저작자표시-비영리-변경금지 2.0 대한민국

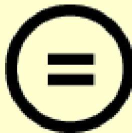
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.


변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

M. Sc. Thesis

Memory Access Pattern Analysis of CNN Type Application on CPU Model

CPU 모델에 대한 CNN 기반 애플리케이션의 메모리 액세스 패턴 분석

by

Konstantin Bick

February 2019

Seoul National University
Graduate School of Engineering
Department of Electrical and Computer Engineering

M. Sc. Thesis

Memory Access Pattern Analysis of CNN Type Application on CPU Model

CPU 모델에 대한 CNN 기반 애플리케이션의 메모리 액세스 패턴 분석

by

Konstantin Bick

February 2019

Seoul National University
Graduate School of Engineering
Department of Electrical and Computer Engineering

Memory Access Pattern Analysis of CNN Type Application on CPU Model

CPU 모델에 대한 CNN 기반 애플리케이션의 메모리 액세스 패턴 분석

지도교수 이 혁 재

이 논문을 공학석사 학위논문으로 제출함

2019년 2월

서울대학교 대학원

전기・정보공학부

빅콘스탄틴

빅콘스탄틴의 공학석사 학위 논문을 인준함

2019년 2월

위 원 장 : 최 기 영 (인)

부위원장 : 이 혁 재 (인)

위 원 : 권 영 준 (인)

Abstract

Neural Networks and especially Convolutional Neural Networks (CNNs) gained a lot of attraction recently as they excel in computer vision fields like object detection and image classification. CNNs generally contain different layer types that are executed alternately. An analysis of such layer based CNN is necessary to draw conclusions on how to modify current computer architecture in order to optimize the performance of CNNs. This thesis analyzes a typical CNN with a focus on the efficiency of caching and prefetching. Based on the findings an adaptive L2 cache prefetching scheme is proposed that combines two existing prefetchers and overcomes their layer specific inefficiencies. Moreover, a cache hierarchy bypassing the L2 cache is proposed that reduces power consumption while only slightly sacrificing overall performance.

keywords: CNN, memory system, prefetching, computer architecture

student number: 2017-23530

To my parents,

who unconditionally supported me

through this eventful phase of my life.

Contents

Abstract **i**

Contents **ii**

List of Tables **iv**

List of Figures **v**

1 Introduction **1**

2 Problem set **2**

2.1 Target Application 2

2.1.1 Layer Description 3

3 Simulation Environment **8**

3.1 Prefetcher 9

3.1.1 Tagged Prefetcher 10

3.1.2 Stride Prefetcher 10

3.2 Model Matching 12

4 Analysis **16**

4.1 Memory Access Pattern 16

4.2 Prefetching Efficiency 21

5 Proposal and Evaluation **25**
5.1 Adaptive Prefetcher 25
5.2 L2 cache bypassing 28

6 Conclusion **33**

A Appendix **34**
A.1 Prefetching algorithm flow diagrams 34
A.2 Linaro disk image bug fix 36
A.3 Model Matching 38

Abstract (In Korean) **42**

List of Tables

2.1 Tiny YOLOv3 layer overview 4

3.1 System specifications of the ARTIK710 hardware development board and the matching gem5 system 13

4.1 Layer type runtimes with different L2 prefetcher types 24

5.1 Layer type runtimes with different L2 prefetcher types 27

5.2 Layer type runtimes with different L2 prefetcher types 31

5.3 McPAT area and power estimates for a single core processor based on Cortex-A53 parameters in 28nm 32

List of Figures

2.1 Pseudo code of the convolution operation 5

2.2 Pseudo code of the max-pooling operation 6

3.1 Tagged prefetcher block diagram 10

3.2 Stride Prefetcher block diagram 11

3.3 Stride Prefetcher flow chart 12

3.4 Tiny YOLOv3 layer runtime comparison of the matched gem5 model and ARTIK 710 development board 15

4.1 Cycles Per Instruction (CPI) and memory intensity of each layer in Tiny YOLOv3 17

4.2 Measured CPI divided into three categories: ideal CPI, CPI caused by pipeline stalls, and CPI caused by memory stalls 18

4.3 L1 data, L2 demand, and overall L2 cache misses 19

4.4 L2 cache MSHR occupancy per layer during the simulation of Tiny YOLOv3 20

4.5 L2 cache data misses comparison for three different systems: no attached L2 prefetcher, L2 Stride Prefetcher, and L2 Tagged Prefetcher . 22

4.6 Stride prefetch abort causes 23

4.7 Runtime speedup normalized on runtimes of a system without L2 prefetcher 24

5.1 The proposed Stagged Prefetcher 26
5.2 L2 cache demand misses comparison for three systems: L2 Stride Prefetcher, L2 Tagged Prefetcher, and L2 Stagged Prefetcher 27
5.3 Runtime speedup normalized on runtimes of a system without L2 prefetcher 28
5.4 L1 data cache and prefetch parameter reevaluation 29
5.5 Tiny YOLO runtimes of ARTIK710 matched system and two systems bypassing the L2 cache 31

A.1 Tagged Prefetcher flow diagram 34
A.2 Stride Prefetcher flow diagram 35
A.3 The linux boot sequence 36
A.4 The linux boot sequence 37
A.5 auto-root-login bash script 38
A.6 auto-console-login bash script 38
A.7 PARSEC 3.0 runtime comparison of the matched gem5 model and ARTIK 710 development board 39

Chapter 1

Introduction

Current computer architecture is built around and optimized with regards to conventional computer applications. Recently, Neural Networks and especially Convolutional Neural Networks (CNNs) gained a lot of attraction as they excel in computer vision fields like object detection and image classification. CNNs generally contain different layer types that are executed alternately. While the depth of the network and the specific techniques in use may differ depending on the network, the layer based structure is characteristic for all CNNs. The memory access pattern is expected to change along with the alternating layer structure. The analysis of the memory access pattern, carried out in this thesis, aims to fully characterize memory utilization and memory requirements of Tiny YOLO v3, a representative CNN. Findings of the analysis will then be used in order to improve a system's architecture with regards to CNN execution.

The remainder of this thesis is organized as follows: proceeding the introduction, Chapter 2 describes the targeted CNN, Tiny YOLO v3, and various CNN layer types. Chapter 3 introduces the simulation environment as well as the hardware matched simulation model used throughout the analysis. Chapter 4 then thoroughly analyzes the targeted CNN's memory access pattern. Proposals and their evaluation are discussed in Chapter 5 while Chapter 6 concludes this thesis.

Chapter 2

Problem set

This chapter discusses the characteristics of Convolutional Neural Networks (CNNs) in the context of Tiny YOLOv3, the targeted application for the research conducted in this thesis. Distinct layer features are emphasized on as they will be the key to architectural changes in order to improve CNN execution.

2.1 Target Application

Convolutional Neural Networks (CNNs) generally contain different layer types that are executed alternately. Every layer type has its purpose, be it to detect features, compress feature maps, or else. Depending on a network's aim, different layer depths or techniques are used in order to optimize performance, accuracy, hardware requirements, etc. However, the convolutional layer is the core layer in every CNN and networks making use of it are categorized as CNNs. The purpose of the convolutional layer is to detect certain patterns or features in the input image through the convolution of different kernels and the input image.

YOLO ("You Only Look Once") has gained a lot of attraction recently as it significantly sped up object detection and classification while maintaining a high accuracy comparable to previous network designs [1]. After its release, the network has been

improved further and currently the third version, YOLOv3, is available [2]. Typical for a CNN, several convolutional layers extract features of the input image into feature maps in YOLOv3.

In total, YOLOv3 has 106 layers, of which 75 are convolutional layers. This makes the network design of the third version of YOLO much deeper compared to previous iterations. Tiny YOLOv3, proposed by the same authors of YOLO, follows the idea behind YOLOv3 but reduces the network depth in order to improve performance on embedded systems or systems with less computing power. Tiny YOLOv3 has just 24 layers in total, of which 13 are convolutional layers, mostly followed by max-pooling layers. Compared to YOLOv3, the accuracy of Tiny YOLOv3 is decreased. However, the smaller network uses the same input image size and contains convolutional layers with identical dimensions. This makes Tiny YOLO ideal for the memory access pattern analysis, as it reduces the required simulation time but overall contains characteristics like dimension, network techniques etc, of its deeper counterpart, YOLO v3.

In this thesis, the memory access pattern of Tiny YOLOv3 is analyzed. Weights, pretrained on the COCO dataset, are used which can be obtained from the author's website. The network with its pretrained weights is able to detect 80 different object classes.

2.1.1 Layer Description

An overview of all 24 layers in Tiny YOLOv3 is given in Table 2.1. Besides convolutional layers, max-pooling, upsampling and routing layers are used. Additionally, layers labeled "yolo" are found towards the end of the network. Those layers evaluate bounding boxes and class predictions based on their previous convolutional layers. Most of the layers in Tiny YOLOv3 are convolutional and max-pooling layers, which are executed alternately. F.e, in Table 2.1 it can be seen that from layer 0 to layer 12, the convolutional layer is executed interchangeably with the max-pooling layer.

Further, a network technique called "skip connection" is used, where previously

Table 2.1: Tiny YOLOv3 layer overview

Layer No.	Type	Filters	Size	Input	Output	BFLOPs
0	conv	16	3x3 / 1	416x416x3	416x416x16	0.150
1	max		2x2 / 2	416x416x16	208x208x16	
2	conv	32	3x3 / 1	208x208x16	208x208x32	0.399
3	max		2x2 / 2	208x208x32	104x104x32	
4	conv	64	3x3 / 1	104x104x32	104x104x64	0.399
5	max		2x2 / 2	104x104x64	52x52x64	
6	conv	128	3x3 / 1	52x52x64	52x52x128	0.399
7	max		2x2 / 2	52x52x128	26x26x128	
8	conv	256	3x3 / 1	26x26x128	26x26x256	0.399
9	max		2x2 / 2	26x26x256	13x13x256	
10	conv	512	3x3 / 1	13x13x256	13x13x512	0.399
11	max		2x2 / 1	13x13x512	13x13x512	
12	conv	1024	3x3 / 1	13x13x512	13x13x1024	1.595
13	conv	256	1x1 / 1	13x13x1024	13x13x256	0.089
14	conv	512	3x3 / 1	13x13x256	13x13x512	0.399
15	conv	255	1x1 / 1	13x13x512	13x13x255	0.044
16	yolo					
17	route	13				
18	conv	128	1x1 / 1	13x13x256	13x13x128	0.011
19	upsample		2x			
20	route	19 8				
21	conv	256	3x3 / 1	26x26x384	26x26x256	1.196
22	conv	255	1x1 / 1	26x26x256	26x26x255	0.088
23	yolo					

calculated feature maps are forwarded to later layers, skipping layers in between. This skip connection technique is used at layers labeled “route”. By reusing previous feature maps, details, that otherwise would become too abstract, can be restored. A short description the main layer types of Tiny YOLOv3 is found below.

Convolutional

The convolutional layer is the core layer of any CNN. Its purpose is to extract features from input images through convolutions of different kernels and the input image. The number of kernels (also referred to as filters), can be seen in the third column of Table 2.1. The parameter "Size" in the fourth column also refers to the filters, while "Input" and "Output" refer to the input image and resulting feature map sizes. For every convolutional layer, Tiny YOLOv3 predicts the number of billion floating point operations, as can be seen in the seventh column of Table 2.1. Pseudo code for the image convolution is shown in Fig. 2.1. The for-loops starting in line 1 and 2 iterate through the input image's height and width. The for-loops in line 4 and 5 iterate through the kernel's height and width. In line 6, the multiplication and addition of image pixel and kernel pixel, required for the convolution, is shown. The result will normalized and saved to the output feature map, as seen in line 9.

```
1  for (y=0; y < i_h; y++) {
2    for (x=0; x < i_w; x++) {
3      out = 0;
4      for (j=0; j < k_h; j++) {
5        for (i=0; i < k_w; i++) {
6          out += B[y+j][x+i] * A[j][i];
7        }
8      }
9      C[y][x] = out / (k_h * k_w);
10   }
11 }
```

Figure 2.1: Pseudo code of the convolution operation.

Max-pooling

The max-pooling layer reduces the dimension of feature maps. In the context of Tiny YOLOv3, column four in Table 2.1, "Size", refers to the size of the max-pooling oper-

ation. For example, a 2×2 size compares four pixel values and stores just the largest. The pseudo code of this operation can be seen in Fig. 2.2. The for-loops in line 1 and 2 are traversing the height and width of the input feature map. The for-loops in line 4 and 5 are iterating the size of the max-pool operation. The max value will written to the output feature map, as seen in line 10.

```
1  for (y=0; y < i_h; y = y+pool_h) {
2    for (x=0; x < i_w; x = x+pool_w) {
3      max = 0;
4      for (j=0; j < pool_h; j++) {
5        for (i=0; i < pool_w; i++) {
6          val = B[y+j][x+i];
7          max = (val > max) ? val : max;
8        }
9      }
10     C[y][x] = max;
11   }
12 }
```

Figure 2.2: Pseudo code of the max-pooling operation.

Yolo, Route and Upsample

Besides convolutional and max-pooling, layers labeled “yolo”, “route” and “upsample” are used in Tiny YOLOv3. Because their runtime is very short, they won’t be analyzed in detail in this thesis. However, a short explanation of their function is given below.

Bounding boxes and classes are predicted inside the **yolo** layer. The evaluation is based on the previous convolutional layer that applies 255 filters of the size 1×1. All 255 channels (depth of the feature map) have their unique information, including bounding box coordinates, objectness scores and class confidences.

The **route** layer forwards the feature map of a certain convolutional layer to a later convolutional layer while skipping all layers in between. The third column in Table

2.1, referring to the number of kernels in other layers, indicates the layer of which the output is forwarded from. F.e. layer no. 17, route, redirects the output of convolutional layer no. 13 to be the input of convolutional layer no. 18. In the case of two indices, as seen in the route layer no. 20, the output of both layers are concatenated and then forwarded to the input of convolutional layer no. 21.

A bilinear upsampling is taking place in layers labeled **upsample**. As this layer is very short compared to convolution and max-pooling layers, this layer will not be analyzed in detail.

Chapter 3

Simulation Environment

A computer architecture simulator is required to not just analyze execution statistics, but to perform a design exploration. During a design exploration, system components and their parameters are modified to investigate the impact on the execution of the targeted application. The simulator is required to be accurate, fast and versatile.

gem5 is a renowned computer architecture simulator [3]. It supports several ISAs (Instruction Set Architectures), including ARM and x86. The simulators offers a SE (System Emulation) and the FS (Full-System) simulation mode. The SE mode supports applications to run directly on a system defined in gem5. In this mode, no operating system (OS) is required and therefore most system calls are unsupported. While this simulation mode grants freedom regarding system design, it lacks the support of most libraries that are handled by the OS. The FS mode, on the other hand, simulates the OS on top of the defined system. Thus, multithread APIs (Application Programming Interfaces) like the pthread library or OpenMP are supported. In this thesis, the FS mode is used because Tiny YOLOv3 execution can be parallelized on multicores through the OpenMP library. Simulation results of the FS mode are well comparable to hardware development board executions, which is shown in the later Section 3.2.

As for the ISA, the ARM architecture is used because of two reasons: firstly, the simulator used in this research, gem5, shows a wide-range support of the ARM ISA,

while linux kernels as well as OS images for the ARM architecture are freely available. Secondly, with the help of ARM's Development Studio 5, ARM processors' PMUs (Performance Monitoring Units) can be monitored. Data collected through the PMUs are crucial for a comparison of the simulator and real hardware. Thus, a hardware matched simulation model can be defined, as discussed in Section 3.2.

ARM supports the development of gem5 and they provide a Research Starter Kit (RSK) with a core model, a gem5 simulation setup script and a compatible kernel and linux disk image[1]. The core model, a gem5 SimObject modeling a 64-bit in-order core called High-Performance In-order (HPI) core, is implemented on top of the gem5 in-order core model [4]. The attached simulation script defines a gem5 system setup that resembles ARM's development board environment. Moreover, a linaro disk image is distributed as well as a compatible kernel. Files distributed in the ARM RSK are used as a starting point for the simulated system used in this research. Section 3.2 discusses necessary parameter modifications in order to match the simulation model to a hardware development board.

In order to calculate power of the processor more accurately, the simulator McPAT (Multicore Power, Area, and Timing) [5] is integrated into the simulation environment.

3.1 Prefetcher

Cache prefetching may significantly hide long memory access times. Hardware cache prefetchers generally predict future miss addresses based on a cache's miss history. Their predictions are fetched whenever the according cache completed handling all outstanding demand misses and before entering an idle stage.

This section introduces the Stride Prefetcher and Tagged Prefetcher, which are implemented into gem5 by default.

[1]The linaro image **linaro-minimal-aarch64.img** distributed within the **aarch-system-20180409.tar.xz** reads attached .rcS scripts twice during a gem5 simulation. The fix to prevent this from happening is described in Appendix A.2

3.1.1 Tagged Prefetcher

The Tagged Prefetcher identifies prefetch candidates on two conditions [8]: either, a demand fetched cache block is accessed for the first time, or, a prefetched cache block is accessed for the first time. The original proposal discusses an additional tag bit for every block in the cache (hence the name Tagged) that is set to 1 upon reference. For every demand fetched or prefetched block, the tag bit is initially set to 0. On every 0 to 1 transition, a new prefetch is generated. Thus, the Tagged Prefetcher considers not only cache misses but also accesses to prefetched blocks when generating prefetches. The simple block diagram of the Tagged Prefetcher is shown in Fig. 3.1. In gem5 the cache object is notifying the prefetcher component whenever above mentioned conditions are met and the prefetcher generates as many next block addresses as its *degree* specifies. Those generated addresses will be queued in the so called "prefetcher queue" and prefetched when all outstanding cache demand misses are handled.

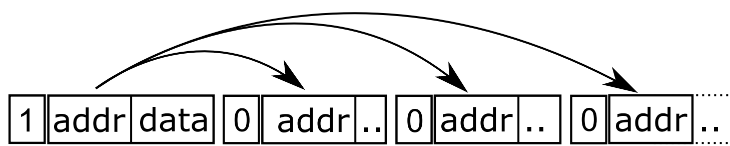


Figure 3.1: Tagged prefetcher block diagram. Bit transitions from 0 to 1 trigger the prefetcher to generate next block addresses according to the degree.

3.1.2 Stride Prefetcher

Compared to the Tagged Prefetcher, the Stride Prefetcher [9] is a more sophisticated prefetch algorithm. Cache misses will be registered in a Program Counter (PC) Table that keeps track of the PCs of instructions that reference a missing cache block. Additionally, a stride between the previous missed memory address and the current missed memory address is calculated. Moreover, every PC Table entry holds a confidence value that is increased whenever the current stride equals the stride recorded in

the PC Table.

Fig. 3.3 shows the simplified flowchart of the prefetch algorithm. On a PC Table miss, an entry in the PC Table is created with the following information: the instruction PC, the memory address that was missing, the stride between the previous and the current missing memory addresses (initially set to 0), and the starting confidence for the given stride. If the instruction with the same PC tries to fetch another cache block that is missing (PC Table hit), a new stride will be calculated based on the PC Table entry's address and the current referenced address. Now, the PC Table entry's confidence is either increased (old and new stride match), or decreased (old and new stride don't match). If the confidence is above a certain threshold, a number of prefetches is generated, again, according to the prefetcher's *degree*. If the confidence is below the threshold, however, the new calculated stride will replace the stride recorded in the PC Table entry. In gem5, the cache object notifies the prefetcher component through a notify() function. Fig 3.2 shows the block diagram of the Stride Prefetcher. Exemplary, two different degrees and strides are shown. The complete flowchart of the Stride Prefetcher is found in Appendix A.1.

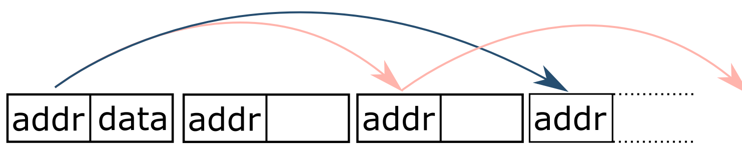


Figure 3.2: Stride Prefetcher block diagram. Prefetches are generated based on the missing cache block address. The number of prefetches generated depends on the degree (blue shows a degree of 1 while pink shows a degree of 2). The stride determines the address offset (blue shows a stride of 3 blocks, while pink shows a stride of 2 blocks).

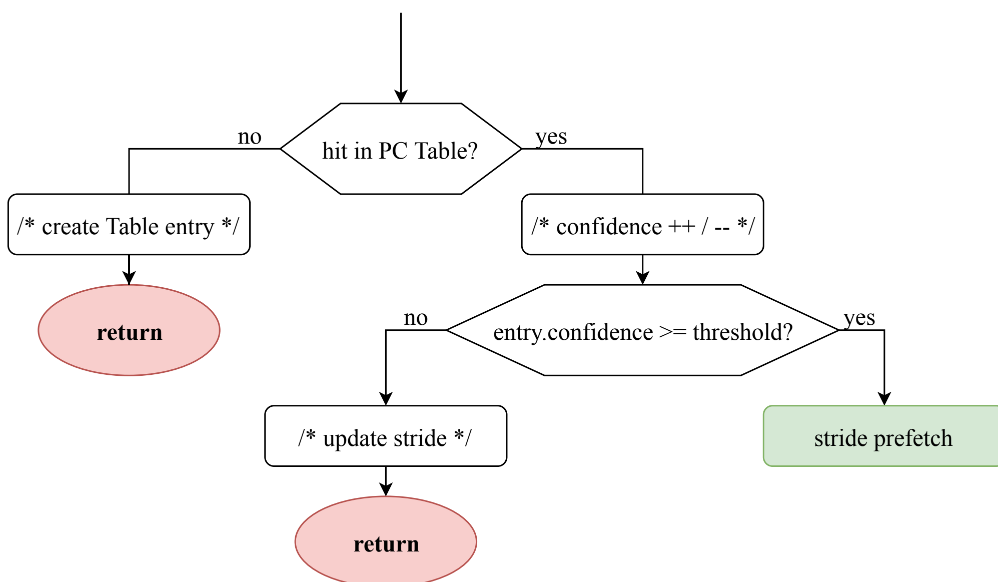


Figure 3.3: Stride Prefetcher flow chart.

3.2 Model Matching

In order to create a realistic simulation model, the system defined in gem5 and its performance are closely matched and compared to the ARTIK710 hardware development board [citation]. As discussed above, the files included in the ARM Research Starter Kit (ARM RSK) are used as a starting point for the simulation model matching. The ARTIK710 hardware development board implements a ARM Cortex-A53 octa-core processor and runs Ubuntu as default operating system. Publicly disclosed specifications of the board are shown in Table 3.1. Next to the ARTIK710 specifications are the gem5 system specifications that are used for the memory access pattern analysis in Chapter 4. “n/a” in Table 3.1 indicates that information was not openly available. In those cases, default parameters of the HPI CPU are assumed, which are given in the ARM RSK. Rows highlighted in gray in Table 3.1 show modifications to the generic HPI system that are made in order to match the ARTIK710 board.

Table 3.1: System specifications of the ARTIK710 hardware development board and the matching gem5 system. Rows highlighted in gray show modifications to the generic HPI system.

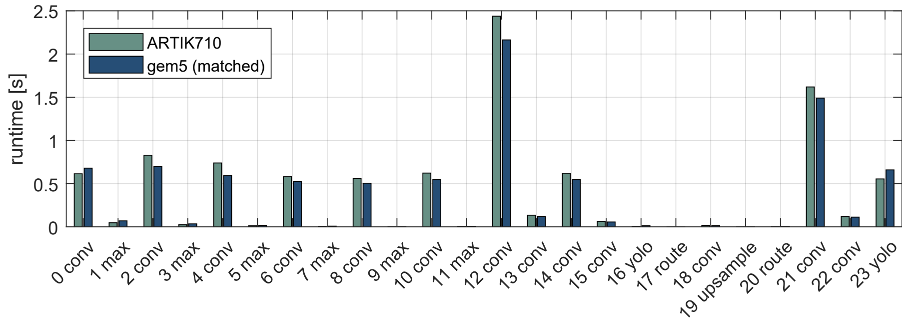
		ARTIK710	gem5 sys.
CPU	core type	Cortex A-53	HPI
	no. of cores	up to 8	up to 4
	core frequency	1400 MHz	1400 MHz
L1I	size	32 kB	32 kB
	latency	n/a	1 cycle
	associativity	2 [1]	2
	mshrs	n/a	2
	write buffers	n/a	8
L1D	size	32 kB	32 kB
	latency	n/a	1 cycle
	associativity	2 [1]	2
	mshrs	n/a	4
	write buffers	n/a	4
Prefetcher	type	n/a	Stride
	queue size	n/a	4
	degree	n/a	4
L2	size	2× 512 kB	512 kB [2]
	latency	13 [1]	13 cycle
	associativity	16 [1]	16
	mshrs	n/a	4
	write buffers	n/a	16
Prefetcher	type	n/a	Stride
	queue size	n/a	4
	degree	n/a	4
DRAM	type	DDR3	DDR3
	size	2 × 512 MB	2 × 512 MB
	bus width	32 bit	32 bit

[1] source: J. L. Hennessy, and D. A. Patterson, *Computer Architecture: A Quantitative Approach* [10]

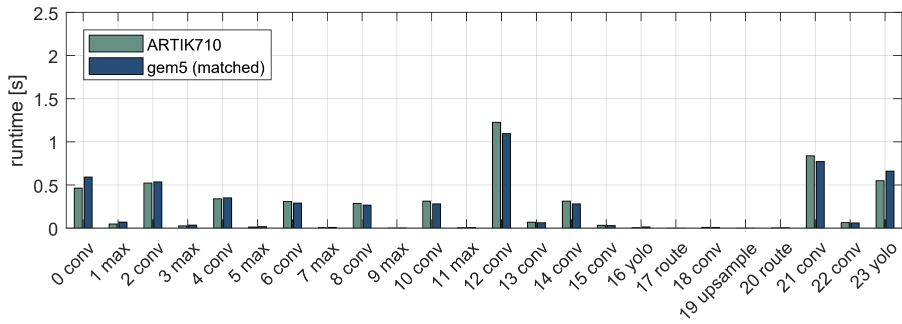
[2] 512 kB are allocated per cluster in ARTIK710. The matched gem5 model is setting up just a single cluster, thus just 512 kB are specified.

Modifications to the ARM RSK system setup are mainly concerning the main memory model and the L2 cache prefetcher. While the ARM RSK system sets up the main memory in a Dual Inline Memory Module (DIMM) fashion, consisting of 8 DRAM devices with a 8 bit data bus each, the ARTIK710 board specifies just two DRAM devices with a total bus width of 32. As for the prefetcher, the exact algorithm in use inside the ARM Cortex A-53 is not publicly disclosed. However, the technical reference manual of the ARM Cortex A-53 processor is stating that the prefetcher is able to recognize misses in a "fixed stride pattern" [6]. Thus, the Stride Prefetcher, that is also defined in the ARM RSK setup, is attached to the L1D cache for the gem5 model. Further, the ARM RSK is not attaching any prefetcher to the L2 cache. Tiny YOLOv3 runtime and main memory access counter differences of the hardware board and the gem5 system, however, suggest that L2 cache prefetching is at work inside the ARM Cortex A-53 processor. Thus, another Stride Prefetcher with a queue size and degree of 4 is attached to the L2 cache, as seen in Table 3.1.

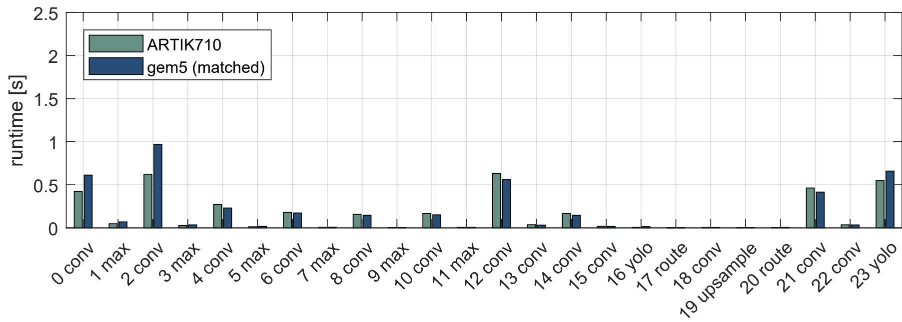
Fig. 3.4 shows a comparison of Tiny YOLOv3 layer runtimes of the ARTIK710 development board and the matching model defined in gem5. Accumulated absolute mean error for the total runtime with single core execution is 4.95%, while dual and quad core mean absolute errors are at 2.61% and 3.82%, respectively. For single core execution, the matched gem5 system model runtimes are slightly shorter on average, with the exception of layer 0 (conv) and layer 23 (yolo). The model is further validated running the PARSEC (Princeton Application Repository for Shared-Memory Computers) 3.0 benchmark suite [7]. Runtime comparisons of the matched gem5 model and the ARTIK710 board are found in Appendix A.3.



(a) Single core; mean absolute error = 4.95%



(b) Dual core; mean absolute error = 2.61%



(c) Quad core; mean absolute error = 3.82%

Figure 3.4: (a) Single core, (b) dual core, and (c) quad core Tiny YOLOv3 layer runtime comparison of the matched gem5 model and ARTIK 710 development board. For (a) the absolute mean error in total runtime is 4.95%, while for (b) and (c) it is 2.61% and 3.82%, respectively.

Chapter 4

Analysis

This chapter discusses the layer based memory access pattern analysis. While the first section analyzes the general utilization of the memory system during the execution of the Convolutional Neural Network (CNN), the second section emphasizes on data prefetching and its performance gains.

4.1 Memory Access Pattern

Different layer types in a CNN result in alternating system performances. This becomes evident in Fig. 4.1, where Cycles Per Instructions (CPI) and the memory intensity of each layer in Tiny YOLOv3 are shown. The memory intensity refers to the ratio of memory read and write operations (including micro operations) to the total number of operations. As for the performance, measured in CPI, an alternating pattern can be recognized for layers 1 to 12. While all the max-pooling layers show a CPI of 1.4 or less, the convolutional layers in between show a relatively constant CPI of 1.5. This pattern, clearly distinguishing max-pooling and convolutional layers, is also evident for the memory intensity: while convolutional layers show a memory intensity of between $40\,\%$ and $50\,\%$, max-pooling layers remain at around $10\,\%$. Thus, Fig. 4.1 indicates that higher layer based memory intensity results in worsened performance.

In layer 16, the first set of predictions is made by the network, and thereafter no max-pooling layer is used. Thus a similar pattern as observed above is missing.

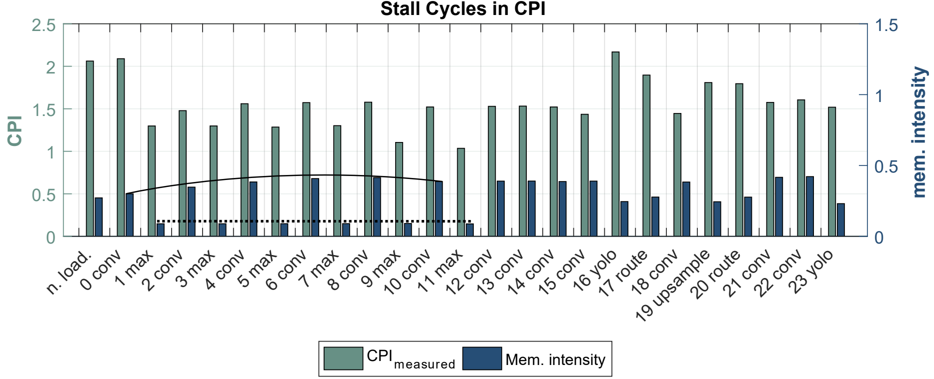


Figure 4.1: Cycles Per Instruction (CPI) and memory intensity of each layer in Tiny YOLOv3. The memory intensity shows the ratio of memory operations per total operations. The solid line marks the memory intensity trend of the convolutional layers, while the dotted line marks the memory intensity for max-pooling layers.

To analyze the CPI based performance of Tiny YOLOv3 further, Fig. 4.2 divides the measured CPI (based on micro operations) into three categories: the ideal CPI of the CPU, here ARM Cortex-53, CPI caused by pipelines stalls, and CPI caused by memory stalls. The CPI categorized as pipeline stalls includes stalls due to incorrect branch predictions, insufficient resources etc. Memory stalls, on the other hand, include stalls caused by memory latency affecting all types of L1 caches: L1 data, L1 instruction, but also the walker caches for the data pages translation buffer (DTB) and instruction page translation buffer (ITB). Equation 4.1 shows how the gem5 measured CPI is divided into the three categories. While the measured CPI is given and the ideal CPI is constant, both the memory stalls related CPI and pipeline stalls related CPI have to be determined:

$$CPI_{\text{measured}} = CPI_{\text{ideal}} + CPI_{\text{mem.}} + CPI_{\text{pipe}}. \tag{4.1}$$

The memory related CPI is then derived as follows:

$$CPI_{\text{mem.stall}} = (AML_{\text{L1}} \cdot MR_{\text{L1}} + AHL_{\text{L1}} \cdot HR_{\text{L1}}) \cdot memory\, intensity, \quad (4.2)$$

where AML refers to average miss latency, MR to miss rate, AHL to average hit latency, and HR to hit rate. Fig. 4.2 shows that in general convolutional layers show a larger proportion of memory stalls compared to max-pooling layers and thus supports the above made observation that increased memory intensity is directly related to worsened performance of the targeted CNN.

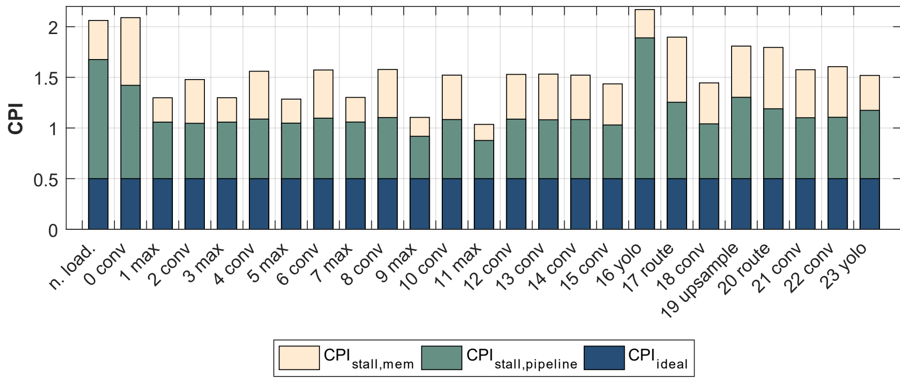
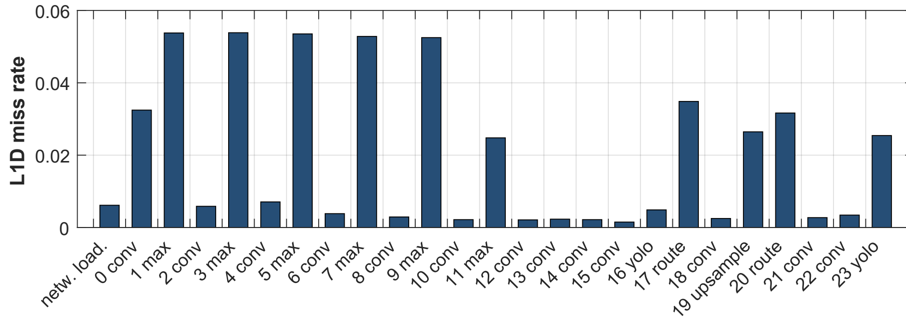
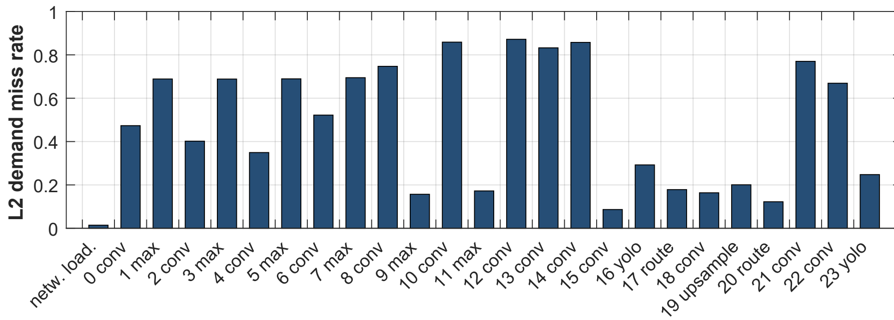


Figure 4.2: Measured CPI divided into three categories: ideal CPI, CPI caused by pipeline stalls, and CPI caused by memory stalls.

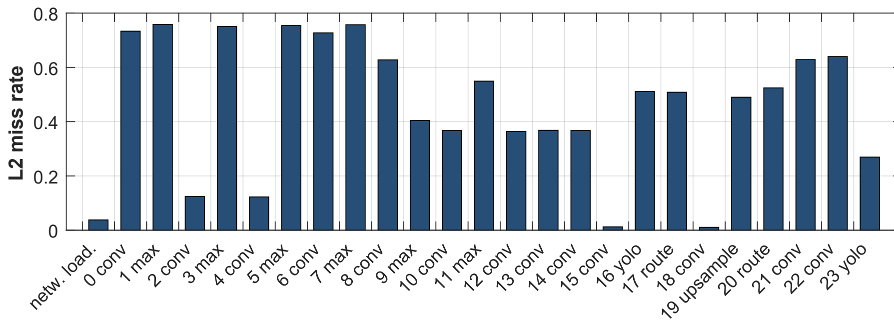
In Fig. 4.3, the L1 data cache miss rate, L2 demand miss rate, and L2 overall miss rate is shown. It can be seen that similar to CPI and memory intensity, also miss rates of L1 data and L2 cache show a layer based regularity. At $5\,\%$, max-pooling layers show larger L1 data cache miss rates compared to convolutional layers, that show miss rates below $1\,\%$. The high data reuse in convolutional layers, where kernel and feature map data is loaded once and used multiple times, is resulting in lower miss rates. Max-pooling layers, on the other hand, require new data to be loaded for every pooling operation. On average, the L1 data cache miss rate is lower than $5\,\%$ and peaks for max-pooling layers. In Fig. 4.3 (b), the L2 cache demand miss rate is shown. The



(a)



(b)



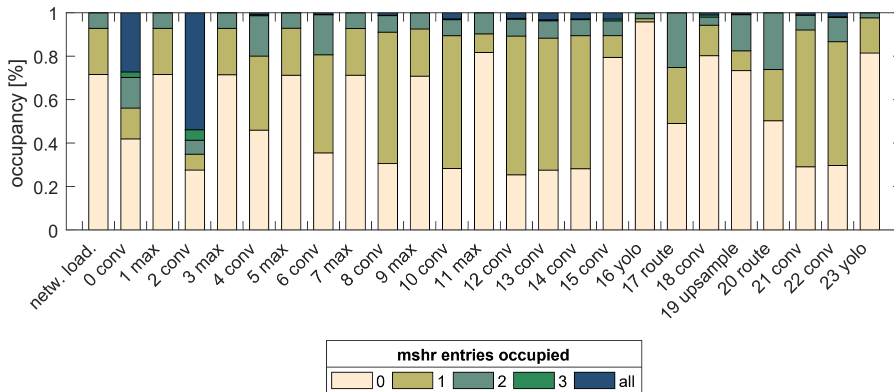
(c)

Figure 4.3: (a) L1 data, (b) L2 demand, and (c) overall L2 cache misses.

demand miss rate excludes misses caused by L1 cache prefetching accesses. The L2 demand miss rate is often tenfold higher than the L1 data cache miss rate for the same layer. It peaks for convolutional layers 10, 12, 13, and 14, where it is higher than 80 %.

While early convolutional layers show L2 demand miss rates of between 40 % and 50 %, they increase to about 80 % from layer 10. Max-pooling layers show a reversed pattern when it comes to L2 demand miss rates, as early layers are at 60 % while deeper layers show demand miss rates of just around 15 %. Lastly, Fig. 4.3 (c) shows the overall L2 cache miss rate. High miss rates in Fig. 4.3 (b) and (c) suggest a high inefficiency of the second cache level.

(a)



(b)

Figure 4.4: L2 cache MSHR occupancy per layer during the simulation of Tiny YOLOv3 for (a) single core and (b) octa core. Fully occupied MSHR entries result in the cache entering a blocking state.

Fig. 4.4 (a) and (b) show the layer based L2 cache Miss Status Holding Register (MSHR) occupancy for single and octa core, respectively. In gem5, any cache component enters a blocking state when all MSHR entries are fully occupied. In that case, increasing memory latencies are expected because incoming requests are blocked. Because the gem5 system has only one L2 cache object (opposed to two L2 cache instances on the ARTIK710 hardware development board), the MSHR occupancy represented in Fig. 4.4 (b) might be higher than on the hardware counter part. However, it is assumed that the figure shows the correct trend of MSHR occupancy. For the single core simulation of Tiny YOLOv3 shown in Fig. 4.4 (a), fully occupied MSHR entries are not evident. For octa core execution, however,fully occupied MSHR entries are evident for layer 0, 2, 10, 12, 13, 14, and 15. This is expected because during the octa core simulation eight L1 data caches send requests to a single L2 cache. Especially during the execution of layer 2 (conv), the MSHR show a high occupancy. As a result, the L2 cache is being blocked for more than 50 % of the layer runtime.

4.2 Prefetching Efficiency

The previous section discussed that the targeted CNN, Tiny YOLOv3, shows overall low L1 data cache miss rates but at the same time high L2 cache miss rates. Thus, when analyzing prefetching efficiency, the focus is on the second cache level.

As pointed out in [11], one important metric for prefetching efficiency is the proportion of misses that have been prevented by the cache prefetcher. Fig. 4.5 shows L2 data cache misses that occured as a consequence of accesses made by the L1 data cache. The figure shows accumulated misses for three different systems: one having no L2 cache prefetcher, the other two having either Stride or Tagged Prefetcher attached to the second level cache. All systems include a L1 data cache Stride Prefetcher with a queue size and degree of 4. While layer 0 (conv) is an exception, systems with a L2 cache prefetcher show greatly reduced overall L2 misses for all deeper convolu-

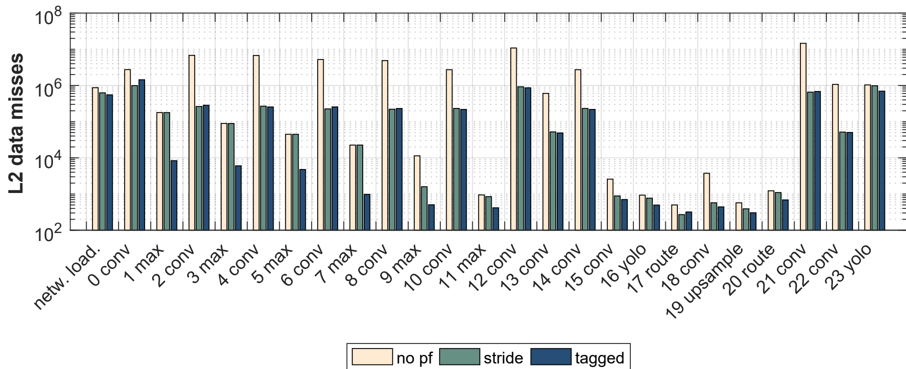


Figure 4.5: L2 cache data misses comparison for three different systems: no attached L2 prefetcher, L2 Stride Prefetcher, and L2 Tagged Prefetcher. All systems have a Stride Prefetcher attached to the L1 data cache.

tional layers. Other layer types like yolo, route, and upsample show overall reduced L2 misses even though the magnitude of reduced misses differs. The most crucial observation is that for max-pooling layers 1, 3, 5, and 7, the Stride Prefetcher fails to reduces any L2 cache misses. At the same time, the Tagged Prefetcher reduces misses efficiently. As shown in section 3.1.2, Fig. 3.3, the Stride Prefetcher aborts prefetch address generation on two conditions: either when no Program Counter (PC) Table hit occurs, or when the confidence for a given stride is too low. These abort conditions make the Stride Prefetcher fail to reduce misses during above mentioned max-pooling layers. Thus, the Tagged Prefetcher shows a clear advantage over the Stride Prefetcher for max-pooling layers. For other layers, however, the Stride Prefetcher shows a similar efficiency to the Tagged Prefetcher (e.g. layer 6, 8, 20, 21, 22) or even outperforms it like in the case of layer 0 (conv).

In order to further analyze why the Stride Prefetcher fails to reduce any misses during max-pooling layers, the cause of prefetch aborts has to be counted. Additional counters have been implemented into gem5, counting whether aborts occured due to PC Table misses or low confidence in the recorded strides. Fig. 4.6 shows the

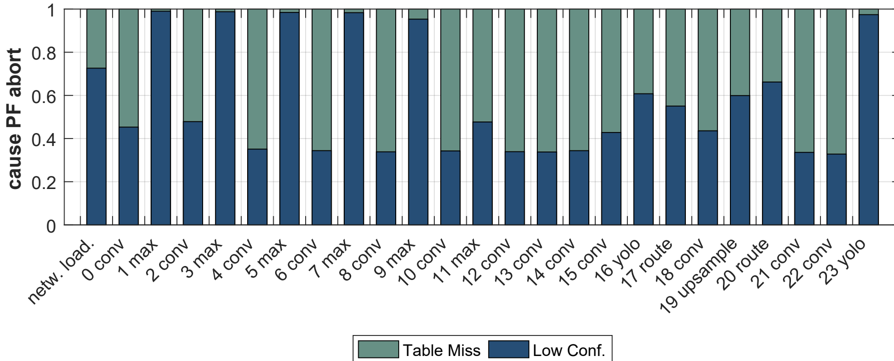


Figure 4.6: Stride prefetch abort causes.

causes of aborted prefetch generations for the L2 Stride Prefetcher when simulating Tiny YOLOv3. While for most layers a missing PC Table entry is the major cause of prefetch generations being aborted, low confidence in the recorded stride leads to the abort of almost all prefetches during max-pooling layers (1, 3, 5, 7, and 9).

Having discussed reduced L2 cache misses, the expectation is to see reduced layer runtimes proportional to the reduced misses. The layer specific speedup normalized to the system without L2 prefetcher is shown in Fig. 4.7. For most layers Stride and Tagged Prefetcher speed up the performance to an equal magnitude. For max-pooling layers 1, 3, 5, and 7, however, the Tagged Prefetcher shows higher improvements compared to the Stride Prefetcher. During network loading, on the other hand, the Tagged Prefetcher is causing a slow down. Thus, all performance gains the Tagged Prefetcher shows during the inference are canceled out due to longer network and weight loading times, as is shown Table 4.1.

Although L2 misses are reduced during network and weight loading, the runtime is increased when the Tagged Prefetcher is attached to the second level cache. This may be caused by the prefetcher generating more prefetches compared to the Stride Prefetcher. Whenever the main memory responds to requests from the L2 cache prefetcher, weight loading into the main memory is stalled.

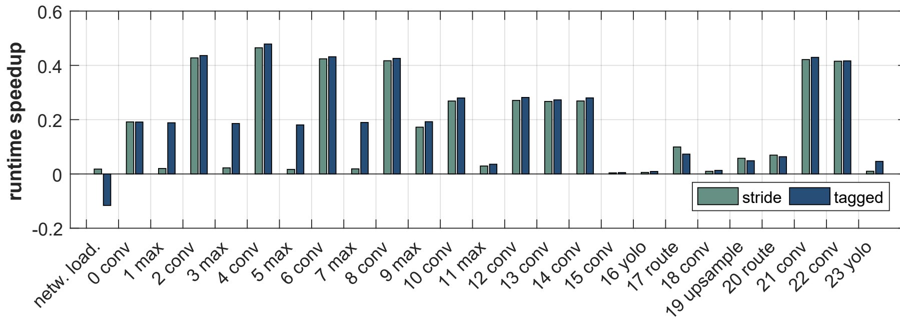


Figure 4.7: Runtime speedup normalized on runtimes of a system without L2 prefetcher.

Table 4.1: Layer type runtimes with different L2 prefetcher types

L2 prefetcher	net. load.	convolution	max-pooling	others	total
none	4.98 s	12.28 s	0.14 s	0.68 s	18.08 s
stride	**4.88 s**	7.95 s	0.14 s	0.67 s	**13.64 s**
tagged	5.55 s	**7.84 s**	**0.11 s**	**0.65 s**	14.15 s

Key observations in this section are that the Stride Prefetcher failed to predict a stride for max-pooling layers. Due to changing strides during max-pooling the threshold confidence is not reached and the Stride Prefetcher proves inefficient for this layer type. Another observation is that the Tagged Prefetcher showed overall better performance improvements for inference layers than the Stride Prefetcher. However, network and weight loading is slowed down compared to a system having no L2 prefetcher. This shows that the aggressive prefetching scheme of the Tagged Prefetcher is unsuitable for network and weight loading.

Chapter 5

Proposal and Evaluation

Chapter 4 discussed the targeted CNN's layer based regularities in its memory access pattern. F.e., layer type dependent memory access statistics like L1 miss rates and L2 miss rates are show a recurring pattern and are thus predictable. Moreover, it was shown that the Tagged Prefetcher may prove efficient for inference layers but the Stride Prefetcher proves more efficient for network and weight loading.

Based on these findings, this chapter discusses two proposals: an adaptive prefetching scheme that combines two existing prefetchers and a new cache organization that minimizes power consumptions.

5.1 Adaptive Prefetcher

Section 4.2 discussed performance improvements related to L2 cache prefetching. As shown in Fig. 4.5, the network and weight loading and all inference layers of Tiny YOLOv3 show reduced L2 cache misses when a prefetcher is attached to the L2 cache. However, resulting runtime improvements still depend on the type of prefetcher and the specific layer.

The proposed Stagged Prefetcher combines two prefetchers, Stride Prefetcher and Tagged Prefetcher, in order to optimally reduce L2 misses and increase performance

of Tiny YOLOv3. In Section 3.1.2 it was discussed that the Stride Prefetcher had two conditions on which the generation of a prefetch is aborted: either, on a PC Table miss, or, when a PC Table entry's stride is below the confidence threshold. The idea of the Stagged Prefetcher, shown in Fig. 5.1, is to prefetch in two different fashions: ① if a stride is found and the confidence for it is above the threshold, the Stagged Prefetcher generates prefetches in the same way like the Stride Prefetcher. ② However, on occasions where the Stride Prefetcher would abort prefetch generations, the Stagged Prefetcher prefetches data according to the Stagged Prefetcher, with a fixed address offset and degree.

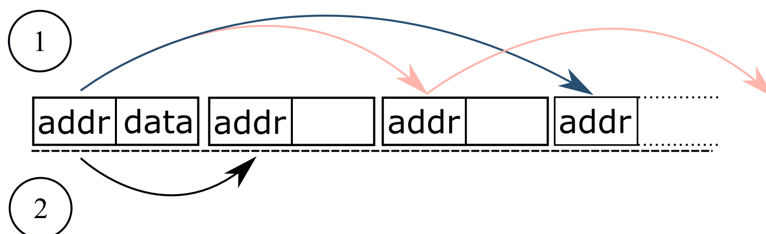


Figure 5.1: The proposed Stagged Prefetcher. ① Prefetch generation is equal to the Stride Prefetcher if a stride is found. ② If no stride is found or confidence is too low, prefetches generation is equal to the Tagged Prefetcher.

In Fig. 5.2 L2 cache demand misses, excluding accesses by the L1 data cache prefetcher are shown. Demand misses are a stronger indicator for the overall system performance than overall L2 cache misses. The proposed prefetcher works as intended: in the worst case, the proposed prefetcher reduces L2 demand misses equal to the Stride or Tagged Prefetcher, whatever prefetcher shows less cache miss reduction (see layer 22 (conv)). However, in the best case, the proposed prefetcher outperforms both, Stride and Tagged Prefetcher (see layer 17 (route)). For max-pooling layers, where the Stride Prefetcher fails to reduce misses compared to having no L2 prefetcher, the proposed Stagged Prefetcher reduces misses at the same magnitude of the Tagged Prefetcher (see layers 1, 3, 5, 7, 9, and 11).

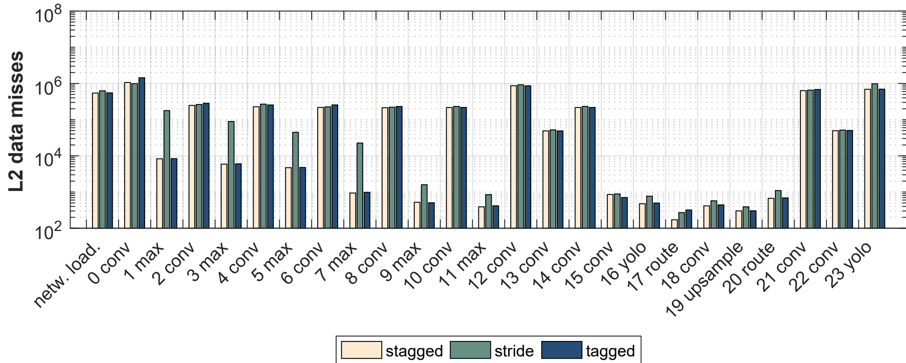


Figure 5.2: L2 cache demand misses comparison for three systems: L2 Stride Prefetcher, L2 Tagged Prefetcher, and L2 Stagged Prefetcher. All systems have a Stride Prefetcher attached to the L1 data cache.

To conclude the discussion of the proposed Stagged Prefetcher, Fig. 5.3 shows the speedup of all three different prefetching algorithms normalized on the system having no L2 prefetcher. The first observation that a slowdown is only evident for network and weight loading when the Tagged Prefetcher is used. Max-pooling layers 1, 3, 5, 7, 9, and 11 are sped up to the same degree by Stagged and Tagged prefetcher, while Stride Prefetcher lacks behind. For layer 17 (route), the Stagged Prefetcher even outperforms Stride and Tagged Prefetcher. Thus, as intended, shortcomings of both existing prefetching algortihms, Stride and Tagged Prefetcher, are eliminated in the combined prefetcher proposal. Exact layer runtimes are shown in Table 5.1.

Table 5.1: Layer type runtimes with different L2 prefetcher types

L2 prefetcher	net. load.	convolution	max-pooling	others	total
stride	**4.88 s**	7.95 s	0.14 s	0.67 s	13.64 s
tagged	5.55 s	**7.84 s**	**0.11 s**	**0.65 s**	14.15 s
stagged	**4.88 s**	7.93 s	**0.11 s**	**0.65 s**	**13.57 s**

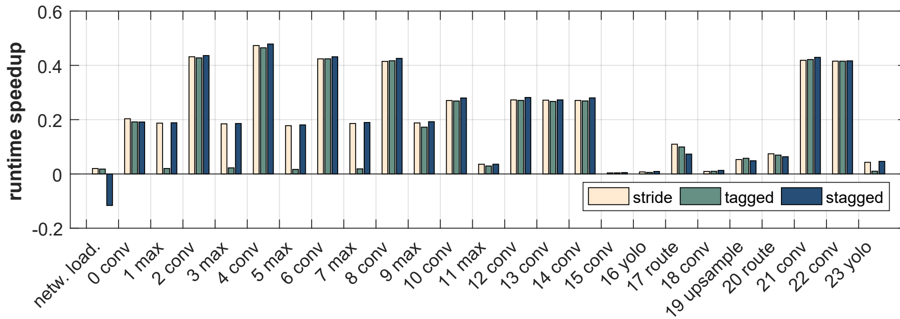


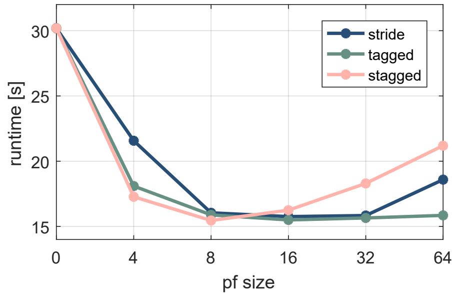
Figure 5.3: Runtime speedup normalized on runtimes of a system without L2 prefetcher.

5.2 L2 cache bypassing

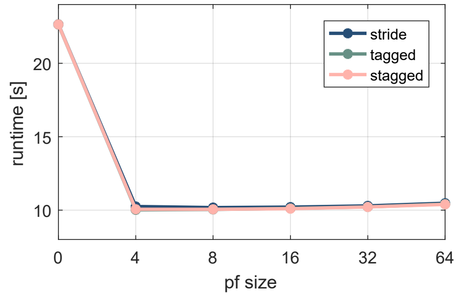
Two observations are crucial for the proposal of a system bypassing the L2 cache during the execution of a CNN. Firstly, Section 4.1 shows the overall inefficiency of the L2 cache during the execution of Tiny YOLOv3 on the gem5 system matching the ARTIK710 board. Miss rates of more than 70 % for certain layers reveal the small proportion of L1 misses mitigated by the second level cache. Secondly, Section 4.2 shows the improved performance of Tiny YOLOv3 when a hardware prefetcher is attached to the L2 cache.

The idea of bypassing the L2 cache is to save L2 cache lookup latency. The higher the cache miss rate, the more lookup latency is spent needlessly to the point where the benefits of a L2 cache are eliminated completely. With miss rates as high as 70 % or 80 %, the majority of L2 accesses needlessly adds cache lookup latency. Moreover, when bypassing the L2 cache dynamic and leakage power consumption can be reduced. In this case, prefetching at the first level caches becomes crucial: because the L1 data cache is smaller in size compared to the L2 cache, the data of large feature maps and kernels cannot be stored in its entirety. Thus, L1 prefetching is not only vital but has to solve the problem of long access latencies to the main memory.

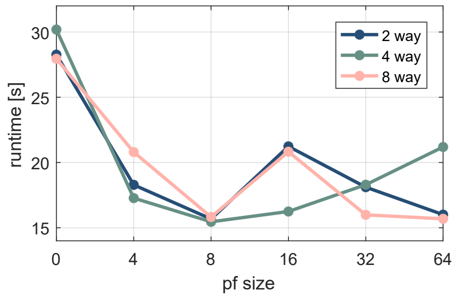
Fig. 5.4 compares different L1 prefetching algorithms and cache parameters, in-



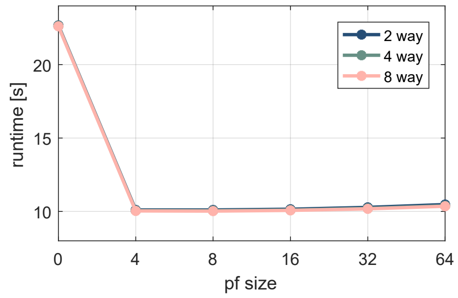
(a)



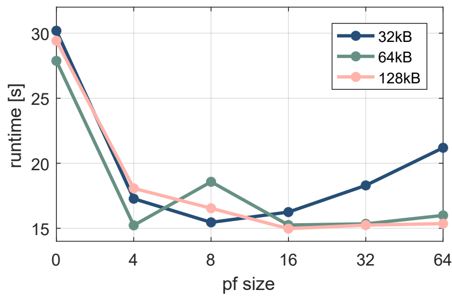
(b)



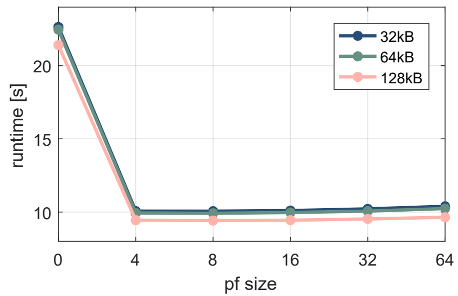
(c)



(d)



(e)



(f)

Figure 5.4: L1 data cache and prefetch parameter reevaluation. Figures on the left show runtimes including network and weight loading as well as inference layers. Figures on the right show only runtimes inference layers. The x-axis, labeled "pf size", shows the queue size and the degree of a given prefetching scheme. (a) and (b) compare prefetching algorithms, (c) and (d) compare cache associativity, (e) and (f) compare cache sizes.

cluding size and associativity, by evaluating Tiny YOLOv3 runtimes on different systems. Runtimes shown on the left of Figure 5.4 include the network and weight loading stage as well as the inference layers, while plots on the right side show only the inference runtimes. The x-axis, labeled “pf size”, shows the queue size and the degree of a given prefetching scheme. In Figure 5.4 (a) and (b), three prefetching schemes introduced in this thesis are compared: Stride, Tagged and Stagged Prefetcher. In (a), the Stagged Prefetcher shows the shortest runtimes for smaller prefetcher sizes of 4 and 8 while at larger prefetcher sizes the Tagged Prefetcher shows the best performance. However, at sizes larger than 8, the Tagged Prefetcher shows saturation in (a) while runtime even increases for Stride and Stagged prefetcher. Performance saturation is also evident in 5.4 (b) where runtimes are almost constant for prefetcher sizes between 4 and 16. Based on Fig. 5.4 (a), remaining L1 cache parameters are reevaluated with the use of the Stagged Prefetcher. Fig. 5.4 (c) and (d) compare different cache associativities. The 4-way associativity, again, shows shortest runtimes for a prefetcher size of 8 while other associativities show spikes in runtimes when the prefetcher size is increased from 8 to 16. These spikes occur due to the network and weight loading as they are not found in Fig. 5.4 (d). Selecting the 4-way associativity, Fig. 5.4 (e) and (f) compares different L1 cache sizes. The smallest cache size of $32\,\mathrm{kB}$ has a clear optimum at a prefetcher size of 8. Increasing the cache size further does not reduce runtimes significantly. Overall it is evident that if only inference layers are regarded, runtime improvement seems saturated at a prefetcher size of 4. However, if both, network loading and inference runtimes are regarded, most optimums can be found at a prefetcher size of 8.

Fig. 5.5 compares the gem5 system matching the ARTIK710 board with two systems bypassing the L2 cache: one without any prefetcher attached and the other with a Stagged Prefetcher of size 8 attached to the L1 data cache. The selection of the cache and prefetcher parameters is based on the findings of Fig. 5.4. At over 30 seconds, the system bypassing L2 without any L1 prefetcher shows a runtime that is more

than twice as long as the ARTIK710 matched system. However, when the Stagged Prefetcher is attached to the first level cache, only a mere increased runtime of $10\,\%$ is observed.

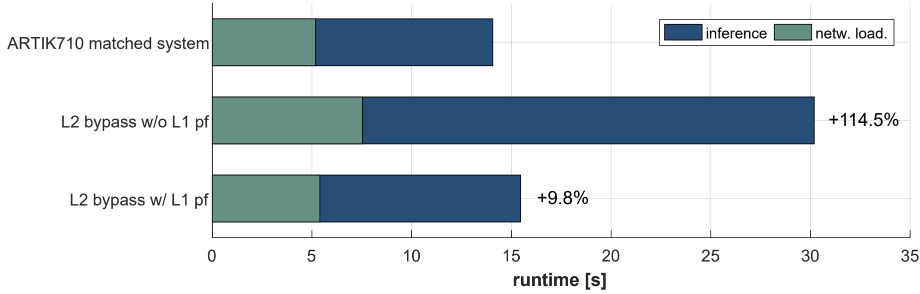


Figure 5.5: Tiny YOLO runtimes of ARTIK710 matched system and two systems bypassing the L2 cache.

When the runtimes of Fig. 5.5 are further divided into network loading and specific layer types, as shown in Table 5.2, it becomes evident that prefetching is able to reduce the runtime significantly during convolutional layers. Max-pooling layers, on the other hand, are even faster on the system bypassing the L2 cache without any prefetcher than on the ARTIK710 matched system.

Table 5.2: Layer type runtimes with different L2 prefetcher types

system	netw. load.	convolution	max-pooling	others	total
ARTIK710 matched	5.19 s	8.06 s	0.1416 s	0.68 s	14.07 s
L2 bypass w/o L1 pf	7.54 s	21.59 s	0.1373 s	0.92 s	30.19 s
L2 bypass w/ L1 pf	5.40 s	9.04 s	0.1241 s	0.90 s	15.46 s

In a system bypassing the L2 cache, power consumption can be reduced significantly. In order to estimate power consumption more accurately, the McPAT (Multicore Power, Area, and Timing) simulator has been integrated into the simulation environment. McPAT subdivides the total power consumption of a processor in sub-

components like core and caches. Thus, savings related to L2 cache bypassing can be estimated. Table 5.3 shows power consumption estimates for a single core processor closely resembling a ARM Cortex-A53 processor fabricated in a $28\,\mathrm{nm}$ process. Runtime dynamic power take component utilization, based on gem5 simulation statistics, into account. When bypassing the L2 cache, all runtime dynamic power can be saved. However, when additional techniques like power gating is used, not only runtime dynamic but also close to all leakage power can be saved when information stored at the L2 cache can be sacrificed. In that case the total power saving bypassing the L2 cache accounts of up to $27\,\%$, as seen in Table 5.3.

Table 5.3: McPAT area and power estimates for a single core processor based on Cortex-A53 parameters in 28nm

component	area	runtime dynamic power	leakage power	**total power**
processor	$3.78\,\mathrm{mm}^2$	$84.4\,\mathrm{mW}$	$115.2\,\mathrm{mW}$	$\mathbf{199.6\,mW}$
— core	$1.52\,\mathrm{mm}^2$	$61.8\,\mathrm{mW}$	$50.3\,\mathrm{mW}$	$112.1\,\mathrm{mW}$
— L2 cache	$1.78\,\mathrm{mm}^2$	$2.76\,\mathrm{mW}$	$51.7\,\mathrm{mW}$	$\mathbf{54.46\,mW}$

Chapter 6

Conclusion

In this thesis, the memory access pattern of a characteristic CNN, Tiny YOLOv3, is analyzed and proposals are made based on the findings. Memory and system related statistics show reoccurring patterns according to the layer structure of the network. Thus, the adaptive prefetching scheme of the proposed Stagged Prefetcher can improve overall performance by combining two existing prefetching schemes. While cache prefetching during the execution of the CNN is improving overall system performance, it currently cannot remedy high L2 cache miss rates. Thus, a system proposal is made where the L2 cache is bypassed to save unnecessary L2 cache lookup latency.

In conclusion, current memory optimization techniques like certain prefetching algorithms may prove efficient for one layer type while not affecting or worsening other layer types. Thus, finding adaptive techniques for the execution of CNNs may improve overall system performance. Moreover, even the current memory organization needs to be reevaluated with regards to CNNs as it is shown that traditional cache components like the L2 cache prove inefficient compared to the execution of conventional applications.

Chapter A

Appendix

A.1 Prefetching algorithm flow diagrams

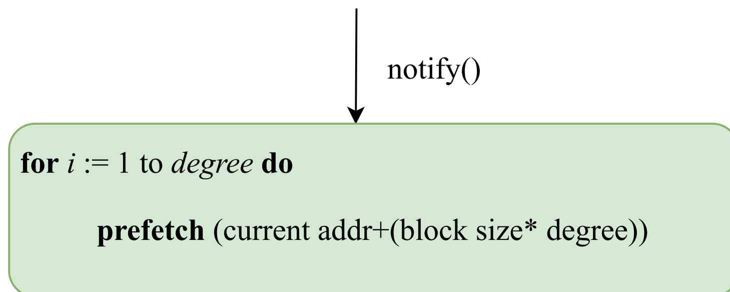


Figure A.1: Tagged Prefetcher flow diagram.

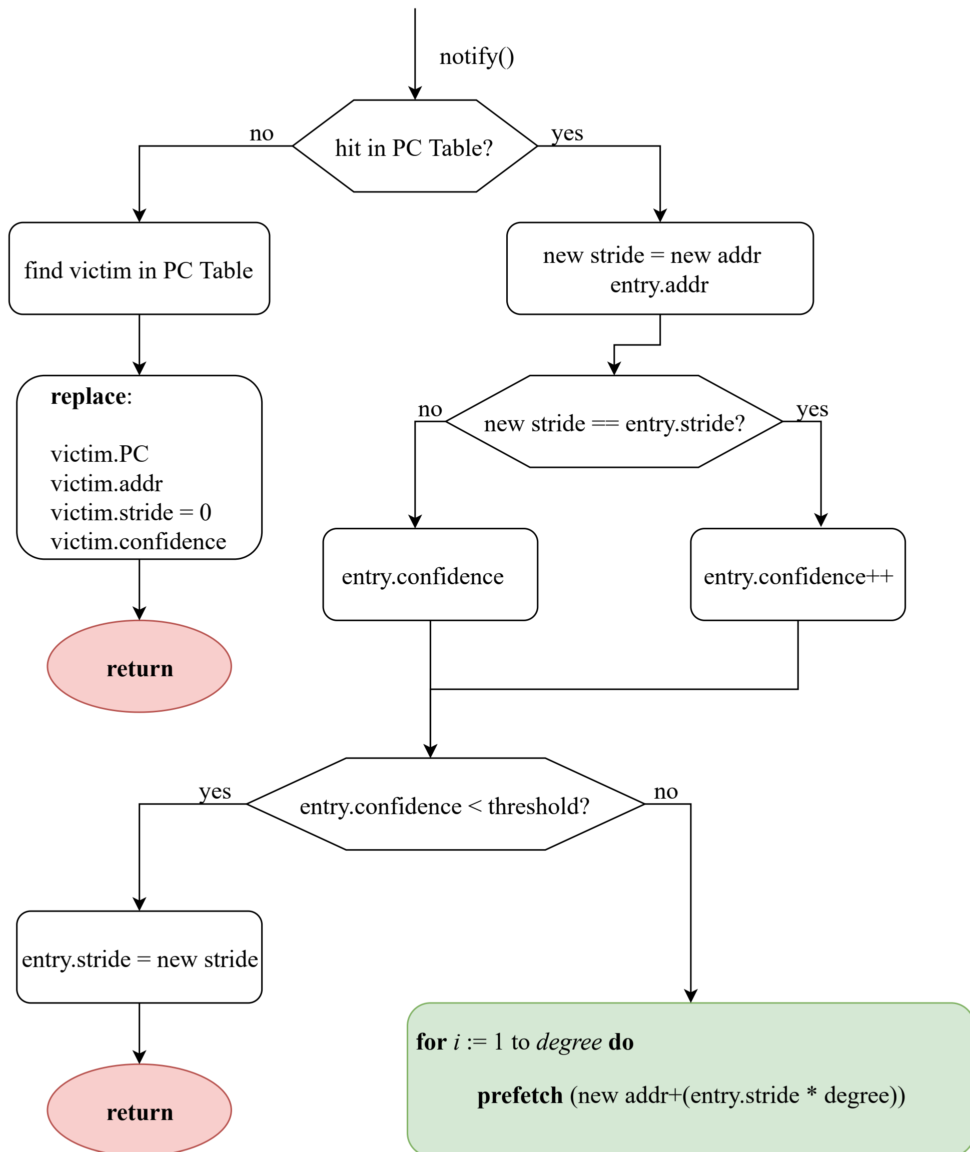


Figure A.2: Stride Prefetcher flow diagram.

A.2 Linaro disk image bug fix

In Full-System (FS) simulation, gem5 has the option to automatically read an attached run command script (.rcS) and execute the commands specified in it. The rcS script will be read during the simulation after linux has been booted and saves the user from supervising the simulation and manually inputing commands. The reading of the script is done through a gem5 pseudo instruction, **m5 readfile**, where the simulator reads the script and saves its contents to a file inside the system. The **m5 readfile** command and its call has to be implemented into the disk image used for the FS simulation. In case of the linaro image **linaro-minimal-aarch64.img** distributed within the **aarch-system-20180409.tar.xz** file by ARM, the script reading is erroneously executed twice which distorts the execution and simulation statistics measured by gem5.

To fix this duplicated script reading, the code responsible for the second rcS script reading has to be uncommented inside the **linaro-minimal-aarch64.img** disk image. The procedure of script reading and where it takes place is briefly explained hereafter. Fig. A.3 shows the typical linux boot sequence. After the Kernel is fully loaded it

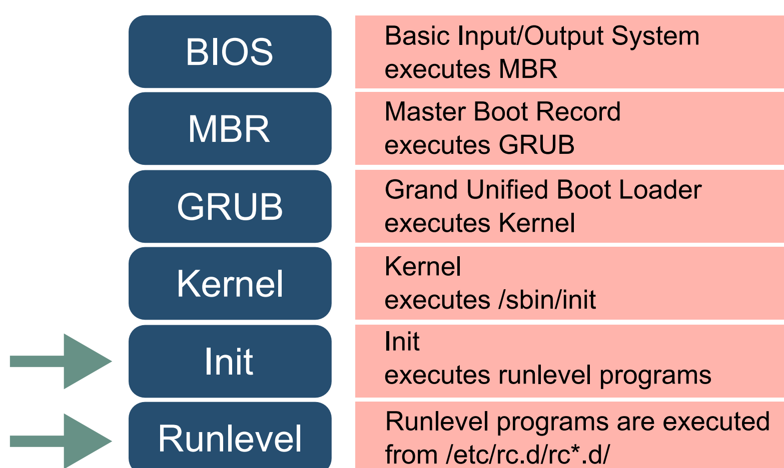


Figure A.3: The linux boot sequence. gem5 script reading takes place when Init executes the runlevel bash scripts before console login.

will execute the init process, that in turn executes runlevel programs before the login prompt.

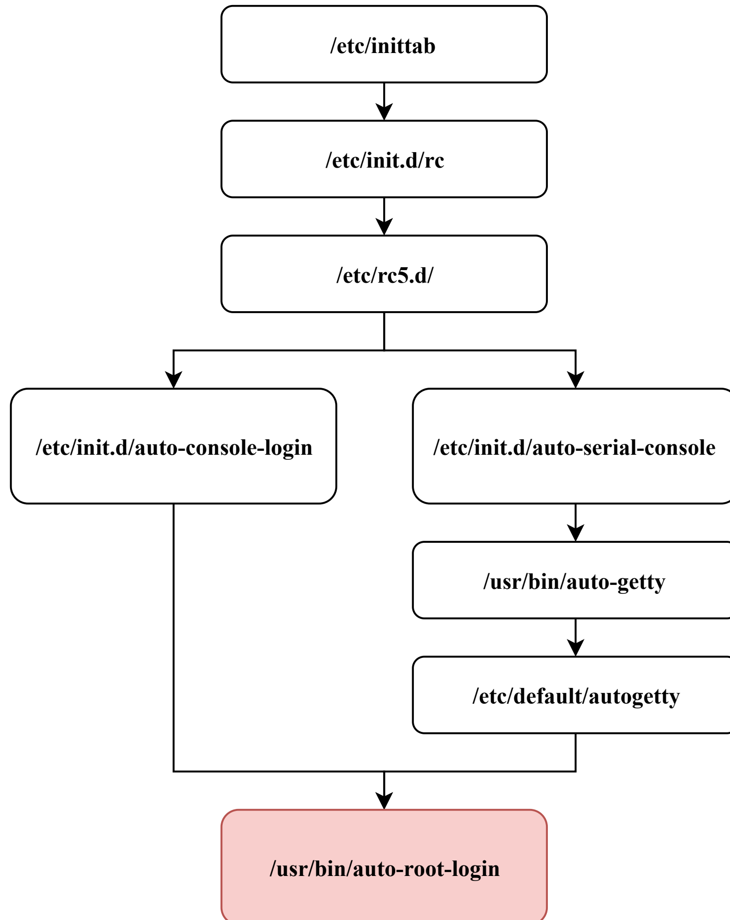


Figure A.4: The linux boot sequence. gem5 script reading takes place when Init executes the runlevel programs before console login.

Fig. A.4 shows the rather complex sequence of scripts that are called sequentially by the Init process. First, the runlevel is determined in **/etc/inittab**. According to the runlevel, **/etc/init.d/rc** is calling **/etc/rc5.d**, as here the specified runlevel is 5. There are a couple of services started in **/etc/rc5.d**. Among them are **/etc/init.d/auto-console-login** and **/etc/init.d/auto-serial-console**. Both end up calling **/usr/bin/auto-root-login** that is responsible for the script reading, as shown in Fig. A.5. The call to

/usr/bin/auto-root-login has to be uncommented at one of the two source scripts.

```
1  [...]
2  /sbin/m5 readfile > /tmp/script
3  chmod 755 /tmp/script
4  if [ -s /tmp/script ]; then
5          exec /tmp/script
6  fi
7
8  exec /bin/login -f root
9  ~
```

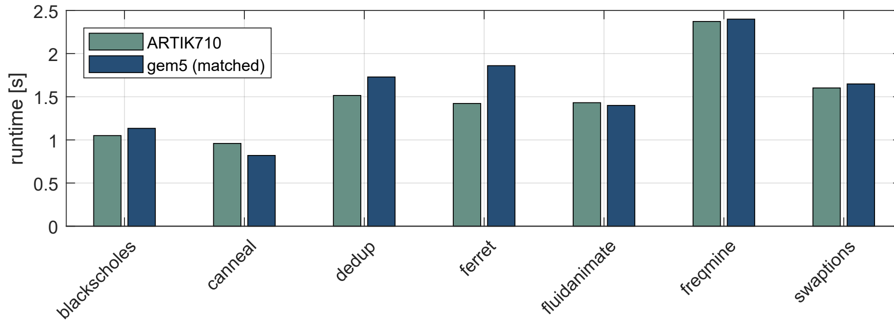
Figure A.5: auto-root-login bash script.

What makes this duplicated script reading hard to detect is the fact that in **/etc/init.d/auto-console-login** a virtual terminal is opened and the second reading of the rcS script does not show in the terminal output of the gem5 simulation, and neither do the following executions inside the virtual terminal. Thus, uncommenting line 6 in **/etc/init.d/auto-console-login**, as shown in Fig. A.6, prevents a new virtual terminal from being created and a duplicated reading and execution of the rcS script from happening.

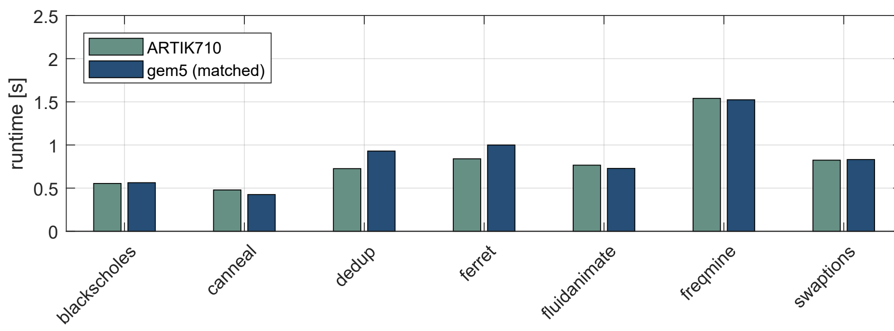
```
1  #!/bin/sh
2
3  PATH=/sbin:/bin:/usr/sbin:/usr/bin
4
5  # Start auto-login for root at consoles
6  # exec openvt -c 1 -- /usr/bin/auto-root-login
7  # exec /usr/bin/auto-root-login
```

Figure A.6: auto-console-login bash script.

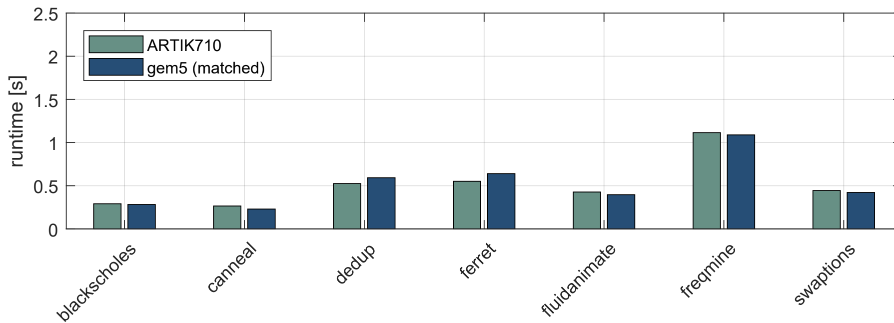
A.3 Model Matching



(a) Single core; mean absolute error = 13.81%



(b) Dual core; mean absolute error = 6.95%



(c) Quad core; mean absolute error = 4.24%

Figure A.7: (a) Single core, (b) dual core, and (c) quad core PARSEC 3.0 runtime comparison of the matched gem5 model and ARTIK 710 development board. For (a) the geometric mean ratio is 4.63%, while for (b) and (c) it is 3.74% and -0.27%, respectively.

Bibliography

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Computer Vision and Pattern Recognition (CVPR)*, 2016.

[2] J. Redmon, A. Farhadi, ”Yolov3: An incremental improvement,” Technical Report, 2018.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. “The gem5 simulator,” May 2011, *ACM SIGARCH Computer Architecture News*.

[4] A. Tousi and C. Zhu, ”Arm Research Starter Kit: System Modeling using gem5,” 2017. Available online: https://github.com/arm-university/arm-gem5-rsk/blob/master/gem5_rsk.pdf (accessed on 24 November 2018).

[5] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen and N. P. Jouppi, ”McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures,” 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), New York, NY, 2009, pp. 469-480.

[6] ARM Cortex-A53 MPCore Processor, Technical Reference Manual, revision r0p3. Available online: `http://infocenter.arm.com/help/topic/`

com.arm.doc.ddi0500e/DDI0500E_cortex_a53_r0p3_trm.pdf (accessed on 29 November 2018).

[7] C. Bienia, ”Benchmarking Modern Multiprocessors,” doctoral dissertation, Dept. Computer Science, Princeton Univ., 2011.

[8] J.D. Gindele, “Buffer block prefetching method.” July 1977, *IBM Tech Disclosure Bull. 20, 2*, 696-697

[9] J.-L. Baer and T.-F. Chen, “An effective on-chip preloading scheme to reduce data access penalty,” *Proc. Supercomputing ’91*, pp. 176-186, 1991.

[10] J. L. Hennessy, and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2017, pp. 130

[11] S. Mittal. “A Survey of Recent Prefetching Techniques for Processor Caches.” *ACM Computing Surveys*, 49(2):35–69, 2016.

초 록

신경망 및 특히 CNN (Convolutional Neural Networks)은 최근 객체 감지 및 이미지 분류와 같은 컴퓨터 비전 분야에서 탁월한 성과를 나타낸다. CNN은 일반적으로 번갈아 실행되는 서로 다른 레이어 유형을 포함한다. CNN의 성능에 최적화된 컴퓨터 아키텍처를 만들기 위해서는 위와 같은 레이어 기반 CNN에 대한 분석이 필요하다. 이 논문은 캐싱 및 프리페치의 효율성에 중점을 두고 전형적인 CNN을 분석한다. 분석 결과를 토대로 기존의 두 프리페처를 결합하여 계층별 비효율을 극복하는 적응형 L2 캐시 프리페치 방안을 제안한다. 또한 L2 캐시를 우회하는 캐시 계층 구조를 사용하여 전반적인 성능의 큰 손해 없이 전력 소모를 줄일 수 있다.

주요어: CNN, memory system, prefetching, computer architecture
학번: 2017-23530