MASTER

System exploration of next generation flash devices

de Jong, I.J.

Award date:
2012

Link to publication
System Exploration of Next Generation Flash Devices

by

Irenéus Johannes de Jong

(0720114)

A thesis submitted in partial fulfillment for the degree of Master of Science

in the Faculty Electronic Engineering and the Faculty Wiskunde & Informatica

August 2012
“You live and learn. At any rate, you live”

Douglas Adams (1952-2001)
Abstract

Nowadays the memory system is a major factor in determining the performance and the power consumption of embedded, as well as server systems. Typically the memory latency is significantly higher than the cycle time of the processor. Even modern out-of-order CPUs are not able to hide the stalls caused by interactions with the memory system. In mobile devices flash memory is nowadays mostly used as non-volatile memory. Flash differs from standard memory by its asymmetric access latencies and its maintenance functions which are needed to keep the flash effectively usable for as long as possible time.

Mobile systems have become more complex; different layers of abstraction obscure the transaction flow of the disk accesses, making it very hard for the application to predict the location where the data will eventually end up, making performance optimizations very hard.

There is therefore a need to explore system configurations and simulate the designed solution to evaluate their impact. Currently there is no timing accurate full system simulator that has a flash disk model.

To improve the system performance a new interface, UFS has been introduced by JEDEC. This new interface has not yet been evaluated on a system level, and it is therefore not sure what the impact will be when a complete system is considered. At the moment of writing, no UFS device is commercially available.

This work will introduce a UFS flash disk model for gem5. The Flash model is a major contribution of this thesis, programmed in the language C++ as part of the open source BSD simulator gem5. It enables full system analysis of a mobile system that embeds a UFS device.

The UFS device model allows the analysis of the UFS protocol itself, and reasoning about the best usage of the protocol. The UFSA, the association which promotes the UFS protocol, claims that this protocol is most effective when it is used with a few intensive data transfers, rather than a lot of small transactions. This claim has been evaluated in this work.

This work demonstrates the model, shows it relevance to the current research fields and proposes an improvement to the software system in order to effectively implement UFS in mobile devices.
Acknowledgements

It has been over a year, since I first heard about the opportunity to do my master project at a company abroad. Until then the notion of graduation had seem a distant and abstract idea, something that I would only have to worry about after a long time. The opportunity that ARM, and Andreas Hansson in particular, proposed to me was challenging on many levels. It meant relocating to Cambridge, far away from the familiar environment, far away from friends and family.

I have had a lot of support from my friends and family during my stay in Cambridge via email, Facebook, Twitter, as well as postcards. I would like to thank everyone for their support, but in particular my parents, Miriam Vermey and Paul de Jong, and my sister, Maaike de Jong for their support, understanding and their visit during this period.

Also my friends, Yannick de Lange and Sebastien Fabrie, need to be thanked in this part. They have journeyed with me throughout the masters program, and have taught me new tricks and have given me new insights, which I needed for the successful completion of this project.

I would like to thank Kees Goossens for supervising me, as I realise that this was not too easy, given the geographical distance between me and him. Also I like to thank Henk Corpmaal and Rob Hoogerwoord for their feedback in the initial phase, and their willingness to evaluate my work.

The largest part of the work that has been done for this project took place at ARM in Cambridge. I would like to thank my co-workers and friends, Sascha Bischoff, Uri Wiener, Radhika Jagtap, William Wang, Peter Aldworth and Akash Bagdia for being there. Thanks to you I have had a great time during my internship. Also outside of the department, Brian Foley, Charlie Brej and Matt Horsnell, have made the experience of working at ARM a real pleasure.

A special thank you goes to my supervisor, Andreas Hansson, without whom I would not even have an assignment at ARM. Thanks for the great project and the wonderful chance to see and explore a very interesting research field. Thanks for your dedication and guidance, in both coding style and reporting style.

Finally I would like to thank Sanne Schoenmakers, for the best suggestion about how to write an acknowledgement section I have ever had.
# Contents

Abstract

Acknowledgements

List of Figures

List of Tables

Abbreviations

1 Introduction

1.1 Non-Volatile Memory types

1.1.1 NAND and NOR flash

1.1.2 PCM

1.1.3 MRAM

1.2 Aim of this research

1.2.1 Observations

1.2.2 Research Questions

1.2.3 Hypotheses

1.3 Contributions

1.4 Structure of the document

2 Background

2.1 The Linux file system

2.2 The SCSI layers

2.3 Entering the hardware

2.4 Flash management

2.4.1 Address translation

2.4.2 Garbage collection

2.4.3 Wear levelling

2.5 Management

2.6 Simulations

2.6.1 System exploration

2.6.2 gem5

3 UFS: A new interface

3.1 UFS Host Controller Interface

3.2 Communication

3.3 UFS device

3.4 SCSI command set
Contents

9.4.4 Exact FTL from disk ........................................ 62
9.4.5 Power modelling .......................................... 62

10 Reflection ...................................................... 63

Bibliography ....................................................... 65
# List of Figures

1.1 Memory hierarchy on mobile platforms .......................... 1
1.2 Android architecture [1] ........................................... 2
2.1 Overview of a typical mobile system architecture ................. 9
2.2 The linux file system structure ................................... 10
2.3 Different types of management .................................... 17
3.1 Top level architectural model[2] ................................. 20
3.2 The UFS system model[2] ......................................... 21
3.3 Transaction flow sequence diagram ............................... 25
4.1 Simulation model architecture .................................... 29
4.2 Queueing system overview ....................................... 31
5.1 System setup ...................................................... 34
6.1 Throughput of random read transfers ............................. 38
6.2 Throughput of random write transfers ............................ 38
6.3 Throughput of read transfers per file size ....................... 39
6.4 Throughput of write transfers per file size ..................... 40
6.5 Amount of data transferred during read actions, and the number of instructions executed per 100 ms interval; while running IOzone .................. 40
6.6 Amount of data transferred during write actions, and the number of instructions executed per 100 ms interval; while running IOzone .................. 41
6.7 Latency experienced per offset in the transferred file, for 16MB transfers with a chunk size of 4kB ........................................ 41
6.8 Latency experienced per offset in the transferred file, for 16MB transfers with a chunk size of 4kB ........................................ 42
6.9 Latency experienced per offset in the transferred file, for 16MB transfers with a chunk size of 2MB ........................................ 42
6.10 Amount of data transferred during write and read actions, and the number of instructions executed per 100 ms interval; while booting Android  .......... 43
6.11 Graphs of Figure 6.10 are combined to show the dependency ........ 43
6.12 Amount of data transferred during write and read actions, and the number of instructions executed per 100 ms interval; while running Bbench ........... 44
7.1 The hardware diagram for the hardware SSD simulator ............. 47
List of Tables

1.1 Comparison of different flash types ........................................... 3

2.1 Comparison of different wear levelling techniques ........................ 15

7.1 Comparison of related work ..................................................... 46
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AEL</td>
<td>ARM Embedded Linux</td>
</tr>
<tr>
<td>AMBA</td>
<td>Advanced Microcontroller Bus Architecture</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>apk</td>
<td>Android Package file</td>
</tr>
<tr>
<td>App</td>
<td>Application</td>
</tr>
<tr>
<td>ARM</td>
<td>Advanced RISC Machines</td>
</tr>
<tr>
<td>BBench</td>
<td>Browser Benchmark</td>
</tr>
<tr>
<td>CAT</td>
<td>Cost Age Time</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic Random Access Memory</td>
</tr>
<tr>
<td>eMMC</td>
<td>Embedded MultiMediaCard</td>
</tr>
<tr>
<td>FTL</td>
<td>Flash Translation Layer</td>
</tr>
<tr>
<td>gem5</td>
<td>A simulator combining GEMS and M5</td>
</tr>
<tr>
<td>GEMS</td>
<td>General Execution driven Multiprocessor Simulator</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU is Not Unix</td>
</tr>
<tr>
<td>HCI</td>
<td>Host Controller Interface</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>ID</td>
<td>Identification</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Drive Electronics</td>
</tr>
<tr>
<td>IOPs</td>
<td>Instruction Operations</td>
</tr>
<tr>
<td>IR</td>
<td>Internal Representation</td>
</tr>
<tr>
<td>JEDEC</td>
<td>Joint Electron Device Engineering Council</td>
</tr>
<tr>
<td>LBA</td>
<td>Logical Block Address</td>
</tr>
<tr>
<td>LU</td>
<td>Logical Unit</td>
</tr>
<tr>
<td>LUN</td>
<td>Logical Unit Number</td>
</tr>
<tr>
<td>MIPI</td>
<td>Mobile Industry Processor Interface</td>
</tr>
<tr>
<td>MLC</td>
<td>Multi Level Cell</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>MPHY</td>
<td>MIPI Physical Layer</td>
</tr>
<tr>
<td>MRAM</td>
<td>Magnetoresistive RAM</td>
</tr>
<tr>
<td>NAND</td>
<td>NAND flash memory</td>
</tr>
<tr>
<td>NIC</td>
<td>Network Interface Controller</td>
</tr>
<tr>
<td>NOR</td>
<td>NOR flash memory</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Controller Interface</td>
</tr>
<tr>
<td>PCM</td>
<td>Phase Change Memory</td>
</tr>
<tr>
<td>PIO</td>
<td>Programmers View I/O</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>RL benchmark</td>
<td>RedLicense Labs Benchmark</td>
</tr>
<tr>
<td>RTT</td>
<td>Round Trip Time</td>
</tr>
<tr>
<td>SAM</td>
<td>SCSI Architecture Model</td>
</tr>
<tr>
<td>SCSI</td>
<td>Small Computer System Interface</td>
</tr>
<tr>
<td>SD</td>
<td>Secure Digital (card)</td>
</tr>
<tr>
<td>SLC</td>
<td>Single Level Cell</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SQLite</td>
<td>SQL Lite</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static Random Access Memory</td>
</tr>
<tr>
<td>SSD</td>
<td>Solid State Disk</td>
</tr>
<tr>
<td>TLC</td>
<td>Triple Level Cell</td>
</tr>
<tr>
<td>UCS</td>
<td>UFS Command Set</td>
</tr>
<tr>
<td>UFS</td>
<td>Universal Flash Storage</td>
</tr>
<tr>
<td>UFSA</td>
<td>Universal Flash Storage Association</td>
</tr>
<tr>
<td>UFSHCI</td>
<td>UFS Host Controller Interface</td>
</tr>
<tr>
<td>UNIPro</td>
<td>United Protocol</td>
</tr>
<tr>
<td>UPIU</td>
<td>UFS Protocol Information Unit</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>UTP</td>
<td>UFS Transport Protocol</td>
</tr>
<tr>
<td>VFS</td>
<td>Virtual File System</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Dedicated to my parents and sister, who have supported me throughout
Chapter 1

Introduction

Nowadays the memory system is a major factor in determining the performance and the power consumption of embedded, as well as server systems. Typically the memory latency is significantly higher than the cycle time of the processor. Even modern out-of-order CPUs are not able to hide the stalls caused by interactions with the memory system.

Memories can be divided into two major classes: non-volatile and volatile memory. The difference between those memories is that volatile memory will lose its data when the power is turned off whereas non-volatile memory will not. Typically the access time of non-volatile memory is longer, which means that it is not ideal for quick memory operations for quick memory operations. In mobile computer systems the memory hierarchy typically looks as in Figure 1.1 [3].

![Figure 1.1: Memory hierarchy on mobile platforms](image)

Memories come in a lot of different flavours, e.g. SRAM, DRAM and flash. Flash is a non volatile memory type that is mostly used as a block device or a disk. Different solutions for memory have been designed for different applications, optimizing performance and power. Non-volatile memory is traditionally used to store code and data that is not often accessed. Now that parallelism and multitasking has become the standard, non-volatile memory is more used as a swap storage i.e. it stores both code and data that is not immediately needed by the operating system. Unfortunately, the current non-volatile memories, e.g. flash memory, are not designed for this purpose. Typically sequential flash accesses have a better performance than random medium accesses, this can make the response time unpredictable, and the performance slow. Moreover, moving the data from volatile memory to non-volatile memory is costly both in terms of performance and power.
Until recently flash storage was partially managed. The processor had full responsibility in handling the memory including handling all the errors that occurred in the flash. Nowadays new fully managed controller interfaces like eMMC and UFS are emerging, which allow the processor to treat flash as a normal read/write block device. The interfaces are also getting increasingly advanced and introducing features like re-ordering, command queuing and prioritisation. The downside of this approach is that it is difficult to predict the resulting system performance. Which solution is best depends on the application and the capabilities of the particular device. If one takes a mobile system as an example, then this mobile system has an operating system, e.g. Android Jelly Bean. The architecture of the operating system will have a monolithic software architecture (Figure 1.2). The Android system layers are built on the Linux kernel, which act as a hardware abstraction layer. To fully understand what is happening in the system and solve this challenge a full system evaluation, were all the different abstraction layers can be pulled apart and analysed, is required. This simulation has to consider the complete system, from the architecture model to the operating system and its drivers, and finally the user application.

![Android architecture](image)

The need for a full system analysis has become more apparent since the work of Kim et al. [4] showed that the performance of Android systems depends more than was expected on the performance of the disk. Even the performance of mobile browsers, which one expects to be completely independent of the disk accesses, suffered from a bad system organization. When the database of the system was moved from the flash disk to the RAM disk, the Webbench benchmark runtime improved from almost 500 seconds to below 200 seconds. It showed Androids use of the SQLite database system, and it concludes by stating: “We believe the storage system on mobile devices needs a fresh look”[4]. This thesis will enable the possibility to take a fresh look into the storage system by introducing a flash simulation model for gem5, a full system simulator.
Chapter 1. Introduction

1.1 Non-Volatile Memory types

There are different types of non-volatile memory. An overview of the types that are expected to be the most commonly used in the next 5 years can be found in table 1.1 [5] [6] [7] [8].

<table>
<thead>
<tr>
<th>Memory type</th>
<th>NAND</th>
<th>NOR</th>
<th>PCM</th>
<th>MRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write latency</td>
<td>200(\mu)s</td>
<td>1 – 10(\mu)s</td>
<td>1(\mu)s</td>
<td>4(\mu)s</td>
</tr>
<tr>
<td>Write energy</td>
<td>0.1 – 1(nJ/)bit</td>
<td>1(nJ/)bit</td>
<td>1(nJ/)bit</td>
<td>1(nJ/)bit</td>
</tr>
<tr>
<td>Read latency</td>
<td>10 – 25(\mu)s</td>
<td>100(ns)</td>
<td>50 – 100(ns)</td>
<td>4(ns)</td>
</tr>
<tr>
<td>Read energy</td>
<td>1(nJ/)bit</td>
<td>1(nJ/)bit</td>
<td>1(nJ/)bit</td>
<td>1(nJ/)bit</td>
</tr>
<tr>
<td>Idle power</td>
<td>0.1W</td>
<td>0.1W</td>
<td>0.1W</td>
<td>0W</td>
</tr>
<tr>
<td>Endurance</td>
<td>(10^5)</td>
<td>(10^5)</td>
<td>(10^8)</td>
<td>Infinite</td>
</tr>
</tbody>
</table>

Table 1.1: Comparison of different flash types

1.1.1 NAND and NOR flash

NAND architecture is organised in independent blocks. Each block is divided into a number of pages. The number of bits per page, and the number of pages per block differs per type of NAND flash. For a 4Gb single layer cell NAND memory a single page typically contains 16895 bits and a block will contain 64 pages. NAND flash can typically be erased or read per block, and programmed per page. Some NAND manufacturers will however allow partial page programming in some of their products, but this is not a common practise. For this reason NAND flash memory is not accessible on any location that a programmer might want, and thus unsuitable for random access memory.

NOR flash is typically slower and has a lower density than NAND flash. However, it provides the use of busses and addresses, allowing the user full access to all the memory [5] [9].

1.1.2 PCM

Phase Change Memory, (also known as PCRAM or PRAM) is a new type of memory. PCM exploits the phase changing behaviour that occurs in some materials when an electric current passes through. Usually chalcogenide glass is used for this effect. The heat that is generated when such a current passes changes this material between a crystalline and an amorphous state. PCM is faster, more energy efficient and has a longer endurance than NAND and NOR flash. The downside is however that this technology is more vulnerable for heat changes in the environment and that the technique is relatively expensive [6] [8].

1.1.3 MRAM

Magnetoresistive Random-Access Memory is a new type of memory that has been under development since the 1990s. In MRAM data is stored by magnetic storage elements. Two ferromagnetic plates form a cell, separated by a thin insulating layer, which can hold a magnetic current. One
of the two plates is a permanent magnet set to a certain polarity, the other’s field can be changed to match that of an external field to store data. This configuration is known as a spin valve. A memory device is built from a grid of cells. Because of its very low access time and non-volatile aspect it is believed that magnetoresistive RAM is a good candidate to eventually become a universal memory, and make the distinction between volatile and non volatile unnecessary [5] [7].

1.2 Aim of this research

There are a number of flash simulators used. Unfortunately those focus at best at the disk itself and do not consider the system around it. The general full system simulator makes no distinction between a flash disk and a normal hard drive. This is a generalisation which is dangerous to make, since the flash disk behaves differently. There is a need to evaluate software systems on hardware platforms, and it is in the best interest of the evaluation to make this as realistic as possible, to attain best results.

This research has focussed on the implementation of a flash disk model with a UFS interface in the gem5 simulator. This simulation model has been evaluated and some initial system exploration has been done. To date there are no UFS flash disk simulators available, apart from the one described in this document. A flash simulation model is attractive, as it is possible to analyse every component of it, were as this would be expensive with a real device.

In this document the impact of UFS within a system will be analysed. The aim of UFS is to have a high throughput while being power efficient. According to the UFSA, the association responsible for the promotion of UFS, the best usage is to have short periods of burst transactions and long periods where the device is in a sleeping state [2]. This thesis will evaluate the performance of a UFS device and analyse its place in a mobile system.

1.2.1 Observations

Considering a mobile system their are many indirections between the application and the storage [10]. It solves a number of problems, but this structure may obscure a number of other problems. At every level there might be a bottleneck and it is hard to specify where. The separate layers may not even be a bottleneck individually, but combining the layers may cause them to become one. For instance, a file system can make very clever usage of very small data transfers, if it turns out that it increases the number of random transactions to the flash, then the performance may be worse then when another less intelligent file system is used.

Given this information UFS does not seem to be able to do much about it. At a first glance it seems to improve things on one level only, the interface. But UFS is more than that. The UFS specification is set up in such a way that the most attractive usage of the standard is to send lots of data in short periods of time, with a relatively long pause between the transfers. This will cause the UFS interface to be energy efficient, and allows the device to do its garbage collection
between the transactions. An interface of $5.8\text{Gb/s}$ [2] seems to be fast enough for any mobile device. The question is if it is not causing any congestion somewhere along the way.

The flash device itself seems to be a possible bottleneck. A quick calculation shows that even with the most optimistic access times, the data transfer is not going to be near the interface speed, except when one uses multiple dies in parallel. This means that a lot of data to the flash device will be stored in the volatile memory and the disk cache. When this buffer gets full, the performance will drop. Hence it is wise to do a burst of data transfers, but then pause to give the disk enough time to empty its buffer.

Flash devices need maintenance functions to keep the memory device usable for as long as possible. These maintenance functions should ideally happen when there is no ongoing transaction, such that it will not influence the throughput. This is only possible when there is enough time in between transactions to do this. It would be interesting to see the trade off one has to make to set a point in time were the maintenance functions will be invoked regardless of the setting in UFS. Although this is not within the scope of this research, there will be functionality in the simulation model to research this.

Ideally the operating system has some functionality in place that considers the garbage collection, and tries to access the locations in such a way that the maintenance burden will not be too heavy on the flash device. However, with all the possible devices: unmanaged, partially managed, or fully managed; all the possible maintenance algorithms, and all the different layers of indirection, this seems like an impossible demand. Especially when the end user, in this case the application, does not want to have anything to do with the system set up. There are many parameters, and the possibility for a one size fits all operating system seems to be unlikely.

1.2.2 Research Questions

This research will focus on the full system analysis of a mobile system which embeds a UFS flash memory device. The aim of this research is to make apparent where the performance bottleneck in the system is, and if UFS seems to be useful in the system. This thesis is written in attempt to answer the following question:

*What effect may sensible usage of UFS have on the overall system performance, as noticed by the end user?*

In the process of answering this question, the following questions will be answered:

- What throughput can realistically be reached when using UFS?
- Can the UFS performance potential be fully utilized in a real system?
- What is the main bottleneck in the system?
- How will the user experience applications when the system makes use of UFS?
- What can the application do about the performance?
- What is the best usage of UFS?
1.2.3 Hypotheses

Given all the information presented in this document so far, the following hypotheses can be derived:

Hypothesis 1. UFS will have a positive effect on the system performance. It will be demonstrable that it is not the bottleneck in the system.

Hypothesis 2. Since it is to be expected that UFS is fast enough to facilitate the flash memory performance, the throughput that can be reached is dependent on the performance of the flash memory.

Hypothesis 3. The main bottleneck of the system will turn out to be the flash performance.

Since UFS has a maximum throughput of 5.8Gb/s and not much overhead, it should be clear that when it takes 200us to fetch one page from the flash, and pages are typically between 1kB and 8kB in size, that the maximum throughput needed is around 240MB/s or 1.9Gb/s. Clearly, a throughput limit has been selected which can accommodate the future speed improvements of flash devices, or multiple units in one device. For this reason it is not expected that the interface will be the bottleneck.

Hypothesis 4. There will be time between transfers that is overly sufficient for maintenance functions.

Since the garbage collection takes a significant amount of time, there should be enough time between transfers to accommodate this. If this is not the case, the application will stall longer, waiting for data. It is expected that the operating system will have a mechanism to support this.

Hypothesis 5. The application will often stall to wait for information when it is executed. This is because the application often need to access the disk for information, at random locations.

Hypothesis 6. The application may improve its performance by forcing the file system to store its information in one location.

If all the information is stored in one location rather than scattered all over the disk, then this is expected to improve performance.

Hypothesis 7. The best usage of UFS is to use it as advised by the UFSA: short active transaction periods followed by long sleep periods.

The advice of the UFSA seems very sensible when all is considered. It will save power, because it allows long sleeping periods, and if there is a need for garbage collection a long period of rest will allow the maintenance function to execute this action without troubling the performance.

1.3 Contributions

This thesis contributes a simulation model of a UFS device, including a UFS host, a UFS device and a flash device. It shows the overall performance of a flash device and splits it up into different
layers to show the bottlenecks in the system. The model can be tweaked and adapted in such a way that anybody can use it, and that anyone can research their own design choices. Given that UFS seems to become the next embedded flash standard, this means that gem5 can be a useful tool to explore the different software systems for mobile devices. No other simulator at this point in time supports this.

This thesis will demonstrate the use of the simulation model and the possibilities it enables to explore different mobile system configurations which embed flash devices. It will reason on how UFS can be used most effectively and demonstrate this by using the simulation model.

1.4 Structure of the document

The document is built up as follows. Chapter 2 gives a more detailed background about the different abstraction layers of a system, simulation systems in general and gem5 in particular. Chapter 3 introduces UFS and explains the benefits of this new interface. Chapter 4 explains the simulation model that has been added to gem5. Chapter 5 and Chapter 6 describe the experiments and their results respectively. Chapter 7 presents the related work that has been done in the different research fields that are related to this work. Chapter 8 will summarize the conclusion that can be drawn. Finally Chapter 9 and Chapter 10 present the future work and reflect upon the project.
Chapter 2

Background

To analyse this system, one must understand each layer of abstraction. Knowledge about the system might show the bottleneck in this hierarchy, or at least should give some insight as to where to look for it. This chapter will explore the system top down, starting with the file system down to the flash itself.

To provide all of the functionality that embedded computers have nowadays, multiple abstractions layers are implemented to solve the different problems that where encountered within systems. The computer scientist David Wheeler (1927-2004) did put it as follows: “All problems can be solved by another layer of indirection, but that usually will create another problem”[10]. Wheeler has got a good point here, because although the problems will be solved, it will make the actual action more obscure. When considering a modern computer system which embeds a flash device, the overview looks like Figure 2.1.

Figure 2.1 shows that a system contains multiple abstraction layers. At the top there is the application, which needs to store information on the disk. This application acts within the environment of the operating system which embeds, amongst other things, file systems and drivers. The drivers control external hardware, which allow the software to communicate with external devices. A system may contain multiple devices with multiple partitions.

Flash memory can be accessed faster sequentially than randomly. With this knowledge, the downside of abstraction layers become apparent. If the application can not know what address it has accessed on the storage device, and if it ends up waiting until the transaction is finished, then this will influence the system performance, and ultimately the Quality of Service.

2.1 The Linux file system

The Linux file system is designed to abstract the complexity of the different types of file systems from the application developer. As a developer, one does not have to have knowledge of the type of storage medium device, interface, or even the type of file system itself. The API layer provides all the functionality needed to support every configuration permutation. When a file
handling function, such as read, is called all the underlying handling is taken care of by the lower layers. This system is the first layer between the application and the storage. It tries to improve the ease of use for the application developer by taking away the details of the storage handling mechanisms. Nevertheless, the performance impact of this system may show when the performance of the complete system is analysed. Not only are there abstraction layers, but also software caches placed in the system, dedicated to make the file accesses faster. This system is however designed originally to accommodate a system with a hard disk rather than a flash disk. The question remains, whether this system is evolved in a system that is effective for flash disks as well.

A file system is an organization of data and metadata on a storage device. Different devices and applications demand different file system structures and there is no solution which is optimal
for all configurations. The Linux file system components structure is a layered structure and is shown in Figure 2.2.

![The Linux file system structure](image)

**Figure 2.2: The Linux file system structure**

The majority of the file system code is embedded in the kernel. The user space contains the applications and the GNU C library. The GNU C library provides the user interface with the file system calls (e.g., open, write, etc.). This interface manages all the system calls it receives, makes sure that everything is handled correctly and accesses the correct storage device.

The system calls access the Virtual File System, which is the main handling mechanism of all the supported flavours of file systems. Each different file system implementation provides a set of functions to have an interface with the VFS. New types of file systems can be added or removed by making use of register functions. The VFS also keeps a list of every file system in use and which device is associated with it. The VFS makes use of two caches, inode and directory cache. Both of these caches are allocated in the main memory. An inode represents an object in the file system with a unique identifier. The file system provides a method to translate a file name into a unique inode identifier and an inode reference. The most recently used directories and files are stored in this cache.

Below the file system implementation is the buffer cache. This cache stores the most recently used pages from the disk devices, such that it does not have to go all the way back to fetch the same data all the time. This cache is coupled with the block device driver[11].
2.2 The SCSI layers

The file system is coupled with the device drivers. Although the aim of this chapter is to provide an understanding of why a storage system is so complex and not to describe all the functional layers in great detail, it is interesting to show a general overview of a complex block device driver in Linux. SCSI is a part of the UFS protocol, which makes it useful to have a basic understanding of the protocol at this stage. For this purpose, this section will focus on the SCSI driver and how it is built up.

SCSI stands for Small Computer System Interface and it is an overwhelming interface at first sight. It is one of the oldest interfaces still evolving to date. The specification consists of over 900 pages, and it supports a wide range of storage devices. When it was created in 1979 it defined an 8 bit parallel interface which provided a maximum throughput of 5MB/s. Nowadays the SCSI standards can provide a throughput of more than 640MB/s.

In those 33 years of development the basic command set and system architecture has remained almost unchanged. A SCSI interface consists of hosts (referred to as initiators in the SCSI standard) and devices (refereed to as hosts in the SCSI standard). Each host can request a certain actions, and each device can provide the functionality for an action. The host is in this system the processor and the device is the disk. In total there are over sixty commands available, making SCSI able to support all types of storage devices, from tapes to hard drives.

The SCSI driver in Linux is a monolithic architecture consisting of three layers. The upper layer makes the distinction between the flavour of devices. A character device needs to be accessed differently than a block device, simply because the addressing is per byte rather than per page. It ensures that the right type of actions are send to the right type of devices. It also handles the completion of commands. It checks the status of the SCSI acknowledgement and notifies the callers about the status.

The services are provided by the mid layer. It acts as the unification layer between the type of devices and the interfaces which connect them to the system. This abstraction layer exists because one does not want to bother with the exact register bits that need to be set for the transaction when setting up the command for a disk, and when the low level driver initiates the transaction it would be ideal to not worry about the message set up but just put the buffer on the interface. All transactions set up by the upper layer are queued in the middle layer.

The lower layer provides the access point to the hardware. The hardware is notified by this layer about the command that needs to be executed[12].

2.3 Entering the hardware

Different interfaces demand different handling. The most generic interface consists of an initiator and a host, where the host initiates the commands and the device will handle them. Again, many flavours and setups exist. Some interfaces allow multiple devices, some multiple hosts, and some allow both. Advanced interface protocols will have some discovery mechanism which will allow
the exploration of all the devices on the interface. The host will provide the functionality for this
discovery, and will communicate the set up to the software system, such that it can be handled
correctly.

With large data transactions, the data is usually communicated between the memory and the
disk. This is done via DMA transactions. Because these type of transaction do not have to be
monitored by the processor all the time, the transactions can be done directly between the I/O
and the memory, requiring less overhead and providing a higher throughput.

Depending on the interface, caches may be implemented between the device and the storage space
of the device, and between the processor and the main memory. People familiar with memory
consistency can see that this is a potential hazard. It is therefore important to guarantee the
completion of transfers, and to have handshakes between the host controller and the device. This
ensures that the data in the memory is indeed the data that existed on the disk, and that it did
not get delayed in one of the abstraction layer in between.

Each device can consist of multiple logic units. A logic unit does not have a unique unambiguous
definition, but in this thesis it will only be used as defined by the SCSI standard, i.e.: “...an
externally addressable entity within a SCSI target device that implements a SCSI device model
and contains a device server”[13]. Because different LUs have different device servers, it is not
possible for two logic units to control the same physical address space. Each LU can control
multiple partitions, but multiple LUs cannot control the same partition[13].

\section{Flash management}

Flash is usually organized in planes, blocks and pages. A flash die contains multiple planes,
a plane contains multiple blocks, and a block contains multiple pages. The numbers are very
dependent on the type of device that is used. Flash devices can be written per page, but have
to be erased per block. Furthermore, a page has to be erased before it can be written again.
To compensate for the limited durability of the flash, different flash management modules are
commonly used. The management of the flash consists out of three different modules:

\begin{itemize}
  \item Address translation
  \item Garbage collection
  \item Wear levelling
\end{itemize}

The different strategies that can be used in the model are described in this section. It is im-
portant to gain an understanding of the different possible strategies to solve these problems, to
understand that it is hard to predict on a high level where the data is going to end up without
understanding the flash device. At a high level one is not aware of the implemented maintenance
algorithms, and yet these functions have a different effect on the flash performance and the data
locality. This section will describe the different flavours of maintenance functions used in flash
devices. It will illustrate how difficult it is to capture every possible flash device because of the
diversity of these functions.
2.4.1 Address translation

In address translation the strategies that have been created can be divided into three main groups:

- Page level mapping
- Block level mapping
- Hybrid mapping

In every group different variants have been created. In this section the focus will be on the difference between the different groups. In page level mapping every logical page location known by the driver is translated into a physical page address. This strategy is very costly, because it requires a large table with information of every page. Take for example a 16GB flash device with 16,384 blocks and 64 pages per block, the total size of the table would be 1,048,576 entries. This would take at least 24 bits per entry to store. In total this requires a minimum of space of about 3 MB just for storing the translation table. In most embedded systems this is infeasible. Since every page can be accessed in $O(1)$ the performance of this approach is good. To overcome the problem of needing a lot of RAM to store the page mapping table, the page table is usually stored in the flash memory. A small RAM is then used to store the frequently accessed page locations. These type of devices usually have an extra arbiter to indicate frequently used addresses.

Another way to map the location is on a block level. In block level mapping a logical page address is made out of a logical block address and an offset. For example one could take the address of any page, divide it by the number of blocks and round that number down to the nearest integer. The offset can be derived by performing a modulo operation by the number of blocks. The number of entries that the table needs to store the mapping is equal to the number of blocks. If the previous example is considered this would result in a table size of 32kB. The general read time will be about the same as with the page level mapping, but when a page needs to be rewritten, the whole block should be moved to another location, since NAND flash does not allow single page erasure. This makes writing in general very costly, since copying the data to another block takes in a worst case scenario the number of pages per block times the write access time. In the page level mapping this was not the case because the page location would be invalidated, and the page would be moved to another physical location.

The third approach is a hybrid address mapping. Most hybrid mapping algorithms divide the memory into a number of log blocks and a number of data blocks. The data block area will take up the greatest part of the flash memory and the address translation of this part will be on a block level. The log blocks will take a relatively small amount of space, typically 5% of the total amount, and will be indexed on a page level. Typically the log block area is used to store the overwriting data and to merge log blocks into the data area. This will create empty log blocks which can be used again. There are different types of merge actions that can be performed on log blocks. A full merge involves a data block and a log block. Both blocks have too many invalid entries to write one over into the other, but the valid entries would fit together in a new block. The full merge will then allocate a new data block, copy the entries into the new block...
and erase the two old blocks. This is very expensive because it requires the erasing of two blocks. Another option is a partial merge. This merge takes a log block and a data block, and one of the blocks has enough valid storage spaces to contain the pages of the other block. The pages will be transferred to one block, making this the new data block, while the other one will be erased and will added to the free block pool. The last option is a switch merge. In essence this involves a log block that is fully occupied with all its entries. When this will occur the log block will be converted to a data block[14] [15].

The hybrid address mapping solution seems to be the flavour which captures the behaviour of the other two. Which solution is chosen in the flash device used depends largely on what the designer is trying to achieve, and what trade-off has been made. If this behaviour needs to be modelled then it would be a good idea to try to make an abstraction that ideally captures the behaviour of all three solutions.

### 2.4.2 Garbage collection

Over the years there have been many different methods proposed that can be used for garbage collection. Three of those methods will be described in this section. Garbage collection can occur when there are no more free pages that can be accessed or it can be running in the background, only performing actions when no transaction to the flash is made.

A greedy approach to garbage collection would be to clean the block with the most invalidated pages. This will result in the highest profit that can be made from one clean action, i.e. there is no block that would give more free space after an erase action. On the other hand it does map frequently changed data to the same block, so in terms of wear levelling this is not a good approach.

A cost benefit approach can also be considered. This approach determines the cost of an action and compares it with the amount of benefit that could be gained from doing that action. This can be done by using the following equation: \(\frac{a(1-u)}{2u}\). In this equation \(u\) is the percentage of valid entries in the block and \(a\) is the time since the last modification. Ideally the erasure of the block would occur as late as possible, allowing the block to fill up with invalid entries and making the clean action less costly. When the result of the equation rises above a predefined cleaning threshold, the cleaning of the block can be initiated.

A third approach would be to determine the Cost Age Time (CAT) of a block. In essence the data is divided in three types of data: read only data, cold data and hot data. Hot data is defined as data that is frequently updated, cold data is defined as data that is infrequently updated. In this approach newly written data is given a high hot degree, which decreases as its age grows. When the data is updated its degree will increase again. A block is considered to be cold when its hot degree is lower than the average hot degree. The read only data is separated from the writeable data, and the hot data is clustered apart from the cold data. The algorithm chooses to clean segments that minimize the CAT formula: \(\text{CleaningCostFlashMemory} \times \frac{1}{\text{Age}} \times \text{NumberofCleaning}\). The cleaning cost of the flash memory is defined as: \(\frac{u}{1-u}\). Where \(u\) is the number of valid pages in a block. The \(\text{Age}\) is defined as the elapsed time since the segment was
created. The *NumberOfcleaning* indicates how many times the block has been erased [16] [17] [18].

The diversity of garbage collection functions make it difficult to create a function that captures the behaviour of all the possible garbage collection algorithms. For this reason it would be better to generalize the behaviour of the garbage collector, as a function of the moment at which a block should be erased, and the impact of the erase action on the system. This will allow researchers to investigate a system configuration, while assuming that they have a garbage collection algorithm with a defined behaviour. This way, all garbage collectors can be modelled, albeit on a high level.

### 2.4.3 Wear levelling

Flash blocks have a lifetime of approximately 100000 program and erase cycles [5]. If the same block was erased and programmed over and over again, the lifetime of the block would be a matter of days. To ensure that the flash lasts longer than that it is wise to spread the data that has to be stored over the complete memory. This strategy is called dynamic wear levelling.

Not all the data is accessed equally often. This means that it is possible for one block of data to wear down faster than other blocks. To prevent this and keep the wear levels the same over all the data blocks it can be convenient to shift the data that is not often accessed to a block that is. As a consequence the block will be accessed less and wears down slower. This approach is called static wear levelling. Table 2.1 gives an overview of the different approaches.

<table>
<thead>
<tr>
<th>Item</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Endurance</td>
<td>Longer life expectancy</td>
<td>Shorter life expectancy</td>
</tr>
<tr>
<td>Performance</td>
<td>Slower</td>
<td>Faster</td>
</tr>
<tr>
<td>Design complexity</td>
<td>More complex</td>
<td>Less complex</td>
</tr>
<tr>
<td>Typical use</td>
<td>SSD</td>
<td>USB flash drives</td>
</tr>
</tbody>
</table>

*Table 2.1: Comparison of different wear levelling techniques*

Good wear levelling can increase the life expectancy of the memory from a few days to a couple of decades, depending on the application [9].

Dynamic wear levelling keeps track of the erase cycle counts of all the blocks and indicates which available block has been erased the least. This will ensure that wear levelling of the dynamic blocks is done evenly, however, if there is a large fraction of blocks that contain static data, then this approach will wear down the dynamic blocks very quick while the static blocks are hardly ever erased.

Static wear levelling can be done by applying the dual pool algorithm. This algorithm moves cold data from young blocks to older blocks. This will allow older blocks to cool down, as the cold data is less likely to be accessed. For this approach the flash memory is divided in a hot pool, which contains blocks with hot data, and a cold pool. The algorithm determines the difference of the block with the largest erasure cycle count of the hot pool and of the block with the smallest erasure cycle count of the cold pool. When this difference is too big, it swaps the data in the
blocks. Both of the blocks are then put in the other pool. It takes a considerable amount of
time for the memory pools to become stable, and the algorithm does not consider any cleaning
functionality, but overall this approach does result in a even wear distribution.

Another static algorithm is rejuvenator. The approach seems similar to the dual pool algorithm,
with as main difference that it explicitly identifies hot data, instead of indicating which pages
contain hot data. To this extend it will make use of a hybrid FTL, where hot blocks are mapped
at a page level, and cold blocks on a block level. Rejuvenator uses one list where every entry
in the list corresponds with a number of erase cycles. Blocks are stored in the list at the entry
 corresponding to their erase count. Every time a block is erased it moves one position further
in the list. Entries in the list may contain no block at all. Between the lowest not empty entry
and the highest not empty entry a point $m$ is defined. This point $m$ is used as a threshold to
discriminate not often accessed blocks from frequently accessed blocks. The algorithm tries to
store as much hot data in blocks with an entry below $m$, and as much cold data in blocks with
an entry above $m$. The essence of this algorithm is that the number of entries and therefore the
possible age of the block can be tweaked, as well as the fraction of hot and cold data migration.
A drawback of this algorithm is that it uses a lot of storage\[19\] [20].

Moving all the data around has an impact on the performance. When the wear levelling algorithm
is modelled, the access pattern of the disk is of great influence. Not only the access pattern of
the transactions that are happening, but also that have occurred in the past. The initial state
information of the disk is of equal importance, as it influences the hot or cold state of the data.
Modelling the wear leveller without this knowledge is not reliable, and has no guarantees about
what would have happened in real life.

### 2.5 Management

There are different configurations to organize the responsibility of handling the issues. These
configurations can be divided into three categories: unmanaged, partially managed and full
managed flash devices. These options are shown in Figure 2.3. In the previous paragraph the
issues and their solutions have been introduced. In essence the following strategies should be
applied:

- Error correction
- Block management (assuring all is treated equally)
- Wear levelling

If it is chosen to handle all these strategies in the host device, e.g. the processor, then the
controller is unmanaged. This is a burden for the host device, but it will remain fully in control,
and will therefore always know how the data has been stored inside the memory device.

Another option is to let the memory device handle the error correction. This means that the host
device does not have to check correctness of the blocks after every read, and takes a significant
part of the burden away from the host. Still the host should control its read and program commands such that errors are rare and other issues are omitted. Another downside is that this solution will cost extra hardware. This method makes the memory device partially managed.

Another option is to let the memory device take care of all the issues in the flash. This lets the host control the memory as standard non-volatile memory. Unfortunately it is not totally clear anymore if every memory operation that happens over a number of memory locations that are sequentially addressed according to the host, can also be sequentially approached according to the memory device. The wear levelling control, and the command management could result in jumbling the memory addresses around. This will make the behaviour of the device over time unpredictable. In general however the access time will improve, because reads and programs do not have to occur very often for non-volatile memory, which gives the processor extra time in the long run since it does not have to handle with any issues any more [21].

It is important to be aware of the system configuration of the flash device. If an unmanaged flash device is used with a controller configuration of a managed flash device, then the configuration will not work because the assumed handling of all the maintenance functions will not be executed. On the other hand, if an unmanaged device is assumed by the controller and the system turns out to be a managed device, then configuration will work. However, since the burden of maintenance functions is moved to both the devices, the performance will be worse.

2.6 Simulations

When discussing simulators a few things are important to consider:

- Accuracy
- Precision
- Run time
- What is simulated

The difference between accuracy and precision is subtle. Precision means that the outcomes do not vary significantly; multiple runs give the same outcome. Accuracy on the other hand means
that the results are close to the actual result; in this case the result in real life. Which one is more useful depends on the application. One could argue that as long as the result a known bias throughout the experiments it is useful, because it shows the right trend when multiple settings are examined, and one could reason what the actual result is going to be; hence precision is important. On the other hand, as long as the accuracy is acceptable, one could perform the same experiment multiple times and average the results.

When simulators are used in projects one key selling point is run time. Assume that both the precision and accuracy could be ideal, when it takes a very long time to run the experiments, it is not going to be much use if the project deadline is approaching in near future.

2.6.1 System exploration

The question “what should be simulated?” is less trivial than it appears to be. A lot of simulators seem to give performance figures just based on the processor behaviour. But the processor may stall, waiting for I/O which hasn’t been modelled. Given that most embedded systems nowadays contain multiple processors, it seems strange to base the system performance prediction on only one of them.

Usually one would want to verify the design of a system. This has a functional aspect, but also a performance and energy aspect. With different subsystems in one system this is hard, and a simulator can be useful in this case. The advantage of a simulator over a device in real life is that the simulation model is easily altered. If the memory turns out to be a bottleneck, then the memory parameters can be changed without cost and the experiment can be run again to verify this theory. When a real device has been built then the modification of the device will always come at a cost. Another advantage is that in a simulation model measuring points can be added whenever and wherever needed. This is convenient when a system has multiple abstraction layers, for it gives a clear indication where the bottlenecks are.

The simulator in this research should be able to simulate not only the system but the flash disk and the flash interface as well. The system should consist out of a realistic memory hierarchy model and I/O attributes that one would expect in a mobile device. In short it should be able to simulate a mobile platform. There is currently no simulator that models a UFS interface available. There are a few flash disk simulators available, but those do not consider a complete system.

The best way to investigate the impact of UFS is to extend an existing full system simulator. The goal of the model would be to show a trend when the system is modified, and to indicate what causes the delays. This does not demand a cycle accurate model were every transistor in the system is modelled. It is important that the different transactions behave in correspondence with real life.
2.6.2 gem5

The gem5 simulation infrastructure came into existence by combining the best aspects of the M5 simulator with the best aspects of GEMS. Gem5 is a modular discrete event driven computer system simulator platform. As a consequence, one can rearrange and extend the different system components in such a fashion that it precisely models the system one wants to simulate. The passing of time is simulated as a discrete series of events. The flexibility of the simulator is one of its important features. It supports different platforms such as x86 and ARM processors.

Currently the gem5 simulator includes two different memory system models, Classic and Ruby. Ruby is a model that was originally used in GEMS, and it provides a flexible infrastructure which allows accurate simulations of cache coherent memory systems. The classic model was originally designed for M5 and provides a fast and easily configurable memory system.

Gem5 supports several I/O devices such as timers and network interface controllers. In the current version common device interfaces, such as PCI, are supported. This simplifies implementing new devices and avoids code duplication. Examples of device models that are currently implemented are IDE controllers and NICs.

The simulator is still under development and not all functionality has been implemented yet. One of the aspects that is currently worked on is a good power model. External power models such as McPAT have been used with GEMS and M5, and indeed have been made usable under gem5. However at the moment a more comprehensive, modular, and integrated power model for gem5 is being worked on.

Gem5 does not support eMMC or UFS interface controllers yet. It also lacks performance models of flash memory. The modelling of this memory is challenging because the behaviour depends very much on how the memory has been handled in the past. The memory controllers will react differently depending on how they manage the flash. To get the right performance and omit accessing different parts of the memory too often, extra hardware will be added with this controllers. This will make the behaviour very opaque and hard to simulate, but at the same time this is the main reason that it has to be simulated to understand what is happening[22].
Chapter 3

UFS: A new interface

The interface to flash devices within mobile systems is about to be upgraded. Currently the standard for flash devices within mobile systems is eMMC. The new interface that is introduced by JEDEC in June 2012 is called Universal Flash Storage, UFS. This standard attempts to bring one universal controller standard to suit all the flash devices, by trying to blend the best of eMMC with SSD devices. It aims at having a higher speed than eMMC, maximally 1.25 Gb/s but it will be extended to 5.8Gb/s in the future. The power consumption is expected to be in the same order as that of eMMC.

The top level architectural model of UFS is given in Figure 3.1. This model shows a layered communication architecture. It is based on the SCSI SAM-5 architectural model [2].

The application layer consists of the Task Manager, device manager and the UFS command set handler. The command set handler handles, amongst other commands, all the read and write commands. UFS supports multiple command sets, but version 1.0 supports SCSI as a baseline protocol layer. A simplified command set has been selected for controlling the device. The application layer is handled by the SCSI driver, the transport protocol layer is handled by the device driver, and the interconnect layer is handled by the interface between the host and the device.

The system model is shown in Figure 3.2. On the host, an application wishes to communicate with the device. It will call the driver, which will use the host controller for this purpose.
Communication will happen by writing to the UFSHCI, which will involve writing to a set of registers.

UFS is able to offer a significant speed advantage over eMMC as it is able to queue the commands, and then communicate those asynchronously to the device itself. If a command has been initiated then the driver has no need to keep track of the status of the command until it has been completed. The transaction will take place and a response is given when it has failed or succeeded. An eMMC device should first complete one transaction, i.e. wait until the response has been sent, before it can continue. It is also possible to queue commands, which will give a great advantage in case of parallel flash devices. It is then possible that the queue is searched for a command for an idle flash device. This optimizes the parallel performance. The power advantage will be achieved by making the transfers as brief as possible, allowing the device to be in a sleep state for most of the time[2][23][24].

3.1 UFS Host Controller Interface

The UFS Host Controller Interface is the bridge between the UFS device and the driver. The Host Controller Interface Architecture defines two interface spaces. The first space is the Memory Mapped I/O space. This space is dedicated to a set of hardware registers which provide the interface between the Host and the system software. The registers in this space can be divided into three types:

- Host controller capability registers: These registers describe the host controller capabilities such as UFS version and the size of the command queue.
- Runtime and operation registers: These registers provide an interface to control the interrupts, the host controller status, the transfer request management, the task request management and the command management.
- Vendor specific registers: These registers implement some vendor specific options

The second space is the host memory space. In this space there are data structures that provide information about the commands for execution and their data buffers [25].
Chapter 3. UFS: A new interface

3.2 Communication

The communication between the software and the UFS devices is managed by the host controller, which provides three interfaces to handle transfer requests. The first interface is the UTP transfer request list. This list includes support for all SCSI command sets adopted by UFS, Native UFS command set and device management function via query request UPIU/query response UPIU (UFS Protocol Information Units). The list consists of a data structure which describes a command that is to be executed and the data that is associated with it. When a command is placed on the list, the software will ring the doorbell to ensure it will be handled. The commands on the list will be handled in the order that they are placed there. The devices however may cause the commands to be completed out of order. The host handles all the data transfer actions that are required to finish the transaction. While the host is running, the software can still add commands to the list. The host supports interrupt aggregation, which means that it can generate one interrupt for command completion for a subset of the commands on the list.

The second interface is the task management request list. This list is used by the software to implement a task management service access point. This list consists of a data structure which describes a task management function that the software system wants the attached device to execute. All the task management requests will be prioritized over the transfer requests. This list has a separate doorbell register which is rung when a new task is placed on the list. The handling is similar to the handling of the transfer request, with the exception of interrupt aggregation, which is not supported for this list.

The third interface is the command register, which allows the software to command the host controller directly. These transactions have the highest priority on the host controller.

3.3 UFS device

The UFS device is the actual memory card which stores the data, plus some extra functionality. It communicates with the UFS host controller via the M-PHY interface, and the UNIPro protocol. This protocol provides three services, namely data transfer, config/control/status service and interrupt/notification service. A UFS device may consist of multiple logical units, a device manager and descriptors. The device manager allows the handling of device specific tasks, such as power management, and the logical units perform actions like read and write. The descriptors store the configuration information.

The transfer of the packets is designed in such a way that the device will control the pacing and the state transitions needed to satisfy the data transfers and status completions of the request. The device will start the transaction when the conditions are optimal. The main benefit of this approach is that it cuts down logic and firmware needed in the host controller to support a device. With this method it is also possible to get an optimal bus transfer performance with a minimum of transfer actions.

The SCSI architecture model is used as the general architecture model for the UFS transfer protocol. This architecture provides a client-server model were the clients are called initiators,
in the case of UFS this will be the host controller, and the servers will be the targets, in this case the logical units within the UFS devices. Initiators and targets are mapped into UFS physical devices. This system does not imply that SCSI commands have to be exclusively used. A UFS device will contain one or more Logical Units. A logical unit is an independent processing entity within the device.

Communication between the initiator and the target occurs as a series of messages. These messages are formatted into UPIU. All UPIU structures contain a common header area at the beginning of the structure. The remaining fields can vary, according to the type of UPIU.

UFS devices are able to perform background operations when there is no transaction to the device. These actions include maintenance functions such as wear levelling and garbage collection. From this fact it can be derived that UFS devices are fully managed devices.

### 3.4 SCSI command set

UFS embeds a subset of SCSI commands. The SCSI command set has been chosen to enable UFS to be implemented easily in existing technology. If there is already a SCSI driver in the system, then there is only need to write a low level driver, without having to write of a lot of middle ware drivers. UFS is in essence designed to be protocol agnostic, which means that also other protocols can be used for the higher transportation layers. Nevertheless, in the first version of the UFS standard SCSI is described as the standard protocol[2], and the first published driver seems to follow that specification. For full functionality of UFS the following subset of SCSI commands need to be implemented according to the standard[2]:

- Inquiry: This command requests general information about a specific LU
- Read(6)/Read(10): Read information from disk
- Read Capacity(10): Read the size of the disk
- Report LUNs: Report the number and ID of LU available
- Start Stop Unit: This command requests the unit to stop, in order to load or eject the device, or change its power
- Test Unit Ready: This command is sent to test whether the unit is ready; this command is not a request for a self test
- Verify(10): Verification that the specific blocks mentioned in the command can be accessed
- Write(6)/Write(10): Write to the disk
- Format Unit: Request the device to format into the structure specified by the command parameters
- Send Diagnostic: Send information about the error code that has been generated
Chapter 3. UFS: A new interface

- Synchronize cache: This command ensures that the device has the up to date version of the specified blocks in its cache
- Mode select: Write to mode pages to set device options
- Mode sense: Request the mode pages to gain more information about the device
- Unmap: Unmap a specific set of pages; they have been deleted by the software system

The following commands are optional and do not have to be implemented. They are described in the UFS standard and will provide full control over the flash device when implemented[2].

- Read(16): Read disk
- Read Capacity(16): Read disk capacity
- Write(16): Write disk
- Read buffer: Read cache
- Write buffer: Write cache

3.5 Transaction flow

A typical transaction flow occurs as shown in the sequence diagram in Figure 3.3. The driver will initiate a transfer by searching for an empty doorbell register bit and filling the corresponding slot with a UTP frame. This UTP frame will have a field which contains a pointer to the UPIU data structure. After the data structures have been filled the doorbell bit is raised and with that, the host is notified of the message. The transaction itself has different stages where messages are being sent. The transfer of the UTP and UPIU frames is a DMA action. Since the frames are both about 64 bytes in length, the transaction time will be in the order of 0.1us to 1us. The second stage is the actual disk data transaction and disk transaction. This will take significantly longer, and depends on the size of the transfer, and the size of the page. Typically a read will take in the order of 25us per page, and a write 200us per page. The responses of the UTP and UPIU frames take the same order of time as the original messages, between 0.1 and 1us.

3.6 How to drive UFS

The UFS device driver for Linux was initially written by developers from Samsung. This driver connects the Linux system with the Host controller interface of the UFS device via PCI. From there the HCI communicates everything to the appropriate device, but this transaction will be opaque to the system. The driver that has been provided makes use of the SCSI standard to control the UFS controller, including some UFS specific commands. To transfer data to the device it makes use of DMA.
3.6.1 Initialization and Removal

Every Linux device module has to have at least two functions: A module initialization routine, and a module removal routine. The functions that act as such are defined by special kernel macros that are called by “module_init” and “module_exit”. In the UFS driver, these functions call “pci_register_driver” and “pci_unregister_driver” respectively. The PCI driver that is registered has a register field, describing among other things a name, a PCI table and a probe function. The PCI table registers how a certain device can be identified, such that when there is a match, the probe function can be started.

3.6.2 Probe Function

When the probe function is called, the kernel will provide this function with a device ID as well as a device handle. The function will then enable the device and check whether this action has been successful. Enabling the device will wake the PCI device up and should be called in any PCI probe function before accessing the PCI address regions. After the device is successfully woken up, The bus master functionality on the device will be enabled. This will allow the device to initiate DMA transfers, without interference of the processor.

Now that the Host Controller Interface is set as PCI master it can be registered as SCSI host. If that is successful, then the PCI address regions can be allocated. With this function the memory regions of the PCI device is associated with this PCI device and be safely accessible for the driver. It is then evaluated whether these regions are actual memory locations instead of IO regions.
Next thing that needs to be initialized is the DMA functionality. It will be tested whether the PCI device can support it and the DMA mask will be set. This DMA mask will identify which address lines will be associated with the PCI device. When all this is achieved the host bus adapter field will be filled in with the most up to date information, all the capability fields will be read and the memory space for the host will be allocated.

The SCSI host then needs to be configured such that the capabilities of the host device are known. This initialization considers things like maximum number of Logic Units, maximum number of channels and the maximum size. After this has been completed a waiting queue for SCSI tasks is created. This queue is initially empty.

For transactions from the PCI device, interrupts and tasklets are generated. This will allow the PCI device to actually generate, start and complete a transfer. Work queues are created along with these to queue commands, errors and transactions. A tag map is then created to keep track of all the different items in the queues. If anything goes wrong during the execution of the probe function, then that will result in undoing anything it created to that point and returning an error value. If everything is executing without errors, then the probe function will return zero.

### 3.6.3 Tasklet handling

Whenever a SCSI or query command needs to be completed a transfer request completion function is initiated. This function will ensure the handling and completion of transfers. UFS has a register which is called the doorbell register. The purpose of this register is to allow the system to indicate if there is a command on the waiting list that can be executed. Each bit in this register corresponds with a slot in the transfer request queue. If a bit is zero then this indicates a free slot, if a bit is one, then that indicates a slot that is occupied. At the beginning of this function the driver will check whether the register has been updated and updates the local outstanding request indicator accordingly. This update will happen atomically to ensure that no transactions that are not yet known by the doorbell register are deleted.

For all the completed requests the driver will determine the overall command status of each of the completed requests. When a request has been handled successfully according to the Host Controller interface then the SCSI status of the message will be tested. The SCSI driver in the kernel will be notified of the status of the transaction. If a transaction was not completed successfully then the appropriate SCSI error messages are generated.

### 3.6.4 Interrupt Handling

The host controller will generate an interrupt whenever it has finished a command, a task, a transfer, or when it has encountered an error. The driver will handle the interrupt accordingly.

Whenever an error occurs, the driver will try to determine what kind of error has occurred. If it is a fatal error, then the driver will reset the controller. If it was a command error, and this error is related to an error in the physical layer, then the driver will reset the controller as well. In all other cases, no additional actions are taken.
In the case of command completion the driver ensures itself that the command has been handled well. In the case of a device that is starting up the driver makes sure that the device is successfully enabled and ready to use. When a task that has been scheduled is completed, a dedicated handler will take care of his removal from the queue. It wakes up sleeping tasks and removes the ones that are completed.

3.6.5 Issuing Commands

The driver supplies the software system with a function which makes it possible to generate its own commands in order to control the task management. The working of this function is as follows. Firstly the doorbell register is checked. This evaluation will check whether there are any available slots that can be used for the new command. When the queue is full then an error will be returned. If the queue is not full, then a free slot will be selected and the task request descriptor will be constructed.

When this is finished the task request is configured and built to be readable for the UFS Protocol Information Unit. Ultimately the command is send of to the controller interface. When this happens the doorbell register bit that corresponds with the position that in the queue where the task will be put is set to one. The function waits until the action has been completed. When this action has been completed, then the function calls the task request completion routine and terminate.

3.6.6 General Data Transfers

All the data transfers are done via SCSI and DMA. By registering the device as a SCSI host, the operating system has a way of handling the device. The SCSI driver will eventually call the function “ufshc_d_queuecommand_leck” when communication with the UFS device is needed. When the SCSI midlayer approaches the driver it will supply the driver with a SCSI command structure and a pointer to the field where a completed transaction can be indicated. The function will fill in the necessary transaction structure and test whether the device is occupied or not. After the UFS Protocol Information Unit has been set and the DMA controller has been initiated, the driver rings the doorbell of the device.
Chapter 4

UFS Flash Model

The Flash model is a major contribution of this thesis, programmed in the language C++ as part of the open source BSD simulator gem5. Its design has the layered structure as shown in Figure 4.1, where each layer is modeled as a class. The model is written in the files “UFSDevice.cc” and “UFSDevice.hh”. The files contain 1855 and 644 lines of code respectively. The model has been created from scratch, and the development of the model took about four months. It has been tested by making use of the UFS driver that is included in the Linux kernel 3.4. The model inherits functionality from the DMA device in gem5, which provides the model with the functionality for operations via the PIO bus and with an interrupt. As an input the model needs a disk image file. The disk image can be any format that is supported by the software system that is used in the simulated environment.

The top layer provides the interface with the simulated system and resembles the host controller. The simulation system is connected to the model via the PIO bus and via the DMA bus. The device that initiates the transactions via UFS will approach the model via the PIO bus and set the appropriate registers. All the necessary data will then be transferred to the device via the DMA. This is in accordance with what would happen in the real world.

The middle layer models the device itself and determines which actions are appropriate to take. This section includes SCSI message handling, reading the disk, writing to the disk and all the device specific tasks that need to be handled, such as providing the initiator with information about the capabilities of the device.

The lowest layer provides the model with a method of calculating the appropriate waiting time. This functionality includes keeping track of random and sequential reads, determining when the garbage collector is invoked, and the basic consequences of any action.

The architecture of the model is chosen such that it represents the real world as realistic as possible. There are of course, as with any abstraction, some simplifications in comparison with the real world. During this section the simplifications will be discussed, disputed and the choices that have been made will be defended.
4.1 Host control layer

This layer tracks every stage of the transaction. As explained earlier, UFS makes use of multiple stages of transfers. Every transfer would need an DMA action to complete, and this means that the host controller should wait for completion, before it can continue. This has been achieved by creating finish events for every transaction.

The first action is the setting of the doorbell register. The initiator sets the appropriate bit in this register, which evokes an action from the host controller. The controller will determine what has changed, and what actions should be taken. It then starts, if necessary, the first transaction.

When the first transaction has completed the transfer handle function is evoked. This function has the information about the physical location of the next data structures. Hence it can initiate
the next transfer, which will transfer a device command, information about either the initial or
the final location of the data that needs to be transferred, and the data location of the reply
that will be send.

When this transfer has been handled, then the function which calls the SCSI function and handles
the demanded data is started. This function can handle four different scenarios.

1. Data needs to be read from the disk
2. Data needs to be written to the disk
3. Information about the device needs to be transferred
4. No transfer of data is needed

This function ensures that the right data is selected and transported to the right destination by
starting the appropriate transfer handle action. Regardless of what action is started, eventually
the replies will be constructed and sent.

The first reply is transported from the device to the response UPIU structure. After this the
second reply is scheduled. Due to the nature of the DMA functionality in gem5 it is advisable,
when sending two messages, to send the second one at least a tick later than the first one; this
because there are no guarantees when two messages are scheduled within the same simulation
tick about the order in which they will be handled.

After both messages have successfully been transferred into the main memory the DMA function
will initiate an event. This event generates an interrupt and all the memory allocated to facilitate
the transfer is released. This is the last event in the transaction. The number of interrupts
generated will be tracked to provide an indication of what went wrong if the system crashes. For
effective use of this indication the number of interrupts in the driver should also be checked.

4.2 Queueing system

There are a number of events not discussed in the previous section. These events are the data
transfer events, and take care of the logistics of read and write transactions. Such a transaction
can be split up into a disk action and a DMA action. In fact, two events need to be created to
control the timing handling of both of the events. An overview of this set up is shown in Figure
4.2.

When a new transaction is scheduled, the first action is to check whether the queue is empty.
If the queue is empty, then the first event will be scheduled and the event information will be
pushed to the queue. If the queue is not empty the event information will be pushed to the
queue.

Whenever an event is triggered, the event will pop its information from the queue, and if the
queue is not empty schedule the next event. It will then resume its tasks. These tasks vary
per event, but ensure the actual transportation flow of the different transactions through the queue. The general idea of the queueing system is to track the position of the messages, and not necessarily do the actual transfer to and from the disk image.

For every transfer the amount of transferred data is tracked such that the finalization event can be started at the correct moment.

4.3 Device layer

The middle layer provides the device functionality. This means the handling of the SCSI messages, and the reading of the disk image. This layer makes the distinction between the transactions that might have an influence on the disk data, and the transactions that influence the device. It includes test functionality to inform the initiator about the status of the device, and provide warnings for errors that may occur along the way. The transactions to and from the disk model will be performed by doing file operations to the disk image file. At the end of the simulation the resulting disk image can be inspected, to see the result of the executed transfers.

Not every message affects the device, there are commands that are answered, but due to the unmeasurable result at the initiator side, and considering the complexity it would impart, it has been chosen to abstract from these messages. These messages are related to generating device information rather than actual functionality. An example is the Mode sense command. This command should provide up to date information about the device. Instead a standard data packet is being replied, without any further tests.

4.4 Flash layer

The lower layer provides the calculation of the access times. It keeps track of the page table and determines whether an access was random or sequential and if the garbage collector has
been activated. For every action the appropriate delay will be calculated which is then used to schedule events in the higher layers.

If every page has to be tracked in the model then the page table that is needed will grow rapidly when bigger disks are used. For a good page table, the evolution of the disk image, i.e. in what order was every file placed on the image and were there any remove actions in the process, should be known. This model is provided with a disk image that has already been created, so it is unknown how the image has been built, or how old the disk actually is. It has to be assumed that the disk is in perfect order when it is attached to the system. Because of this it can not be determined how the disk would have evolved when the page table had been known, and thus the timing calculation is always incorrect. Resolving this issue is left for future work.

Because the assumption has been made that the disk is perfectly mapped at the start, the initial accesses will most likely be all sequential and the performance will be higher then in real life. This model is thus optimistic.

Another issue with the disk is that since an image is used, it is unknown to the model where the empty pages are. This makes the modelling of page replacement strategies quite a challenge, because it does not know what destination can be used for the pages. The best way to deal with this problem is to implement an initialization which maps the disk and creates tables according to the information that it finds on the disk. However, to build a file system decoding algorithm is quite time consuming, and is therefore outside the scope of this project. Since the model is optimistic, the solution for this issue needs to be optimistic as well.

The operating system can interpret the file system, and does have information about where the free addresses are. Every first access to a new page will be considered to be an action to an empty slot, from a timing point of view. After that, the location is considered to be known. When a write action is made to a known page, it will increment the invalidated pages count of that block with one, and the page map address is set to be higher than the actual number of pages per block. This ensures that no sequential access can be made to this page. When a block consists completely of invalidated pages, it has to be cleaned. A parameter set by the user defines how much the system will be influence the garbage collection. After a clean, all invalidated pages are reset, and all page addresses are set back to their original location. This improves locality, which is what happens in real life garbage collection algorithms.

The moment at which the maintenance functions are activated in the simulation model, might not be the same moment as in real life. A garbage collection algorithm might chose to erase a block when it does contain free pages, because it estimates that this is the most cost effective action to take. For that same reason it might decide to leave the block as it is, to erase it at a later time. Since it is unknown how the garbage collector within the device would operate, this abstraction has been made. The abstraction is optimistic, since it does only erase a block when it is needed, and it improves spatial locality in the process.
Chapter 5

Experimental setup

The UFS flash disk model is an extension of gem5 that enables the exploration of different system configurations that make use of a flash disk. This enables the research of the performance of mobile systems. It is now possible to find out where in the system the bottleneck is, as the simulation model allows the user to probe between every abstraction layer. This chapter describes the experiments that have been set up to demonstrate that the model performs realistically and to show the impact of flash memory on the system performance. The simulated system is very configurable. But since time is as always limited, only a small selection of configurations can be verified.

This chapter will provide a recipe of how these experiments should be done. The chapter is divided into three parts. The first part explains the system configuration, the second part explains the operating system configuration and the third part discusses the different workloads that have been run in the simulated environment.

5.1 Gem5

To set up the python script in such a way that a realistic system is being simulated, the ARM Realview PBX system configuration has been used. This configuration is extended with the model of the UFS device, which is linked to the PIO bus at address 0x1c1b0000. The interrupt number that has been assigned to this device is 48. The values of this device have been chosen for no particular reason other than that these values were still unused in the system. The alternative disk model, the IDE disk, is still attached to the system. To monitor all the bus activity a communication monitor is attached between the UFS device and the PIO bus. The sample period of this communication monitor is set to be 100 simulated ms. As extra metrics the following UFS model statistics have been added to gem5:

- The amount of UFS UTP commands that are currently unanswered in the host controller
- The read/write queue length
• The total amount of data read from/written to the disk
• The average bandwidth used for reads/writes
• The average number of UFS UTP commands that are unanswered in the host controller
• The average read/write queue length

The system setup is shown schematically in Figure 5.1. The system is set up with the following configuration:

• A CPU model of a cortex A15 processor, (–cpu-type arm,detailed);
• A clock frequency of 2GHz;
• L1 instruction cache size of 32kB;
• L1 data cache size of 32kB;
• L2 cache size of 2MB;
• Physical non volatile memory of 64MB (IOzone) or 256MB (BBench);
• Disk size of 512MB (IOzone) or 1GB (BBench);
• Flash disk model has one die and one LUN;
• 64 pages per flash disk block;
• 2kB per page;
• 25 us read latency;
• 200 us write latency;
• 1.5 ms erase delay;
• 50% garbage collector influence.

![Figure 5.1: System setup](image-url)
5.2 Operating system

Since a new device has been added to the system, the Linux kernel needs to be modified. From kernel 3.4 onwards a workable UFS driver has been provided with the kernel. However, this driver assumes that the model is connected via a PCI interface. This is because this is an initial version of the driver. Future versions should support UFS host controllers connected via AMBA or connected as platform devices as well. The patch to support this is already available in the Linux patch queue at the moment of writing, but has not yet been applied to the kernel.

To modify the kernel the specific C files which define an ARM Realview PBX system should be accessed. These files can be found in “arch/arm/mach-realview/realview_pb.c” in the kernel directory. This file contains the registration functions of all the devices associated with the platform. When a device is not registered and not hot pluggable, the probe function will be unable to find it. When the probe function does not find the device, it will not be possible to use the device. It is important to assign the UFS model at the same address and use the same interrupt number as defined in gem5. Registering the device should be done in the function “realview_pb_init” by calling the function “platform_device_register(&PD_struct)” where “PD_struct” is the structure which defines the platform device structure of the UFS device. The kernel then needs to be compiled. In the kernel used for the system tests the IDE device has been removed, to make sure that this device does not interfere with the system.

The Linux kernel by itself does provide an Operating System, but without a file system. The file system that can be attached to the UFS model will be in the form of a disk image. The images available on the gem5 website will work with this set up. For the benchmark the image of the Linux AEL system and the Android Gingerbread disk. These images have been enhanced with benchmarks.

5.3 Benchmarks

To capture both general system performance numbers as well as user performance experience benchmarks are run on the simulated system. The different benchmarks, and what those may show about the system, are discussed in this section.

5.3.1 IOzone

IOzone is a file system benchmarking tool, which uses multiple tests to measure the disk performance. These actions include: write, rewrite, random write, read, reread and random read. The benchmarking tool is useful for full system analysis, because it can be linked in the application layer. It communicates via different system call functions with the lower layers, and is therefore able to show the overall impact of all the different layers combined. The benchmark has different parameters to influence the behaviour. This allows the callee to get the output data in an excel compatible format, to select specific tests, or to define the file sizes that are to be used.
IOzone has been added to the Linux AEL image. The command that has been used to start the benchmark is "./iozone -Ra -g 128M -i 0 -i 1 -i 2 -Q" This starts IOzone with excel data generation, a maximum transfer size of 128MB, latency numbers, and with read/write, reread/rewrite and random read/random write tests enabled[26].

Normally it is not advisable to run IOzone for flash devices. This is because the life expectancy of a flash device will decrease significantly after so many write operations. Since this is a simulation, there is no need to consider this disadvantage.

5.3.2 BBench

BBench is an automated benchmark, created to test the rendering performance of a browser. It achieves this by letting the browser show a sequence of the most popular web pages of 2011. Only rendering is tested with this benchmark, so a network connection is not required. With the selection of pages that need to be rendered it has been attempted to cover a selection of web pages with a diverse style, and the most visited websites. With these criteria it is expected that the benchmark is not only thorough in testing the technologies a browser supports, but also relevant to the experience of the end user. The following web pages have been used:

- www.amazon.com
- www.bbc.co.uk
- www.cnn.com
- www.craigslist.org
- www.ebay.com
- www.google.com
- www.msn.com
- www.slashdot.com
- www.twitter.com

The websites have been captured using HTTrack. This tool, allows the user to download complete websites including pictures and JavaScript. The browsers built in JavaScript engine is used to automate the navigation between pages. The benchmark moves to the next page as soon as the onLoad event is triggered. This event scrolls down the page to ensure complete rendering, and then loads the next page[27].

As shown by Kim et al.[4], the browser performance is dependent on the disk accesses, and thus on the dis performance. Benchmarking browser performance can therefore show more information about how the disk would be used in a practical environment.
Chapter 6

Results

As described in the previous chapter, two types of experiments are run, to give a general overview of the performance of a mobile device. As described there are two benchmarks used to accomplish this, IOzone and BBench. These benchmarks show both the performance at user level, namely Browser performance, as well as system mechanic performance, such as experienced throughput. This chapter will present the numbers that were collected from the experiments, and explain them.

6.1 General analysis

In this section we will analyse the general performance of the file system from an operating system perspective. To create an as plain as possible operating system, a Linux AEL system has been used. When IOzone finishes its run on this platform, the graphs given in Figure 6.1 and 6.2 are generated.

The highest achieved throughput is 1.5GB per second for reads, and 940MB per second for writes. These high throughputs can occur, because the amount of data that is transferred fits in the physical memory or, in the case of the write transaction, in the level 2 cache of the system. This system configuration has 64MB of RAM, and Linux reserves 16MB of this memory for DMA write transaction. Since the file is first written in IOzone and then read back, smaller files can be read back from the physical memory. This memory is 64MB minus 16MB for DMA transactions in size, and that is the reason of the quick decline when the file size is enlarged from 32MB to 64MB. For the write transaction the decline starts gradually after 2MB file sizes, which is the size of the level 2 cache, and seem to reach the lowest performance with a file size of 16 MB. The performance that remains is the performance of the flash disk itself. In the case of write transfers this is 9MB per second and for read transfers this is 80MB per second. Considering that the page size is 2kB, this translates to 4.5 kIOPS and 40 kIOPS respectively. Given that every page access takes 25 us for a read action and 200 us for a write action, the expected performance would be 40 kIOPS for reads and 5 kIOPS for writes. It can be explained that the write is a
little lower, as it suffers more from the garbage collector functionality. Thus it can be stated that UFS disk model behaves in a way that could be called predictable if it was a real device.

Drawing the graphs as multiple lines, Figures 6.3 and 6.4, the effect of the chunk sizes can be observed more clearly. The gradual increase in throughput with increasing chunk sizes is largely due to a smaller overhead. Every transfer has to be done to the appropriate section in the memory at first, and if this can be done in less actions then this will mean less iterations, and
less interleaving of read and write commands. After the file size grows too big to fit in the cache or the memory, the throughput experienced on a system level drops. The IOzone seems to have suffered an interference while transferring a file of 64kB in chunks of 16kB. This is considered to be a measurement error of IOzone.

During read transfers (Figure 6.3) all the files with a file size smaller than 32MB fit nicely within the physical memory and can therefore increase their speed, until the optimal point is reached. A file of 32MB is the turning point in this process. When the chunks get bigger than 1MB then the throughput decreases dramatically.

The write transfers seem to fine their optimal transfer point for transfers of 2MB, with a chunk size of about 1MB. This seems to correlate with the level 2 cache size. File transfers of 4MB and 8MB seem to have an interesting increase at 128kB chunks and 256kB chunks respectively. When it is calculated what this actually implies one finds that this means that the number of transfers at this point is equal to 32. If the chunk size is increased, then the number of transfers decreases. 32 is the number of transfer slots available in UFS. After this turning point the driver does not have to wait for the transfers to finish to schedule a new transfers, but can dispatch the complete batch at once. When the file size is increased to 16MB this is no longer possible, because this file plus overhead is too big to fit in the DMA memory. This means that as long as the physical memory can accommodate the data that needs to be transferred, it pays off to fill it as much as possible. Another conclusion that can be derived is that it pays off to have larger transfers rather than many transfers. This is consistent with the advice of the UFSA.

Consider the throughput as seen at a disk level, that is gathered by the statistics included in the queue system of the model at the disk level. These statistics are gathered during runtime by sampling and reseting the values every 100 simulated ms. In Figures 6.5 and 6.6, the number
Chapter 6. Results

Figure 6.4: Throughput of write transfers per file size

Figure 6.5: Amount of data transferred during read actions, and the number of instructions executed per 100 ms interval; while running IOzone

of operations and the amount of transferred data is shown. This is the visualization of the boot process of linux and the execution of IOzone with file sizes up to 4MB. IOzone is started at 4.2 seconds. Interestingly, there do not seem to be many read transfers (Figure 6.5), except one of 1kB. However, a significant number of write transfers can be seen. This is according to expectations, and shows that the data is indeed fetched from other places in the memory system, and not from the disk, when it is read. There are a lot of file transfers to the disk during the IOzone run, which means that the files are indeed written to disk.

The latency between issuing the command and the execution of the command as experienced by the application, has an effect on the user experience as well. In Figure 6.7 the latency for transfers of 16MB has been drawn. The value of 16MB has been chosen because it seems to be a corner case in this set up. The write performance has drop to its minimum in this set up whereas it is also the first transaction where the read performance is beginning to decline. When considering small transfers, in Figure 6.8 transfers of 4kB, the latency seems to be low, apart from a few outliers, where the garbage collector has been activated.
Figure 6.6: Amount of data transferred during write actions, and the number of instructions executed per 100 ms interval; while running IOzone

Figure 6.7: Latency experienced per offset in the transferred file, for 16MB transfers with a chunk size of 4kB

Considering larger transfers such as shown in Figure 6.9 which shows chunk sizes of 2MB, the latency has increased. Note that the y axis of this graph has a logarithmic scale. The x axis differs from the previous graph because there have been less transfers in this setup. With a file of 2MB one will only access eight different offsets in the file, whereas with 4kB one will access 4096 different offsets. The difference between the read and write transactions has become bigger, as the number of bytes that has been transferred increases. This is to be expected, since more data is being transferred so the access latency difference between reads and writes is amplified. In the larger write transfers however there seems to be some additional overhead for transfers at a certain location.

6.2 Mobile system analysis

Now the system has been explored on kernel level, the disk image is changed for a Android Gingerbread disk image. Android operates at a higher level, and introduces a number of extra abstraction layers to the system. The effect of these layers will be analysed in this section.
Figure 6.8: Latency experienced per offset in the transferred file, for 16MB transfers with a chunk size of 4kB

Figure 6.9: Latency experienced per offset in the transferred file, for 16MB transfers with a chunk size of 2MB

Figure 6.10 plots the number of instructions per sample, versus the number of data that has been read and written to the flash disk respectively during the boot sequence of Android. These statistics are gathered in the same way as in Figures 6.5 and 6.6. One sample is 100 ms long. The number of instructions that are executing plummets down at the beginning of the simulation. This is during the boot procedure where the disk is being loaded to explore the file system and to start Android. The number of instructions that has been executed in the interval starts to increase the moment that the data is transferred. This implies that the software was waiting for the data during this process.
From around 6.2 seconds the number of instructions decreases again. At this point in time the benchmark waits for 10 seconds before it initiates BBench. At 15.6 seconds BBench is started, 9.4 seconds later. The BBench run is shown in Figure 6.12.

During the run, the data seems more random and chaotic. However, when it is inspected, it becomes clear that every peak correlates with the beginning of a data transfer, the processor seems to be stalling a number of times for data transactions. Also the data transfers are very sporadic. There are a lot of data transfers and, given that no sample seems to reach the maximum bandwidth as established with IOzone, the suspicion rises that those transactions could have been bundled together. Take for instance the group of read transactions at the 15.6 second mark, these transactions each reach barely 1MB each, while the maximum data transfer that can be get in one timeslot is, according to IOzone, 8MB. Thus these small transfers in quick succession could probably have been combined in one eighth of the current number of transaction. The UFS standard has been designed to handle a small number of transfers with a large amount of data per transfer. The question is whether the small write transactions could have been combined to one large transaction. According to [4] this is due to the fact that Android uses SQLite to keep track of the application data. The problem with this is, that the SQLite functionality seems to store everything on the disk. IOzone shows that the best performance is achieved by keeping small transactions in the physical memory and not directly on the disk. If there are many small
write transfers, then Figure 6.4 shows that the performance is worse. Application performance will suffer due to this, as it becomes clear that the system waits for these transactions to finish. Also the flash disk need to be active a great part of the time. If it can not enter its sleep state, it will consume significantly more power than expected. This is another reason why the transactions should be bundled.

6.3 Discussion

Gem5 is a simulation system that is in development. There is not much work available on the validation of the simulation results, so the accuracy of the simulations is unclear at this point in time. This is the Achilles heel of the project. If the assumptions, or in this case, the environment is not valid, then what has been proven?

The validation problem holds especially for the UFS model. There is no UFS interface that has been created yet, so how will the accuracy be defended? At best it can only be considered a reasonable method for initial design space exploration.

Nevertheless, it is clear that gem5 yields results that give a good indication of the system performance in general. This is supported by the most recent validation work that has been performed on gem5. This work concludes with the notion that although the DRAM model is too simplistic, the overall accuracy remains satisfying, i.e. varying between 1% and 20% difference with real life scenarios, which means that it is a suitable simulator for initial exploration of the system design [28].

One could wonder whether it was wise to work with an optimistic model. This is useful to show the horrible performance of a model, because it tells you it is not going to get any better than
this. It does not tell how well it can perform, because it is impossible to know if it is possible to get to this performance.

During this document it has been claimed that between every abstraction layer the behaviour could be measured. This still holds, but it has not been measured between every layer. One could claim that some behaviour remains unobserved by the measurements. The results show however a clear overview of the status of the system.
Chapter 7

Related work

This work makes a contribution to a number of research fields; Flash simulators, system analysis and solutions. Table 7.1 gives an overview of the differences between the contributions that have been made by this thesis and the related papers.

7.1 Flash simulators

There are not many flash disk simulators available, and most of them focus on cycle accurate models. Related work where a full system simulator embeds a model of an SSD is even more rare, and no published work could be found of a full system simulator which focussed on performance and embedded an SSD model. In this work three disk simulators are discussed. The simulators presented are to low level to be embedded in gem5. But in the future they may be used as guideline to make the current model more detailed, and more realistic.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Thesis</th>
<th>[29][30][31]</th>
<th>[32]</th>
<th>[4]</th>
<th>[33]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>Transaction</td>
<td>cycle</td>
<td>real life</td>
<td>real life</td>
<td>Empirical</td>
</tr>
<tr>
<td>Modifiable</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>Arguably</td>
</tr>
<tr>
<td>Full system</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>UFS support</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>General performance</td>
<td>nozone</td>
<td>traces</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Mobile performance</td>
<td>browser</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 7.1: Comparison of related work
7.1.1 Flashsim

One of the simulators that can simulate flash behaviour is Flashsim. Flashsim is an event-driven modular simulator, designed to capture Solid State disk behaviour. It focuses mainly on NAND flash because this is according to the developers of Flashsim the most popular flash type.

Flashsim makes use of the Flash translation layer (FTL), which is mainly composed of three software components, namely address translation, garbage collector, and wear levelling. From a developer’s point of view however the Flash translation layer is mostly thought of as an address translator. The mapping table of this translation is stored in a small and fast on board SSD RAM in the physical world. An overview of all the components that are simulated is given in figure 7.1.

![Diagram of hardware SSD simulator](image)

**Figure 7.1:** The hardware diagram for the hardware SSD simulator

The overall structure shows apart from the FTL, the wear leveller and the garbage collector, the actual SSD divided into different components. The package class represents a group of flash dies that share the same bus channel. A die is a single flash memory chip that is organized in a set of planes. A single plane consists of blocks and a page register. In this register a complete page is loaded before it is written to an actual page [29].

The model is cycle accurate, but models the transactions of the data on a bit level. This would be too computation intensive for gem5. The detail of the model, as the structure and the components are all modelled, is an ideal for the current model. If it can be remodelled in this form without too much computational overhead, then this should be pursued.

7.1.2 A cycle accurate SSD simulator

Disksim enables system architects to research their systems during the design phase. However, the simulation tool is highly abstracted and the simulation accuracy is therefore often insufficient. Disksim simplifies the operation of the hardware components and increases the abstraction level to decrease the simulation latency. For SSD simulation there is a need for a cycle accurate model.
Chapter 7. Related work

The contribution that [30] makes is the implementation of a simulator that considers all SSD hardware components in order to allow for accurate and quantitative analysis when an algorithm or a controller component is enhanced. This simulator models the detailed characteristics of hardware components such as the operation clock frequency and resource conflicts.

[30] explains how flash in general and how a multichannel device work. There is an evaluation in this chapter of the throughput of a single flash cell, which appears to be 5 MB/s for writes and 24MB/s for reads. The simulator provides the simulation models for individual hardware and software components. Hardware includes the CPU, RAM, host interface, and NAND flash.

When the simulation model is evaluated it turns out to be an accurate simulator which allows the designer to verify his algorithm at a cycle level. The architecture is highly adjustable, such that it allows the user to define the architecture of the device in great detail[30].

The aim of [31] is to provide a quantitative evaluation of SSD for mass storage to be used as guidelines for SSD based storage system design or access management software development. The researchers hope that they have enabled future researchers to fulfil the lack of accurate knowledge about solid state disks.

To achieve this the DiskSim simulator has been used. DiskSim has been developed by Parallel Data Laboratory and Carnegie Mellon University. The extension that was made by Microsoft, to support SSD and hybrid storage system, has been used. To test the performance different IO traces are being used, which have been generated using bonnie, a free BSD benchmark. The evaluation is done by simulating different storage system configurations. The configurations differ in number of chips, erase latency, page size and disk size. In the experiments they influence the request size and the request type to see the different consequences for the performance.

Their findings are that the response latency of data seems to increase if they increase the current transfer size by more then eight pages. This is claimed to be due to the meta data and load management overhead that is included in the transactions.

Their other experiments try to implement parallelism at different levels to find out where it would be most optimal. The first configuration is at chip level, the second configuration is at die level and their third configuration is a division between multiple blocks, which are smaller than one die. As it turns out the third configuration is the best way of paralleling the disk. It performs 2 times better than the original configuration in terms of write throughput[31].

7.2 Mobile system analysis

The problem of slow performing mobile systems has been picked up by a number of researchers recently. A number of these researchers seem to point to the disk as main bottleneck of the system. Furthermore, some researchers claim that the days of SSD as it is currently known are almost over. This section discusses a number of recent publications which have picked up on this, and shows their different approaches to this subject.
7.2.1 Slow mobile browsers

The web browsers on mobile phones are known to be slow. If it is better understood why this is the case, then it would be easier to improve this performance. Previous studies indicate that in web browsers for personal computers have computing intensive operations and the long Round Trip Time (RTT) of wireless network hops as main bottlenecks. The authors of those studies didn’t consider the internal functionality of the browsers and had, because of that, limited insight of what actually happened. [32] has the following as main contributions:

- It examines the internals of mobile web browsers by analysing web pages visited by 25 iPhone users over a period of three months. The analysis shows that over half of the pages were not optimized for mobile browsers.
- The study utilizes two techniques to analyse the browser performance: dependency time line characterisation and what if analysis.
- Their technique enables the capturing of the user perceived delay of opening a webpage, reveal the dependency and concurrency of browser operations, and evaluate the impact of optimizations.

Furthermore the study shows that compute intensive operation improvement has a marginal impact on the overall performance, resource loading instead is the key point in browsers on smart phones. Given a certain resource, the delay on loading this resource turns out to be determined by the network condition, the browser loading procedure and the processing power of the device. The network RTT is partly responsible for the overall browser performance. The network bandwidth has only effect on the performance up to a certain point. The browser performance will not improve much beyond typical 3G network.

When a browser opens a page, it loads multiple web resources, builds an Internal Representation (IR) and converts this representation to the graphical representation. Opening a page involves a set of interdependent operations that can be dynamically scheduled and executed. The three steps can be interleaved. While loading a resource e.g. a script, it may be discovered that another resource is required and needs to be loaded before the rest of the actions can be taken. Therefore the overall process cannot be thought of as a simple pipeline.

Mobile web pages tend to have smaller CSS files and include less Java scripts, which leads to a lighter workload. Overall these pages are not as content rich as normal web pages. Surprisingly enough a majority of pages typically visited by mobile users are not optimized for mobile devices. This research therefore focusses on both mobile pages and non mobile pages.

To characterize the user perceived delay the researches measure the time between pressing the GO button and the moment when the browser percentage bar reaches 100%. The time in between should capture the elapsed time of all the steps that where necessary to open the web page. They try to answer the following questions:

- how do various operations collectively contribute to the browser delay?
To answer these questions the researchers employ dependency time line characterisation and what if analysis. The dependency characterisation provides two insights into the browser performance. First off it offers a detailed latency breakdown at operation level, and second the dependencies can serve as a foundation for the what if analysis. The what if analysis determines what will happen to the performance if a browser operation is accelerated.

The experiments are performed on two smart phones; the Google Nexus one and the HTC dream. These devices have the same software environment, but different hardware configurations. Three different networks are emulated: enterprise Ethernet, 3G and adverse network. The benchmark websites that have been used consist of the ten most visited mobile websites reported by an online blog and the ten most visited non mobile websites as found during their own research. To visit all of these pages the application PageCycler has been used.

The results show that, in general, mobile browsers are slow. The performance of the browser was better on a platform with better hardware. The IR operations have little impact on the overall performance. When these operations were optimized, the performance gain was below one percent. Resource loading on the other hand has a greater impact. Speeding up this process by 2x, yielded a gain of 70%. Loading resources can unfortunately not happen in parallel as their need is often discovered while loading another resource. Improving the RTT had effect up unto the point that is comparable with a typical 3G network. Which indicates that 3G is sufficient for the current standard of mobile devices[32].

7.2.2 Analysing mobile storage flavours

Storage performance has not yet been the focus of many researchers when researching mobile devices. This is because storage is considered not to be a critical component of those devices, in terms of performance. The paper asks the following question: does storage affect the performance of mobile applications? As long as storage performance exceeds the network performance, the answer to this question is expected to be no. However the paper presents evidence to the contrary, even for interactive applications. Due to the fact that the network speed in the past years has increased while the latency has remained the same, certain accesses to cloud functionality have been split between the cloud and the device. This has increased the burden on the local resources. [4] presents a detailed analysis of the I/O behaviour of mobile applications on android based smart phones and flash storage devices. The focus lies on applications that are frequently used by mobile users, e.g. Google maps and Facebook. To perform the analysis a measurement infrastructure for Android has been built. The paper proposes a set of pilot solutions that improve the performance.

The paper has three major contributions. First it describes a measurement infrastructure that enables custom setup of the firmware and software stack on android to perform in depth analysis. Second a detailed analysis of storage performance is presented. Third solutions that address this problem are presented.
The Android stack contains several layers. The most relevant layers to this paper are flash storage, operating system and Java middleware. The OS is based on Linux and contains low-level drivers, Dalvik virtual machine for application isolation and memory management, several libraries, and an application framework for development of new applications using system services and hardware.

Most mobile devices contain an internal flash device, an external SD card slot and a certain amount of RAM. Some devices have in addition to this a non removable SD partition inside the phone. Such storage is generally treated as external. The internal flash partitions contain all the important system partitions, such as the partitions for the bootloader and kernel, recovery, system settings, and installed applications. The external memory is used for storing user specific files such as movies or music. Applications can store data on the internal and the external storage systems. For the storage of structured data, Android typically uses SQLite database. SQLite is a lightweight database engine, which is often used in embedded and mobile operating systems.

To measure the performance the researchers have come up with a particular measurement setup. The experiments are performed on a Nexus one running Gingerbread 2.3.4, the experiments were repeated on a HTC desire, LG G2X and HTC EVO, all with similar results. To measure the performance the following changes have been made: The driver is slightly modified to help indicating whether the storage device is busy, a background monitor tool has been written to periodically read the proc file system and store summary information into a log file, and blktrace has been used to collect block level traces for device I/O. The overhead of these additions proved to be less than 2% of the total runtime. To execute interactive applications monkeyrunner is used. The following applications have been used to benchmark the mobile device behaviour: WebBench, AppInstall, AppLaunch, Facebook, Google Maps, Email, RLBench, Pulse News and Background. The experiments have been repeated multiple times, and the local cache has been deleted between every iteration.

When the experiments are performed the performance of the internal flash is compared to the performance of the external flash. Depending on the brand the performance loss varied from about 200% performance difference to about 2000% performance difference. The performance turns out to be greatly affected by the storage performance. More data is written to the flash then that is read, and more transactions are performed sequentially than randomly. Because random transactions take a considerable longer amount of time than sequential actions, the majority of the time is occupied by random writes. This results in a long waiting time for the CPU and is therefore bad for the overall performance.

Four scenarios have been proposed as solutions for this problem. The first solution places the cache on the RAMdisk, the second solution places the Database on the RAMdisk, the third solution places both the cache, and the database on the RAMdisk and the fourth solution transforms the writes to the cache and the database to be asynchronous. Putting the database on the RAMdisk and transforming all the writes to asynchronous writes seem to be the solutions with the biggest impact on the performance. This analysis leads to the following conclusions: The key bottleneck in mobile device performance seems to be the storage system and the interface choices of Android amplify the effect that this has on the system[4].
7.2.3 The scalability of NAND

Flash based solid state disk have enabled a revolution in mobile computing. It has substantial performance improvements compared to disks, but the cost is limiting adoption in cost sensitive applications. The hope is that future improvements will scale the size of the flash down and making SSD storage cheaper. The current trend however seems to suggest that this is unlikely. While flash density and feature size scaling seems to increase rapidly, all other figures: performance, endurance, energy efficiency and data retention, decline as density rises. It seems like the limit is being reached of what flash management techniques can deliver.

To enable higher densities manufacturers try to store multiple bits in one cell. The most advanced cell on the market at this point in time is the triple level cell, capable of storing 3 bits in one cell, but techniques to enable the storage of four or more bits in one cell are being developed. The price per bit decreased rapidly in SSDs in the past years, with 40 to 50\% a year. Since 2011 however the price trend seems to flatten out. However the storage of multiple bits per cell is expected to improve the price further, the reliability and performance do suffer from limitations. Theory and empirical evidence also seem to indicate lower performance for denser chips, specifically for program operations.

To model the effect of evolving flash characteristics on flash chips the researchers combine empirical measurement of flash chips in an SSD architecture with a constant die count. FTL overhead is modelled by counting an optimistic penalty to transactions. The overhead is thought to be 30 us.

For seven metrics projections are made: Capacity, read latency, write latency, read bandwidth, write bandwidth, read IOPs and write IOPs. All assumptions that have been made when creating these projections are made to maximize performance of the SSD. In other words, the projections a considered to be optimistic. Four different technologies are being investigated: SLC-1 MLC-1, MLC-2 and TLC-3.

The results show that density will increase as the number of bits per cell rises. However there is an upper bound for the capacity. The SSD can only scale beyond 4.6 TB with two or more bits per cell. TLC would push the boundary for the capacity to 14TB. Beyond this the cells will become too small. The latency, a huge advantage of SSDs over conventional hard disks, will become less advantageous when the flash technology is changed. Both read and write latencies will increase when capacity increases. A 4.6TB disk will have a write latency of 1ms for MLC-2 and 2.1ms for TLC. Read latency will rise to 70us for MLC-2 and 100us for TLC. Bandwidth will drop as well as a result of this. The projections show that bandwidth drops with 21\% and latency increases with 26\%[33].

7.3 Solutions

Multiple attempts have been made to solve the problems that arise with the use of SSDs. This section focusses on the solutions that are relevant for this research, namely, page replacement schemes, fast interfaces, and indirection removal. The ideas presented give a good overview of
the design space that is being researched at this point in time. There does not appear to be one clear front were researchers search for solutions, other than disk performance.

### 7.3.1 Finding solutions for SQLite

As mobile systems evolve, the demand for high performance and high capacity storage becomes higher. This has made flash memory an indispensable component in memory systems. Because of all the flash peculiarities, storage management techniques were developed. The most popular flash memory database at this point in time is SQLite. It is a relational database management system designed for efficient data management and easy data access in embedded mobile devices. This paper focusses on optimizing the transaction commit strategy and the parameter configuration.

In the current situation SQLite performs an atomic commit of a transaction against a single database file. It uses different kinds of locks to guarantee data consistency. Prior to making any changes SQLite will create a rollback journal of the original content of the database pages that will be altered. The purpose of this is to be able to restore the database to its original state when needed. After the journal has been built, the data is modified in the database, and the journal is flushed to the non-volatile memory. Because this involves write transactions, this takes up a significant amount of time.

The aim of this paper is to optimize the commit strategy for SQLite. The aim is to use this commit in the lowest frequency possible. To accomplish this the following things have been implemented: All sequences of SQL statements during mobile device connection initial are wrapped, and all sequences of SQL statements for transferring each file are wrapped. Only after a whole command is completed a commit is initiated. If there is any error then the transaction will be aborted. This strategy improves the efficiency with about 15%.

Another way of improving the performance is by setting the sector size of SQLite to the blocksize of the flash device. This will yield an improvement of about 5%.

When the parameters of SQLite are tweaked more performance can be gained. The Synchronous setting has the biggest impact when it is modified. If it is set to OFF the performance improves with about 10% in relation to the setting FULL. With all the modifications considered about 25% performance improvement can be gained[34].

### 7.3.2 High performance and out of order

NAND flash is broadly used in many applications. While integrated circuit techniques keep improving, the flash memory cell improves as well. Meanwhile the Organization of flash cells has evolved from single plane and single die to multi plane and multi die. In comparison with all these advances the actual operation time of NAND seems to improve slowly. To avoid these operation times becoming a bottleneck, one could try to upgrade the the I/O frequency, but this will make timing closure hard to achieve. To take full advantage of the possibilities of multi die
chips, the paper introduces two techniques: Out of order execution on different dies/chips in parallel and two-plane address translation with simple hardware implementation.

With a flash chip there is typically a NAND flash controller which receives a command from the host via a certain interface, and translates this command to a set of commands for the flash chip. Typically this set of commands is in order. So when a device is accessed, the rest of the devices are idle. With this setup it is impossible to profit from the available parallelism.

To achieve higher performance a NAND controller is proposed which fully supports advanced features provided by flash drive vendors. Three techniques are highlighted: Multi die commands are issued in parallel where possible, out of order execution for commands in queue is enabled to squeeze out more command parallelism, and through the proposed address translation technique two plane commands can be exploited to reduce the average sequential data access time.

When out of order execution is applied there may be some data dependencies between commands. The analysis of these data hazards is very similar to that of CPU pipelines. When the read commands are classified as reads and the program and erase commands are classified as writes the following hazards can occur: Read after Read, Write after Read, Write after Write and Read after Write. For each hazard a detection mechanism is implemented and the controller will ensure in order execution when needed.

In an original multi die setup the address translation was organized in such a way that the pages were sorted by address over the blocks, i.e. the set with the lowest addresses in the block with the lowest address, the highest addresses in the highest numbered block. The chance that the next page is selected in another block is however slim. To optimize for sequential access the researchers have chosen to divide the pages on the least significant bit of the address over the blocks. When a sequential access happens it is now guaranteed that it is possible to do this over multiple dies en therefore in parallel.

The results of the experiments show a greater improvement for random accesses than for sequential accesses. This seems to indicate that the out of order execution has a greater impact than the address translation. The impact of the changes varied from a speed up of 20% for a sequential read action, to a 300% speed up for a random program action. Overall the improvements seem to have a significant effect [35].

7.3.3 New page replacement scheme

As mobile devices provide a variety of functions, virtual memory systems are becoming increasingly important. Meanwhile mobile systems are employing flash memory as swap space. Since NAND flash is significantly different from a conventional hard disk, new file systems for flash have been studied. This research is still in its initial stage. This research focusses on page reference characteristics of a virtual memory space that uses flash as its swap space, and presents a new page replacement algorithm for its environment.

Page references in a virtual memory environment have a temporal locality property and thus most of the algorithms used nowadays are based on the most recent accessed pages. Those algorithms
generally do not take I/O costs into account. This research analyses the characteristics of virtual read and write references separately in terms of temporal locality. Based on their observations a new algorithm is proposed.

When analysing the locality of different transactions it becomes apparent that in general the algorithm makes a good judgement of the frequency with which a certain page is accessed. However when these accesses are split up then it seems that write references are a bit more irregular than read accesses, and the algorithm typically makes wrong predictions in those cases. Further analysis shows that most pages are written to frequently, and a few are accessed less. This implies that a page accessed frequently in the past, is likely to be accessed again in the future.

The new page replacement algorithm that is proposed is called CRAW (clock for read and write). For each area replacement is performed similar to the CLOCK algorithm. To select a victim page, CRAW exploits the read and write characteristics of virtual memory: For read accesses temporal locality is exploit and for write accesses both temporal locality and write frequency are used to predict the reference likelihood. Write pages are given priority because the replacement is more costly, but if a read page is referenced enough it still can be preserved.

To verify the algorithm, trace driven simulation is performed. Traces are obtained via the cachegrind tool of Valgrind. The paper [36] unfortunately does not disclose what simulator has been used to play these traces. The System that is simulated runs Linux Xwindow with the applications gedit, freecell, xmms mp3 player and kgohstviw pdf file viewer. CRAW is compared with the algorithms CLOCK, CAR and CFCLOCK. Overall CRAW performs best. Although it makes more read I/O transfers, it saves a significant amount of write transfers which causes it to have the best overall performance. In general the performance is improved by 20-66%[36].

7.3.4 What’s in a name?

Indirection is a fundamental technique in computer systems. It is used as well in SSDs to map the logical addresses to a physical address. This has as major drawback that this has a performance cost and space overhead. Different FTL techniques have been proposed, and all are a trade-off between performance overhead, space overhead and device complexity. The paper presents nameless writes as a strategy to remove a big part of the cost of indirection. Unlike most writes nameless writes pass the data to the device without mentioning the address. The device then can choose any available block to put the data in and isn’t bound by the address. After the device decides which block to use, it informs the client of its choice. The client then records the name for future reads. One potential problem of this approach is that if all writes are nameless, then any update to the filesystem requires a recursive set of updates in the file system tree. This problem is resolved by introducing a segmented address space. This introduces a large nameless write space and a small space for traditional writes.

The nameless write functionality can be split up into the following actions: nameless write, nameless overwrite, physical read and free. The nameless write interface replaces the existing write operation. A nameless write supplies the data to the device, and the device answers with
the address at which the data is stored. The nameless overwrite passes apart from the data also the old address of the data. Read operations are performed as in conventional systems. A nameless write can be considered to be an allocation operation. For that reason a free actions is provided.

This strategy means modifying both the device as well as the initiator. The actions have an impact on the filesystem, FTL, garbage collector and wear leveller. To test the system, an SSD emulator has been build. To simulate this the simulator David has been used. This emulator is utilized by a modified ext3 filesystem on a 64 bit Linux 2.6.33 kernel. The experiments are run on a 2.5GHz Intel quad core CPU with 8GB memory. The results of this new system are compared with a Hybrid FTL and a page level FTL.

The application performance turns out to be in between page level and hybrid, but is almost on par with the performance of page level translation. Page level translation is considered to have high performance, but is very costly in terms of capacity overhead. All the three approaches have the same performance when the file system is sequentially accessed, but with random accesses the hybrid translation performs much worse.[10]

7.4 Summary

In this chapter the work of other researchers have been presented. This research is closely related to the work presented in this thesis. The simulation models of flash disks are good examples for the flash model in gem5, but turn out to be too low level for actual implementation. Cycle level accuracy comes at a performance price. The approach and structure can very well be used as ideal, as the functionality is modelled in quite some detail.

Overall there seems to be a lot of recent interest in the topic of mobile performance. A number of researches seem to blame the disk rather than the CPU for the bad performance. This highlights the need for a simulator as presented in this work. Mobile systems have become complex and a lot of the related work want to continue their investigation for the cause of this performance problem. The work of Kim et al. [4] is the closest related to this research. It focusses on the performance of mobile systems and uses a comparable approach on real devices to determine the bottleneck in the system. The simulation model presented in this thesis however has a number of advantages, which allow the user to quickly adept to model to test new theories. This may enable more research in this area. The simulation model presented is not as exact as the models presented in this chapter. Perhaps some improvements can still be made in this direction. The simulation model does enable full system simulation, an advantage that the other simulation models cannot deliver.

As a solution to the performance problems of mobile suystems a lot of researchers seem eager to propose a lot of different solutions related to the disk accesses. All seem to experience some improvements, which indicates that there is a lot of optimizations to be made in this area. A good understanding would only help a structured exploration of this field. And given the contributions of this work, this is currently possible.
Chapter 8

Conclusions

There is a need for a full system simulator that contains a flash disk model. This is because the system is too complex to understand what the implication of a system solution is without simulating it. Furthermore a new flash interface UFS is on the horizon. It is unsure how this new interface will affect the performance of a system when it is implemented in a system. A simulator would be a good method to evaluate this.

This work has mainly been an analysis of mobile system behaviour, in particular disk performance. Throughout the document the system behaviour related to the file system handling has been explained. Not all hypothesis could be evaluated, as explained in the previous chapter, but the overall analysis and experiments did provide answers and new insights.

As demonstrated in Chapter 6, the throughput of the disk is dependent on the flash throughput rather than the interface throughput. This means that UFS does not introduce a new bottleneck in the system. UFS is however also not a miraculous interface, which can boost the performance to 5.8Gb/s, just by being implemented. To fully utilize UFS, one has to design the system properly. This was according to Hypothesis 1, and Hypothesis 2.

The throughput that is experienced could be directly calculated by calculating the throughput to the flash disk. This suggest that the flash disk is the main bottleneck in a transaction. This was in accordance with Hypothesis 3.

With IOzone it could be demonstrated that it is better to have big transfers than many transfers. Once the file size was too big to fit in the cache, the filling of the transfer slots was noticeable in the application level in the system. When more transactions are planned than can fit in the slots, the performance drops. This has an optimum, once the file that is to be transferred does not fit within the reserved memory any more, the performance drops. This was predicted by Hypothesis 7.

It also makes sense from a power perspective to sleep for long periods of time when the device is not in use. To make these periods of rest longer, one can group data transfers together. How this should be done is a question left for future work, but the accesses made to the disk in current mobile systems are the exact opposite of what the UFSA advises; there are many accesses, each
transferring a small amount of data. To implement UFS successfully, a change has to be made in the way that the data is currently managed. To do this, different solutions need to be examined.

To examine the different solutions a full system analysis is required, an analysis were it can be shown how the processor activity and the disk behaviour can be related.

Overall the disk model seems to behave in a predictable manner, which shows that it may be consistent with real life. To know this for sure, more evaluation is needed. It provides a good tool to get a better understanding of the disk behaviour in a system.
Chapter 9

Future work

In this document a few hypothesis have been evaluated, and a few have been left unanswered. The project itself has generated some new questions and has unveiled some new interesting areas for future research projects. This chapter will summarize the future work and will propose improvements for the simulation model.

9.1 Unresolved hypotheses

Due to the complexity and the time limit of this project, not all hypotheses could be validated. It was not possible to generate enough data to see the transfers at a smaller granularity than 100 ms. This already generated a statistics file with a size of more than a gigabyte, and the actual disk transfers happen within a granularity of milliseconds. Therefore it is impossible to say if there was enough time between accessing the same location for the garbage collector to do its work. Hypothesis 4 can therefore not be validated.

It was not possible to determine the exact location of the transaction with the gathered data. Therefore Hypothesis 5 and Hypothesis 6 have not been evaluated.

In future work these hypothesis will be tested.

9.2 New questions

When considering the BBench test it seems that the data transferred can be organized more efficiently. Why could it not be possible to save these transactions for one big transfer? This would make the disk access pattern more attractive for UFS devices. For a follow-up research the following questions could be asked:

- What is the best strategy to organize data transfers?
  - Is it better to gather them in software or hardware?
Which device should be responsible for the transfer initiation?

Is it advisable to introduce a new part of the memory hierarchy to store these data transfers?

- How can the software system be organized in a way that improves the performance?
  - How can applications be more aware of the performance impact of their actions with respect to file system accesses?
  - Is SQLite the main bottleneck in an Android system?
  - Can the performance of SQLite be improved?

- How does UFS compare with other interfaces?

- What advantage does having multiple LUs have?

## 9.3 New benchmarks

For the continuation of this research a few more benchmarks have been designed and developed. These benchmarks are expected to shine a little more light on the Android environment and the user experience. This section will describe the working of these benchmarks.

### 9.3.1 RL benchmark

SQLite is used on many applications to store application data on an Android device. It is the way for applications to store data in the non volatile memory and some application store data very often. It is therefore useful to have an indication on how well the SQLite performs in a mobile system. For this purpose RedLicence Labs has developed RL Benchmark.

This benchmark can be automated by writing a script that is launched after Android is booted. This script installs RL Benchmark and launches it via the intent action Main. After this the key event sequence: down, up, up, enter is given to the application, which selects and presses the start button. By doing so there will be no need for a graphical interface automation. When RL Benchmark is completed, it creates an XML file with the results. When this file has been created, the script will print its contents to the terminal, and exit gem5. Since the terminal can still be read after gem5 has terminated, the results can be obtained. [37]

### 9.3.2 App install

Before an application can be executed it should be installed. The installation of an application typically involves downloading an apk file from the market and install it automatically. This requires connection to the internet. Unfortunately an internet connection is not available in gem5. The alternative method to install applications is to download the apk from the developers website and install the app by using the “adb instal” command. Between every installation the
statistics of gem5 are dumped and reset, which will give a good indication of how much time the installation took.

9.4 Model improvements

A model can always be improved. As described, the model suffers from a few shortcomings, which makes it less accurate then it potentially could be. In the near future, it is envisioned that improvements will be made. To help future developers this section will provide a number of suggestions to improve the model. The suggestion are not an exhaustive list, but rather an indication of what the current developer finds the most important to add.

9.4.1 Multiple Logic Units

The current model supports only one LU. All the functionality is in place to support multiple LUs but it is not implemented. To get the multiple Logic Units one needs to add extra checks to the SCSI routine in the device layer of the model. Also the waiting queues of SCSI need to be modified, as the current mechanism stalls as soon as the SCSI function is instantiated. Adding multiple LUs could potentially show a better performance exploitation of UFS, as it would be able to grab data from more than one location. Not implementing this can be seen as not fully utilizing UFS, however, since every LUN has control over its own LBA and therefore its own partition, it is not simply a matter of “being able to reach a certain multiple of the current performance”. In the current state the model provides an excellent insight in what the base performance of the UFS device is and where in the system the bottleneck of the performance for a certain action lies.

If multiple LUs have been built then the disk image need to be split in different partitions to fit within the system setup. Those sections may not overlap.

9.4.2 Disk caches

Currently there is no cache modelled on the disk. A disk cache acts like a buffer between the IO port and the disk. It is implemented to bridge the gap between access times to different parts of the data. The size of this buffer can therefore have an impact on the overall performance. To keep control of the size of this cache, one can modify the queueing system in such a way that it also keeps track of the size of this cache. Basically an extra check of what is in the queues to and from the disks would be sufficient.

Another point is the DRAM within the flash device. Currently this has not be modelled as it is deemed too low level. It is assumed that the DRAM access time is instant. Future work might implement an extra buffer which simulates the actual fetch from the flash before the DMA transaction. This will create a more realistic transaction flow of the device, and might have an impact on the performance.
Chapter 9. Future work

9.4.3 Cache on write

Currently when a write action is done to the disk, then the disk image is modified. This means that after a run the disk image must be returned to its original state when the test is to be repeated. To circumvent this issue the IDE disk model in gem5 implements a Cache on Write disk model. This model makes no changes to the image, and keeps track of the changes in a separate buffer, such that any read actions to modified data can be correctly handled. Being able to repeat experiments without having to replace the disk image is an advantage. On the other hand, modifying a disk image enables the researcher to look back in the changed files to get more information about the process. A cache on write disk does not have this advantage. Therefore when this functionality is implemented it is important to keep the old functionality.

9.4.4 Exact FTL from disk

As mentioned the current FTL model does not provide a very in depth table of every location in the disk image. This is mainly because the table for the disk image can not be derived from just the disk image itself. It would be nice if future applications were able to come up with such a table, based on the age, the algorithm used and the disk image. Such a tool should be built outside of gem5, such that for every disk a table can be made beforehand. The SSD model is to be extended with a number of methods which can be used to keep track of all the changes from that point on. Such an approach would give a more realistic behaviour of the flash device, although it is expected that this behaviour would not be far off from the current behaviour.

9.4.5 Power modelling

As the previous chapter has shown, the current way that disk transactions are done in a way that is not likely to be power effective. The impact of this has not yet been researched because the power analysis in gem5 is still in development. For the UFS disk model, no power model has yet been created. The results show that research in this direction is very useful. It would be interesting to see if there are more design choices to be made if the power efficiency of the device is to be considered.
Chapter 10

Reflection

In the six months that I spent at ARM, I have seen a great deal of new tools and software, and have refreshed a lot of knowledge from past courses and experience. This thesis has been written as partial fulfilment for the degree of Master of Science in the Embedded Systems at the Technische Universiteit Eindhoven. This program has the following goals:

- The graduate has a holistic view on systems and system development. On the one hand he is capable of an abstract view to understand and master systems of huge complexity. On the other hand he is able to describe and study the structure and the behaviour of the (embedded) systems in precise detail. He understands the position and importance of the system during its lifetime.

- The graduate has thorough knowledge of contemporary techniques to realize embedded systems. He has sufficient academic background to understand and apply techniques that will become available within the next decades. He is cost and environment aware, thus capable of making optimal use of available means (software/hardware).

- The graduate has a sufficient basis to design embedded systems at the required level of quality, or assess a priori that such a design cannot be constructed. This presupposes thorough knowledge of requirement engineering, modelling, testing and implementation techniques.

- The graduate has a flexible and inquisitive mind. He understands the theories, techniques and tools in this field in such a way that he is able to adapt these to optimally fit their purpose. He is able to invent his own tools, theories and techniques if these are not available.

Given all these goals and reflecting on this thesis, I cannot come to a different conclusion than that this thesis checks all the boxes. In this thesis a full and complex system has been presented, the performance of disk actions have been analysed, a simulation model has been designed, and the behaviour has been researched.
The essence of embedded systems is in my understanding about exploring the design space where software and hardware meet. This thesis covers a standard mobile system, that communicates with another embedded system, the flash drive, in order to store its data; an essential action for a functional mobile device. When considering this action from a layman’s point of view this seems trivial. I remembered thinking at the start of this project that it should not be much more complicated than a standard hard disk. As has been shown, it is more difficult, and it indicates how complex embedded systems are nowadays.

Both the software as well as the hardware needed to be analysed to come up with a decent model. The project has given me a lot more insight in Android and Linux operating systems, especially regarding kernel hacking activities. My C and C++ programming skills have improved over the last year, and I have been introduced to the knowledge of simulator mechanics.

The project has been valuable, for me, for ARM, and for the gem5 community. The project has enlarged my interest in and understanding of non volatile memory systems. It appears to be one of the more exciting topics of the coming years, given that there seems to be a lot of development for the next generation non volatile memory. A few alternatives have already been mentioned, but when one looks further ahead, as in developments for 2020 and later, then other alternatives such as RRAM and CNT, and true universal memory might be available. If one considers the evolution of NAND, and all the solutions that it has brought in current operating systems as described in Chapter 2, then one can see that the solutions already are deeply embedded in the system. How different will the next non volatile memory solutions be, and how many changes should there be made to the operating system to use them effectively?

This work may be a starting point of non-volatile memory research using gem5. Giving the developments within this field and the growing interest from myself and people around me in this subject, this seems to be an exciting and interesting field to contribute to.
Bibliography


