a new RPC transaction.

5. Record the scheduling priority s of the current real-time thread.

6. Store the transaction data in the `rt_todo` queue, instead of the original `todo` queue.

8.1. Request the creation of a dedicated Looper thread, if it does not yet exist.

8.2. Notify the `rt_wait` queue, instead of the `wait` queue to wake up the new Looper.

10.1. Set the priority of the new Looper thread to s.

10.2. Poll the data from the `rt_todo` queue, instead of the original `todo` queue.

17. Sleep on the `rt_wait` queue, instead of the `wait` queue.

This approach establishes a priority inheritance mechanism during remote procedure calls in Android, while keeping architectural changes at minimum. All threads involved in the execution of an RPC are guaranteed to have the same priority, which allows privileged execution across process boundaries. Additional Looper thread on the callee side inherits the priority of the caller and benefits from the real-time capable mutex, which preserves the correct execution order for high-prioritized threads. Furthermore, the new thread also complies with the original Looper/Binder life cycle and uses a separate job queue, minimizing the integration and maintenance overhead. Other resources required at runtime are allocated analogously to other Looper threads, such that they are released by the Binder driver automatically, after the termination of the corresponding process.

6.3 Experiments

This section evaluates the introduced system modifications in multiple tests. It begins with a short description of the testing setup and a definition of the measured values. The first part of the evaluation compares the behavior of the original and of the modified system depending on the system load. Finally, the last part discusses the scalability of the new approach in setups with multiple real-time threads.

Testing was performed on the Google Nexus 10 (codename `manta`) tablet computer, which is built with a Samsung Exynos 5250 (2x 1.7 GHz Cortex-A15) SoC and 2 GB of RAM. This device runs the Linux kernel `android-manta-3.4.5-rt15` as part of the modified Android 4.2.2. Test cases were implemented in conventional Android applications with introduced extensions as explained in previous sections.

Applications A and B are used to perform remote procedure calls via the Binder driver as shown in Figure 6.3. Application A spawns k_{rt} real-time threads with priority of 80 and k_{nrt} regular threads. All threads periodically invoke the method `getTimestamp()` on an object in the context of application B. This method is implemented to return the current timestamp t_{res} from the high-resolution timer enabled by `PREEMPT_RT`. In order to determine the processing time, each thread records a timestamp t_{pre} immediately before the method is invoked and another timestamp t_{post} immediately after method `getTimestamp()` returns. The processing time can be split into the invocation delay T_{CALL}, method execution time T_{EXEC} and the return delay T_{RET}. Since T_{EXEC} is application-specific, it is excluded from the evaluation. Remaining delays are calculated based on the returned result value t_{res}: $T_{CALL} = t_{res} - t_{pre}$ and $T_{RET} = t_{post} - t_{res}$.

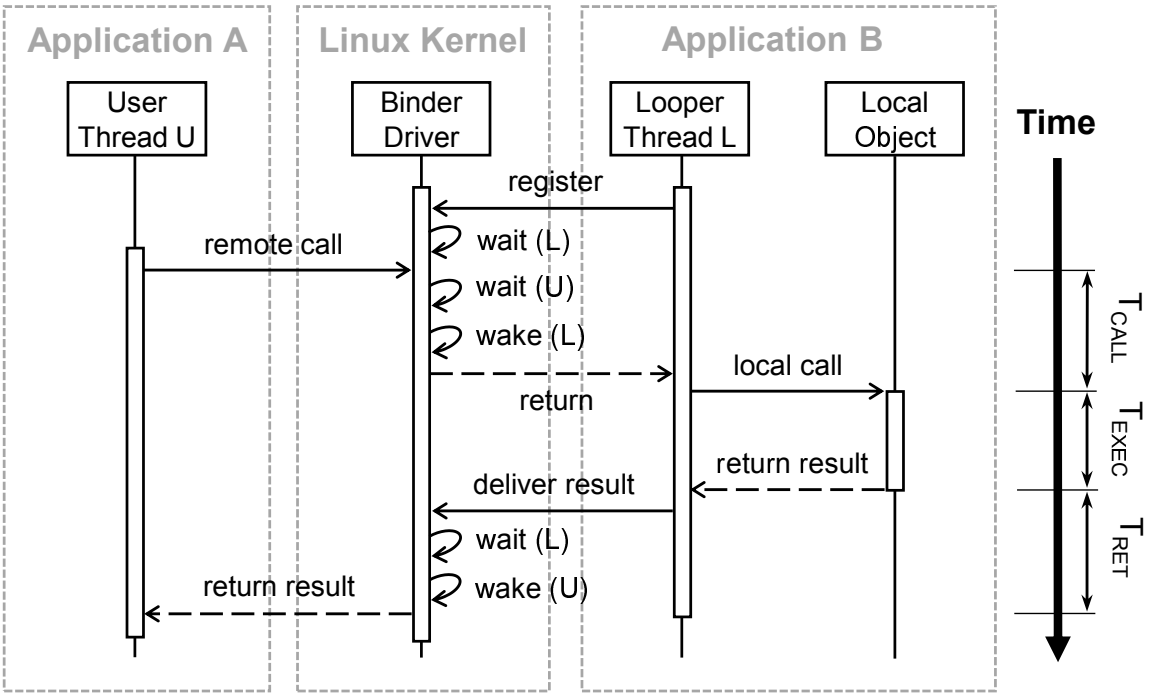


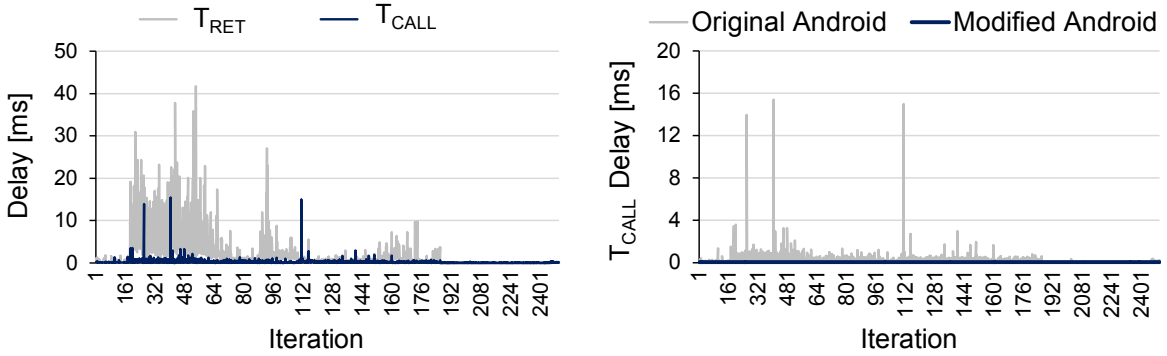
Figure 6.3: Measuring time for a single RPC invocation in Android.

6.3.1 Performance Evaluation under Load

This section evaluates the performance of the Binder driver for rising number of regular threads. In the first test, one real-time thread and $k_{nrt} = 20$ regular threads are used to invoke the RPC as described above. All non-real-time threads are spawn only to generate additional system load, their processing delays are neither recorded nor evaluated. Invocation delays T_{CALL} and return delays T_{RET} are recorded in 2500 iterations.

Values measured with the original implementation are summarized in Figure 6.4(a). Obviously, the unmodified Binder architecture cannot guarantee predictable timings. While recorded invocation delays show only a few outliers up to 15.4 ms, returning a value from the remote method blocks the executing thread up to 41.7 ms. The vast majority of return delays are significantly higher than the invocation delay in the same iteration. In both cases, the handling is negatively affected by other Looper threads, as the priority of the caller is not inherited by the callee. Thus, new method invocations or returning the data may be blocked for a non-deterministic time until the Looper thread carrying the privileged RPC acquires the mutex.

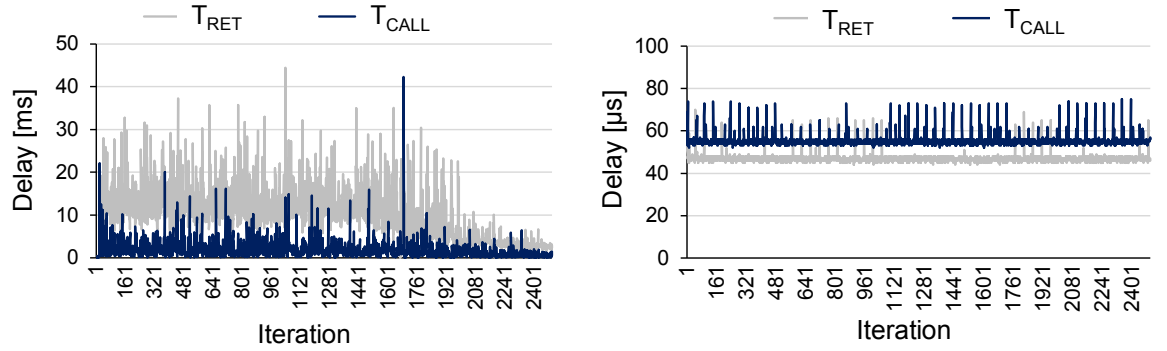
Binder modifications proposed in this chapter allow a more predictable system behavior. A comparison between T_{CALL} delays in the original and in the new implementation is shown in Figure 6.4(b). Although the original driver typically handles the RPC invocation in less than 2 ms, random outliers are distributed without a recognizable pattern. The modified Binder is able to avoid these outliers and notably reduce the invocation delay. Integrated priority inheritance mechanism improves the system predictability and introduces an upper bound below 1 ms.



(a) Both Binder delays in the original Android.

(b) Comparison of T_{CALL} for $k_{nrt} = 20$.

Figure 6.4: Analysis of Binder delays for $k_{nrt} = 20$ regular threads [62].



(a) Binder delays in the original Android.

(b) Binder delays in the modified Android.

Figure 6.5: Analysis of Binder delays for $k_{nrt} = 100$ regular threads [62].

The system performance in high load scenarios on the original and on the modified Android is evaluated in the next test. It reuses the testing environment presented earlier, but increases the number of regular threads to $k_{nrt} = 100$. Although all threads execute the same remote method in a separate process and perform the same calculation, T_{CALL} and T_{RET} values are only recorded for the single real-time thread.

The behavior of standard Android is shown in Figure 6.5(a). Increased number of regular threads causes major disturbances in RPC processing and leads to higher delays, when compared with results of the previous test in Figure 6.4(a). Although returning a result value from a remote method performs notably worse than its invocation, both delays fluctuate during the test. The diagram shows random distributions with multiple outliers up to 42.3 ms for T_{CALL} and up to 44.5 ms for T_{RET}. Since the unmodified Binder cannot guarantee a reasonable upper bound, it is concluded to be unsuitable for predicable RPC execution in real-time applications.

Figure 6.5(b) depicts the results of this test on the modified system. Both graphs show a constant processing time without major negative effects caused by 100 regular threads. Since invocations from the real-time thread are executed by a privileged Looper thread

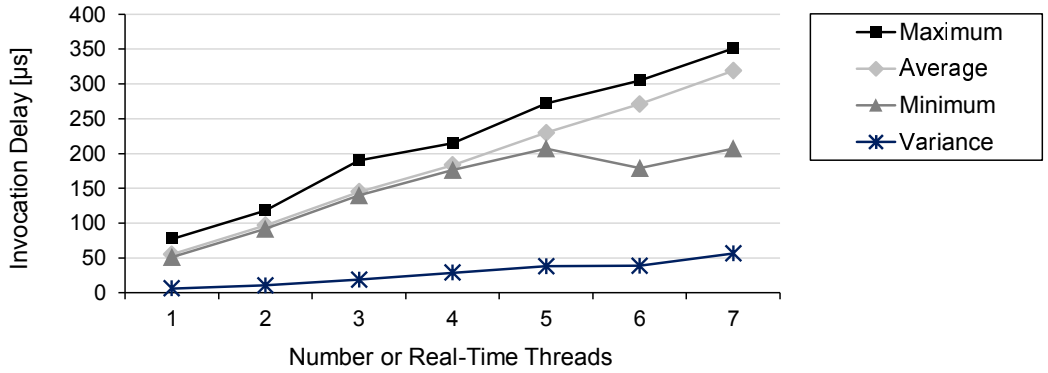


Figure 6.6: Invocation delays for rising number of real-time threads [62].

on the remote side, the RPC can be processed immediately without further blocking by other threads. This does not change in high-load situations, such that both delays are bounded by 51 $\mu s \leq T_{CALL} \leq 75$ μs and 41 $\mu s \leq T_{RET} \leq 70$ μs. Periodic increase of both delays by approximately 30 μs illustrate cases where the critical section is blocked by another thread and cannot be entered immediately. In the first test with $k_{nrt} = 20$ regular threads there were approximately 30 such outliers for each T_{CALL} and T_{RET}. With $k_{nrt} = 100$ there are more than 45 outliers. This behavior was expected, as the probability of a mutex being already acquired increases with higher number of competing threads. Nevertheless, the extra blocking time is reliably bounded, as the real-time thread is guaranteed to receive the ownership of the `rt_mutex` as soon as it is released by a regular thread. This assumption is confirmed by recorded measurements, since the real-time thread is not blocked for longer than 30 μs during the test execution. Thus, the new Binder driver is shown to provide reliably bounded invocation and return delays for remote procedure calls even under major load from concurrent regular threads.

6.3.2 Multiple Real-Time Threads

The final test is performed to evaluate the scalability of the proposed approach in scenarios with $1 \leq k_{rt} \leq 7$ real-time threads. Additional RPC load is generated by spawning $k_{nrt} = 100$ regular threads, which execute the same RPC analogously to the previous test. To achieve a better readability of the resulting diagram, it only presents the invocation delays of all real-time threads combined. For each $k_{rt} \in \{1, ..., 7\}$, the statistical evaluation in Figure 6.6 is based on the total number of $k_{rt} \times 2500$ recorded values.

The diagram shows a recognizable linear trend of all depicted values. Worst-case delays have only a slight deviation of about 30 – 40 μs from the corresponding average, confirming a bounded and predictable processing. The average duration of the RPC invocation reaches from 55 μs for $k_{rt} = 1$ to 319 μs for $k_{rt} = 7$. Notably, these values are arranged on an apparent straight line with an approximate increase of 44 μs for each additional real-time thread. Previous tests have shown that this overhead roughly corresponds to the execution duration of one RPC invoked by a real-time thread. Considering that only one privileged Looper thread is created on the remote side to handle

high-prioritized transactions, the rise of the average values can be explained by the sequential data processing. The introduced `rt_mutex` preserves the correct execution order, such that no further blocking delays are produced by RPCs from regular threads.

6.4 Summary and Discussion

This chapter has presented an analysis of Binder's internal architecture with an approach to improve its suitability for the real-time application domain. The investigation of the related platform components is performed in Section 6.2.1. Identified flaws in the original implementation were considered during the new design for both Binder subsystems: the low-level driver and the high-level wrapper. The new architecture enables a more robust IPC based on remote procedure calls by inheriting the priority of the executing thread across process boundaries. Thus, real-time applications can avoid priority inversion while executing code on the remote side. In contrast to the original implementation – which handles all incoming RPCs by a regular Looper thread – the enhanced Binder architecture relies on a dedicated real-time thread to process method calls from high-prioritized threads. Additionally, the new approach relies on a real-time mutex from `PREEMPT_RT` to protect the critical sections in the Binder driver. Instead of the non-deterministic selection of the next thread to acquire the mutex, the access is now granted to the waiting thread with the highest priority.

Evaluation of the proposed modifications to the low-level Linux driver and the high-level wrapper was performed in multiple tests. The original framework was shown unsuitable of providing predictable behavior, making it impractical for use in real-time applications. The Binder driver included in the original Android platform introduces random delays up to 4 ms during a remote method call even on an idle system. Oppositely, latencies caused by the modified implementation were shown to be bounded under different testing conditions. Test results highlight that the new approach provides a deterministic behavior even with high CPU load and multiple competing real-time threads. Although the WCET for a remote method call is rising with the number of active real-time executors accessing the same method, the upper bound shows a strict linear dependency on the number of competing threads. The maximum value stays predictable and can be calculated a priori during the system design phase. In future, the performance of the proposed extension may be further increased by creating a pool of real-time threads with different priorities.

Integration of the extended Binder driver – which is involved in different kinds of interaction between user applications and the Android platform – as suggested by the presented approach improves the overall system responsiveness. The new subsystem for remote procedure calls is used by all active applications implicitly, leading to a more reliable and deterministic data exchange across process boundaries. A special case of such IPC is Intent broadcasting, which is analyzed in detail in the next chapter.

7 Prioritized Intent Broadcasting

The last chapter has presented a way to bound the overhead caused by remote procedure calls, used all native communication mechanisms in Android. It is especially important for broadcasting Intents between different Android applications or their components, which is one of the most important concepts in Android [3, 82]. Since a real-time capable platform has to provide reliable and predictable methods for inter- and intraprocess communication, this chapter analyzes the capabilities of Android's Intent broadcasting. As it will be shown in the evaluation, the built-in `FLAG_RECEIVER_FOREGROUND` for preferential treatment of foreground Intents does not prevent the system from causing non-deterministic delays during the message delivery. Thus, this chapter presents a new approach to bound the execution overhead and reduce the broadcasting delay for both global and local broadcasts. Unordered data structures used in the original implementation for first in – first out Intent handling are replaced in order to allow priority-based processing. Additionally, blocking times during the internal process flow are reduced by minimizing the corresponding critical section. This leads to a better preemptibility and guarantees predictable delays and higher scalability for real-time applications.

The remainder of this chapter is structured as follows. An overview of the related work is provided in Section 7.1. The design and integration of the introduced modifications are covered by Section 7.2. After a short analysis of Android's original architecture, Section 7.2.2 summarizes the proposed approach of explicit Intent prioritization. Implementation details are presented separately for global broadcasting in Section 7.2.3 and for local broadcasting in Section 7.2.4. The impact of the new approach is evaluated in multiple tests in Section 7.3. Finally, Section 7.4 concludes this chapter with a brief discussion of the presented enhancements.

7.1 Related Work

Interprocess communication in Android was evaluated in a number of scientific publications presented in recent years. However, publicly available research results are limited to the analysis of security-related questions. For example, measures for the identification of security risks arising during the communication between separate processes were presented by Chin et al. [19]. An empirical study of inter-component interaction with focus on robustness was performed by Maji et al. [76]. Their work proposes a modified data structure used by Intents, which was designed to achieve higher compatibility between sending and receiving components. Other publications have presented similar ideas for improving Android's IPC architecture in terms of security and privacy protection [65, 90, 106].

Until today, only a few studies have examined the predictability of existent IPC methods in Android or proposed new approaches suitable for real-time applications. Mauerer et al. was the first to present a data exchange mechanism between real-time Linux processes in Android [79]. It uses shared memory regions with additional synchronization algorithm based on publish-subscribe pattern. However, the authors only consider processes on the top of the Linux kernel and do not evaluate high-level Android components.

Another approach was implemented as part of RTDroid [133]. It relies on an extended application architecture using modified `Handler` and `Looper` classes. This allows real-time applications to post prioritized events into the internal message queue. At a later stage, these events are dispatched and translated into user-defined local method calls for corresponding actions. This mechanism allows predictable delivery of messages and runnables within a single application process in a bounded time. However, RTDroid does not provide any mechanisms for real-time capable data transmission across process boundaries. Furthermore, the corresponding research does not cover Intent broadcasting architecture.

7.2 Extended Broadcasting Architecture

Android's broadcasting system is a powerful mechanism for sending and receiving generic messages. It is one of the most flexible ways for the exchange of structured data between the system and user applications, or between multiple applications from different vendors or even between separate components of the same application. Although Android supports various types of Intent broadcasts (see Section 2.2.2), generic communication is typically implemented by relying on parallel broadcasts, also referred to as *normal broadcasts*[1].

This section evaluates the architecture behind Android's Intent processing, which covers the central system components like the Activity Manager Service (AMS) used for global broadcasts and the Local Broadcast Manager (LBM) used for broadcasts inside the same process. Since there is no official documentation for the internal Intent handling, this section provides an architecture analysis of respective components based on their source code in Android 4.2.2. After presenting the implementation details, the original architecture is extended with explicit Intent prioritization mechanism for both global and local broadcasts.

7.2.1 Analysis of Intent Broadcasting

A schematic illustration of the global broadcast handling is given in Figure 7.1. It shows that the application A can transmit Intents by using the `sendBroadcast()` method in any of its components. Since Android's security policy prevents user applications from accessing all registered receivers directly, the Binder driver is used to forward the data to a persistent system service implemented in the `ActivityManagerService` class. The AMS manages all known applications and their active broadcast receivers, which are

[1] Official documentation: `https://developer.android.com/guide/components/broadcasts.html`

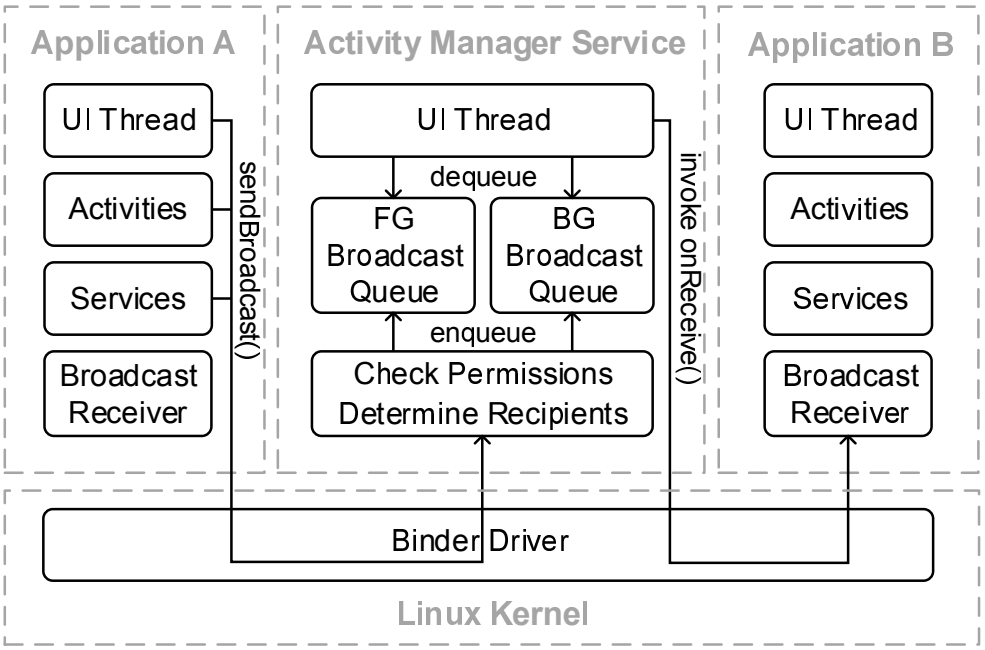


Figure 7.1: Schematic delivery of global Intents using the Activity Manager Service.

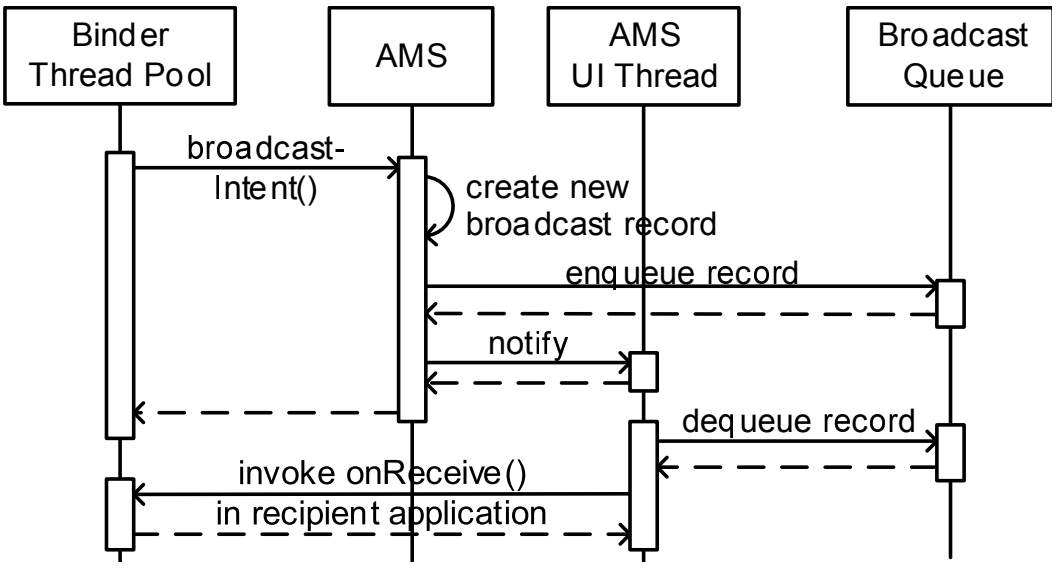


Figure 7.2: Internal broadcast processing in the Activity Manager Service [61].

used to resolve all recipient processes for a given broadcast. This information and the original Intent object are encapsulated in a new `BroadcastRecord` object and saved in the internal `BroadcastQueue` to be handled later. Android manages a *foreground* queue und a *background* queue, both designed for simple FIFO processing. The foreground queue stores only Intents broadcasted with enabled priority flag `FLAG_RECEIVER_FOREGROUND`. All other broadcasts are stored in the background queue. The Activity Manager Service relies on its UI thread to dequeue the pending broadcast records and pass them to the recipient application via Binder. In Figure 7.1 this step is denoted by the transition to the application B, which invokes the method `onReceive()` in the registered Broadcast Receiver.

The process of internal record processing inside the Activity Manager Service is additionally illustrated in Figure 7.2. As presented above, the broadcast is delivered by the Binder driver and handled on the service side by one of the Looper threads. The Looper executes the method `broadcastIntent()` in the AMS, which provides the record insertion logic. Notably, the original implementation of this method is not thread-safe. Race conditions are avoided by protecting its major part with a `synchronized`-block, such that only one Looper thread can insert new broadcast record at a time. In order to minimize blocking times, further processing is done by the UI thread of the service. The Looper sends a notification about the pending broadcast and exits the critical section to allow other records to be inserted. The Binder driver performs the actual broadcast delivery as described above.

If the user application does not have to receive global broadcasts, but only relies on the internal communication between its own components, the Local Broadcast Manager can be used. As shown in Section 2.2.2, the LBM is designed to provide a similar interface for registering Broadcast Receivers and sending Intent objects within the same process. An overview of the LBM's main components is presented in Figure 7.3. Analogously to the global approach, all application components can use the singleton instance of the Local Broadcast Manager to generate a new broadcast, which is encapsulated by a broadcast record and inserted into the local FIFO queue. In contrast to the AMS, the LBM does not provide a separate data structure for foreground broadcasts and saves all records in the same queue by default. Selection of the thread to handle the pending records depends on the type of the local broadcast. If the `sendBroadcast()` method was used, the application's own UI thread will be notified to deliver the Intent. Otherwise, if the `sendBroadcastSync()` method was used, the processing will start immediately in the context of the calling thread. Notably, the latter approach does not only deliver the current Intent, but rather dequeues and processes all pending records stored in the waiting queue. This may lead to a situation where another Intent object, originally broadcasted using the method `sendBroadcast()`, is actually delivered to its receiver by the current thread, instead of the UI thread. This process illustrated by the sequence diagram in Figure 7.4. Furthermore, the presented diagram highlights that the `onReceive()` method of the corresponding Broadcast Receiver may be called from different threads, depending on which broadcast function was executed. This difference is important, as specific actions in Android may only be performed by the UI thread (e.g. accessing visible elements).

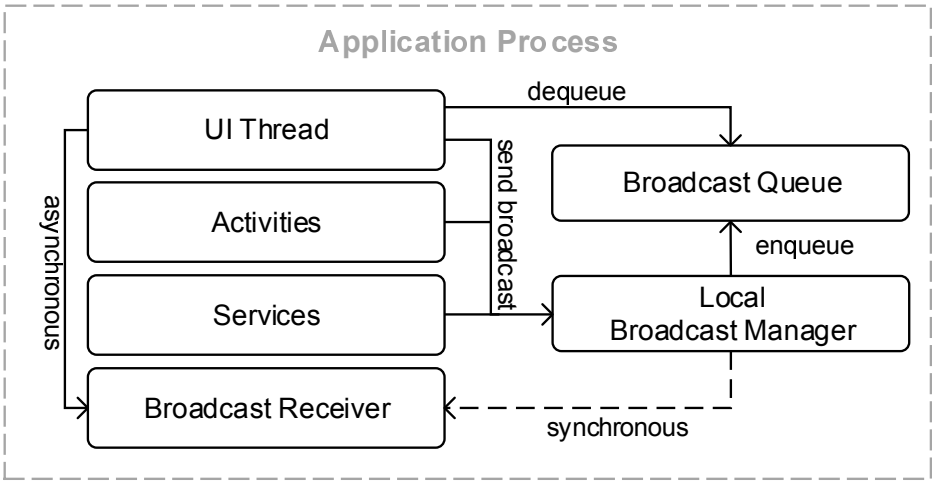


Figure 7.3: Internal architecture of the Local Broadcast Manager [61].

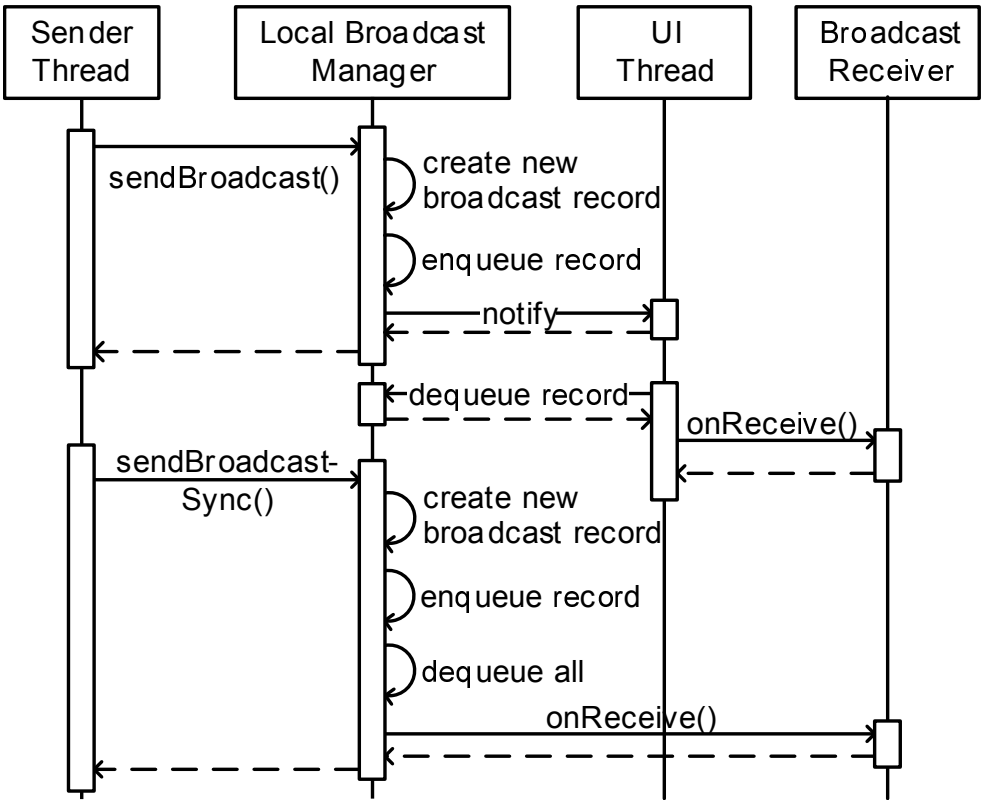


Figure 7.4: Broadcast handling in the Local Broadcast Manager [61].

```
// transmit a low-prioritized Intent
Intent intentL = new Intent(ACTION);
intentL.putExtra(INTENT_PRIORITY_EXTRA, 5);
sendBroadcast(intentL);

// transmit a high-prioritized Intent
Intent intentH = new Intent(ACTION);
intentH.putExtra(INTENT_PRIORITY_EXTRA, 80);
sendBroadcast(intentH);

// implicitly use the priority of the current thread
Intent intentT = new Intent(ACTION);
sendBroadcast(intentT);
```

Listing 7.1: Sending prioritized broadcasts in Android [61].

7.2.2 Intent Prioritization Mechanism

In contrast to local broadcasts, which are always handled with the same priority, global approach of Intent transmission can make use of the `FLAG_RECEIVER_FOREGROUND` to request preferential treatment (see Section 2.2.2). Such broadcast records are inserted in the foreground queue and transmitted prior to Intents stored in the background queue. However, it will be shown in the evaluation that the usage of this flag guarantees neither a deterministic data processing, nor bounded delivery delays. The lack of a coherent prioritization mechanism may introduce significant blocking times for pending broadcasts. Intent objects sent by privileged processes can only be processed after all other Intents stored in the same queue have been delivered. Since all processes can set the foreground flag regardless of their priority, this method cannot protect real-time broadcasts from undesired delays. This drawback is addressed in the following by introducing a priority queue to ensure the correct processing order. It allows broadcast records to be sorted based on the priority of the encapsulated Intent object, instead of being sorted by their creation time.

In order to make use of this approach on the application side, Intent objects generated from the thread i are augmented with the broadcasting priority value b_i. As illustrated by the example in Listing 7.1, the priority value b_i can be directly assigned in the application logic. If no explicit value is provided, it will be set automatically to the scheduling priority s_i of the corresponding sending thread. Additionally, the method `sendBroadcast()` is extended to avoid privilege escalation when passing the Intent object to the broadcasting manager. The validation process uses the current priority s_i of the thread i as an upper bound for possible b_i. This prevents real-time threads with low priorities from jeopardizing the internal Intent management by always choosing the highest possible value $b_i = s_{max} = 99$ (see Section 2.3.1).

As explained in the last section, both AMS and LBM enqueue incoming broadcasts using critical regions. Corresponding `synchronized`-blocks protect the identification of matching Broadcast Receivers, the creation of a new broadcast record and its insertion

in the correct broadcast queue. On the one hand, this guarantees the consistency of the internal state in cases where multiple applications generate broadcasts simultaneously. On the other hand, this setup allows only one Looper thread to enter the critical section at a time, leading to a major bottleneck and neutralizing the benefits of a multithreaded environment. As a real-time operating system has to maximize the overall system predictability and responsiveness for privileged applications, portions of source code enclosed into `synchronized`-sections must be kept as short as possible [18, 85]. For this reason, the relevant critical sections in AMS and LBM are reconstructed in order to ensure the priority-based processing and to reduce the blocking time.

Another important aspect of the Intent delivery is the availability of the processing thread. The sender application and the corresponding Looper thread only can insert a new Intent object into the broadcast queue, but not deliver it to the Broadcast Receiver[2]. This task is typically performed by the UI thread of either the Activity Manager Service (for global broadcasts) or of the application itself (for local broadcasts). However, in both cases the respective UI thread is also responsible for processing a high number of other events. For example, the AMS interacts with all active applications at the same time, while the UI thread of a single application handles its life cycle transitions and processes external events. Such architecture can lead to significant delays before the respective UI thread receives a sufficient amount of the CPU time to deliver the next broadcasted Intent. This limitation can be addressed by modifying the interaction with the local UI thread or by introducing a dedicated thread, which is only used to process the pending broadcasts. Following sections present the integration of the proposed extensions into the global and into the local broadcasting subsystems.

7.2.3 Handling of Prioritized Global Broadcasts

Intent objects broadcasted by user applications or system processes are passed to the Activity Manager Service by the Binder driver. A Looper thread deserializes the transmitted payload and executes the method `broadcastIntent()` for further processing, which handles the permission validation, the identification of target receivers and the insertion into the corresponding broadcasting queue (see Figure 7.1). As it was explained above, this section is largely protected by a `synchronized`-block to ensure the data consistency if multiple threads are accessing the queue at the same time. Since only one Looper thread is allowed to enter this critical section, other Looper threads carrying privileged broadcasts may be blocked until the `synchronized`-block is released again. Furthermore, the next processing thread is chosen randomly out of all waiting threads, which may cause unbounded delays during the delivery of high-priority broadcasts. As pointed out by Mercer and Tokuda [85], reducing the size of critical regions may prevent occurrences of priority inversion and improve the soft real-time performance. However, introducing a thread-safe method while reusing the architecture is challenging, because of the high complexity of Android's original implementation and strong interconnection of the related system components. Although a clean slate redesign would improve the system

[2]Except for the `sendBroadcastSync()` method in the `LocalBroadcastManager` class

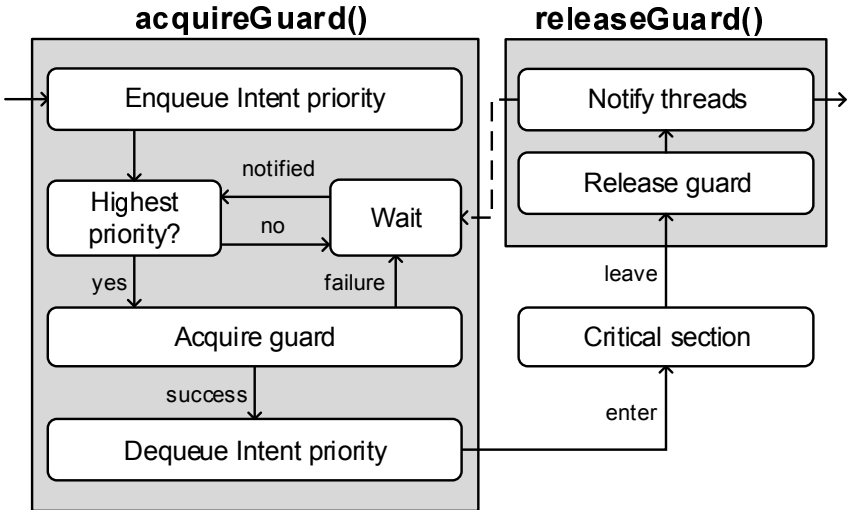


Figure 7.5: Encapsulation of the critical section in the Activity Manager Service [61].

processing performance, it may also threaten the backward compatibility and increase the integration overhead in future releases. For this reason, instead of the dissolving the critical section, this dissertation presents a different approach to ensure the correct activation order for Looper threads waiting to insert the next broadcast record. The introduced modifications guarantee a deterministic access to the `synchronized`-block based on the priority of the carried Intent object b_i. Looper threads with privileged broadcasts receive preferential treatment and allowed to enter the critical section first. This functionality is implemented by a new *Guard* object with two additional methods `acquireGuard()` and `releaseGuard()` in the Activity Manager Service:

- The Guard object is represented by a global variable of type `AtomicInteger`.
- The method `acquireGuard()` initializes the Guard and allows the current thread to proceed, if no broadcasts with higher priority are being currently blocked. It is designed to provide a deterministic selection among all waiting thread, depending on the priority of the carried Intent. A call to this method is inserted immediately before the `synchronized`-block.
- The method `releaseGuard()` releases the Guard and emits a controlled notification that the critical section is empty again. A call to this method is inserted immediately after the `synchronized`-block.

This process is illustrated in Figure 7.5. Each Looper thread, which enters the method `broadcastIntent()` (see Figure 7.2), is registered in an additional waiting queue. The queue is sorted by the priority of the waiting Intent objects, which were inserted by corresponding Looper threads. Only the thread carrying the Intent with the highest priority is allowed to proceed to the Guard object. Otherwise, the thread is put asleep on the waiting queue until the next notification arrives. In the next step, the thread

validates the state of the Guard object. If it already contains a value, another Looper thread currently blocks the critical section. In this case, the current thread is also suspended by calling the `wait()` method. Only if the Guard object is in the released state, the current thread attempts to atomically store its thread ID. This operation may fail, if another Looper thread with the same or higher priority acquires the Guard first. Low-prioritized broadcasts are guaranteed to be blocked by the waiting queue, avoiding further interferences with privileged threads. As soon as a Looper thread has successfully acquired the Guard, its Intent is encapsulated by a new broadcast record and inserted into the broadcast queue. Finally, the thread exits the critical section, resets the value of the Guard object and wakes up the waiting threads.

Section 7.2 has presented the internals of the broadcast processing, including the creation of new `BroadcastRecord` objects. In the original implementation, these objects are stored either in the foreground or in the background queue, which are managed by the UI thread of the AMS. This dissertation introduces a new class `SortedBroadcastQueue` to respect the priority of the enqueued Intents. However, instead of replacing the original queues, the new data structure is used additionally to store broadcasts generated by real-time threads only. The correct queue for the insertion is selected automatically at runtime, based on the priority of the incoming broadcast. Furthermore, the waiting time of the pending records is minimized by creating a separate real-time thread, which is dedicated to the delivery of broadcast records stored in the `SortedBroadcastQueue`. Thus, Intents with real-time priority receive the preferential treatment and are delivered by the dedicated real-time thread, avoiding additional waiting time.

The presented approach does not resolve the bottleneck caused by the critical section, but it creates a strict ordering mechanism to control the processing Looper threads. Methods `acquireGuard()` and `releaseGuard()` enable a more predictable insertion of global broadcasts, as only the thread carrying the highest-priority Intent can proceed to the `synchronized`-block. This minimizes the number of competing threads in front of the critical section and avoids priority inversion scenarios. In comparison to the original implementation, the proposed extension does not cause additional blocking times. Instead, it ensures that the selection of the next thread to enter the critical section is performed in a controlled and deterministic manner.

7.2.4 Handling of Prioritized Local Broadcasts

As presented in Section 7.2.1, broadcasts between the components of the same application can be implemented by relying on the `LocalBroadcastManager` class. Since no data has to be transferred across process boundaries, the broadcasting is more efficient, which also leads to shorter handling delays.

Although the original implementation of the LBM is less complex than the AMS, it still makes use of synchronization on a global monitor object. In fact, the content of the method `sendBroadcast` is almost completely enclosed in a `synchronized`-block. Source code analysis indicates that this may be done for memory saving purposes. For example, this method avoids object allocations, relying on pre-allocated global variables instead. This approach demands the usage of critical sections to ensure the data consistency and

makes concurrent execution impossible. User threads carrying broadcasted Intent objects have to be serialized and blocked while another thread occupies the corresponding `synchronized`-block, creating a major performance bottleneck. In contrast to the AMS, the Local Broadcast Manager is not part of the actual Android framework, but is distributed separately with the Support Library package[3]. It allows the class architecture to be easily modified without jeopardizing the compatibility to previous or future platform releases. In the following, the method `sendBroadcast()` is redesigned by making use of local object allocations, which allows to shorten the `synchronized`-block significantly. This improves both the application responsibility and the performance of the broadcasting system in multithreaded environments. By relying on a real-time capable, concurrent garbage collection presented in Section 5, the proposed modifications do not affect the predictability of the process behavior.

Android's LBM also uses a FIFO list to store the pending broadcast records, being unsuitable for priority-based processing. Similarly to the AMS, this list is replaced by a priority queue, sorted by the priority value of the contained Intents. Since pending broadcasts are delivered by the application's UI thread, its priority is increased to correspond the sending real-time thread. In contrast to the global approach, no additional real-time thread for the queue management is created. Providing a separate real-time thread as part of the LBM may negatively affect the behavior of running applications. Furthermore, execution of the `onReceive()` method should be performed by the UI thread by default, as it might interact with visible elements.

Another drawback arises from the design of the method `sendBroadcastSync()`, which transmits the given Intent object in the context of the current thread. Instead of the immediate delivery of the carried Intent to the matching receiver, it uses the method `sendBroadcast()` method to create and enqueue the record into the broadcast queue. In the next step, the sending thread dequeues all pending Intents and delivers them consecutively, bypassing the UI thread. The original implementation does not distinguish between broadcasts created by `sendBroadcast()` or by `sendBroadcastSync()` and processes all records in a single run. This way a high-priority broadcast can be significantly delayed until other records – which were already stored in the queue, but not handled by the UI thread yet – are delivered first. For this reason, the method `sendBroadcastSync()` cannot guarantee an upper bound for the internal processing time. In order to make it suitable for real-time applications, its implementation is modified to handle the created record immediately, instead of adding it to the priority queue. Thus, the carried Intent object is passed to the target receiver even if the broadcasting queue contains other elements. This approach is preserved for local broadcasts generated by low-priority threads, as the synchronous delivery can only be performed during the own computation time of the sender. The time slot is reliably bounded by the system for both regular and real-time threads, depending on the corresponding priority. This approach minimizes the interferences during the handling of local broadcasts and transfers the control over the synchronous data delivery to the Linux scheduler.

[3]Official documentation for the Android Support Library: `https://developer.android.com/topic/libraries/support-library/index.html`

7.3 Experiments

The modifications presented in previous sections are evaluated separately in multiple tests. The first part analyzes the delays for global broadcasts caused during their processing in the Activity Manager Service. This includes measurements of the default prioritization using the `FLAG_RECEIVER_FOREGROUND` and of the system behavior in different load situations. Furthermore, the correctness of the proposed approach for handling explicitly prioritized broadcasts is validated in detail. Finally, the new architecture of the Local Broadcast Manager class is evaluated and discussed in the second part.

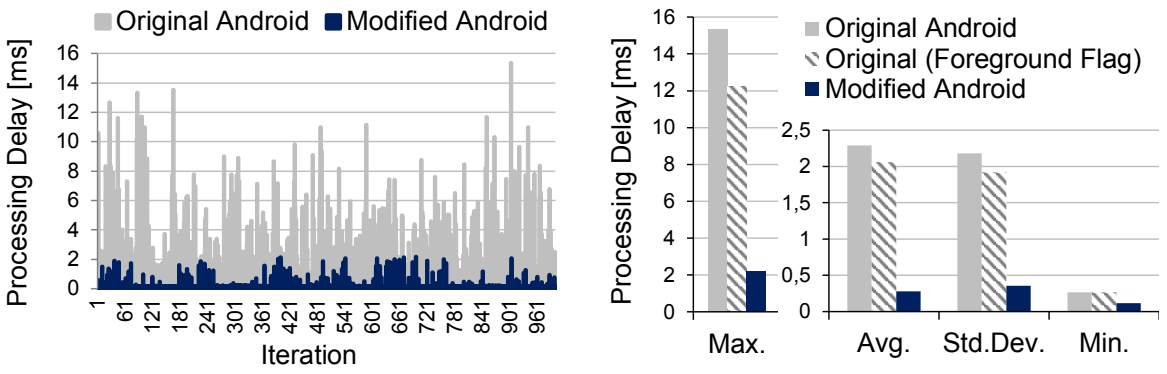
Testing was performed on the Google Nexus 10 (codename `manta`) tablet computer, which is built with a Samsung Exynos 5250 (2x 1.7 GHz Cortex-A15) SoC and 2 GB of RAM. This device runs the Linux kernel `android-manta-3.4.5-rt15` as part of the modified Android 4.2.2. Test cases were created in conventional Android applications with introduced extensions as explained in previous sections. The method `onReceive()` of the implemented Broadcast Receiver only records the timestamp of data arrival t_{arr} using the high-resolution timer, which is enabled by `PREEMPT_RT`.

7.3.1 Performance Evaluation of Global Broadcasts

The evaluation begins by measuring the processing time T_{PROC}, which is spent by a globally broadcasted Intent object inside the AMS. It specifies the total delay between the invocation of the method `broadcastIntent()` in the Activity Manager Service at time t_{inv} and the actual arrival of the broadcasted Intent in the corresponding receiver at time t_{arr} with $T_{PROC} = t_{arr} - t_{inv}$.

Two applications are used to perform the first test. Application A spawns one real-time thread with priority of 80 and $k_{nrt} = 10$ regular threads to transfer data to the application B using global parallel broadcasts. Each of the threads sends a new Intent object every 100 ms, performing 1000 iterations in total. No explicit priority values are used in this test, such that broadcasts are prioritized implicitly in the modified system. Processing delays T_{PROC} are calculated based on Intents broadcasted by the real-time thread only. A comparison between the original implementation and the extended platform is presented in Figure 7.6(a).

The values recorded in the original implementation seem to be randomly distributed in the range between 264 µs and 15.3 ms. Although more than 80% of all broadcasts were delivered in less than 3 ms, 55 outliers higher than 6 ms have been detected. This behavior indicates the lack of predictability in the original design of the Activity Manager Service. Non-deterministic blocking caused by the critical section and the FIFO processing enforced by the `BroadcastQueue` cannot guarantee a preferential treatment for broadcasts generated by the privileged thread. The effectiveness of the proposed modifications is shown by the behavior of the extended system. Introducing a sorted data structure in combination with a dedicated processing thread leads to a notable improvement of the time an Intent object spends in the AMS. The worst-case delay is reduced to only 2.2 ms, whereas 95% of all high-priority Intents were delivered to the corresponding receiver in less than one millisecond.



(a) Comparison of the processing delay [61].

(b) Evaluation of the processing delay [61].

Figure 7.6: Processing delays for global real-time broadcasts with $k_{nrt} = 10$.

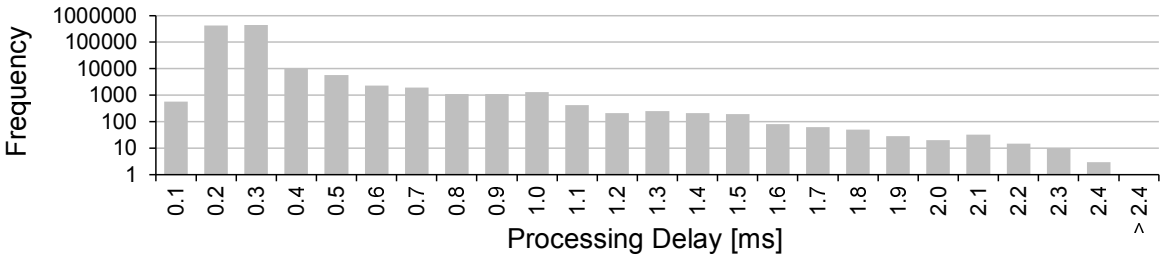


Figure 7.7: Distribution of T_{PROC} for real-time broadcasts during a long-term test [61].

7.3.2 Impact of the Foreground Priority Flag

The next test compares the behavior of the extended implementation to the explicit prioritization of global Intents using the `FLAG_RECEIVER_FOREGROUND`. In addition to the setup presented in the previous section, the real-time thread sets the foreground flag for its own Intents. Since this test was only performed with unmodified Android, the result comparability is increased by summarizing the recorded measurements with data from the previous test as shown in Figure 7.6(b).

Android's native prioritization is able to reduce the worst-case processing delay by a small margin. However, even with activated foreground flag the standard deviation stays almost as high as the average processing time, indicating major delay dispersion and the poor system's predictability. The original implementation is concluded to be unable to provide any timing guarantees for the delivery of global broadcasts.

Introduced changes allow a significant improvement of the system behavior. The average value is reduced to only 278.5 µs. Furthermore, a long-term test with $1,000,000$ broadcasts generated by each of the running threads[4] has shown an upper bound for the processing time of about 2.4 ms. The full delay distribution of the long-term test is depicted in Figure 7.7.

[4]The long-term test had a total running time of about 28 hours.

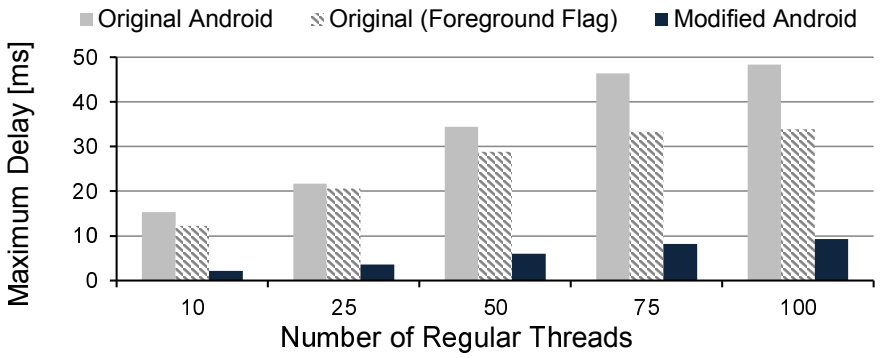


Figure 7.8: Worst-case processing delay for real-time Intents with rising k_{nrt} [61].

7.3.3 Handling of Global Broadcasts under Load

The processing time T_{PROC} was shown reasonably bounded, when global broadcasts generated by a real-time thread were handled with low system load. It can be assumed that $k_{nrt} = 10$ regular threads – which concurrently generate additional broadcasts at the same rate as the real-time thread – create an adequate amount of throughput to simulate a typical usage scenario. Nevertheless, this test evaluates the behavior of the new Activity Manager Service in high-load situations. The test is conducted with the same setup as used in previous sections, but with a varying number of regular threads $k_{nrt} \in \{25, 50, 75, 100\}$ creating 1,000 broadcasts each. The value T_{PROC} is calculated for Intents generated by the real-time thread in three different scenarios: using the original Android platform, using the original platform with activated `FLAG_RECEIVER_FOREGROUND` and using the extended AMS with implicit prioritization. The worst-case delay of each run for different k_{nrt} values is presented in Figure 7.8.

The diagram shows a rising worst-case delay for higher number of active senders in all configurations. The highest delay was observed in the original implementation, where the longest Intent processing took 48.3 ms for $k_{nrt} = 100$. Native prioritization using the foreground flag did not prevent the load from negatively influencing broadcasts generated by the real-time thread. Although the measured worst-case value was slightly lower than when the foreground flag was not used, this value increased almost threefold from $k_{nrt} = 10$ to $k_{nrt} = 100$, reaching about 33.8 ms in the worst case. The extended implementation shows significantly lower delays with the maximum of about 9.3 ms for $k_{nrt} = 100$ regular threads. However, in comparison to the unmodified system, the maximum delay shows a faster growth. It can be assumed, that the new implementation cannot guarantee a fixed upper bound for the worst-case delay, if the number of senders is not limited. This behavior may be caused by the critical section, which creates a significant performance bottleneck. Nevertheless, implemented methods for acquiring and releasing the Guard object ensure that the access to the `synchronized`-block is granted based on the priority of the waiting threads, reducing the overall overhead by a factor of 3 to 4. Since long critical sections neutralize the benefit of a multi-threaded platform, the scalability of the system may be further improved by reducing their size to the minimum. This assumption is evaluated for local broadcasts in Section 7.3.5.

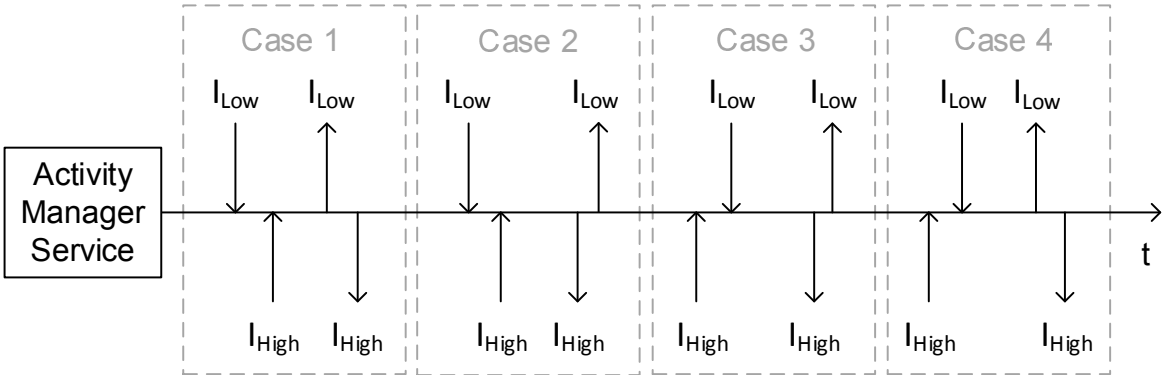


Figure 7.9: Case 4 violates the processing order of prioritized Intents [61].

7.3.4 Evaluation of the Explicit Prioritization

Tests presented in previous sections have mainly focused on the evaluation of processing timings and the overall broadcasting performance. Another obstacle to the reliable and predictable Intent handling in the original implementation was created by the FIFO queue. Sorting the pending broadcast records based on their insertion time cannot guarantee the correct order of delivery for prioritized Intents. Broadcasts generated by real-time threads may be blocked until the processing of already enqueued records – possibly generated by threads with lower priority – is finished. Section 7.2.3 has presented the modification approach to solve such conflicts by introducing an ordered queue, to sort pending broadcasts based on their priorities, and the Guard object, to ensure the correct thread activation order.

This test evaluates the effectiveness of the introduced changes using $k_{rt} = 10$ real-time threads without additional system load. Each thread generates a new broadcast every 5 ms with an explicit broadcasting priority randomly chosen between $b_{min} = 0$ and $b_{max} = 99$. In comparison to previous tests, the sending period was reduced in order to facilitate the occurrence of conflict situations. As the presented approach automatically limits the effective broadcasting priority (see Section 7.2.2), the scheduling priority of the sending threads is set to $s_{max} = 99$ in order to make use of the full possible broadcast priority range. Figure 7.9 illustrates the possible combinations for handling of two Intents with different priorities in the Activity Manager Service.

The first case shows a low-priority Intent being delivered to its receiver earlier than the high-priority Intent. This behavior might be correct, if the transmitting Looper thread has already entered the critical section and inserts its record in the broadcast queue first. If the queue is currently empty, the dedicated real-time thread handles the low-priority Intent immediately. Hence, Case 1 is assumed valid under given conditions. Cases 2 and 3 are valid too, since I_{High} is delivered to its receiver earlier than I_{Low} regardless of the insertion order (as enabled by the introduced priority queue). The last case depicts a scenario where the high-priority Intent is blocked by a low-priority Intent, violating the correct processing order. This behavior indicates a failure in the prioritization mechanism, as I_{Low} receives preferential treatment over I_{High}. The following equation describes

	Count	Failures
Original Android	6344	2271
Modified Android	7485	0

Table 7.1: Analysis of the broadcast handling order [63].

the correct system behavior for two Intents H and L based on their priority values b, the invocation times t_{inv} and arrival times t_{arr}:

$$b^H > b^L \wedge t^H_{inv} < t^L_{inv} \Rightarrow t^H_{arr} < t^L_{arr}$$

Current test is performed to validate this property in the original and in the modified Android platforms for all generated broadcasts. It records the total number of situations in which I_{Low} was invoked in the AMS later than I_{High} (Count) and how many of these I_{Low} were delivered to the receiver before I_{High} (Failures). Table 7.1 presents the measured results for both systems.

As the broadcasting priority is generated randomly, the total number of conflicting situations is different for both tests. During the test in the unmodified Android, 6344 possible race conditions were observed, but despite of the FIFO processing, only 2271 of them resulted in the low-priority broadcast to be delivered first. This is due to fact that the Looper thread carrying I_{High} has a higher priority (see Section 6) and more time to enter the critical section and insert a new broadcast record. However, as the evaluation shows, this mechanism may fail if another thread currently executes the critical section. This creates a non-deterministic scenario, where a Looper thread with I_{Low} may enter the critical section earlier than another Looper thread carrying I_{High}. Although a higher number of race conditions was detected during the test of the extended implementation, all Intents were handled in the correct order. This indicates that the integrated approach effectively prevents prioritization errors during the processing of global parallel broadcasts.

7.3.5 Performance Evaluation of Local Broadcasts

The final test evaluates the performance of the intraprocess communication using the extended Local Broadcast Manager as described in Section 7.2.4. The setup is chosen similar to the first test from Section 7.3.1, where a single real-time thread and $k_{nrt} = 10$ regular threads generate 1000 implicitly prioritized Intents each. Since no data has to be passed across process boundaries, the timestamp t_{inv} for the calculation of T_{PROC} is taken before the Intent is passed to the LBM, such that the processing delay covers the full handling period of a broadcast. The evaluation is performed separately for the asynchronous Intent delivery by the UI thread using the method `sendBroadcast()`, as well as the synchronous delivery in the context of the calling thread using the method `sendBroadcastSync()`. Figure 7.10 summarizes the recorded results for the original and for the extended Android.

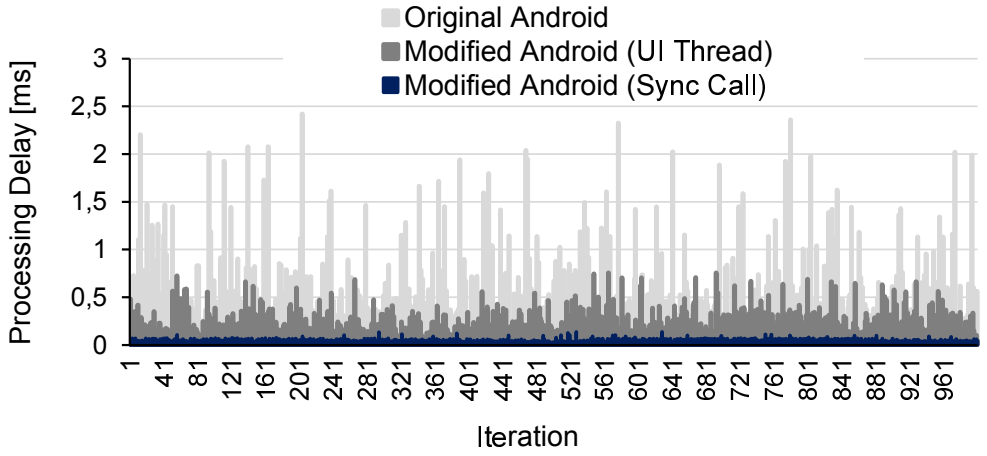


Figure 7.10: Processing delay for local broadcasts with $k_{nrt} = 10$ regular threads [61].

In comparison to the global broadcast processing using the AMS, local broadcasting is confirmed to be significantly more efficient (compare Figure 7.6). The worst-case processing delay under the same conditions is reduced from 15.3 ms to only 2.4 ms, while over 90% of all Intents were delivered in less than 1 ms. However, recorded delays are distributed randomly, making a reasonable prediction impossible. Asynchronous delivery using the extended LBM shows a bounded overhead with the maximum delay of 765 µs. For synchronous broadcasting using the method `sendBroadcastSync()`, this time is further reduced to only 139 µs without significant outliers. Although invoking the receiver in the context of the sending thread shows the most reliable behavior, this approach can be used only if the receiver does not directly interact with UI elements.

Modifications for the Local Broadcast Manager introduced in Section 7.2.4 aimed at achieving higher performance under CPU load due to the considerably shortened `synchronized`-block. This property is evaluated in the following by repeating the last test with $k_{nrt} \in \{25, 50, 75, 100\}$ regular threads. Each scenario is tested with the original implementation (O), the modified implementation using asynchronous handling by the UI thread (`Mu`) and the modified implementation using the synchronous delivery by the sending thread (`Ms`). For better readability, Figure 7.11 presents the boxplots for all recorded values using logarithmic scaling.

The responsiveness of the original implementation strongly depends on the number of active threads. While the median value is decreasing in situations with higher load, the worst-case processing delay rises from 2.4 ms to 5.1 ms for $k_{nrt} = 100$. A similar observation can be made for the modified Intend delivery in the context of the sending thread. Although the maximum values in each run are significantly smaller than in the corresponding test of the original implementation, the worst-case delay increases from 139 µs for $k_{nrt} = 10$ to 249 µs for $k_{nrt} = 100$. The increase of the processing delay for `Mu` is marginal. While the maximum value rises to 972 µs for 50 regular threads, it stays below 1 ms regardless of the additional load, offering a reasonable upper bound for local broadcasts. Hence, this method is concluded to provide the optimal performance for Intent delivery while enabling predictable broadcasting in real-time applications.

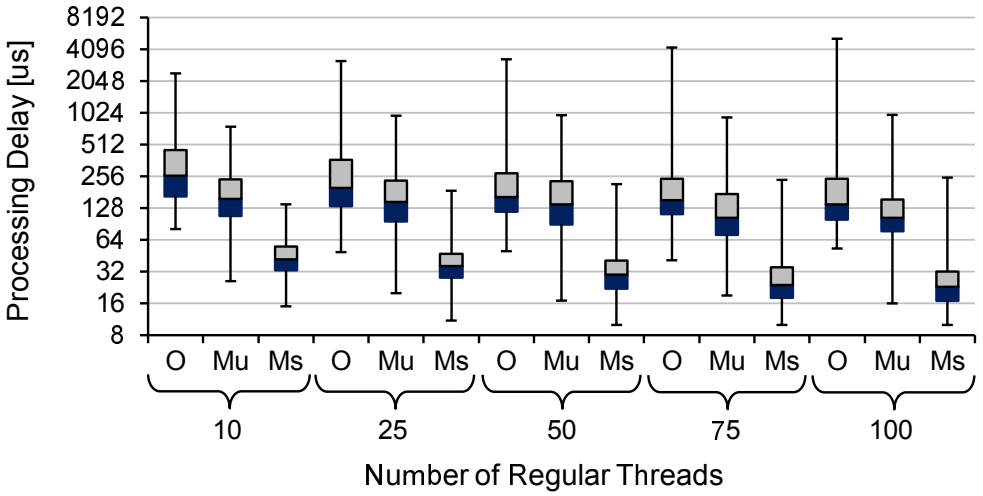


Figure 7.11: Statistics for local broadcast processing [61].

7.4 Summary and Discussion

Intent broadcasting constitutes one of the most important mechanisms for data exchange in Android, allowing to take advantage of Android's rich application framework. The original implementation of AMS and LBM was evaluated and extended in order to enable priority-based processing. Instead of FIFO-based Intent delivery based on the insertion time, Intents broadcasted from real-time applications are managed automatically by a separate data structure, being sorted by their explicit or implicit priority. Furthermore, processing delays were significantly reduced by minimizing the responsible critical sections and introducing privileged delivery threads. Evaluation results recorded in multiple tests highlight advantages of the modified system components for both global and local broadcasts.

The original implementation was shown to behave unreliably and to cause unbounded delays during Intent processing. Global parallel broadcasts in standard Android cannot be delivered within a predictable time regardless of the `FLAG_RECEIVER_FOREGROUND` priority flag. Original components cause worst-case transmission delays of up to 15.3 ms in the default configuration and up to 12.2 ms for foreground broadcasts with $k_{nrt} = 10$ regular threads. During the test of the modified system over the course of 28 hours all Intents broadcasted by a real-time thread were delivered in less than 2.5 ms. These results demonstrate the effectiveness and scalability of the presented approach. Priority-based ordering and the introduced Guard object guarantee the deterministic thread management in front of the critical section and thus the correct processing of messages transmitted by real-time applications.

Positive effects gained from resolving critical sections are illustrated by evaluation results of the new local broadcasting manager. Instead of the worst-case delay of about 2.4 ms observed during the asynchronous Intent delivery in the original implementation, the new approach has shown bounded transmission delays below 1 ms. As explained during the architecture analysis, synchronous Intent broadcasting using standard LBM

class may result in unpredictable blocking times since previously inserted objects have to be dispatched first. The modified concept allows an immediate delivery, which makes the method `sendBroadcastSync()` to be a better way of message passing between components of the same real-time application. Evaluation results show that the modified LBM class is able to dispatch local broadcast and process them in context of the sending thread in less 250 µs for $k_{nrt} = 100$ regular threads. The only limitation of this approach is the missing support for the direct interaction with user interface. As presented in Section 2.2.2, only the UI thread may access or modify the state of UI elements. Thus, broadcasted actions that affect the UI cannot be handled by synchronous processing, since they require additional synchronization mechanisms with the UI thread.

While the proposed approach for global broadcasts ensures a deterministic Intent processing inside the system core, it has no influence on the UI thread of the recipient application, which passes the data to the corresponding Broadcast Receiver. Although high-prioritized Intents are delivered to their target applications first, the responsible UI thread may be currently blocked either by the system or even by other threads inside the same application. This would lead to the delayed processing on the application level, which might have undesired consequences in context of real-time computing. In future, an extended Intent architecture with a possibility to temporarily adjust the priority of the receiving thread has to be evaluated. However, such approach requires detailed analysis and careful design, since enforced modifications of thread priorities may affect the internal application logic if the process also contains active real-time threads. New API for the specification of allowed priority ranges for the UI thread of each application could provide a solution to avoid negative effects. Another improvement of the data processing on the application side may be achieved by incorporating the modified `Looper` and `Handler` classes [133], as proposed by Yan et al. Finally, importance of sticky and ordered Intents has to be evaluated in context of real-time applications. In case their processing delays can jeopardize the deterministic execution in real-time scenarios, responsible system components have to be analyzed and redesigned too.

8 Evaluation

Current survey results show that the number of annually sold mobile devices is rising[1], while the sales of desktop PCs continuously decrease. Although desktop PCs still play a key role in consumer electronics, the importance of mobile devices is growing rapidly[2]. As an increasing number of users works with mobile devices on a daily basis, intuitive handling becomes the defining factor for new electronic products. It can be assumed that high-quality user experience will eventually become an important non-functional requirement for the design of interfaces for industrial machinery. Hence, a reliable solution based on a modern OS has the potential to change the current market situation and to provide an intuitive and extensible platform, encouraging the design of user-friendly products. Today, the Android platform holds the largest market share of over 81%. This popularity of Android arises from the satisfaction of the majority of developers' and customers' expectations, as it was presented by the survey results [122]. Additionally, its source code is available under the *Apache 2.0* open-source license, which facilitates the development of innovative applications, commercial products and scientific research. Since the platform is developed by a consortium of mobile device manufacturers, Android benefits from a wide range of supported hardware from different vendors like Samsung, HTC and Motorola. Furthermore, Google has recently presented the *AndroidThings*[3] – an Android-based platform for the *Internet of Things* – designed as a customizable solution for new embedded products. This is a crucial step to establish Android in the industrial market and to provide high-level applications with access to advanced periphery like ADC and I^2C, which increases the competitiveness and attractiveness of the off-the-shelf embedded boards like Raspberry Pi.

As shown in Chapter 3, Android is already used in more than 20% of new embedded projects today. As it was originally developed for needs of smartphones and tablets, Android's architecture was deliberately designed to provide the best possible user experience even on devices with limited resources. Thus, it includes mechanisms to influence application priorities based on their current visibility and to reduce energy consumption by terminating long-running background processes. While these are beneficial properties for consumer devices, such behavior is not compatible with predictability and determinism required in industrial time-critical environments. This dissertation aimed at combining the advantages of a reliable execution environment for industrial applications and a flexible modern mobile platform running on general-purpose hardware. Different methods to improve Android's behavior towards bounded processing latencies

[1] Worldwide Smartphones Sales in 2016: `http://www.gartner.com/newsroom/id/3609817`

[2] Usage of mobile vs. desktop devices: `http://www.telegraph.co.uk/technology/2016/11/01/mobile-web-usage-overtakes-desktop-for-first-time/`

[3] Peripheral I/O in AndroidThings: `https://developer.android.com/things/sdk/pio/index.html`

were developed over the course of this work. The experiment results indicate that real-time applications can be effectively shielded from unwanted influence of system activities by modifying platform components responsible for process scheduling, CPU frequency control, memory management and interprocess communication.

This chapter provides a qualitative assessment of the introduced changes and evaluates different functional and non-functional criteria. The challenge of conflicting requirements between the user-oriented design and the requirements for embedded systems is discussed in Section 8.1 using the example of dynamic CPU frequency scaling. Additionally, this section presents an overhead estimation and impact analysis for each of the integrated extensions. Finally, Section 8.2 concludes this chapter with a brief summary.

8.1 Impact Analysis of the Introduced Extensions

From the consumer's point of view, Android's mechanisms for strict energy saving play a crucial role in an application's life cycle. Historically, it is one of the most prominent features, which is adjusted and extended in almost every Android release. The dynamic CPU frequency scaling was the first approach to reduce the energy drain if no application is currently active. In the next Android version, the device was put into sleep mode as soon as its screen was turned off. *Wakelocks* were designed for allowing background activity and preventing the screen timeout in exceptional cases. However, third-party applications often forget to release a Wakelock, which forced Google to introduce a new *Doze* mode in Android 6. It is activated automatically when the device remains stationary for a given period of time, suppressing active Wakelocks and disabling all networking activities[4]. In Android 7, the Doze mode was extended and it can be enabled by the system even if the device is not kept stationary.

While this feature is useful to reduce the device's energy consumption, it contradicts the main principle of real-time computing. In the industrial domain, the control logic should not be affected by user interaction or by other applications running concurrently. Since the CPU frequency determines the execution time of CPU instructions, dynamic changes influence the behavior of all active processes. As demonstrated in Chapter 4, lower values of the CPU frequency cause applications to require more time to finish current calculations, possibly leading to deadline misses. To maintain the energy consumption benefits of frequency scaling, this thesis has proposed a method to disable the dynamic CPU frequency adjustments during the execution of real-time applications.

Advantages of mobile platforms become apparent when considering modern non-functional requirements like intuitive user interfaces and a simplified development cycle. However, as the example of enforced power saving in Android shows, improving the platform's predictability inevitably requires substantial modifications on a functional level. The following sections analyze the introduced extensions and their role in making consumer-class electronic devices eligible for use in the industrial domain as summarized in Table 8.1. In this context, each extension is evaluated for its contribution to reaching

[4]Official documentation about the new Doze mode in Android: `https://developer.android.com/training/monitoring-device-state/doze-standby.html`

the primary goal – enabling real-time support in Android. Besides the implementation and integration overhead, the discussion analyzes non-functional aspects like compatibility to future Android releases and providing access to Android's rich application framework (e.g. use of standard platform components and third-party applications in real-time applications). Finally, relevance of the respective extension outside of the real-time domain is estimated, indicating whether it might also be useful for conventional Android applications running on consumer devices.

Application of the PREEMPT_RT Patch

The PREEMPT_RT patch provides all relevant features and does not require implementation of additional techniques. As shown by different scientific studies and because of its wide acceptance in the industry, this patch provides adequate results for soft real-time applications [5, 87]. However, the integration of this patch into Android generates a notable overhead, which strongly depends on the version of the device-specific Linux kernel and vendor extensions. Although patching succeeded in all examined devices, proprietary drivers and version inconsistencies may cause severe integration problems, as mentioned by Yan [131] and Zores [135]. Furthermore, this approach increases the risk of incompatibilities to future releases. Since the PREEMPT_RT patch only introduces real-time support on the lowest architectural level, it is detached from the Android framework. This extension is assumed not suitable for the consumer market, as deterministic scheduling comes at cost of performance, flexibility and fairness for user applications.

Mechanism for the CPU Frequency Locking

The frequency lock is implemented as a wrapper for Linux CPU frequency governors. It encapsulates the activation logic and provides a high-level application interface. This allows to keep the integration overhead at minimal level and to migrate the presented solution easily to other governor types or newer Linux versions. As shown in Chapter 4, this extension is important to reduce scheduling latencies caused by dynamic CPU frequency scaling and achieve deterministic process behavior. Similarly to the PREEMPT_RT patch, frequency locking only extends the basic real-time support of the Linux kernel. It does not allow real-time applications to make use of the Android application framework and may be disadvantageous for consumer devices, as locking CPU frequency can negatively impact the battery life and lead to increased heat development.

Non-blocking Garbage Collection

Among all proposed extensions, the real-time capable automatic memory management has the highest development complexity. All required algorithms[5] had to be implemented from scratch by taking into account the specifics of Android. Furthermore, the implemented garbage collector has to operate on Android's native data structures, which significantly increases the integration overhead. This also leads to a poor upgradability

[5]Except the local root scanning, which is part of the original Android implementation.

	1	2	3	4	5	6
	Application of `PREEMPT_RT`	CPU Frequency Lock	Non-blocking Garbage Collection	Direct use of `OOM_ADJ`	Extended Binder Driver	Prioritized Broadcasting
Overhead for implementation and integration into Android	medium	low	high	low	high	medium
Importance for real-time support in Android	high	high	high	high	medium	medium
Estimated compatibility to future Android releases	medium	high	low	high	low	medium
Importance for interaction with the Android framework	low	low	low	low	high	high
Relevance for consumer market	low	low	medium	low	high	medium

Table 8.1: Impact analysis for all introduced platform extensions.

of the new GC, as any change in future releases of Android may break the compatibility. At the same time, non-blocking garbage collector is one of the most important components to achieve real-time support on higher architectural levels. It ensures the interruption-free execution of real-time applications implemented in Java programming language. To avoid suspensions in out-of-memory situations, this functionality might also be useful in regular applications on consumer devices. It is, however, far less important than in time-critical scenarios, as short blocking times in regular applications do not cause any damage and usually even lay beyond human perception.

Control of Memory Adjustment Values

Implementation and integration overhead for controlling memory adjustment values is low, as it only manipulates Android's mechanism for process management. Similar to the wrapper for the CPU frequency governors, this extension is created on top of the internal API and provides high portability and updatability to future Android releases. Although this approach does not directly improve scheduling latencies or process responsiveness, it offers a new interface to influence Android's internal prioritization of running applications. Hence, it is important to guarantee the predictability of the system and to protect real-time background processes from being killed during automatic memory usage optimization. This extension does not contribute to the interaction between real-time application and other platform components. Furthermore, it is not reasonable for the consumer market, as bypassing Android's control mechanisms creates a substantial risk of abuse.

Extended Binder Driver

Introducing priority inheritance for Binder-based remote procedure calls involves considerable development effort. The overhead for implementation and integration is high, as multiple components including the Linux kernel and the application framework must be modified. It also requires additional data exchange across different architectural layers, which increases the complexity and reduces the expected compatibility to future releases. Although high-level applications can meet soft real-time requirements by using the extensions presented above, they have no reliable communication channel to invoke the official Android API. Hence, this extension plays a key role, as it bounds the RPC overhead and enables the interaction with third-party components without jeopardizing the process behavior. This way external functionality can be incorporated in real-time applications in a predictable manner. As priority inheritance is generally favorable for all kinds of applications, it might also be useful for the consumer market.

Prioritized Intent Broadcasting

Implementation of the broadcast prioritization is done on high level, encapsulating the new functionality in just a few additional classes. The original platform interface is preserved by relying on existing `Extra` fields and the `BroadcastQueue` class. Although

the proposed extensions can be easily integrated into Android, their effectiveness is significantly decreased by critical sections, which protect the Intent handling in the original implementation. While a redesigned class for local broadcasting can be provided in a separate package, deterministic processing of global Intents requires major platform adaptations. Nevertheless, the presented approach provides a standardized way for predictable data exchange between real-time applications and external components. Implicit Intent prioritization might be also reasonable for conventional applications on consumer devices, as the original processing is based on the FIFO strategy only. However, such functionality must be designed carefully, as low-priority messages may be blocked for an indefinite period of time in high-load scenarios.

8.2 Summary and Discussion

In accordance to the objectives of this dissertation, the primary goal was reached by extending the main platform components of Android, instead of redesigning its architecture. This approach supports the fulfillment of the secondary goal, as real-time applications can preserve the original application model and make use of the new API to leverage the integrated functionality in the standard Android manner. The presented modifications cover the most important parts of the internal infrastructure:

- Extensions 1, 2 and 4 establish a reliable execution environment for soft real-time support at the lowest architectural level. Real-time Android applications executed in dedicated Linux processes are precisely scheduled by the patched kernel based on their priorities. Furthermore, they are protected from dynamic changes of the CPU frequency or unexpected terminations by the memory optimization mechanism.

- Extension 3 minimizes interferences between the application logic and the automatic garbage collector. This change affects Android Runtime and allows usage of Java programming language for the development of real-time applications.

- Extensions 5 and 6 provide a possibility to link components from different applications and reuse existing functionality in a predictable manner. They allow applications on the highest architectural level to benefit from a predictable exchange of structured data across process borders.

Although the popularity of Android for embedded projects is rising, using it in time-critical environments requires substantial platform extensions. Evaluation of the presented modifications indicates that augmenting Android for usage in the industrial domain is generally possible, albeit with considerable development overhead. Furthermore, as the discussion of extensions 1, 2 and 4 shows, such modifications may conflict with standard behavior of consumer devices, where important non-functional requirements include the responsiveness of the foreground application and extended battery life. Since the original platform is optimized to provide the best possible user experience, its architecture does not consider typical use cases for industrial operating systems like static

priorities or splitting the application logic from the UI handling. While these are known techniques to guarantee a predictable process behavior regardless of user input and visualization overhead, they often lead to a limited flexibility and extensibility of the resulting platform.

Android is based on the Linux kernel, which is broadly used in the industrial domain today. The extensions proposed by this dissertation unveil the potential of the platform in terms of deterministic behavior and allow Android-based embedded products to serve industrial purposes beyond simple visualization and user interaction. Preserving the compatibility to external components and third-party applications facilitates the integration of modern technologies, reducing the product complexity and resulting in faster software development by relying on a wide range of available extensions, support tools and third-party libraries.

9 Conclusion

The example of industrial Linux proves that deterministic and predictable application behavior is possible using the combination of high-level operating systems and general-purpose hardware. Currently, Android is continuously gaining importance not only in consumer electronics, but – as indicated by the current trends in embedded products – also in the industrial application domain (see Section 3). While it is mainly used for HMI purposes today, included Linux kernel raises the question of Android's suitability for reliable monitoring and control. A more predictable process and memory management help to expand Android's field of application to time-critical domains. Ongoing interest in augmenting Android with real-time support is reflected by the rising number of related scientific publications [74, 79, 86, 91, 104, 131, 133]. Prior to this work, the research was mainly focused on the evaluation of the real-time Linux kernel and application-specific extensions, creating separated execution environments for regular and real-time applications. Only little attention was paid to the extension of Android's native high-level components and their interaction at runtime. This dissertation provides a holistic view on the Android platform by identifying and modifying components required for the predictable and deterministic execution of real-time Android applications. Furthermore, the implemented approach preserves the original API, allowing real-time applications to interact with system components in a controlled manner.

9.1 Summary

Each chapter of this work presented an analysis and extension of original components related to one specific aspect of the platform's behavior. For achieving the optimal process behavior and preserving application design flexibility on different architectural levels, the final embedded product has to incorporate all of the introduced extensions:

1. A fully preemptible Linux kernel with worst-case scheduling latencies of about 63 µs. This corresponds with the optimal behavior of a patched Linux kernel as evaluated in the scientific literature (compare Section 4.1) and seems suitable for a wide range of time-critical use cases.

2. A concurrent memory management with an iterative, non-blocking garbage collection based on reference counting. It addresses the undesired non-deterministic suspensions and unfolds the full power of high-level programming languages and frameworks. Additional protection prevents real-time processes in Android from being killed during a long-term execution in the background.

3. Enhanced mechanisms for data exchange across the application's process borders. Since remote procedure calls and Intent broadcasting represent the two most important approaches for interprocess communication in Android, extended methods reduce corresponding blocking times and guarantee predictable upper bounds for the communication delay.

9.2 Future Work

Providing a real-time capable platform based on Android facilitates the development of modern user interfaces and reduces the complexity of software and hardware products. The approach proposed in this dissertation creates the minimal foundation for soft real-time support in Android. Possible improvements related to each of the introduced features were already briefly presented in the corresponding chapters. However, further research on other topics is required in order to successfully establish the Android platform running on general-purpose hardware in the industrial environment and to use it as a substitute for the individual application-specific solutions.

Besides of security audits and the identification of privacy-related issues, embedded products typically have to provide APIs for low-level hardware control. In addition to GPIO, such interfaces often include access to peripherals like PWM, ADC or UART. They are commonly used for actuator control and raw data exchange with peripheral sensors. Recently, Google has introduced peripheral I/O extensions designed for Android. While this is a reasonable way to exploit the full potential of general-purpose embedded boards like ODROID and Raspberry Pi, the internals of these extensions have to be evaluated in the context of real-time applications. For example, predictable data retrieval was partially covered by Yan et al. [132], who presented a promising approach for real-time capable access to sensor data at runtime.

Another issue arises from the limited support for industrial communication. Android devices provide interfaces for consumer-grade protocols like Bluetooth, USB and WIFI, which are designed for higher throughput, while industrial setups typically rely on more robust protocols like Modbus, CAN and PROFINET with focus on predictability. Several projects were identified to evaluate the predictability of the available interfaces [45, 88] or to integrate industrial protocols into the Android platform [4, 105, 120] in the past. Such approaches are important for the seamless integration of off-the-shelf embedded devices into the industrial environment.

Bibliography

[1] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole. A Measurement-Based Analysis of the Real-Time Performance of Linux. In *Proceedings of the 8th Real-Time and Embedded Technology and Applications Symposium*, RTAS '02, pages 133–142, 2002.

[2] Luca Abeni and Giuseppe Lipari. Implementing Resource Reservations in Linux. In *Proceedings of the 4th Real-time Linux Workshop*, 2002.

[3] Frank Ableson, Charlie Collins, and Robi Sen. *Unlocking Android: A Developer's Guide*. Manning, Greenwich, Connecticut, USA, 2009.

[4] Maximilian Adducchio. *Portierung eines Android-Betriebssytems auf ein Embedded-Multi-Prozessor System sowie Anbindung an die industriellen Schnittstellen CAN und Ethernet*. Master's thesis, Hochschule Esslingen - University of Applied Sciences, Esslingen, Germany, 2014.

[5] Jan Altenberg. Using the Realtime Preemption Patch on ARM CPUs. In *Proceedings of the 11th Real-Time Linux Workshop*, pages 229–236, 2009.

[6] Björn Andersson, Konstantinos Bletsas, and Sanjoy Baruah. Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, RTSS '08, pages 385–394, 2008.

[7] Ashraf Armoush, Dominik Franke, Igor Kalkov, and Stefan Kowalewski. An Approach for Using Mobile Devices in Industrial Safety-Critical Embedded Systems. In *Proceedings of the 5th International Conference on Mobile Computing, Applications, and Services*, MobiCASE '13, pages 294–297, 2013.

[8] Siro Arthur, Carsten Emde, and Nicholas McGuire. Assessment of the realtime preemption patches (RT-Preempt) and their impact on the general purpose performance of the system. In *Proceedings of the 9th Real-Time Linux Workshop*, 2007.

[9] Stefan Assmann. Einführung in Real-Time Linux (Preempt-RT), 2010. Slides. `https://chemnitzer.linux-tage.de/2010/vortraege/shortpaper/517-slides.pdf`. Accessed on 10.03.2017.

[10] Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-Spend: Democratic Scheduling for Real-time Garbage Collection. In *Proceedings*

of the 8th International Conference on Embedded Software, EMSOFT '08, pages 245–254, 2008.

[11] Michael Backes, Sven Bugiel, and Sebastian Gerling. Scippa: System-Centric IPC Provenance on Android. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC'14, 2014.

[12] David F. Bacon and Vadakkedathu T. Rajan. Concurrent Cycle Collection in Reference Counted Systems. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 207–235, 2001.

[13] Stephen M. Blackburn and Kathryn S. McKinley. Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 22–32, 2008.

[14] Raja Bose, Jorg Brakensiek, Keun-Young Park, and Jonathan Lester. Morphing Smartphones into Automotive Application Platforms. *Computer*, 44(5):53–61, 2011.

[15] Lisa Carnahan and Markus Ruark. Requirements For Real-Time Extensions For The Java™ Platform, 1999. Special publication by National Institute of Standards and Technology. `http://www.nist.gov/rt-java`.

[16] Arben Çela, Adam Hrazdira, Abdellatif Reama, Rédha Hamouche, Silviu-Iulian Niculescu, Hugues Mounier, René Natowicz, and Rémy Kocik. Real time energy management algorithm for hybrid electric vehicle. In *Proceedings of the 14th International IEEE Conference on Intelligent Transportation Systems*, ITSC '14, pages 335–340, 2011.

[17] Che-Wei Chang, Jian-Jia Chen, Waqaas Munawar, Tei-Wei Kuo, and Heiko Falk. Partitioned Scheduling for Real-time Tasks on Multiprocessor Embedded Systems with Programmable Shared Srams. In *Proceedings of the 10th ACM International Conference on Embedded Software*, EMSOFT '12, pages 153–162, 2012.

[18] David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager. Thoth, A Portable Real-Time Operating System. *Communications of the ACM*, 22(2):105–115, 1979.

[19] Erika Chin, Adrienne P. Felt, Kate Greenwood, and David Wagner. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, 2011.

[20] Onur Cinar. *Pro Android C++ with the NDK*. Apress, Berkeley, CA, USA, 1st edition, 2012.

[21] George E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–657, 1960.

[22] Angelo Corsaro and Douglas C. Schmidt. Evaluating Real-Time Java features and performance for real-time embedded systems. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '02, pages 90–100, 2002.

[23] Andrea Dardanelli, Mara Tanelli, Bruno Picasso, Sergio M. Savaresi, Onorino Di Tanna, and Mario D. Santucci. A smartphone-in-the-loop active state-of-charge manager for electric vehicles. *IEEE/ASME Transactions on Mechatronics*, 17(3):454–463, 2012.

[24] L. Peter Deutsch and Daniel G. Bobrow. An Efficient, Incremental, Automatic Garbage Collector. *Communications of the ACM*, 19(9):522–526, 1976.

[25] Edsger W. Dijkstra, Leslie Lamport, Alain J. Martin, Carel S. Scholten, and Elisabeth F. M. Steffens. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[26] Bruce P. Douglass. *Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit*. Newnes, Newton, MA, USA, 1st edition, 2010.

[27] Nikil D. Dutt and Kiyoung Choi. Configurable Processors for Embedded Computing. *Computer*, 36(1):120–123, 2003.

[28] Christof Ebert and Capers Jones. Embedded Software: Facts, Figures, and Future. *Computer*, 42(4):42–52, 2009.

[29] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 1–6, 2010.

[30] Frédéric Fauberteau, Serge Midonnet, and Manar Qamhieh. Partitioned Scheduling of Parallel Real-time Tasks on Multiprocessor Systems. *SIGBED Review*, 8(3):28–31, 2011.

[31] Hasan Fayyad-Kazan, Luc Perneel, and Martin Timmerman. Linux PREEMPT-RT V2.6.33 Versus V3.6.6: Better or Worse for Real-time Applications? *SIGBED Review*, 11(1):26–31, 2014.

[32] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, 2005.

[33] William Foote. Theory versus Practice in Real-Time Computing with the Java(tm) Platform. In *Proceedings of the 2nd International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '99, pages 105–108, 1999.

[34] Jonas Fortmann. *Requirements Analysis and Prototypic Implementation of a Development Environment for Structured Text on RTAndroid*. Bachelor's thesis, RWTH Aachen, Aachen, Germany, 2014.

[35] Dominik Franke. *Testing Life Cycle-related Properties of Mobile Applications*. PhD thesis, RWTH Aachen, Aachen, Germany, 2015.

[36] Lee Garber and David Sims. In Pursuit of Hardware-Software Codesign. *Computer*, 31(6):12–14, 1998.

[37] Thomas Gerlitz. *Test and Stabilisation of RTAndroid*. Master's thesis, RWTH Aachen, Aachen, Germany, 2012.

[38] Thomas Gerlitz, Igor Kalkov, John F. Schommer, Dominik Franke, and Stefan Kowalewski. Non-Blocking Garbage Collection for Real-Time Android. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, pages 108–117, 2013.

[39] Philippe Gerum. The Xenomai Project: Implementing a RTOS Emulation Framework on GNU/Linux, 2004. White paper. `http://www.xenomai.org/documentation/xenomai-2.5/pdf/xenomai`. Accessed on 09.04.2017.

[40] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing, Boston, MA, USA, 2000.

[41] Stefan Gradl, Patrick Kugler, Clemens Lohmüller, and Björn Eskofier. Real-time ECG monitoring and arrhythmia detection using Android-based mobile devices. In *Proceedings of the 34th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, EMBC '12, pages 2452–2455, 2012.

[42] Bill Greene. Agile Methods Applied to Embedded Firmware Development. In *Proceedings of the Agile Development Conference*, ADC '04, pages 71–77, 2004.

[43] Raphael Guerra, Stefan Schorr, Gerhard Fohler, Enrico Bini, G. Buttazzo, Vanessa Romero, Karl-Erik Arzen, Claudio Scordino, and Johan Eker. Adaptive Resource Management for Mobile Terminals - The ACTORS approach. In *Proceedings of 1st Workshop on Adaptive Resource Management*, WARM '10, 2010.

[44] Rajesh K. Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design Test of Computers*, 10(3):29–41, 1993.

[45] Alexandru Gurghian. *Flexible Konfiguration eines mikrocontrollerbasierten Feldgeräteadapters für RTAndroid*. Bachelor's thesis, RWTH Aachen, Aachen, Germany, 2013.

[46] Jan P. Haller. *Utilizing Bluetooth for Supporting Real-Time Wireless Communication*. Bachelor's thesis, RWTH Aachen, Aachen, Germany, 2015.

[47] Russell J. Harding and Mark A. Whitty. Employing Android devices for autonomous control of a UGV. In *Proceedings of the 9th European Conference on Mobile Robots*, ECMR '15, pages 1–6, 2015.

[48] Yunan He, Chen Yang, and Xiao-Feng Li. Improve Google Android User Experience with Regional Garbage Collection. In *Proceedings of the 8th IFIP International Conference on Network and Parallel Computing*, NPC '11, pages 350–365, 2011.

[49] Joachim Henkel. Selective revealing in open innovation processes: The case of embedded Linux. *Research Policy*, 35(7):953–969, 2006.

[50] John Hennessy and Mark Heinrich. Hardware/Software Codesign of Processors: Concepts and Examples. In Giovanni de Micheli and Mariagiovanna Sami, editors, *Hardware/Software Co-Design*, pages 29–44. Springer Netherlands, Dordrecht, 1996.

[51] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund University, Lund, Sweden, 1998.

[52] Thomas A. Henzinger and Joseph Sifakis. The Embedded Systems Design Challenge. In *Proceedings of the 14th International Conference on Formal Methods*, FM'06, pages 1–15, 2006.

[53] James J. Hunt. Realtime Java technology in avionics systems. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 138–147, 2010.

[54] James J. Hunt. A New I/O Model for the Real-time Specification for Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 26–33, 2012.

[55] IEC. *IEC 61131-3: Programmable Controllers — Part 3: Programming Languages*. International Electrotechnical Commission, 3rd edition, 2003.

[56] Wallace Jackson. *Android Apps for Absolute Beginners*. Apress, Berkeley, CA, USA, 3rd edition, 2014.

[57] Yiming Jing, Gail-Joon Ahn, Ziming Zhao, and Hongxin Hu. RiskMon: Continuous and Automated Risk Assessment of Mobile Applications. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 99–110, 2014.

[58] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, San Francisco, CA, USA, 1996.

[59] Igor Kalkov. *Design and Integration of Real-Time into the Android Platform*. Master's thesis, RWTH Aachen, Aachen, Germany, 2011.

[60] Igor Kalkov, Dominik Franke, John F. Schommer, and Stefan Kowalewski. A Real-time Extension to the Android Platform. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 105–114, 2012.

[61] Igor Kalkov, Alexandru Gurghian, and Stefan Kowalewski. Predictable Broadcasting of Parallel Intents in Real-Time Android. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 57–66, 2014.

[62] Igor Kalkov, Alexandru Gurghian, and Stefan Kowalewski. Priority Inheritance During Remote Procedure Calls in Real-Time Android Using Extended Binder Framework. In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '15, pages 5:1–5:10, 2015.

[63] Igor Kalkov, Alexandru Gurghian, and Stefan Kowalewski. Explicit Prioritization of parallel Intent Broadcasts in Real-Time Android. *Concurrency and Computation: Practice and Experience*, pages 1–21, 2017.

[64] Wataru Kanda, Yu Murata, and Tatsuo Nakajima. SIGMA System: A Multi-OS Environment for Embedded Systems. *Signal Processing Systems*, 59(1):33–43, 2010.

[65] David Kantola, Erika Chin, Warren He, and David Wagner. Reducing Attack Surfaces for Intra-application Communication in Android. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 69–80, 2012.

[66] Krzysztof Kołek. Application of Android OS as Real-Time Control Platform. *Automatyka / Automatics*, 17(2):197–206, 2013.

[67] Kushal Koolwal. Myths & Realities of Real-Time Linux Software Systems, 2009. White paper. `https://static.lwn.net/images/conf/rtlws11/papers/proc/p20.pdf`. Accessed 11.08.2016.

[68] David Kushner. The Making of Arduino: How five friends engineered a small circuit board that's taking the DIY world by storm. *IEEE Spectrum Online*, 2011.

[69] Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline Scheduling in the Linux Kernel. *Software: Practice and Experience*, 46(6):821–839, 2016.

[70] Yossi Levanoni and Erez Petrank. An On-the-Fly Reference-Counting Garbage Collector for Java. *ACM Transactions on Programming Languages and Systems*, 28(1):1–69, 2006.

[71] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ, USA, 1st edition, 2000.

[72] Hans H. Lovengreen, Anders P. Ravn, and Hans Rischel. Design of embedded, real-time systems: developing a method for practical software engineering. In *Proceedings of the IEEE International Conference on Computer Systems and Software Engineering - Systems Engineering Aspects of Complex Computerized Systems*, COMPEURO '90, pages 385–390, 1990.

[73] Ruhui Ma, Fanfu Zhou, Erzhou Zhu, and Haibing Guan. Performance Tuning Towards a KVM-based Embedded Real-Time Virtualization System. *Journal of Information Science & Engineering*, 29(5):1021–1035, 2013.

[74] Cláudio Maia, Luís Nogueira, and Luís M. Pinho. Evaluating Android OS for Embedded Real-Time Systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, OSPERT '10, pages 62–70, 2010.

[75] Cláudio Maia, Luís Nogueira, and Luís M. Pinho. Experiences on the Implementation of a Cooperative Embedded System Framework: Short Paper. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 70–72, 2010.

[76] Amiya K. Maji, Fahad A. Arshad, Saurabh Bagchi, and Jan S. Rellermeyer. An Empirical Study of the Robustness of Inter-component Communication in Android. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '12, pages 1–12, 2012.

[77] Paolo Mantegazza, Lorenzo Dozio, and Steve Papacharalambous. RTAI: Real Time Application Interface. *Linux Journal*, 2000(72es):10, 2000.

[78] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, 2008.

[79] Wolfgang Mauerer, Gernot Hillier, Jan Sawallisch, Stefan Hönick, and Simon Oberthür. Real-Time Android: Deterministic Ease of Use. In *Proceedings of Embedded Linux Conference Europe*, ELCE '12, 2012.

[80] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[81] Reto Meier. *Professional Android 2 Application Development*. John Wiley & Sons, San Francisco, CA, USA, 2nd edition, 2010.

[82] Reto Meier. *Professional Android 4 Application Development*. John Wiley & Sons, San Francisco, CA, USA, 1st edition, 2012.

[83] Prashant M. Menghal and A. Jaya Laxmi. Real-time Control of Electrical Machine Drives: A Review. In *Proceedings of the 122nd International Conference on Power, Control and Embedded Systems*, ICPCES '10, pages 1–6, 2010.

[84] Clifford W. Mercer, Raguanathan Rajkumar, and Hideyuki Tokuda. Applying Hard Real-time Technology to Multimedia Systems. In *Proceedings of the Workshop on the Role of Real-Time in Multimedia/Interactive Computing System*, 1993.

[85] Clifford W. Mercer and Hideyuki Tokuda. Preemptibility in Real-Time Operating Systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposuim*, pages 78–87, 1992.

[86] Bhupinder S. Mongia and Vijay K. Madisetti. Reliable Real-Time Applications on Android OS. White paper. `https://www.researchgate.net/publication/266286558_Reliable_Real-Time_Applications_on_Android_OS`. Accessed 22.10.2016.

[87] Morten Mossige, Pradyumna Sampath, and Rachana Rao. Evaluation of Linux `RT-PREEMPT` for Embedded Industrial Devices for Automation and Power Technologies: A Case Study. In *Proceedings of the 9th Real-Time Linux Workshop*, 2007.

[88] Mathias Obster. *Ausführung und Simulation von SPS-Programmen auf RTAndroid*. Master's thesis, RWTH Aachen, Aachen, Germany, 2013.

[89] Mathias Obster, Igor Kalkov, and Stefan Kowalewski. Development and Execution of PLC Programs on Real-Time Capable Mobile Devices. In *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation*, ETFA '14, 2014.

[90] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22nd USENIX Conference on Security*, SEC '13, pages 543–558, 2013.

[91] Hyeong-Seok Oh, Beom-Jun Kim, Hyung-Kyu Choi, and Soo-Mook Moon. Evaluation of Android Dalvik Virtual Machine. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 115–124, 2012.

[92] Nicolas Oros and Jeffrey L. Krichmar. Neuromodulation, Attention and Localization Using a Novel Android™ Robotic Platform. In *Proceedings of the 2nd Joint IEEE International Conference on Development and Learning and Epigenetic Robotics*, ICDL-EPIROB '12, pages 1–6, 2012.

[93] Yoav Ossia, Ori Ben-Yitzhak, and Marc Segal. Mostly Concurrent Compaction for Mark-Sweep GC. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 25–36, 2004.

[94] Sanghyun Park, Tran T. T. Ha, Jadhav Y. Shivajirao, Minsoo Hahn, Jaehyung Park, and Jinsul Kim. Smart Wheelchair Control System Using Cloud-Based Mobile Device. In *Proceedings of the 3rd International Conference on IT Convergence and Security*, ICITCS '13, pages 1–3, 2013.

[95] Edwin Peguero, Miguel Labrador, and Brittany Cook. Assessing Jitter in Sensor Time Series from Android Mobile Devices. In *Proceedings of the 2nd IEEE International Conference on Smart Computing*, SMARTCOMP '16, pages 1–8, 2016.

[96] Luc Perneel, Hasan Fayyad-Kazan, and Martin Timmerman. Can Android be Used for Real-Time Purposes? In *Proceedings of the International Conference on Computer Systems and Industrial Informatics*, ICCSII '12, pages 1–6, 2012.

[97] Luc Perneel, Hasan Fayyad-Kazan, and Martin Timmerman. RTOS Evaluation Project: Comparison of QNX Neutrino, Windows CE7, Linux RT and Android (RT) Operating Systems on ARM Processor, 2012. Project report. `http://embeddedpro.ucoz.ru/app_notes/rtos/EVA-2_9-CMP-ARM-v300.pdf`. Accessed on 20.02.2017.

[98] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-level Programming of Embedded Hard Real-time Devices. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 69–82, 2010.

[99] Filip Pizlo, Lukasz Ziarek, and Jan Vitek. Real-Time Java on Resource-constrained Platforms with Fiji VM. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 110–119, 2009.

[100] Rodion Podorozhny. Design and Verification of Cellphone-based Cyber-Physical Systems: A Position Paper. In *Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 603–607, 2013.

[101] Mareen Przybylla and Ralf Romeike. Physical Computing and its Scope - Towards a constructionist Computer Science Curriculum with Physical Computing. In *Proceedings of the 3rd International Constructionism Conference*, volume 301, pages 278–288, 2014.

[102] Stefan Rakel. *A Visual, Block-Based Structured Text Editor for Twistturn*. PhD thesis, RWTH Aachen, Aachen, Germany, 2015.

[103] Jussi Ronkainen and Pekka Abrahamsson. Software Development under Stringent Hardware Constraints: Do Agile Methods Have a Chance? In *Proceedings of*

the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering, XP '03, pages 73–79, 2003.

[104] Alejandro P. Ruiz, Mario A. Rivas, and Michael G. Harbour. CPU Isolation on the Android OS for Running Real-Time Applications. In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '15, pages 6:1–6:7, 2015.

[105] Stefan Schake. *Integration of the PROFINET Protocol Stack into the RTAndroid Platform*. Master's thesis, RWTH Aachen, Aachen, Germany, 2016.

[106] Peter Schartner and Stefan Bürger. Attacking Android's Intent Processing and First Steps Towards Protecting It, 2012. Technical report. `https://syssec.at/download/misc/TR-syssec-12-01.pdf`. Accessed on 03.12.2016.

[107] Martin Schoeberl. Real-Time Garbage Collection for Java. In *Proceedings of the 9th International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '06, pages 424–432, 2006.

[108] Martin Schoeberl. Scheduling of Hard Real-time Garbage Collection. *Real-Time Systems*, 45(3):176–213, 2010.

[109] Martin Schoeberl and Wolfgang Puffitsch. Nonblocking Real-time Garbage Collection. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(1):1–28, 2010.

[110] Peter Scholz. *Softwareentwicklung eingebetteter Systeme: Grundlagen, Modellierung, Qualitätssicherung*. Springer, Berlin, Germany, 2005.

[111] Ken Schwaber. SCRUM Development Process. In *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 117–134, 1995.

[112] Claudio Scordino and Giuseppe Lipari. Energy saving scheduling for embedded real-time linux applications. In *Proceedings of the 5th Real-Time Linux Workshop*, 2003.

[113] Claudio Scordino and Giuseppe Lipari. Using Resource Reservation Techniques for Power-aware Scheduling. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 16–25, 2004.

[114] Claudio Scordino and Giuseppe Lipari. Linux and Real-Time: Current Approaches and Future Opportunities. In *Proceedings of International Congress on Methodologies for Emerging Technologies in Automation*, ANIPLA' 06, 2006.

[115] Jesper Smith, Douglas Stephen, Alex Lesman, and Jerry Pratt. Real-Time Control of Humanoid Robots Using OpenJDK. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 29–36, 2014.

[116] John A. Stankovic. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *Computer*, 21(10):10–19, 1988.

[117] David Stepner, Nagarajan Rajan, and David Hui. Embedded Application Design Using a Real-time OS. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 151–156, 1999.

[118] Jürgen Teich. Hardware/Software Codesign: The Past, the Present, and Predicting the Future. *Proceedings of the IEEE (Special Centennial Issue)*, 100(CI):1411–1430, 2012.

[119] David Thoenessen. *Steuerung einer Fertigungsanlage mit RTAndroid*. Bachelor's thesis, RWTH Aachen, Aachen, Germany, 2012.

[120] David Thönnessen. *Integration of a Real-Time Ethernet Protocol into the RTAndroid Platform*. Master's thesis, RWTH Aachen, Aachen, Germany, 2014.

[121] Hiraku Toyooka. Evaluation of Real-time Property in Embedded Linux, 2014. Slides. `https://events.linuxfoundation.org/sites/events/files/slides/toyooka_LCJ2014_v10.pdf`. Accessed on 10.03.2017.

[122] UBM Electronics. Embedded Markets Study: Changes in Today's Design, Development & Processing Environments, 2015. Market study report. `https://devzone.nordicsemi.com/attachment/a61052ff4978f8c42b4f6f4b11a1b0e0`. Accessed on 18.01.2017.

[123] Pedro D. Urbina Coronado, Horacio Ahuett-Garza, and Vishnu-Baba Sundaresan. Design of an Android Based Input Device for Electric Vehicles. In *Proceedings of the 2nd International Conference on Connected Vehicles and Expo*, ICCVE '13, pages 736–740, 2013.

[124] Joseph Weizenbaum. Symmetric List Processor. *Communications of the ACM*, 6(9):524–536, 1963.

[125] Johan Wideberg, Pablo Luque, and Daniel Mantaras. A smartphone application to extract safety and environmental related information from the OBD-II interface of a car. *International Journal of Vehicle Systems Modelling and Testing*, 7(1):1–11, 2012.

[126] Wayne H. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, 1994.

[127] David T. Wright and David J. Williams. Object-like software design methods for intelligent real-time process control. In *Proceedings of the 8th IEEE International Symposium on Intelligent Control*, ISIC '93, pages 144–149, 1993.

[128] Bin Xiao, Muhammad Z. Asghar, Tapani Jamsa, and Petri Pulii. Canderoid: A mobile system to remotely monitor travelling status of the elderly with dementia. In *Proceedings of the International Joint Conference on Awareness Science and Technology and Ubi-Media Computing*, iCAST-UMEDIA '13, pages 648–654, 2013.

[129] Karim Yaghmour. *Embedded Android*. O'Reilly Media, 2013. Index.

[130] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum. *Building Embedded Linux Systems*. O'Reilly, Beijing, China, 2nd edition, 2008.

[131] Yin Yan, Shaun Cosgrove, Varun Anand, Amit Kulkarni, Sree H. Konduri, Steven Y. Ko, and Lukasz Ziarek. Real-time Android with RTDroid. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 273–286, 2014.

[132] Yin Yan, Shaun Cosgrove, Ethan Blantont, Steven Y. Ko, and Lukasz Ziarek. Real-Time Sensing on Android. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 67–75, 2014.

[133] Yin Yan, Sree H. Konduri, Amit Kulkarni, Varun Anand, Steven Y. Ko, and Lukasz Ziarek. RTDroid: A Design for Real-Time Android. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, pages 98–107, 2013.

[134] Victor Yodaiken and Michael Barabanov. A Real-Time Linux. *Linux Journal*, 34, 1997.

[135] Benjamin Zores. The Growth of Android in Embedded Systems: The Linux Foundation Training Publication, 2013. Online article. `http://www.emacinc.com/sites/default/files/lft_pub_growth_of_android.pdf`. Accessed on 30.01.2017.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen
Email: `biblio@informatik.rwth-aachen.de`

2015-01 * Fachgruppe Informatik: Annual Report 2015

2015-02 Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications

2015-05 Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity

2015-06 Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"

2015-07 Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon

2015-08 Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization

2015-09 Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models

2015-11 Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus

2015-12 Christoph Matheja, Christina Jansen, and Thomas Noll: Tree-like Grammars and Separation Logic

2015-13 Andreas Polzer: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen

2015-14 Niloofar Safiran and Uwe Naumann: Symbolic vs. Algorithmic Differentiation of GSL Integration Routines

2016-01 * Fachgruppe Informatik: Annual Report 2016

2016-02 Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems

2016-03 Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl: Lower Runtime Bounds for Integer Programs

2016-04 Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution

2016-05 Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization

2016-06 Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud

2016-07 Sebastian Biallas : Verification of Programmable Logic Controller Code using Model Checking and Static Analysis

2016-08 Klaus Leppkes, Johannes Lotz, and Uwe Naumann: Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features

2016-09 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic

2016-10 Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering

2017-01 * Fachgruppe Informatik: Annual Report 2017

2017-02 Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity

2017-04 Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE

2017-05 Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems

2017-06 Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy

2017-07 Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++

2017-08 Thomas Gerlitz: Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts

2017-09 Muhammad Hamad Alizai, Jan Beutel, Jó Ágila Bitsch, Olaf Landsiedel, Luca Mottola, Przemyslaw Pawelczak, Klaus Wehrle, and Kasim Sinan Yildirim: Proc. IDEA League Doctoral School on Transiently Powered Computing

2018-01 * Fachgruppe Informatik: Annual Report 2018

2018-02 Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.