A Preliminary Exploration of Memory Controller Policies on Smartphone Workloads

Goran Narančić

Master of Applied Science

The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
University of Toronto

2012

Abstract

This thesis explores memory performance for smartphone workloads. We design a Video Conference Workload (VCW) to model typical smartphone usage. We describe a trace-based methodology which uses a software implementation to mimic the behaviour of specialised hardware accelerators. Our methodology stores dataflow information from the original application to maintain the relationships between requests.

We first study seven address mapping schemes with our VCW, using a first-ready, first-come-first-served (FR-FCFS) memory scheduler. Our results show the best performing scheme is up to 82% faster than the worst. The VCW is memory intensive, with up to 86.8% bandwidth utilisation using the best performing scheme. We also test a Web Browsing and a set of computer vision workloads. Most are not memory intensive, with utilisation under 15%.

Finally, we compare four schedulers and find that the FR-FCFS scheduler using the Write Drain mode [8] performed the best, outperforming the worst scheduler by 6.3%.
Acknowledgments

I would like to thank my supervisor, Andreas Moshovos, for the guidance he provided during the making of this thesis, as well as all the knowledge he imparted to me. I would also like to thank prof. Enright Jerger, Serag, Serag Gadelrab, Myrto Papadopoulou, Jason Zebchuk and other members of prof. Moshovos group for the interesting discussions and the advice. Finally, I would like to thank all my friends, both in Canada and Croatia, for putting up with me.
# Table of Contents

Acknowledgments ............................................................................................................. iii

Table of Contents ................................................................................................................. iv

List of Tables ........................................................................................................................ vii

List of Figures ....................................................................................................................... viii

Chapter 1 Introduction ......................................................................................................... 1

Chapter 2 Main Memory Subsystem ..................................................................................... 3

2.1 The Main Memory Subsystem Context ......................................................................... 3

2.2 DDR SDRAM ................................................................................................................... 4

2.2.1 DRAM Commands ..................................................................................................... 5

2.2.2 Data Granularity ....................................................................................................... 6

2.2.3 Timing Constraints .................................................................................................... 7

2.3 Logical Organisation of DRAM ...................................................................................... 9

2.3.1 Logical channel ....................................................................................................... 9

2.3.2 Logical rank ............................................................................................................. 10

2.3.3 Logical bank ........................................................................................................... 10

2.3.4 Logical row ............................................................................................................. 11

2.3.5 Logical column ....................................................................................................... 11

2.3.6 Address Mapping Process ....................................................................................... 11

2.4 DRAM Behaviour Metrics .......................................................................................... 12

2.5 Memory Controller ....................................................................................................... 13

2.5.1 In-Order Scheduler ................................................................................................. 14

2.5.2 First Ready - First Come, First Served memory scheduler ................................... 16

2.5.3 Thread-Fair memory scheduler ............................................................................... 18

2.5.4 Thread Clustering memory scheduler .................................................................... 20
2.6 Related Work

2.6.1 Performance-oriented schedulers

2.6.2 Power and energy oriented schedulers

2.6.3 Prefetch-oriented schedulers

2.6.4 Related Work Summary

Chapter 3 Methodology

3.1 Motivation

3.2 The Trace Collection Tool

3.2.1 Creating the dependency graph between the instructions

3.2.2 Tracking the dataflow within the cache

3.2.3 Determining memory access dependencies

3.2.4 Trace collection tool summary

3.3 Trace-Based Traffic Generator

3.3.1 Waiting requests queue

3.3.2 Dependency handling

3.3.3 Issued requests queue

3.3.4 Finished reads queue

3.3.5 Trace-based traffic generator summary

3.4 Trace Format

3.5 Video Conference Workload

3.5.1 Trace collection and preparation

3.5.2 Traffic generation

3.5.3 Video Conference Workload summary

3.6 DRAMSim2

3.7 Summary

Chapter 4 Evaluation
4.1 Workload Parameters.................................................................48
4.2 Address Mapping Schemes ........................................................52
4.3 Workload Characteristics ..........................................................56
4.4 Memory Scheduler Comparison .................................................58
4.5 Summary....................................................................................61
Chapter 5 Conclusion and Future Work.............................................62
References.......................................................................................64
List of Tables

Table 1 DRAM timing constraints ........................................................................................................8

Table 2 Memory controller parameters.................................................................................................47

Table 3 Computer vision workload summary........................................................................................48

Table 4 Video Conference Workload device configuration .................................................................52
List of Figures

Figure 1 Typical computer organisation ................................................................. 4

Figure 2 Schematic of a DRAM cell ................................................................. 5

Figure 3 DIMM module with DRAM devices ......................................................... 7

Figure 4 Logical organisation of DRAM ............................................................ 10

Figure 5 Example of the impact of scheduling on performance ................................ 15

Figure 6 Decision flowchart for the Thread Fair scheduler ...................................... 19

Figure 7 Example of insertion shuffle through time .............................................. 23

Figure 8 Generalised example of a smartphone-like system .................................... 29

Figure 9 Infrastructure overview ....................................................................... 31

Figure 10 Dependency graph .............................................................................. 32

Figure 11 Indirectly satisfied dependencies ........................................................ 35

Figure 12 Example of head and tail segments of a single trace entry ....................... 40

Figure 13 High-level description of the devices involved with a video conference ...... 42

Figure 14 Video Conference Workload trace collection mechanism ....................... 43

Figure 15 Required total amount of data during the encoding and decoding of the first frame, as a function of cache size ........................................................................................................... 50

Figure 16 Percentage of TAD reduction as a function of increasing cache size for our Video Conference Workload .................................................................................................................. 51

Figure 17 Address Bits Usage per Mapping Scheme; $K$ stands for the rank bit ............... 53
Figure 18 Comparison of address mapping schemes using an FR-FCFS scheduler on the Image Segmentation workload .....................................................................................................................54

Figure 19 Comparison of address mapping schemes using an FR-FCFS scheduler on our Video Conference Workload ..................................................................................................................................55

Figure 20 Average number of column commands per ACTIVATE command ..................................................56

Figure 21 Average DRAM utilisation for the Web Browsing and computer vision workloads using FR-FCFS with scheme RKBC ..................................................................................................56

Figure 22 DRAM utilisation for Video Conference Workload using FR-FCFS scheduler ..........................57

Figure 23 Comparison of time banks spent in particular bank state during the execution of the Video Conference Workload ..............................................................................................................58

Figure 24 Comparison of the scheduling policy performance ............................................................................59

Figure 25 Comparison of the scheduling policy performance ............................................................................60

Figure 26 Average latency of read requests for the combination of workloads .........................................61
Chapter 1
Introduction

The proliferation of smartphone usage in the modern society provides an incentive for manufacturers to improve and expand the performance envelope and abilities. The general-purpose abilities of the smartphone have closed in on the abilities of the average personal computer from several years ago although integrated hardware elements such as cameras and microphones are expected. However, the design and operation constraints of smartphones and mobile platforms are different from personal computers. Smartphones run on batteries and battery endurance is a crucial parameter, so the energy envelope is much stricter. Furthermore, smartphones are physically much smaller and thinner than desktop computers with implications on the physical design, such as chip count, pin per chip and memory capacity, among others. One solution to provide acceptable performance for the common tasks while staying within the limitations is to use specialised hardware elements.

Modern smartphones frequently capture and process images and video streams, as well as reproduce them for viewing. In particular, the video conference ability requires capturing, encoding and sending through network a real-time video stream while receiving, decoding and displaying a separate video stream. Additionally, future smartphones may include other functionality, such as face recognition and tagging during the video stream. These tasks present an avenue for the exploration of the smartphone performance design space.

The main memory subsystem, using Dynamic Random Access Memory technology, stores the long-term working dataset and serves as a data exchange mechanism between the computer elements. The subsystem is encapsulated by a memory controller, which receives requests to read or write data to a particular address, and issues commands to the actual chips which store the data. The scheduling of these commands has a great impact on performance, for example, it is often best to schedule requests to adjacent memory locations back-to-back.

The aim of this thesis is to investigate the performance of the main memory subsystem in a smartphone-like system with near-future capabilities. We explore the main memory characteristics of the smartphone workloads, as well as the behaviour of the memory controller under such workloads.
We make the following contributions in this thesis:

- We present a Video Conference Workload, which simulates the memory traffic of the hardware elements within the smartphone during a two-way video call.
- We test the main memory behaviour of our Video Conference Workload, a Web Browsing workload and a selection of computer vision workloads.
- We investigate the impact of the address mapping policy on the performance of the main memory subsystem.
- We explore the performance of a selection of memory schedulers under the smartphone workloads.

The experiments were performed using a trace-based system we developed. The system models the dataflow between requests based on the data collected from an instrumented application. We implemented a request trace collection tool which collects dataflow information from an instrumented program, and a trace-based traffic generator which reproduces the collected traces based on the dataflow information.

Chapter 2 explains the inner working of the main memory subsystem and the memory scheduler policies we test. The same chapter also gives a brief overview of the related work. Chapter 3 presents our trace-based, dataflow-controlled traffic generator, the Video Conference Workload and the overall infrastructure we use for our experiments. Chapter 4 presents our experimental results. The conclusion and a brief discussion of future work are presented in Chapter 5.

Our results show that the Web Browsing and computer vision workloads are not memory intensive, with all but one utilising bandwidth under 15%. Our Video Conference Workload was memory intensive, with up to 86.8% bandwidth utilisation for the First Ready – First Come, First Served (FR-FCFS) [9] scheduler with the best performing address mapping policy. We find that the address mapping policy can have significant impact on the performance, and that the best policy will attempt to accentuate the number of requests which access the same row, while trying to reduce the number of requests attempting to access different rows of the same bank. We tested four memory scheduler policies using a workload composed of the Web Browsing, Face Detection and our Video Conference Workload, and find that modern policies intended for heavily multi-threaded systems do not perform as well under a workload typical of smartphone usage. The FR-FCFS scheduler using the Write Drain mode [8] performed the best, with the Thread Fair scheduler [7] slower by 1.45%. The Thread Clustering scheduler [3] performs the worst, falling behind 6.08% under the maximum resources configuration.
Chapter 2
Main Memory Subsystem

This chapter reviews the modern main memory subsystem, emphasising the aspects relevant to this thesis. The description of the main memory and the underlying technology is heavily based on the book “Memory Systems: Cache, DRAM, Disk” by Jacob et al. [1].

The chapter is organised into the following sections. A high level overview of the main memory subsystem is given in Section 1. Section 2.2 gives a brief overview of the modern DRAM technology and timing constraints imposed by it. The correspondence between the physical and the logical organisations of DRAM, and the process of translating addresses into location within DRAM, is explained in Section 2.3. Common DRAM performance metrics are explained in Section 2.4. Section 2.5 explains the duties and the operation of a memory controller and memory schedulers implemented and tested in this thesis. Related work is discussed in Section 2.6.

2.1 The Main Memory Subsystem Context

The main memory subsystem of a general-purpose computer is an intermediate repository for operating data, located between the processor caches and secondary, persistent storage such as hard drives. It provides temporary storage for the data expected to be useful in the future, or the data that has been evicted from the cache system. To accomplish this task, the main memory system needs to be much faster than the hard drives to improve system performance. Fast access requires the memory to have random access capability. The temporary nature of the data in the main memory means that the hardware needs to be able to withstand long term, high volume of reads and writes. However, this temporary nature also enables the technology used to require periodic refreshing of data.

The main memory is accessed through the memory controller, a hardware device located on the motherboard, although the recent systems are integrating it on the CPU die. The memory controller in turn controls the random-access memory (RAM) through the command, address and data buses. The main memory system may consist of multiple channels, which are completely independent of each other and often have individual memory controllers. The computer elements which produce memory requests and consume data see the main memory system as a black box.
which receives their requests and replies with uncertain latency. The amount of data handled per request is typically set at a single cache line in modern computer systems.

![Typical computer organisation](image)

**Figure 1 Typical computer organisation**

The following section presents the specific technology used to build the main memory subsystem. We discuss the physical aspects of the technology and their influence on the hardware implementation. However, the system sees the storage devices in a slightly different fashion, and this difference is discussed in Section 2.3.

### 2.2 DDR SDRAM

This section looks at the specific RAM technology used in modern computers, the *dual data rate synchronous dynamic RAM*, and how it is used to form the actual devices. The access method for DDR SDRAM is not fully random. The service latency depends both on the current state of the memory system, and the access sequence. The technology requires adherence to a large set of *timing constraints* when issuing commands for the hardware to function correctly.

Modern computer systems use a *dual data rate synchronous dynamic RAM (DDR SDRAM)* technology for the main memory. The *dual data rate (DDR)* part of the name designates that the data bus operates twice as fast as the address and the command buses. *DDR SDRAM* is *synchronous*; the devices act on *row* and *column access strobe* signals (RAS and CAS) at the
falling edge of the clock signal, rather than immediately upon the change of the access strobe
signals. The *dynamic* part of the name designates the technology which uses a single transistor-
capacitor pair to store a single bit of data, as shown on Figure 2. It is called *dynamic* because the
capacitor leaks electrons, limiting the time information can be held in the cell and requiring
periodical refresh to keep the data correct. The single pair is also called a *DRAM cell*.

![Figure 2 Schematic of a DRAM cell](image)

Section 2.2.1 explains the commands for controlling the DRAM devices. The access granularity
and overall physical design is discussed in Section 2.2.2. Timing limitations the hardware
implementation creates are described in Section 2.2.3.

### 2.2.1 DRAM Commands

*DRAM cells* are used to form rectangular *memory arrays*, where each bit of information can be
referenced by a specific *row* and *column* values. Each array also contains a set of sense
amplifiers, one for each *column*, connected to every *row* of the corresponding *column*. This sense
amplifier row is used as a buffer for managing the data and refreshing the cells. To access the
data located at *row* X and *column* Y, sense amplifiers must be prepared to receive the data by
using a “precharge amplifiers” (*PRECHARGE*) command. The next step is to “read out” a *row*
from the DRAM cells to the sense amplifiers using an “activate *row* X” command (*ACTIVATE*).
Once the sense amplifiers have activated a row, *bank* is said to have a *row open*. The data is then
accessed in the sense amplifiers using a “read *column* Y” (*READ*) or changed by using a “write
*column* Y” (*WRITE*) command. Once the *column* accesses are completed, using a *PRECHARGE*
command will prepare the sense amplifiers for receiving another *row*. Activating a *row* then
precharging also serves to refresh the cells. *REFRESH* command performs the required actions
for several sequential *rows*. 
PRECHARGE, ACTIVATE, READ, WRITE and REFRESH are the basic DRAM commands. DDR SDRAM supports a larger set of commands, mostly for managing the power state. We do not cover these in more detail here. However, there are two compound commands which will be relevant in the later sections: READ-PRECHARGE and WRITE-PRECHARGE. These commands instruct the devices to execute as if they received a regular READ/WRITE command followed immediately by a PRECHARGE command. Using these commands is often referred as auto-precharge. The devices behave in all respects as if they received two separate commands, which means there is no timing advantage. However, these commands can be useful from a memory controller perspective by not occupying the command bus twice. This ability is called auto-precharge.

### 2.2.2 Data Granularity

A single access to DRAM always handles a certain, fixed amount of data. In the following paragraphs we explain how the whole design is formed from the memory arrays we previously discussed.

A single memory array provides one bit of information for one column access; to satisfy the needs of a memory subsystem a single DRAM device contains large number of the arrays. Within the DRAM devices, the arrays are divided into the groups called banks. All arrays forming one bank receive and execute commands in lockstep, which allows the device to produce multiple bits of data for a single access. The number of bits produced per cycle is noted in the device name with an “x” followed by the number of bits. For example, a x4 DRAM device provides four bits of data on a single READ command.

The DDR SDRAM devices transfer data in bursts to capture the full benefit of the double data rate capability of the data bus. Each access will cause the data from several neighbouring columns to be transferred to the memory controller. The exact amount of data transferred is dependent on the technology, chosen settings and the data bus width. It is often set to a single cacheline.

Each bank within the DRAM device is isolated from other banks, holding different segments of data and having the ability to have different rows open. However, since the buses are shared among all banks, commands must avoid bus conflicts. Since all rows within the bank need to be
refreshed within a certain period, long downtimes are prevented by issuing \textit{REFRESH} commands for several \textit{rows} in smaller periods. A single \textit{REFRESH} command is executed for all banks of the \textit{device}.

Each \textit{DRAM device} provides only a small amount of bits on a single access. Providing a larger data packets is solved by using multiple \textit{DRAM devices}. The multiple devices are slaved together into groups called \textit{ranks}. Each \textit{device} in a \textit{rank} receives the same commands and operates in lockstep, providing a portion of the total required data. The number of devices will depend on the data bus width and the device width. For example, a \textit{rank} designed for a system with 32 bit wide data bus will need to contain 8 x4 or 2 x16 \textit{DRAM devices}.

![DIMM module with DRAM devices](image)

\textbf{Figure 3} \textit{DIMM} module with DRAM devices

Personal computers and workstations use printed circuit called a \textit{dual inline memory module (DIMM)} to provide the configurability. The module contains several \textit{DRAM devices} divided into one or two \textit{ranks}. A drawing of a \textit{DIMM} with the \textit{DRAM devices} is shown in Figure 3. Physically small systems, such as smartphones, have the devices placed directly on the motherboards to save space. This does not impact the system characteristics.

\subsection{2.2.3 Timing Constraints}

The physical implementation of the \textit{DRAM devices} creates a set of \textit{timing constraints}. The constraints arise from signalling limitations, wire requirements and power profiles. Table 1 lists and defines a subset of the constraints, as well as trigger and affected commands. The example values are for a 1 Gb Micron \textit{DDR3 SDRAM device} operating at 800 MHz (1.25 ns/cycle) [2].
The full list of the timing constraints would contain many constraints subsumed in the listed ones and constraints for modes not used in our experiments, such as for the self-refresh mode, so we list only the most important constraints.

**Table 1** DRAM timing constraints; the table is based on the table 11.1 from the book by Jacob et al. [1] with examples based on 1 Gb Micron DDR3 SDRAM device [2]

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Notation</th>
<th>Example value (cycles)</th>
<th>Trigger command</th>
<th>Impacts future commands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burst Length</td>
<td>t\text{BURST} / t\text{BL}</td>
<td>4</td>
<td>READ, WRITE</td>
<td>READ, WRITE</td>
<td>Duration of data burst on the data bus</td>
</tr>
<tr>
<td>Column-to-Column Delay</td>
<td>t\text{CCD}</td>
<td>4</td>
<td>READ, WRITE</td>
<td>READ, WRITE</td>
<td>minimum distance between issuing two column commands to allow for data burst</td>
</tr>
<tr>
<td>Column Access Strobe / Column Latency</td>
<td>t\text{CAS} / t\text{CL}</td>
<td>11</td>
<td>READ, WRITE</td>
<td>all</td>
<td>Duration between issuing a column-read command and start of data burst</td>
</tr>
<tr>
<td>Row Access Strobe</td>
<td>t\text{RAS}</td>
<td>28</td>
<td>ACTIVATE</td>
<td>PRECHARGE</td>
<td>minimal time required between activating a row and issuing a precharge command to the same bank</td>
</tr>
<tr>
<td>Row-to-Column Delay</td>
<td>t\text{RCD}</td>
<td>11</td>
<td>ACTIVATE</td>
<td>READ, WRITE</td>
<td>minimal time between activating a row and accessing the data</td>
</tr>
<tr>
<td>Command Transport Duration</td>
<td>t\text{CMD}</td>
<td>1</td>
<td>all</td>
<td>all</td>
<td>Duration of transfer over command bus</td>
</tr>
<tr>
<td>Column Write Delay / Column Write Latency</td>
<td>t\text{CWD} / t\text{CWL}</td>
<td>8</td>
<td>WRITE</td>
<td>all</td>
<td>time between issuing a write command and starting a data burst</td>
</tr>
<tr>
<td>Write-To-Read delay</td>
<td>t\text{WTR}</td>
<td>6</td>
<td>WRITE</td>
<td>READ</td>
<td>minimal time after end of write burst before read command can be issued</td>
</tr>
<tr>
<td>Row Precharge</td>
<td>t\text{RP}</td>
<td>11</td>
<td>PRECHARGE</td>
<td>all</td>
<td>time needed to precharge an array</td>
</tr>
</tbody>
</table>
### Logical Organisation of DRAM

The logical organisation of DRAM can be divided into five layers: channel, rank, bank, row and column. The channel and rank layers are equal in the physical and the logical organisations, but the logical bank, row, and column layers differ from the equivalents in the physical organisation. Logical organisation is used during the address mapping, which is a process where the request address is translated into the location within the memory system. The organisation is shown in Figure 4. The grey areas in the open row of the bank 0 of rank 0 represent a single access worth of data.

#### 2.3.1 Logical channel

The first layer, channel, divides the main memory into independent sections of storages. Each channel can be controlled by a single memory controller, or each channel can have its own, separate controller. From the physical perspective, each channel is fully independent, with its own controller ports, buses and DIMM modules. Most proposals envision controllers oblivious to the existence of other controllers/channels. Depending on the implementation, memory controllers of different channels may share some information. For example, Kim et al. [3]
propose a controller design which determines the thread priority globally, requiring synchronisation between *channels*.

![Diagram of DRAM organisation](image)

**Figure 4 Logical organisation of DRAM**

### 2.3.2 Logical *rank*

The second layer of logical organisation, *ranks*, corresponds to the physical *ranks* present in the system. Different *ranks* in one *channel* operate independently from one another, but they share the control, address and data buses connecting them to the *memory controller*. Additionally, switching between the *ranks* requires the resynchronisation of the read-write data strobe, which creates a timing requirement that has to be taken into consideration by the *memory controller*. The timing requirement and the sharing of the data bus limit the amount of parallelism that can be extracted from different *ranks* of the same *channel*.

### 2.3.3 Logical *bank*

*Bank* s form the third layer of the logical organisation of *DRAM*. The concept shares the name with the *banks* of *memory arrays* in the *DRAM devices*, and is based on it. The number of logical *banks* is equal to the number of physical *banks* of a single *DRAM device*. The logical *bank* X is a set of all physical *banks* X of the *DRAM devices* which form the *rank*. This organisation allows
the memory controller to manipulate the logical banks and ignore the individual DRAM devices. Banks independently execute PRECHARGE, ACTIVATE, READ and WRITE commands, which can provide a greater level of parallelism than between ranks, since there is no switching penalty. However, REFRESH commands and power-down modes are managed and executed at the rank level, which limits independence of the banks.

2.3.4 Logical row

The row layer of the DRAM organisation is a large set of columns over one logical bank. Each bank can have one open row at a time. Closing and opening a row takes time, so a request to the currently open row, called row-hit, can be completed in a fraction of time necessary for a column access to the non-open row (row-miss).

2.3.5 Logical column

DRAM operates in burst mode, transferring multiple beats of data for every access. Burst length is a fixed value defined by the DRAM specification. For example, DDR3 SDRAM [4] has a burst length of eight, which takes four cycles to transfer, although it possesses the ability to use a “chopped burst” with the length of four. Logical column specifies the sequential set of physical columns which hold the desired data. Hence, a logical column is a set of burst-length physical columns within one logical row.

2.3.6 Address Mapping Process

The address mapping process takes an address from the system’s memory address space and maps it onto the logical organisation of DRAM. Specifically, the address mapping translates a request address into a set of <channel (L), rank (K), bank (B), row (R), column (C)> values, specifying the location of the data. The address mapping scheme determines which bits of the address are used for which of the values, and this choice has a significant impact on performance. The lowest bits of an address, which specify the offset within a cacheline, are discarded.

For example, two requests accessing different rows of the same bank would need to close the first row and open the second, which increases the latency significantly. Using the example values from Table 1, it would take 61 DRAM cycles to service both requests, provided the bank
was precharged when the servicing started. The number is reached in following manner. First request would take $t_1 = t_{RCD} + t_{CAS}$ to be executed, and then a PRECHARGE command would be sent. However, PRECHARGE can only be sent $t_{RAS}$ cycles after ACTIVATE command, which is greater than $t_1$. Thus, it would be $t_{RAS} + t_{RP}$ before a new ACTIVATE could be issued, for a total of $t_{RAS} + t_{RP} + t_{RCD} + t_{CAS} = 61$ cycles. If the requests instead were to access the same row but different columns, they could be executed one after the other after the initial access. This would take only $t_{RCD} + 2*t_{CAS} = 33$ cycles, in addition to using less energy by avoiding the extra ACTIVATE and PRECHARGE commands.

Another option would be to have the requests access different banks, in which case both would be finished in $t_{RCD} + t_{CAS} = 22$ cycles from issuing the ACTIVATE commands, although the second ACTIVATE would need to be delayed five cycles to satisfy the $t_{RRD}$ timing constraint. This delay would subsume the four cycle delay necessary to avoid the collision on the data bus. Thus the total execution time would be 27 cycles. However, this would use more energy, since two rows would be activated instead of a single one. Accessing two different channels would be even more advantageous in terms of speed, since they are fully independent, but the energy problem would remain.

All these considerations impact the performance, but only in relation to the stream of addresses. Applications typically have spatial locality, which means addresses with difference in lower bits of the address are to be expected within a short time frame. This leads us to using the lower bits for either the column value, allowing sequential accesses and amortising the costs of the ACTIVATE and PRECHARGE more column access commands, or using it for the bank, rank or channel values to leverage parallelism. However, obtaining parallelism through using the lower bits, which we expect to be changed often, can hurt long-term performance by increasing the number of accesses to the same bank which require different rows, thus increasing the need for opening and closing the rows. The frequent changes between the rank and channel values can further hurt the energy usage by reducing opportunities and durations for power-saving modes.

2.4 DRAM Behaviour Metrics

A set of parameters can be used to describe the DRAM usage pattern and performance in addition to the achieved throughput, or bandwidth. Both the request pattern and the memory controller behaviour affect these parameters. This section defines the following parameters:
a. DRAM utilisation
b. DRAM efficiency
c. Row buffer locality

**DRAM utilisation** is the percentage of cycles within a time period the data bus was used to transfer the data [5]. While this value is preferred to be high, it can be misleading as a scheduler performance measure if there were significant intervals with no requests at the controller.

**DRAM efficiency** was defined by Yuan and Aamodt to improve upon **DRAM utilisation** [5]. It is defined as the percentage of the non-idle **DRAM** cycles the data bus was utilised. This removes the time during which the controller and **DRAM devices** had no work to do, and presents a more indicative performance measure if there were significant idle times.

**Row buffer locality** is defined as the percentage of requests which access an already open row. High locality means the cost of opening a row (issuing the **ACTIVATE**, and later **PRECHARGE**) commands is amortised over a large number of serviced requests. However, high locality can have a negative side-effect of delaying requests accessing different rows of the same **bank**. Kaseridis et al. points out that the modern multithreaded systems have a high chance of inter-thread interference by keeping a **bank** occupied through repeated accesses to a single row [6].

### 2.5 Memory Controller

The main memory system, as noted earlier, consists of the storage devices and a memory controller. To hide the hardware details from the rest of the system, the memory controller acts as a translation and control wrapper for the storage devices. It receives memory requests and issues appropriate commands to the **DRAM devices**. It can be divided into two layers: the **outer layer** which communicates with other components of the computer system, and the **inner layer** which controls the devices.

The term “**memory scheduler**” is often used in a way which overlaps with **memory controller**. Here we will use **memory scheduler** to mean the part of a **memory controller** which selects the specific **DRAM** command to be issued, forming the essential part of the inner layer.

The outer layer receives memory requests and places them into a single large **transaction queue** (TQ). Often, writes and their data are placed into a separate **write queue** (WQ). Any incoming read requests need to be checked against the write requests waiting in the **write queue** to maintain
consistency. This also provides a small chance of immediately servicing a read request without the necessity of going to the DRAM devices.

Requests from the transaction and write queues are translated into a series of DRAM commands and handed to the second layer. The simplest translation is to wrap every incoming request into a PRECHARGE, ACTIVATE and a READ or a WRITE command. Depending on the system, a PRECHARGE command will be considered either as the first or as the last command of the sequence. An alternative translation process does not create and place ACTIVATE or PRECHARGE commands into the command queues, but rather creates them as needed, based on the queued READ/WRITE commands and the state of the system.

After translation, these commands are placed in the inner layers’ command queue (CQ). There can be a single command queue per memory controller, one command queue per rank, or even one command queue per bank. The memory scheduler issues the commands to the devices. It keeps track of the timing constraints, only issuing requests once all applicable constraints are met. It also has to make sure that no data is lost due to the capacitor leakage by generating REFRESH commands at the appropriate time.

The rest of this section is organised as follows. Section 2.5.1 presents the simplest scheduling policy, and considers its inefficiencies. Sections 2.5.2, 2.5.3, and 2.5.4 present the schedulers we evaluate in Chapter 4. The baseline modern memory scheduler is First Ready – First Come, First Served (FR-FCFS). In addition, we explain the Thread-Fair memory scheduler and the Thread Clustering memory schedulers. The Thread Fair scheduler [7] was a participant in the Memory Controller Championship [8] organised as the 3rd annual Workshop on Computer Architecture Competition, held in conjunction with ISCA-39. We chose it due to its relatively simple design and success at the Championship. The Thread Clustering scheduler [3], presented in Section 2.5.4, was chosen since it targets modern chip multiprocessors, with mixed, heavily multithreaded workloads.

2.5.1 In-Order Scheduler

An in-order scheduler issues commands in strict order of arrival. The basic variant of an in-order scheduler issues PRECHARGE and ACTIVATE commands before every access, though more
conventionally, those commands are skipped if the appropriate row is already open. In either case, in-order scheduling is highly inefficient.

Figure 5 compares (a) an in-order scheduler which executes PRECHARGE and ACTIVATE before every column access, (b) an in-order scheduler which can skip superfluous commands, and (c) a simple out-of-order scheduler. All commands are directed to the same channel, rank and bank, while the commands in the same colour access the same row. PRECHARGE commands do not use the row value. Issuing PRECHARGE and ACTIVATE commands before every access is extremely inefficient and wasteful, as can be noticed by comparing cases (a) and (b). The time required for the accesses going to the same row past the first is more than tripled by repeated PRECHARGE and ACTIVATE commands, due to the timing constraint for issuing the PRECHARGE after the ACTIVATE command.

Out-of-order scheduling (c) further improves performance. In the figure below, by executing the fourth READ out-of-order when the bank is open, its latency is reduced by over five periods, while the third READs latency is increased by only one time period. More advanced techniques for command scheduling can exploit the advantages provided by multiple row accesses and rank/bank parallelism, but also take into account which requests are more important to the overall system.

---

**Requests in arrival order**

| READ | READ | READ | READ | READ | PRE | ACT | READ | PRE | ACT | READ | PRE | ACT | READ |

---

**a) In-order scheduling**

---

**b) In-order scheduling with PRECHARGE-ACTIVATE only if necessary**

---

**c) Out-of-order scheduling potential**

---

**Legend:**

Access rank r, bank b, row X
Access rank r, bank b, row Y=G

---

**Figure 5 Example of the impact of scheduling on performance**
All requests are not equal in usefulness for the issuing device. Scheduling requests optimally from the DRAM perspective will provide the best bandwidth utilisation, but this is not necessarily the optimal scheduling from the system perspective. Some modern memory schedulers such as the Thread Clustering scheduler attempt to take this into account and often require additional data from the issuing devices [3].

2.5.2 First Ready - First Come, First Served memory scheduler

The First Ready - First Come, First Served (FR-FCFS) scheduler is a simple, greedy request scheduling policy originally presented by Rixner et al. [9]. There are several possible variations which change the behaviour significantly. It is often used as a baseline memory scheduling policy due to its low complexity coupled with acceptable performance. In this section, we describe the basic operation and the possible variations of the scheduler.

The base version of the FR-FCFS scheduler chooses a command every cycle. It considers all ACTIVATE, READ and WRITE that satisfy timing constraints, hence the First Ready part of the name. Out of those able to be issued, the oldest command will be chosen; First Come, First Served. This means an newer command will be issued over an older one if the timing constraints prevent the older from issuing. PRECHARGE commands are only considered if no other commands can be issued. The specific timing of the PRECHARGE command depends on the schedulers’ row-status policy, which we examine next.

2.5.2.1 Row-status policy

The FR-FCFS schedulers’ row-status policy determines the rule for closing the row, which is tied with the choice of maximum number of row accesses. There are three options: single access, open-row and closed-row.

The single access policy allows only a single access to a row before it is closed, resulting in a comparatively simple scheduler. Since a PRECHARGE command will be issued after every access, we could use auto-precharge (READ-PRECHARGE and WRITE-PRECHARGE commands). However, even with the auto-precharge commands, this method of access is highly inefficient in both performance and power/energy terms. Figure 5 shows an example of this inefficiency, although the out-of-order scheduling would allow parallelism across ranks and banks.
Allowing multiple row accesses improves performance, but raises a question of when to close the row. A row could be closed once there are no more accesses to it in the command queue, which would speed up future accesses to the different row of the bank in question by allowing them to start immediately with ACTIVATE command. However, this can also slow down future accesses if they would have hit in the row that was open. Thus, we define the closed-row policy as sending the PRECHARGE command as soon as the command queue contains no accesses to the open row. The open-row policy is defined as sending PRECHARGE command only once the command queue contains no accesses to the open row and at least one access to a different row of the same bank.

Preventing the issuing of the PRECHARGE command while the command queue holds access commands to the open row can lead to starvation of other commands trying to access the same bank but a different row. To prevent this possibility, the scheduler can enforce a maximum number of row hits before PRECHARGE will be issued even if there are more accesses to the same row in the command queue.

2.5.2.2 Preferred command policy

By default, FR-FCFS prefers older command over newer commands. This policy can be extended by preferring column commands (READ/WRITE) over row (ACTIVATE/PRECHARGE) commands, or vice-versa, and is called the preferred command policy. In either case, the commands of the non-preferred group are issued only if there are no commands from the preferred group that can be issued. The choice of commands within the group is still done according to the arrival time. Preferring column commands has the potential to reduce the latency by servicing the requests quickly, but can delay closing or opening of rows, which can potentially reduce available parallelism. Preferring row commands seeks to increase available parallelism by maximizing the number of (useful) rows open.

2.5.2.3 READ versus WRITE commands

The base FR-FCFS policy treats both READ and WRITE commands the same. However, writing data into the DRAM devices requires changing the direction of transfer on data bus, which takes additional time. Ignoring the command types may lead to turning the bus more times than was necessary and delay execution. Simple improvement is to issue all READ commands to a row
before issuing all WRITE commands, which can reduce the number of bus switching for any one row. However, it does not take into account inter-bank interference due to the shared bus. An attempt to solve this problem is to use the Write Drain mode.

2.5.2.4 The Write Drain mode

The Write Drain mode is an extension of the basic FR-FCFS targeted at amortising the bus turnaround over as many writes as possible, and depends on the write queue parameters. Chatterjee et al. [8] use the FR-FCFS with the Write Drain mode as one of the baseline schedulers provided with their infrastructure.

When a write request arrives at the controller, it is placed into the write queue. Write requests are not translated and moved into the command queue as soon as possible, but kept in the write queue until the queue is filled to a specified limit, called high watermark. Once the number of requests in write queue exceeds the mark, the scheduler switches into the write drain mode. While in the write drain mode, only write requests are translated and issued to the devices until their number falls below a low watermark. Read requests are not issued. In case there are no read requests in the queues, write requests will be processed. The delaying of write requests in this manner is acceptable since they are non-blocking in most architectures and can be postponed without impacting performance.

2.5.2.5 FR-FCFS scheduler summary

The simplicity of First Ready – First Come, First Served scheduler is the primary reason it is often used as a baseline scheduler. While in-order scheduler is even simpler, FR-FCFS provides a much better base for comparison by tapping into the available parallelism of the main memory system, while being easy to implement.

2.5.3 Thread-Fair memory scheduler

The Thread-Fair scheduler was presented by Fang et al. [7]. The scheduler follows a set of simple rules, and uses additional information from the issuing device to prioritise oldest request from each thread. This is done since the oldest request is expected to have the highest chance of blocking the device’s progress. The scheduling rules are summarised in Figure 6. Under normal operation, read requests are prioritised over write requests. If the write queue is filled above the
high watermark, or there are no read requests, enter the write drain mode in which writes have higher priority.

While read requests are being serviced, read row hits have the highest priority. If no read row hits can be issued, the request generated by the reorder buffer (ROB) head is given the highest priority and appropriate row is opened. If multiple such requests exist, they are considered in a round-robin fashion. If all requests generated by the ROB heads have the corresponding rows open, or the required banks are busy, rows are opened for other read requests by the oldest-first rule. Finally, if no row opening commands can be issued, write row hits are considered for issuing.

Correspondingly, when in the write drain mode, write row hits have the highest priority. If the queue holds no row hits, remaining writes are serviced according to age. Finally, if no write requests can be serviced, read row hits will be issued.

![Decision flowchart for the Thread Fair scheduler](insert_flowchart)

**Figure 6 Decision flowchart for the Thread Fair scheduler**

The Thread-Fair scheduler takes advantage of the auto-precharge commands. When issuing column access commands, if there are no more pending accesses that would hit in the open row,
last column access is issued as the *auto-precharge* command. This way does have the potential of closing the *row* too soon.

Overall, the *Thread-Fair* scheduler has a straightforward policy which maintains fairness through prioritising the potentially blocking requests when opening a *row*. While this requires additional information to be supplied from the issuing hardware element, the authors offer the alternative of preferring the oldest request from each thread when opening the *row*.

### 2.5.4 *Thread Clustering* memory scheduler

The *Thread Clustering* scheduler was proposed by Kim et al. [3]. It targets modern chip-multiprocessor systems with many concurrently executing threads. The scheduler aims to satisfy both system throughput and fairness by assigning priority based on the coarse-grained, periodical division of the threads into two groups (or clusters). Priority is assigned based on thread behaviour across all *channels* in the system, which requires information to be unified across the *channels*. To determine the threads group, the following information is collected about each thread during the execution period: the total bandwidth usage of the thread, the *row buffer locality* and the average number of *banks* accessed.

The two clusters into which the threads are divided are the *latency-sensitive* and the *bandwidth-sensitive*. The *latency-sensitive* threads are characterised with infrequent memory accesses, which also implies their performance is limited by the speed of the memory subsystem. The scheduler prioritises memory requests from these threads, since it is expected they will be stalled waiting for the few requests they issue. Fairness should not be severely impacted since the presumably small number of accesses from these threads means most of the time can be spent servicing other threads.

The *bandwidth-sensitive* threads issue a large number of memory requests, which implies high memory level parallelism. This means the speed of service for any individual request is not crucial, so they can be delayed behind the *latency-sensitive* threads. The ordering within the group is not as straightforward. A concern is that some *bandwidth-sensitive* threads can behave unfairly to other threads, by keeping a *bank* occupied with one particular *row* and blocking all other threads which want to access the same *bank*. The authors propose a ranking method, *thread niceness*, and a sorting algorithm which they call *insertion shuffle* to regularly reprioritise the
 bandwidth-sensitive threads in relation to one-another. The goal is to preserve fairness by giving each thread equal time at highest priority within the cluster, yet keeping the worst behaving threads with low priority at all other times.

The following section explains how the threads are divided into the two clusters. Section 2.5.4.2 explains the priority assignment within the latency-sensitive cluster, while Section 2.5.4.3 explains the same for the bandwidth-sensitive cluster and looks at the thread niceness metric. Section 2.5.4.4 describes the insertion shuffle, and we conclude in Section 2.5.4.5.

2.5.4.1 Dividing threads into clusters

The scheduling is done according to a thread-level priority mapping assigned at regular intervals. The proposed length of these intervals is one million cycles. Requests from the latency-sensitive threads have fixed priority ordering within the cluster and overall priority over requests from bandwidth-sensitive threads. Bandwidth-sensitive threads are ordered within the cluster, but the priority assignment of these threads is shuffled in regular intervals.

The threads are divided into clusters every time interval in the following manner. A bandwidth percentage limit \( B_{LS} \) for the latency-sensitive cluster is set to \( x/N \), where \( x \) is defined at the system start-up and \( N \) is the number of threads in the system. The authors recommend \( x \) to be between two and six. This limit is relative to the total bandwidth usage in the previous interval, rather than maximum possible bandwidth usage.

The threads are ordered by the miss-per-kilo-instruction (MPKI) value, which is provided by the devices running the threads. The thread with the lowest MPKI value is considered for assignment to the latency-sensitive cluster. If the percentage of the bandwidth used by the thread is less than the \( B_{LS} \) limit, the thread is added to the cluster and the thread with the next lowest MPKI value is considered. If the combined percentage of the threads in the latency-sensitive cluster and the new thread is lower than the limit, the thread will be assigned to the cluster and the next thread considered. If the combined percentage would equal or exceed the limit, the thread is not added and the process ends. The bandwidth-sensitive cluster is then formed out of all threads not assigned to the latency-sensitive cluster.
2.5.4.2 Priority assignment within the *latency-sensitive* cluster

The *latency-sensitive* cluster is formed out of *k* threads with the lowest *MPKI* values, and the priority is assigned according to the order in which the threads were added to the cluster. Thus, the thread with lowest *MPKI* will have the highest priority, but even the lowest priority among *latency-sensitive* threads is still higher than any thread from the *bandwidth-sensitive* cluster.

2.5.4.3 Priority assignment within the *bandwidth-sensitive* cluster

The ordering in the *bandwidth-sensitive* cluster is assigned according to the *niceness* metric. *Thread niceness* attempts to capture both the threads tendency to interfere with other threads and the vulnerability of the thread to interference. A thread with high *row buffer locality* will attempt to keep its *rows* open, thus preventing other accesses to the same *banks*, potentially starving other threads. On the other hand, a thread accessing a large number of *banks* in parallel can easily be interfered by another thread holding a *row* open in one of those *banks*.

Threads from the *bandwidth-sensitive* cluster are assigned ranking in the *row buffer locality* and bank parallelism metrics. Bank parallelism is the average number of different *banks* required by the threads’ requests in the *command queue*. *Row buffer locality* metric is dependent on the *rows* currently open in the *banks*, and in a multithreaded system this value is influenced by the concurrently running threads. Capturing native *row buffer locality* of a thread is done by a per-thread array holding the last row accessed in each *bank*. If the *row* required by the request is equal to the stored one, it is a *row buffer hit*; otherwise it will change the *row* in the array.

*Thread niceness* is calculated as the difference of the thread rankings in these metrics. If the thread is ranked *b*\textsuperscript{th} highest in *bank* parallelism and *r*\textsuperscript{th} highest in *row buffer locality*, the *niceness* value *n* of that thread is defined as *n* = *b* - *r*. The threads with higher *bank* parallelism will have higher *niceness*, while threads with higher *row buffer locality* will have lower *niceness*.

The *Bandwidth-sensitive* cluster is ordered by the *thread niceness*, and the initial priority is assigned accordingly. However, the *bandwidth-sensitive* cluster can suffer from fairness issues, so the relative thread priority is adjusted in regular intervals over the main time quanta. The adjustment is done by the insertion shuffle we explain in the following section. However, if the difference in *row buffer locality* and bank parallelism is less than 10% between all threads in the cluster, the priorities are assigned randomly since it indicates the threads behave similarly.
2.5.4.4 Insertion shuffle

The insertion shuffle is designed to give each thread equal time at highest priority, but keep the worst thread for longer periods with lowest priority. The shuffle is repeated at regular intervals, and follows two alternating phases, named A and B in the Figure 7. Each phase lasts $N$ shuffling intervals, where $N$ is the number of threads in the bandwidth-sensitive group. $s$ is a value between $1$ and $N$, corresponding to the index of the shuffling during one phase.

During the first phase, the shuffle assigns priority for the first $s$ nicest threads by increasing niceness values, while the rest are prioritised by decreasing niceness values. This means that during interval $s$, highest priority will be assigned to the $s^{th}$ nicest thread, while the worst thread will hold the lowest priority until the last ordering. Last ordering will assign priority in the opposite order of thread niceness.

Second phase starts with threads ordered by increasing niceness, placing the worst thread in the highest place. Every interval shuffle orders the $(N - s + 1)$ nicest threads in the increasing order of niceness. The remaining $(s - 1)$ worst threads are ordered in the decreasing order of niceness behind the nicer ones. Thus, during the first shuffle the threads will be left in increasing order of niceness, and every following interval the worst thread of those will be moved into the subgroup ordered by decreasing niceness.

<table>
<thead>
<tr>
<th>$t^*$ (SP)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>Max</td>
<td>T1 ($n = 8$)</td>
<td>T2</td>
<td>T3</td>
<td>T3</td>
<td>T2</td>
<td>T1</td>
<td>T1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T2 ($n = 5$)</td>
<td>T1</td>
<td>T2</td>
<td>T2</td>
<td>T1</td>
<td>T2</td>
<td>T2</td>
</tr>
<tr>
<td></td>
<td>Min</td>
<td>T3 ($n = 2$)</td>
<td>T3</td>
<td>T1</td>
<td>T1</td>
<td>T3</td>
<td>T3</td>
<td>T3</td>
</tr>
<tr>
<td>$S$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Phase</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

**Figure 7** Example of insertion shuffle through time; time is displayed in discrete shuffle periods (SP)

Figure 7 displays an example of insertion shuffle. Threads T1, T2 and T3, with niceness values of 8, 5 and 2, belong to the bandwidth-sensitive group. During the first shuffle period (SP), the
highest priority is assigned to the *nicest* thread, and the lowest priority to the least *nice* thread. Phase $B$ begins in period 4, and switches back to phase $A$ in period 7.

### 2.5.4.5 Thread Clustering scheduler summary

The *Thread Clustering* memory scheduler is designed for systems with many threads. It attempts to maximise both the throughput and fairness in the system by creating a coarse-grained division into the *latency-sensitive* and the *bandwidth-sensitive* clusters. The division is reformed in large time periods to allow for changes in the behaviour of individual threads. *Latency-sensitive* cluster is strictly prioritised for maximum throughput, while the priority of the threads in the *bandwidth-sensitive* cluster is shuffled at regular intervals to maintain fairness.

### 2.6 Related Work

This section presents a brief overview of other modern memory schedulers. We chose the *Thread Clustering* scheduler since it targets chip-multiprocessor systems, a potential direction of smartphone-like system hardware. The smartphones will still have different usage patterns and performance envelopes. The *Thread Fair* scheduler was chosen as a modern yet fairly simple scheduler which takes into account the existence of threads, but does not make it the primary scheduling parameter.

### 2.6.1 Performance-oriented schedulers

The common target for improvement is the performance of either the main memory subsystem or the applications running. The main memory subsystem performance is often tracked as the bandwidth utilisation or efficiency, explained in Section 2.4. While this value is preferred to be higher, it does not necessarily reflect the performance of the applications, which might find some requests more useful than others.

Hur and Lin propose an *Adaptive History-Based* scheduler which adjusts to the read/write ratio of the application [10]. The read/write ratio is collected and periodically used to select one of the available policies. One of the policies issues commands that will cause the least delay, while other policies try to maintain a predefined read/write ratio.

Modern multithreaded workloads need to address a potential problem of threads interfering between each other. Mutlu and Moscibroda target the problem of fairness with a *Stall-Time Fair*
Memory Access scheduler that tracks the slowdown of each individual thread, and prioritises the requests from the threads with the largest slowdown [11].

The Parallelism-Aware Batch scheduler also targets fairness as the primary optimisation goal [12]. The scheduler maintains fairness by creating a batch of requests, members of which are prioritised over other requests. Batching is done at the granularity of per-thread-per-bank to preserve and enhance the inherent bank parallelism, while ensuring all threads have equal access to individual banks.

Kim et al. design a scheduler targeting system throughput [13]. The basic idea of the ATLAS scheduler is to prioritise threads which have attained the least amount of service during the previous time period. Thread ranking is determined over the whole main memory system, based on the service threads acquired from all channels.

The scheduler proposed by Ikeda et al. uses three different mechanisms during scheduling [14]. The primary mechanism is called PC-based Gain Estimation, and estimates the number of cycles the CPU will be able to execute before the next memory request, once the current request is serviced. Requests with higher values have higher priority. However, due to collection mechanism, only a fraction of requests will have this prediction value. All other requests have their priority calculated as a sum of the number of read requests from the thread currently in the queue, and the priority from the thread clustering mechanism as described in the Thread Clustering scheduler [3]. The authors also observed that in the case of an empty read queue, waiting for a preset number of cycles before starting the write-drain decreases the probability of a read request arriving and being stuck behind large number of writes.

Ishii et al. [15] present a scheduler which assigns the thread priority by detecting the phases of thread execution which are memory intensive. The threads not in the memory intensive phase are prioritised over the threads in that phase. Additionally, while refreshing one rank, it executes write operations on the opposing rank, which aims to reduce the interference of writes with reads. This method detects thread execution phases at much finer granularity compared to other memory schedulers trying to do the similar

Memory scheduler design often targets a specific set of workloads, or several workload types with the ability to switch between performance modes. Ipek et al. [16] propose a Self Optimizing
scheduler able to learn the optimal scheduling policy based on the program behaviour. The design is based on a machine learning technique called reinforcement learning, and the schedulers’ goal is to maximise the main memory throughput in the form of bandwidth utilisation.

Building on the Self Optimizing scheduler, Mukundan and Martinez present a selection of techniques for automatically determining the previously fixed parameters [17]. These techniques show how to adjust the scheduler to target different metrics, such as power or energy instead of the bandwidth utilisation.

2.6.2 Power and energy oriented schedulers

Another common target is the power or the energy consumption, which is increasingly important in all modern systems. Energy usage is often tightly coupled with improving performance, whereas power considerations are often harmful to the application performance. The peak power usage is already limited by the four-row activation window to ensure the correct functioning of the hardware.

A simple technique for controlling the power usage of the DRAM subsystem, called memory throttling, is examined by Hanson and Rajamani [18]. Memory throttling restricts the maximum number of accesses within a fixed time period to below the normal hardware capability. The impact of this restriction can vary from minimal to significant.

Hur and Lin [19] use the memory throttling as one of the three mechanisms to manage the power usage of the DRAM. Second part of the proposal is to use a timeout counter to generate power-down commands. Finally, they extend the previously mentioned Adaptive History-Based scheduler [10] with an additional policy which prefers issuing commands only to one rank at a time, allowing for longer power-down periods.

Two energy-saving techniques are presented by Huang et al. [20]. The first proposed technique attempts to improve the self-refresh mode of DRAM. The mode saves a large amount of energy, but has a long delay when returning to normal operation. The idea is to use a history of accesses within last refresh interval to modify the time threshold for entering self-refresh mode. The second proposed technique is to move the most frequently accessed pages into a specified subset
of ranks, which would allow longer and more frequent power-saving modes for the other ranks. However, the page migration negatively impacts performance.

Refresh operations use a small but constant percentage of time to execute. Furthermore, open rows are closed and the whole rank is unavailable during the refresh operations. On the other hand, opening then closing the row refreshes it, meaning some rows will be unnecessarily refreshed. Ghosh and Lee propose a Smart Refresh mechanism which keeps track of the rows’ last refresh through either normal activation or the REFRESH command [21]. Rows are only refreshed as necessary. However, the counters require a large amount of storage in the memory controller.

Deng et al. approach the problem of power usage by reducing the frequency of the DIMM modules to limit power [22]. The scheduler detects the systems tolerance to diminished performance (“slack”), and adjusts the frequency accordingly. In addition, the authors propose extending the dynamic voltage and frequency scaling already used on CPU’s to the memory controller hardware island of the die.

2.6.3 Prefetch-oriented schedulers

Prefetching cache lines has the possibility of improving performance. However, even when it is accurate, its use during high memory utilisation can reduce performance due to the prefetches being executed before regular requests. Managing this problem can be alleviated by schedulers which take into account which requests are prefetches and adjusting the priority accordingly.

Lee et al. propose prioritising prefetches according to the prefetch accuracy of the issuing core [23]. The idea is that prefetches from low-accuracy cores will only be executed if there is plenty of bandwidth. If the bandwidth usage is high, low-accuracy prefetches will be delayed and potentially dropped.

Multiple accesses to the open row provide the opportunity to amortise the power and time cost of activating and precharging the row over multiple access commands. While this is highly beneficial for latency of the commands going to the same row, Kaseridis et al. observe than modern multi-core systems run different applications concurrently, and keeping a row open can diminish fairness and overall performance [6]. They propose an address mapping scheme which limits the number of row hits for sequential addresses. Accompanying the change in the scheme is a
\textit{Minimalist Open-Page} scheduler which prioritises read requests over prefetches. Read requests are also ordered in priority based on the number of outstanding L2 accesses in the issuing core’s Miss Status Holding Register, while prefetches are ordered inversely to the distance from use point.

Another approach is to place the prefetcher in the memory controller, called memory-side prefetching. Hur and Lin propose prefetching cache lines into a small prefetch buffer located within the \textit{memory controller} [24]. The prefetching is done only when there is sufficient bandwidth, with the goal of allowing the newly arrived requests to hit in the prefetch buffer and being serviced immediately.

\subsection*{2.6.4 Related Work Summary}

This section gives an overview of memory schedulers not implemented for this thesis. We roughly divide the schedulers into three groups. The groups are not definitive, with many schedulers falling into two or possibly even all three of these groups, but are used for ease of reading. First group are formed by schedulers whose primary aim is to improve either application or memory subsystem performance. Second group primarily targets energy and power usage. Schedulers from the third group are concerned with handling the prefetches.
Chapter 3
Methodology

This chapter presents the motivation behind the thesis, as well as the infrastructure we used for the investigations presented in chapter 4. Section 3.1 discusses the smartphone systems and presents the problems in investigating them. Section 3.2 presents our tool for creating the traces, while Section 3.3 presents the tool for producing the memory request traffic based on the collected traces. We present the trace storage method used for our infrastructure in Section 3.4. Section 3.5 presents the workload we designed using our traffic generator tool. Section 3.6 presents the main memory simulator and modifications we performed on it, and we give a summary in Section 3.7.

3.1 Motivation

The aim of this thesis is to explore memory controller behaviour for workloads typical of a smartphone-like system. Smartphone capabilities and organisation are somewhat similar to several years old personal computers, but they operate under different constraints and common workloads. The energy and power used during operation is strictly limited, and one of the readily available optimisations is to use specialised hardware for the common operations.
Modern smartphones almost always contain a camera for taking photographs and videos, whose capabilities are increasing with new models. The images or video stream collected by the camera have to be encoded for storage and possibly communication, which is provided by a hardware implementation. In addition to creating the videos, the ability to view images and videos requires a corresponding ability to decode the same. The encoding and decoding elements are particularly heavily stressed if the video is being streamed to or from another device. Another example of the hardware sensors which are increasingly common are the acceleration and motion sensors.

An example of a smartphone-like system is presented in Figure 8. While the operation of each element is independent of any other, the coordination and data exchange is realised through the main memory subsystem. Exploring the behaviour of the main memory under the characteristic workloads would best be served by a timing model of every element in the system. We use an accurate timing model of the main memory subsystem, and we present a trace-based method for modelling disparate elements as a substitute for accurate timing models. The trace-based method we implemented collects information about the data flow between instructions to capture the dependencies between memory requests.

Our infrastructure consists of traffic generators and a main memory timing simulator DRAMSim2, as shown in Figure 9. We need to model the hardware accelerators which would be a part of the typical smartphone-like system. Under ideal circumstances, we would have access to a design of such accelerator and could use it to model the memory access behaviour. At the time of writing this thesis, there were no publicly available designs which we could use for some of the most important accelerators. Specifically, we were interested in the h.264 video encoding and decoding accelerators for the 1080p resolution.

Since we did not have access to necessary accelerators, we aim to model the trace pattern of the accelerators by using the software implementation of the corresponding application. However, the designers of the hardware device would know what data is useful in the future or the running of the program, and what data can be discarded or saved to the main memory. Similarly, the designers could build the hardware to execute several operations in parallel when possible, or introduce other optimisations unfeasible in a software implementation. Thus, the memory request stream of the software application is different from the stream of the corresponding hardware device, since the device would have specialised data structures and computational optimisations.
Our tool approximates the access pattern of the hardware accelerator by using a cache to filter the accesses from the software implementation of the targeted technology. We use a cache to try and capture the working set and thus retain the overall pattern, while removing the excess traffic expected from the software implementation. Furthermore, we introduce the ability to limit long delays between requests since we expect the optimisations such as parallelisation to improve the segments where long computation is limiting the performance.

This still leaves a problem common to trace-based traffic: relations between requests. In a real system, not all memory accesses are equal since information from some is only needed for a few instructions, while the others might be used in a long stream of instructions. Furthermore, some streams may produce many relatively independent requests, while others will require earlier requests to obtain the information before new requests can be issued. Thus, the order in which requests are serviced by the main memory system can either speed up or slow down performance, and in a typical trace-based system, this will not be reproduced. This is further compounded by the limits on the number of requests that a device can issue and the memory subsystem can hold in its queues.

We approach this problem by tracking the dependencies between instructions to discover which requests return information useful for the following request, and thus create a dataflow graph between them. The edges of this graph we term dependencies. We describe how we capture this information process in the next section.
3.2 The Trace Collection Tool

Our trace-based system captures the dataflow relationships between requests using a trace collection tool which creates the memory request trace with the dataflow information based on the real application runtime using Pin [25], a dynamic binary instrumentation tool. Pin instruments the application and manages the insertion of the analysis code between instructions of the original application, though system calls are not instrumented.

We define the terms dependent, dependency and dependency graph for use in the following sections. Node $A$ is dependent, or has a dependency, on node $B$ if node $B$ produces data node $A$ uses. A Dependency graph is formed with instructions as vertices and dependencies as edges. We assign transitive property to dependency, which enables the removal of non-memory instructions while retaining the dependency graph connections. Figure 10 shows an example of a dependency graph among instructions. Removing the non-memory instructions would leave loads $X$, $Y$ and $Z$, and store $P$. Loads $Y$ and $Z$ dependent on load $X$, while store $P$ is dependent on load $Y$. Load $Z$ is not dependent on load $Y$ because the instructions which produced its data did not use the data obtained from load $Y$.

![Figure 10 Dependency graph](image)

Our tool constructs a dependency graph between memory requests based on the data dependencies in the instruction stream. The dependency graph shows data flow between requests, which will be used when reproducing the trace to prevent memory requests from issuing before their dependencies have finished.

Each cache access is assigned a unique identifier, called the cache access identifier (CAI). The identifier is equal to the instructions count of the instruction that caused it. If the instruction
causes multiple cache accesses, the count is increased to maintain the uniqueness. This provides an additional benefit of sorting accesses into a temporal order.

Memory accesses are also assigned an almost unique identifier, the memory access identifier (MAI), which is identical to the cache access identifier of the access which missed. Since we use the writeback cache, this means the write accesses will all have the identifier identical to the paired read access. The identifier is still unique within the group of all read accesses, and within the group of all write accesses. Since only memory reads can be dependencies, the identifier is unique for our requirements.

The trace collection tool tracks the flow of data within the instruction stream, and determines the dependencies between cache accesses. This process is explained in Section 3.2.1. Section 3.2.2 looks at the information stored within the cache. Section 3.2.3 explains how the stored information is used to determine the dependencies between memory accesses. We summarise in Section 3.2.4.

### 3.2.1 Creating the dependency graph between the instructions

Instructions pass the data between themselves using the registers, and we track the dependencies through the registers. Instruction cannot start executing before its source registers are all ready. Once the execution starts, the destination registers of an instruction cannot be used by other instructions until the execution finishes. Thus all destination registers depend on all the source registers of an instruction. Any previous dependencies are removed, since we do not care about the register usage, but rather the flow of data. If the data from the destination register is used in the calculation, it will also be listed as a source register, thus preserving the dependencies.

The only exceptions to this rule are the cache loads. When a cache load happens, it is dependent on the source registers, and this is noted with the cache access. The load needs to complete before the destination registers are available for following instructions. Thus, the destination registers are dependent on the load itself. However, they are not dependent on anything else since all the previous dependencies will need to be satisfied for the load itself to happen, and the load will place the latest data in the register. The next section examines how the dependency information associated with the cache accesses is used within the cache.
3.2.2 Tracking the dataflow within the cache

The cache is the key element of our trace collection tool. In addition to the normal cache operation, it tracks the dependencies of the cache accesses and translates them into dependencies between the memory accesses. We implemented a 4-way set-associative, write-allocate writeback cache with LRU replacement policy, which we extended with the dependency tracking ability. We considered this configuration to be adequate for filtering the temporary data. Each cacheline is 64 bytes to match our main memory subsystem. The total cache size is variable, to allow adjustment for different applications.

The cache needs to track the dependencies between the cache accesses and translate them into the dependencies between the memory accesses. To achieve this, each cacheline holds the information on the memory access which brought it into the cache, the cache accesses which loaded information from it, and the information about the data written to the cacheline. The latter two are kept at a word (64 bits) granularity within the cacheline, to avoid the false dependencies between a store to one word and a later load from another word of the same cacheline. This still leaves potential false dependencies between individual bytes of a word.

We also limit the number of dependencies in any one list at 50 by removing the oldest whenever the list grows over the limit. This approach may lose some dataflow information, potentially important. It was implemented to limit the execution times.

The cacheline holds the following information on the memory access which brought it into the cache. It holds the unique memory access identifier of the access and the identifiers of the dependencies of that access. This information is used to resolve indirectly satisfied dependencies for future accesses which depend on the cacheline. An example of an indirectly satisfied dependency is shown in Figure 11. Request $A$ depends on requests $B$ and $C$, and $B$ depends on $C$. Dependence $A \rightarrow C$ can thus be discarded, since $C$ will have to finish before $B$ is allowed to proceed.

A word of the cacheline can be altered by the cache stores. If this happens, additional memory access dependencies are introduced alongside the memory access which brought the cacheline into the cache. These dependencies are based on the cache access dependencies of the last store to the word. The process of translating the cache access into memory access dependencies is
described in the following section. This process also requires the cache to keep track of which word was read by a particular cache access. Therefore, each word of the cacheline also maintains a list of *cache access identifiers* which accessed the word.

![Diagram](image)

**Figure 11 Indirectly satisfied dependencies**

Once a cacheline is evicted from the cache, it would still hold potentially useful information. Some of the data currently in registers could be dependent on the data originally obtained from the cacheline, even though it is not present in the cache anymore. Ideally, we would track all evicted cachelines for future use. The size of the cacheline history is an adjustable parameter of our tool. Any *dependencies* which require the cachelines evicted from both the cache and this history queue are discarded.

### 3.2.3 Determining memory access dependencies

Memory accesses are formed when a cache access misses in the cache. A read request will be formed for the required address, and possibly a write request if the evicted cacheline was dirty. These two requests will be assigned the misses’ *cache access identifiers* as their own *memory access identifier*. The *dependencies* of the read request are based on the *dependencies* of the cache miss. The *dependencies* of the write requests are based on the *dependencies* of the cache accesses which wrote to the cacheline, as well as the original access which brought the cacheline into the cache.

When a cache miss or a cache store happens, a process for translating the list of cache access *dependencies* into a list of memory access *dependencies* is executed. Every cache access *dependency* is a *cache access identifier* of a previous cache load, and for each *dependency* the accessed cacheline is found. The memory access which brought the cacheline into the cache is marked as the memory access *dependency* of the new memory access. Furthermore, if the word
accessed was written to, the memory access dependencies of the store, held within the cacheline, are also added to the list of memory access dependencies.

Once the list of memory access dependencies is completed, it is checked again against the cachelines to remove indirectly satisfied dependencies, as described before. If the cache miss happened, the finished list is stored in the trace as the memory access dependencies of the resulting read request. If it was a cache store, the list is stored with the target word of the cacheline.

Once the cacheline is evicted, the write requests’ dependency list is formed by combining these lists from all the words within the cacheline. While the memory access which brought the cacheline into the cache is also a dependency, we do not add it to the list since our trace-based traffic generator will satisfy it automatically by preventing the issuing of a write request if an older read request to the same address has not yet been issued.

3.2.4 Trace collection tool summary

In this section we described the way we generate traces. Our traces contain a stream of memory accesses with dependency information. Dependency information specifies which earlier accesses need to finish before the current request can be issued. The method is not perfect and can capture false dependencies. An interesting avenue for future work would be to extend the tool to capture the control flow as well as the data flow. Another possible enhancement would be to track the number of instructions which modified the data between any two dependent accesses.

3.3 Trace-Based Traffic Generator

The trace-based traffic generator is designed to reproduce the trace collected by our tool while maintaining the dependencies between the requests. The generator loads and then issues the requests from the trace. The requests are only issued after all of their dependencies are satisfied and the time requirements are met. Once the main memory services the request, its finishing time is noted and it is removed as a dependency from the following requests. We also provide the ability to approximate the computation time between the dependent requests.

Our generator requires a notion of cycles passing, and keeps a local clock. The ratio between the two is set during the initialisation. The clock is needed to limit the issuing of requests. The delay
between a dependency finishing and the dependent request is calculated through the use of the request identifiers, exploiting their origin as the instruction count. In addition to the information received with the request, it calculates misses per million instructions and provides this data with the request.

The generator loads the requests from the trace and places them into the waiting requests queue. This process is described in Section 3.3.1, while the way their dependencies are handled is described in Section 3.3.2. The issued requests queues, which hold the in-flight requests, are described in Section 3.3.3. Section 3.3.4 briefly describes the finished reads queue, and we conclude in Section 3.3.5. All queues have a maximum size set at initialisation time. The size of the queues influences the performance of the generator.

### 3.3.1 Waiting requests queue

The generator loads requests from the trace and places them into a waiting requests queue. The requests stay in this queue until they can be issued to the main memory. When a request is issued, it is moved to the appropriate issued requests queue, and a new request is loaded into the waiting requests queue from the trace. Increasing the size of the waiting requests queue allows more requests to be considered for issuing.

Requests are considered issuable when all their dependencies are satisfied, the time requirement is satisfied and the appropriate (read/write) issued requests queue is not full. Additionally, we require all earlier requests of the opposite type (read – write) for the same address have been issued. This maintains the proper ordering and models consistency requirements.

We define the time requirement of a request as local cycle being equal or greater than all of the following:

1. Request identifier (original instruction count),
2. For every dependency: finished cycle of every dependency plus difference between identifiers of the request and the dependency.

The delay between requests is to model the time needed to perform the computations with the used data before it could be used in our new request. We provided an option to set a maximum limit on this delay, on the assumption that a hardware implementation will reduce the computational limitations. This option can also be used to turn off the delays.
3.3.2 **Dependency handling**

A request loaded from the trace file may have a set of dependencies. The dependencies are identifiers of preceding requests which provided the data used for the current request. All dependencies have to be serviced by the main memory before the current request can be issued. Since the trace contains the requests in the temporal order, we know all dependencies have been loaded previously. However, we do not know the status of these dependencies.

The dependencies are first checked against the finished reads queue. Any dependency that is found in the queue is satisfied, and the appropriate delay is assigned to the current request. Any remaining dependencies are checked against the waiting requests and the issued reads queues. If the dependency is in one of these queues, it means it has not yet been serviced and the current request needs to wait. Any dependencies not found in any of these queues are simply discarded.

The rationale behind the handling of the dependencies is as follows. All requests are loaded from the trace in the original temporal order, and since all dependencies of a request refer to earlier requests, they were loaded from the trace before the current request. Thus, any dependencies have to be either waiting to be issued, in progress, or already finished. All requests move from the waiting requests, through the issued reads and into the finished requests queue. The only way a preceding request would not appear in any of these three queues is if it was dropped as the oldest entry in the finished requests queue. This implicitly means the dependency itself is satisfied. We also assume that, provided the finished requests queue is not trivially sized, the time requirement of the dependency is implicitly satisfied.

3.3.3 **Issued requests queue**

Once the request has been issued, it is moved to one of the two issued requests queues. The request is kept in this queue until the main memory finishes servicing them. If any of the issued requests queues is filled, no more requests of that type are issued until one of the previous requests finishes.

Once the main memory services the request, it is removed from the issued requests queue. If the returned request was a read, it is moved to the finished reads queue. If it was a write, it is discarded. Requests from the waiting requests queue are checked to see if they were dependent on
the newly finished request. If they were dependent, the dependency is now satisfied, and the time requirement is set to the difference between the identifiers from the current cycle if it was greater than the previous time requirement.

3.3.4 Finished reads queue

The finished reads is a FIFO queue which holds the request identifiers and the time at which the read requests were returned by the system. It is used to check against dependencies of the new entries in the waiting requests queue, so that an appropriate delay can be set.

3.3.5 Trace-based traffic generator summary

We explain how our trace-based traffic generator works. It uses the data flow information to limit the issuing of the requests. The generators performance is determined through a set of parameters and the characteristics of the trace.

3.4 Trace Format

The traces created by our trace collection tool and consumed by the traffic generators contain memory requests with dependency information, written in a binary format of our design. Each memory request is composed of two parts: a mandatory head and an optional tail. Request head specifies the following information:

- a. whether the request is a read or a write,
- b. if the request was caused by a store or a load cache access,
- c. the target address of the request,
- d. a unique request identifier,
- e. a single bit indicating the existence of the tail.

Request tail is composed of segments, with each segment representing a single dependency. Tail segment specifies two pieces of information:

- a. unique identifier of the dependency,
- b. a single bit indicating if this is the last segment.

Trace files can grow very large in size, so we employed several space saving techniques. Each information element in both the head and tail segments is fixed in size, limiting the value range. The lowest six bits of the address specify the offset within the 64 byte cache line. Since our main
memory provides the whole cache line on any access, these bits are removed from the address. The request identifier is stored as a value to be added to the previous request identifier.

If the request identifier is equal to or larger than the maximum value storable in the assigned space, dummy requests with the maximum identifier difference value are inserted. In other words, any request which has a maximum difference value needs to be discarded when reading the trace, since its purpose is only to move the previous request identifier.

Figure 12 Example of head and tail segments of a single trace entry; calculation of the unique request identifier and identifiers of its dependencies is shown with example values

The Tail segments are ordered in strict descending order of identifiers. First tail segment contains the negative difference from the previous request. This is possible since the closest possible dependency is the previous request, which has a distance of zero. All subsequent tail segments hold the negative difference from the immediately previous tail segment, forming a
chain. If the difference is too large for the available space, the dependencies are simply discarded.

Figure 12 shows the *head* and *tail* segments of a single entry. The calculation of the current request and dependency identifiers is shown, and is performed in the following manner. System which loads the trace stores the last request identifier. In the example, the value of last identifier is 245. Once an entry is loaded from the trace file, the new request identifier is calculated as the last identifier (245), plus the identifier difference stored in the trace files (13), for a total of 258. The dependency identifiers are also calculated from the last request identifier. The first dependency identifier is equal to the last identifier (245) minus the identifier difference stored in the *tail* segment (4), for a total of 241. Further dependency identifiers are calculated from immediately previous identifiers. In the example, this corresponds to $241 - 3 = 238$. Once the tail segment indicates it is the last one, the last request identifier is updated to the identifier of the current request, in this case 258.

In this section, we described the format used when saving our traces. The trace format has been designed to save space as much as possible while containing all crucial information.

### 3.5 Video Conference Workload

We designed a workload modelling one side of a two-way video conference using the h.264 encoding standard for the 1080p resolution video. The workload models the memory traffic of five hardware devices: *camera*, *hardware encoder*, *modem*, *hardware decoder* and *display*. The devices and frame processing sequence is shown in Figure 13. The *hardware encoder* and *hardware decoder* devices are modelled using our trace-based traffic generators, with traces collected by a version of our trace-based collection tool modified for this purpose. *Camera*, *modem* and *display* produce a simple sequential stream with addresses matched to the traces used for *hardware encoder* and *decoder*.

Our *Video Conference Workload* models a *camera* saving raw images at a specified frame rate, a *hardware encoder* converting the image from raw format into h.264 format, after which a *modem* reads the image. Parallel to the encoding stage, the decoding stage has a *modem* writing an image in h.264 format to memory, simulating the image being received over the network. The image is
then decoded by the *hardware decoder* into raw format, and finally read by the *display*. The input devices (i.e., *camera/modem*) of both stages can be limited to a maximum frame rate.

**Figure 13** High-level description of the devices involved with a video conference; frame processing sequence is marked E1-E4 for the encoding and D1-D4 for the decoding stage

Section 3.5.1 explains how we obtain the traces for our workload, while Section 3.5.2 explains how those traces are used to generate the traffic for all the devices involved. We summarise in Section 3.5.3.

### 3.5.1 Trace collection and preparation

We collected the traces used to model *hardware encoder* and *decoder* devices using a specialised version of our trace-collection tool to instrument *FFmpeg* [26], an open-source software conversion tool. *FFmpeg* performs the decoding process, while the encoding process is started by *FFmpeg* but the main segment uses *libx264* [27], a software library for h.264 encoding, as shown in Figure 14. We disable the *lookahead* and *b-frames* option to model on-the-fly encoding, since these options use the following frames to encode and decode current frame.

Memory traffic of the conversion process is saved into a separate trace file for every frame. The input and output addresses and sizes of the frame data within the program are noted separately, to be used for *camera, modem* and *display*. Trace files are post-processed to redirect the input and output frame data structures into slightly different memory locations between follower
frames, to allow potential pipelining between devices during reproduction. We perform this operation to simulate two alternating image buffers for input and two for output for both hardware encoder and decoder. Furthermore, we simulate the encoder saving the output image by adding a stream of sequential write requests. This was done since a similar sequence is not evident in the collected trace, due to the use of the system calls in the original code.

![Diagram of Video Conference Workload trace collection mechanism](image)

**Figure 14 Video Conference Workload trace collection mechanism**

### 3.5.2 Traffic generation

We model the hardware encoder and decoder devices using our trace-based traffic generators. Once the camera/modem finishes with writing the frame, the hardware encoder/decoder is initialised with a trace file of the appropriate frame. After the hardware encoder/decoder finishes saving the frame, modem/display devices are started, reading the frame from memory. A device is considered to have finished processing the frame once all requests have been sent and returned from the system.

The parameters for camera, modem and display include a ratio of local to system clock, maximum number of in-flight requests and minimum time between issuing requests. The hardware encoder and decoder devices have the full range of controllable variables as described in the Trace-Based Traffic Generator section. The overall Video Conference Workload is designed to repeat a per-frame process for the specified number of frames. We provide several system-level synchronisation options:

1. a frame is started only after the previous frame has passed through the whole stage,
2. the encoder and the decoder will skip to the latest ready frame (incompatible with 1.),
3. next frame is started at the same time for both stages.
3.5.3 Video Conference Workload summary

The Video Conference Workload we presented here is built to be modular, easily adjustable and representative of a smartphone-system usage model, though we targeted near-future system specifications. An interesting extension to the workload would be inclusion of other optional functionality at appropriate stages in the encoder or decoder stages, to model expected developments of the smartphone-like systems. For example, a facial recognition device might be added to either stage for on-the-fly tagging of participants.

3.6 DRAMSim2

Our main memory is simulated with DRAMSim2 [28], a cycle accurate DDR2/3 memory simulator from the University of Maryland, implemented in C++. It models the memory controller and a set of JEDEC specification compliant DRAM devices, based on configuration parameters specified in the input file. DRAMSim2 automatically determines the configuration and the number of ranks based on the DRAM device specifications and main memory parameters.

DRAMSim2’s memory controller is based around the transaction queue and multiple command queues. Memory controller receives memory requests and stores them in the transaction queue until there are available entries in the command queues. Once the entries are available, the requests are translated into DRAM commands and stored in the appropriate command queue. Every cycle, the memory controller searches the command queues for issuable DRAM commands. The chosen command is issued on the bus to modelled DRAM devices. The DRAM command is considered issuable if all timing constraints are met and the data bus will be free at the appropriate time.

The DRAMSim2 memory controller by default follows the First-Ready, First Come First Served (FR-FCFS) policy for choosing which request to issue. Parameters of the controller include specifying the depths of transaction and command queues, choosing either per-rank or per-rank-per-bank command queues, and choosing between open-page and close-page policy. The close-page policy is equivalent to one access policy with auto-precharge commands, as described in section 2.5. Open-page policy allows multiple row accesses, but precharges the row if the queue
holds no accesses to that row. In addition, provided controller precharges the row if a specified number of accesses has been met. This is implemented to prevent starvation.

We modified DRAMSim2 by adding a write queue as described in Chapter 2 and extended the existing controller with the option to use write drain mode. We also implemented Thread-Fair and Thread Cluster scheduling policies, as described in sections 2.5.3 and 2.5.4. The Thread-Fair policy prefers requests which were generated by the head of the Re-Order Buffer (ROB). However, our infrastructure cannot provide this information, so we follow the suggestion by the authors and label the oldest request from a thread as the one generated by the ROB [7].

We added a write queue which stores incoming write requests, leaving the transaction queue exclusively for read requests. The write queue stores both write requests and the associated data. Translation into DRAM commands does not free the write queue entry; rather, the write queue entry is freed once the write data is sent on the data bus to the DRAM devices. This arrangement limits the write data storage inside the controller to the size of the write queue and allows incoming write requests to the same address to automatically overwrite the old information. This removes the need for a new write while maintaining consistency. Incoming read requests are checked against the write requests present in the write queue. If the queue holds the required data, the read request is immediately returned.

3.7 Summary

This chapter presented an overview of the infrastructure used for the experiments. Smartphone-modelling challenges were discussed, and our approach to them explained. Memory access streams of hardware accelerators are modelled by filtering streams collected from corresponding software implementations. Dataflow information during application runtime is also collected and stored. This information is used by the trace-based traffic generator to limit the issuing of requests. We also present our trace format and the simulator we used to model the main memory. The simulator changes and adaptations are also discussed.
Chapter 4
Evaluation

This chapter contains the evaluation of the workloads we consider typical of the smartphone-like system. We evaluate the workloads for the four memory scheduler policies we described in Chapter 2: First-Ready First Come, First Served with (FR-FCFS+WD) and without (FR-FCFS) the write drain mode, Thread Clustering memory scheduler (TCM), and Thread Fair memory scheduler (TF). We also examine seven different address mapping policies.

Our experiments were performed using the following parameters. We model a system with 4 GB of DDR3 SDRAM, running at 800 MHz speed, in a single channel of two ranks, each rank with 8 banks. Each rank is built out of 16 Micron DDR3 32 Megabit, x4 DRAM devices [2] with 16384 rows per bank and 2048 physical columns, accessible in groups of eight columns. Each logical column holds a cacheline of 64 bytes, which is the basic unit of data to be manipulated within the main memory subsystem. The data bus width is 64 bits, which limits the maximum achievable bandwidth to 11.92 GB/s.

Based on these parameters, we use a single bit of the address to specify the rank, three bits to specify the bank, 14 bits to specify the row and eight bits to specify the column value. The lowest six bits from the request address are discarded since they specify the location within the cacheline. We use the following 26 bits of the address (bit 6 up to bit 31) as the input to the address mapping process which determines the aforementioned values.

The memory controller in our system has several adjustable parameters in addition to the address mapping. We use two sets of these parameters to explore the performance range: limited (LIM) and maximum (MAX) settings. Maximum setting looks at performance when parameters are liberally large to simulate unlimited resources of the memory scheduler. The main aim of the maximum setting is to show if the workload is compute-limited, in which case there will be little to no improvement over the limited setting. Limited setting, as the name implies, looks at the performance of schedulers with constrained resources. Table 2 shows the values for different resources. Our infrastructure uses a single transaction queue per channel, and a single command queue per rank. Each queue is of the specified depth. Write queue holds write data from the moment a write request arrives at the controller until the scheduler issues that particular write to
the devices. Our implementation of the Thread Fair memory scheduler does not use the transaction queue to allow issuing write hits while not in the write drain mode, or read hits while in write drain mode, as required by the scheduler. To allow for this, our settings are slightly altered to approximate the space usage. The Thread Clustering memory scheduler uses a set of additional parameters, which we set to the values suggested in the original paper [3]. An interesting approach for future work would be to attempt to tune these parameters for the smartphone workloads.

Table 2 Memory controller parameters

<table>
<thead>
<tr>
<th></th>
<th>Limited</th>
<th>TF Limited</th>
<th>Maximum</th>
<th>TF Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction queue entries</td>
<td>24</td>
<td>-</td>
<td>512</td>
<td>-</td>
</tr>
<tr>
<td>Command queue entries</td>
<td>8</td>
<td>20</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>Maximum row accesses</td>
<td>32</td>
<td>32</td>
<td>1024</td>
<td>1024</td>
</tr>
</tbody>
</table>

Write Queue:

<table>
<thead>
<tr>
<th>Entries</th>
<th>16</th>
<th>16</th>
<th>64</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Watermark</td>
<td>8</td>
<td>8</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>High Watermark</td>
<td>12</td>
<td>12</td>
<td>60</td>
<td>60</td>
</tr>
</tbody>
</table>

The rest of this chapter is divided into the following sections. Section 0 describes the workloads and specifies the parameters we use in later sections. Section 4.2 presents seven ways of determining the location of the data in the DRAM and their effect on performance. The memory usage characteristics of our workloads are shown in the section 4.3. Section 4.4 presents the
results for the four memory scheduler policies and discusses their advantages and disadvantages. We conclude in Section 4.5.

4.1 Workload Parameters

This section gives an overview of the workloads we selected as representative for a smartphone-like system. We present our Video Conference Workload, a Web Browsing workload, and a set of computer vision workloads. Also described is the workload used for testing the scheduling policies in the Section “Memory Scheduler Comparison”. The workload combines our Video Conference Workload, the Web Browsing and the Face Detection workload from the set of computer vision workloads.

The web-browsing workload and all computer vision workloads are modelled using our trace-based collection and reproduction system, with the cache size set to 64 kB. The web-browsing workload was based on the BBench [29] automated browsing workload. The BBench was run using the “Arora” [30] lightweight web browser version 0.10.2, which we instrumented for two billion instructions.

The computer vision workloads used are described in the Table 3, along with the resolution of their input file and the total amount of data (TAD) required during the processing of a single image. The input image resolution is smaller than the expected camera resolution, as we would expect the real system to scale the full image for processing.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Description</th>
<th>Input Resolution</th>
<th>Total Data Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Denoising</td>
<td>Process or removing noise from the image, such as the result of statistical variation of the number of photons to hit each cell [31]</td>
<td>100 x 100</td>
<td>78.3 MB</td>
</tr>
<tr>
<td>Face Detection</td>
<td>Detects location of the faces within the image [32]</td>
<td>100 x 100</td>
<td>238.63 MB</td>
</tr>
</tbody>
</table>
High Dynamic Range produces a greater range of luminance between the lightest and darkest parts of the image by combining multiple images of varying exposures [33].

Optical Character Recognition detects letters and words in an image [34].

Segregates the image into labeled regions that correspond to objects or boundaries in the image [35].

Scale-Invariant Feature Transform is an algorithm for detecting and describing local features within the image; used as first stage for other computer-vision related algorithms [36].

Speeded Up Robust Features is a high-performance feature detection and description algorithm different from SIFT [37].

Detects the location of the upper body of a person in the image [38].

We configure our trace reproduction tool with the parameters comparable to a modern general purpose computer, expected to be representative of a future smartphone capabilities. We set the ratio to model a 3.2 GHz device running the workload. We set the maximum number of in-flight requests to 8 reads and 8 writes. The history length determines the size of the FIFO request queue for checking against the dependencies of new requests. We set it to 256. The number of future requests determines how many requests are loaded from the trace file and considered for issuing, but not yet issued. We set this value to 16. Maximum distance parameter is optional, and if set, limits the maximum number of cycles a request can be issued after its dependency returns from the main memory to model the computation speedup we would expect from specialised
hardware as compared to a general purpose CPU. We do not use the maximum distance for our computer vision or web-browsing workloads.

![Figure 15 Required total amount of data during the encoding and decoding of the first frame, as a function of cache size](image)

The Video Conference Workload we designed uses FFmpeg [26] as the software implementation of the h.264 encoder and decoder. The FFmpeg uses libx264 [27], a free software encoder library for encoding the video stream, whereas decoding is the integral part of the FFmpeg application. Using our trace-collection tool, specialised for the FFmpeg, we instrument both the encoder and the decoder. We set encoder to process a raw, 1080p resolution video stream, to produce h.264 encoded stream, still in 1080p resolution. Decoder is used on the stream produced by the encoder, and produces a raw 1080p video stream. We modified FFmpeg to obtain the input and output frame address and size during the coding process.

Our trace-collection tool uses a cache to model the hardware implementations specialised data structures and other possible optimisations. We determine the cache size by considering the required TAD. Figure 15 shows the TAD required for the encoding of the first frame at different cache sizes. The TAD difference between cache sizes is shown in Figure 16. The encoding requires higher amount of data than decoding due to the higher complexity of the process, as we would expect.
We choose the smallest cache size for which the TAD difference from next largest cache size is less than 5%. We chose the value of 5% difference as an indicator of a plateau in TAD. We adopt a 256 kB cache size for the encoder, since increasing the size to 512 kB results in the required TAD shrinking by 3.14%. We use a 128 kB cache for the decoder. The total TAD the workload requires for 20 frames is 3.9 GB, this figure including all devices from both stages.

Figure 16 Percentage of TAD reduction as a function of increasing cache size for our Video Conference Workload

The Video Conference Workload is setup to model five devices: camera, modem, display, encoder and decoder. We run it in synchronised mode, meaning a new frame can only proceed after both the encoding and the decoding stages have finished. This limits the number of simultaneously operating devices to two; one from each stage. We set a limit of 60 frames per second. Table 4 presents the parameters under which individual devices operate. The encoder and the decoder share all possible parameters with our trace-reproduction tool, while camera, modem and display use only a subset of these settings due to their simplicity. The parameters were chosen to allow the workload to stress the memory system. Under these settings, the Video Conference Workload is memory-limited.
Table 4 Video Conference Workload device configuration

<table>
<thead>
<tr>
<th></th>
<th>Camera</th>
<th>Encoder</th>
<th>Modem</th>
<th>Decoder</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed [MHz]</td>
<td>160</td>
<td>3200</td>
<td>800</td>
<td>3200</td>
<td>160</td>
</tr>
<tr>
<td>Future requests</td>
<td>x</td>
<td>16</td>
<td>x</td>
<td>16</td>
<td>x</td>
</tr>
<tr>
<td>Maximum distance</td>
<td>x</td>
<td>80</td>
<td>x</td>
<td>80</td>
<td>X</td>
</tr>
</tbody>
</table>

Maximum issued requests

<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Write</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

4.2 Address Mapping Schemes

The seven examined address mapping schemes are presented in Figure 17. The scheme RKBC was presented as the typical address mapping in the paper by the Kaseridis et al [6], while the rest were provided with the DRAMSim2 infrastructure. We do not explore other possible combinations because we consider these to show the potential impact of the mapping process on the performance, and contain the best performer overall.

The names of the schemes code the ordering of the way they use address bits to allow for easy recognition. We use “R”, “C”, “B” and “K” respectively to refer to the portions of the address used to select the DRAM row, column, bank, and rank. The effect of address mapping scheme on performance depends on the memory access stream. A good mapping scheme would take advantage of any spatial locality in the memory stream by mapping these accesses to the same row to reduce the number of rows needed to be opened. Mapping a group of temporally close requests to different ranks and banks can be beneficial by allowing them to be executed concurrently. However, too much concurrency can lead to bank conflicts and overall slowdown if requests frequently require the same bank but different rows.
The first scheme on the list, *KBCR*, uses the lowest bits to determine the *row*, which means sequential or close accesses will require different *rows* of the same *bank*, causing *bank* conflicts. We do not consider other schemes that use the lowest bits for the *row* value as we will see that the effect on performance is abysmal. All other schemes determine the *row* value from higher bits to attempt to limit the *bank* conflicts. Accessing a different *bank* provides the best opportunity for parallelisation of requests, but increase chances for *bank* conflicts considerably. Accessing different ranks is very similar to accessing different banks, but slower due to the rank-to-rank switching time \(t_{RTRS}\). It can also lead to reduced opportunities for putting one *rank* into power-down mode.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Address bits [31-6]</th>
</tr>
</thead>
<tbody>
<tr>
<td>KBCR</td>
<td>K</td>
</tr>
<tr>
<td>RCBK</td>
<td></td>
</tr>
<tr>
<td>KCKB</td>
<td></td>
</tr>
<tr>
<td>KRCB</td>
<td>K</td>
</tr>
<tr>
<td>KBRC</td>
<td>K</td>
</tr>
<tr>
<td>RBKC</td>
<td></td>
</tr>
<tr>
<td>RKBC</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 17 Address Bits Usage per Mapping Scheme; K stands for the rank bit**

We evaluated the schemes using the *FR-FCFS* scheduler with our *Video Conference Workload*. We also evaluated the schemes with the computer vision workloads, but their low utilisations mostly hide the differences between the schemes. Figure 18 shows the results for the Image Segmentation workload, which had the highest *DRAM* utilisation of all the tested workloads. Due to the little noticeable pattern, our discussion will concentrate on the results obtained from the *Video Conference Workload*, shown in Figure 19.

The results show that the *KBCR* scheme has by far the worst performance. This is due to the use of lowest bits as the *row* bits, meaning any sequential accesses need to pre-charge and activate a different *row* in the same *bank*. This problem is exacerbated since the *row* value requires the
most address bits. The MAX configuration results in better performance, but the addressing scheme severely limits the ability of scheduler to improve the execution.

Figure 18 Comparison of address mapping schemes using an FR-FCFS scheduler on the Image Segmentation workload; lower is better

The second worst performer is the KBRC scheme, which uses the lowest bits for the column value, allowing sequential accesses to hit in the same row. The highest bits determine the rank and the bank, allowing for parallelism if the addresses are spaced widely enough. The difference between limited and maximum settings is close to 19.5%, due to the larger pool of requests to select from. A larger pool means new row-hits can be scheduled over older requests which require a different row of the same bank. In addition to increasing the speed with which these new requests are issued, a larger pool also reduces the number of times a row needs to be opened or closed.

The three schemes which use the lowest bits to target parallelisation are RCBK, RCKB and KRCB. While increased parallelism for sequential requests can result in lower latency for these requests, it means the opening and closing rows is amortised over the smaller number of accesses to it; since sequential accesses are spread over different rows, there will be fewer row hits. Furthermore, the increased parallelism increases the chances of bank conflicts. One of the reasons for decreased performance can be seen in Figure 20, which shows the average number of column accesses for every opening of a row. Scheme KRCB has the worst performance of the three schemes targeting parallelism, but the highest row buffer locality. This is due to the use of the highest bit to determine rank; it means the rank parallelism only happens if the addresses are
significantly moved apart, yet the column now covers lower bits, allowing greater row buffer locality.

**Figure 19 Comparison of address mapping schemes using an FR-FCFS scheduler on our Video Conference Workload; lower is better**

The two best performers are the RBKC and RKBC schemes. The characteristic of both schemes is that the lowest bits are used for column selection. Both schemes also use the following four bits to determine rank and bank, the only difference being that RBKC uses the lowest of these four bits to determine the rank, while RKBC uses the highest of the four. RKBC shows a very slight advantage over the RBKC; just over 1% faster on limited settings and under 0.02% faster on maximum settings. This is due to the penalty that needs to be paid for switching between the ranks.

The results for our workload show that using the highest bits for the row value to be the best option overall, with the lowest bits used for the column value. The rank bits should be taken from higher bits than the bank, but still lower than the row value. During the examination of memory schedulers and workloads we will concentrate on the RKBC scheme, since it provides the best performance.
Figure 20 Average number of column commands per ACTIVATE command

4.3 Workload Characteristics

We study the memory usage characteristics of the described workloads using the FR-FCFS scheduler on the described system. Figure 21 shows the average DRAM utilisation during the running of the Web Browsing and the computer vision workloads.

Figure 21 Average DRAM utilisation for the Web Browsing and computer vision workloads using FR-FCFS with scheme RKBC

Most workloads have very low utilisation for the LIM configuration, and the utilisation increases by less than 1% for all workloads under the MAX configuration of the scheduler. The only
workload which shows medium (26.7\%) utilisation is the Image Segmentation. However, it requires a small amount of TAD (17.72 MB), which means it can complete in a short amount of time. On the other hand, Upper Body Detection has second-highest TAD requirement, but only utilised 1.2\% of the available bandwidth.

The low average utilisation shows these workloads do not stress the main memory much on their own, though they may still provide interference when run with other workloads. The minimal performance improvement under the MAX configuration indicates that the request scheduling is of limited influence. We use DRAM utilisation instead of DRAM efficiency since they are effectively equal. The largest percentage of idle time among all workloads was 0.5\%, for HDR.

Figure 22 DRAM utilisation for Video Conference Workload using FR-FCFS scheduler

Figure 22 shows the DRAM utilisation under the LIM and MAX configuration for all seven schemes. DRAM utilisation is above 70\% for all schemes except for KBCR and KBRC. The difference between the two resource configurations is significant under almost all schemes, reaching over 10\% for KRCB and KBRC schemes. This difference indicates that the Video Conference Workload is limited in performance by the memory subsystem, with the potential for improvement through different scheduling policies.

Figure 23 shows the time banks spent in each state as a percentage of total execution time when using the RKBC scheme and FR-FCFS scheduler. Banks are mostly idle, waiting to be utilised.
The time spent powered-down is small, which would follow from high utilisation of bandwidth. The percentage of time spent in the *Row Active* state shows the mapping scheme spreading the accesses evenly.

![Figure 23 Comparison of time banks spent in particular bank state during the execution of the Video Conference Workload; limited configuration using RKBC scheme](image)

### 4.4 Memory Scheduler Comparison

The four memory schedulers we examine are the *FR-FCFS*, *FR-FCFS* using *Write Drain* mode (*FR-FCFS-WD*), the *Thread Fair* memory scheduler (*TF*) and the *Thread Clustering* memory scheduler (*TCM*). The two *FR-FCFS*–based schedulers are thread-agnostic and only consider the requests arrival time during scheduling. The *TCM* scheduler was designed for a system with large number of concurrent threads, a significant portion of which should require high bandwidth. The *TF* scheduler uses a *Write Drain* mode, with scheduling which takes into account information on the originating thread.

We examine the performance of these schedulers in a system with a small number of threads, as we expect smartphone-like systems to behave. A combination of workloads was chosen to correspond roughly to a simplified but expected smartphone usage. We used the *Web Browsing*, *Face Detection* and our *Video Conference Workload*. This combination was chosen to represent a realistic usages scenario of applying face detection to tag the people present in the video conference. The *Face Detection* workload was chosen since it has the highest total amount of data (*TAD*) requirement (238.63 MB) and the second highest ratio of *TAD* required to bandwidth.
utilisation (28.15). The *Image Denoising* workload has the highest ratio (129.94), but we decided not to use it due to its trivial utilisation (under 1%).

![Bar chart showing scheduling policy performance](image)

**Figure 24 Comparison of the scheduling policy performance: number of cycles to service the combination of workloads; the y-axis starts at 300 million cycles; lower is better**

Figure 24 shows the completion time for the combined workload. The odd behaviour of *FR-FCFS-WD* and *TF* schedulers for MAX configuration is due to their use of the *write drain* mode. Specifically, the devices can only have a limited number of incomplete writes at any one time, and the total number of writes all devices together can issue is less than the high watermark value of the *write drain* mode. This means the *write drain* mode is rarely if ever triggered, and writes can only be serviced when there are no read requests in the scheduler. This highlights the disadvantage of using fixed limits if the system is not guaranteed to reach those limits. However, this problem is avoided if the devices consider the write requests finished as soon as they are dispatched to the memory subsystem.

We increase the number of in-flight writes for every device to 128 to allow the *write drain* operation as designed. The scheduler performance for the combination of workloads is shown in Figure 25. The figure shows the number of cycles required to complete the respective workload. All schedulers perform within 7% of the best one. The best performer for both configurations was the *FR-FCFS-WD* scheduler. The *TF* scheduler is the second best, being 0.95% slower for limited and 1.45% slower for maximum settings. The *TCM* policy is the worst performer under
both configurations, falling behind 6.72% for LIM and 6.08% for MAX configuration. The poor performance was to be expected as it was designed for systems with large number of different threads (the original paper [3] assumes 24 threads per workload).

Figure 25 Comparison of the scheduling policy performance: number of cycles to service the combination of workloads with unlimited write requests; the y-axis starts at 300 million cycles; lower is better

The average latencies for read requests are shown in Figure 26. Latencies grow for maximum settings, due to the schedulers reordering requests and keeping some requests much longer, while some of the newly arrived requests are allowed to proceed over those waiting. Comparing the TF scheduler to FR-FCFS, we can see it has higher latencies despite performing better. This indicates its re-ordering is more effective. The FR-FCFS-WD was able to outperform the TF for these workloads, but as the FR-FCFS-WD shares its re-ordering policy with FR-FCFS, different workloads may allow the TF to perform better.

FR-FCFS-WD has higher latency under LIM configuration than FR-FCFS, but lower under MAX configuration. This is due to the longer delays between write drain modes, which allows the scheduler to service more READ requests between the periodical write bursts.
Summary

This chapter presents the results of our evaluations. We compared the seven address mapping schemes, and found the RKBC to provide the best performance, with RBKC closely following. Our examination of the Web Browsing and the computer vision workloads show that they do not stress the memory significantly. However, our Video Conference Workload is highly memory intensive. Out of the four schedulers we tested, the First Ready – First Come First Served with the write drain mode performed the best, followed closely by the Thread-Fair scheduler.
Chapter 5
Conclusion and Future Work

This thesis examines the performance of the main memory subsystem under smartphone workloads. We used DRAMSim2, a main memory simulator, to model the DRAM devices and the memory controller. The traffic was provided by a trace-based generator we designed. The traces contained information on the dataflow in the application from which we collect the traces.

We tested the memory requirements of a web-browsing workload and a selection of computer vision workloads. None of the workloads were memory intensive, with only one workload averaging the utilisation of the main memory bandwidth above 13%.

Using our trace-based traffic generator, we designed and tested a Video Conference Workload to simulate the memory traffic from hardware elements of a single smartphone which would be a participant of a two-way video call. The workload consists out of the encoding and the decoding stages, which operate in parallel. It is designed to stress the memory subsystem by simulating near-future video call capabilities.

The Video Conference Workload is used to investigate the seven address mapping policies. Our test results show that using the lowest address bits to determine the row value significantly damages performance. The best performance reached by using the highest bits for the row value, the lowest address bits to determine the column value, and determining rank and bank from the bits in between.

We compare the four memory scheduler policies using a combination of the Web Browsing workload, one computer vision and our Video Conference Workload. The best overall scheduler is the First Ready-First Come, First Served with the write drain mode (FR-FCFS-WD). The Thread-Fair memory scheduler (TF) is within 1.5% in performance to the best one, followed by the First Ready-First Come, First Served without the write drain mode (FR-FCFS). The Thread-Clustering memory scheduler (TCM) ends up as the worst performer overall. Both the TF and TCM schedulers were designed specifically for multi-threaded systems. The TCM scheduler was designed to detect and make decisions based on the thread behaviour, but we find it requires a large set of threads to fulfill its potential. We conclude that for the smartphone-like systems with
only a few threads, most of which are memory-light, the simpler schedulers provide superior performance.

Adapting the existing schedulers’ parameters to improve performance under the smartphone workloads, or designing a scheduler specialised for such workloads, is an interesting research opportunity. Another interesting avenue for further work would be to expand our infrastructure with additional workloads such as the graphics processing unit or the full timing model of a CPU, as well as tracking power and energy usage in addition to bandwidth utilisation and workload performance. Furthermore, fairness in multithreaded workloads and the effects of schedulers on memory latency would provide a more
References


[2] Micron; DDR3 1Gb x4, x8, x16 DDR3 SDRAM Data Sheet; URL: http://www.micron.com/~media/Documents/Products/Data%20Sheet/DRAM/1Gb_DDR3_SDRAM.pdf


[27] VideoLAN Organisation; x264 software library; URL: http://www.videolan.org/developers/x264.html; Accessed: February 2012


