La Sapienza University of Rome

Doctoral Thesis

Study and development of innovative strategies for energy-efficient cross-layer design of digital VLSI systems based on Approximate Computing

Author:
Giulia Stazi

Supervisor:
Dr. Francesco Menichelli

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Department of Information Engineering, Electronics and Telecommunications (DIET)

January 7, 2020

Declaration of Authorship

I, Giulia STAZI, declare that this thesis titled, “Study and development of innovative strategies for energy-efficient cross-layer design of digital VLSI systems based on Approximate Computing” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"γνῶθι σεαυτόν"

Delphi's Oracle

LA SAPIENZA UNIVERSITY OF ROME

Abstract

Faculty of Information Engineering, Informatics and Statistics (I3S)
Department of Information Engineering, Electronics and Telecommunications (DIET)

Doctor of Philosophy

Study and development of innovative strategies for energy-efficient cross-layer design of digital VLSI systems based on Approximate Computing

by Giulia STAZI

The increasing demand on requirements for high performance and energy efficiency in modern digital systems has led to the research of new design approaches that are able to go beyond the established energy-performance tradeoff. Looking at scientific literature, the Approximate Computing paradigm has been particularly prolific. Many applications in the domain of signal processing, multimedia, computer vision, machine learning are known to be particularly resilient to errors occurring on their input data and during computation, producing outputs that, although degraded, are still largely acceptable from the point of view of quality. The Approximate Computing design paradigm leverages the characteristics of this group of applications to develop circuits, architectures, algorithms that, by relaxing design constraints, perform their computations in an approximate or inexact manner reducing energy consumption.

This PhD research aims to explore the design of hardware/software architectures based on Approximate Computing techniques, filling the gap in literature regarding effective applicability and deriving a systematic methodology to characterize its benefits and tradeoffs.

The main contributions of this work are:

- the introduction of approximate memory management inside the Linux OS, allowing dynamic allocation and de-allocation of approximate memory at user level, as for normal exact memory;
- the development of an emulation environment for platforms with approximate memory units, where faults are injected during the simulation based on models that reproduce the effects on memory cells of circuital and architectural techniques for approximate memories;
- the implementation and analysis of the impact of approximate memory hardware on real applications: the H.264 video encoder, internally modified to allocate selected data buffers in approximate memory, and signal processing applications (digital filter) using approximate memory for input/output buffers and tap registers;
- the development of a fully reconfigurable and combinatorial floating point unit, which can work with reduced precision formats.

Contents

List of Figures

List of Tables

List of Abbreviations

AxC	**A**pproximate **C**omputing
AxM	**A**pproximate **M**emory
ANT	**A**lgorithmic **N**oise **T**olerance
BDD	**B**inary **D**ecision **D**iagram
BER	**B**it **E**rror **R**ate
CNN	**C**onvolutional **N**eural **N**etwork
CPS	**C**yber **P**hysical **S**ystem
CS	**C**hip **S**elect
DCT	**D**iscret **C**osin **T**ransform
DMA	**D**irect **M**emory **A**ccess
DNN	**D**eep **N**eural **N**etwork
DPM	**D**ynamic **P**ower **M**anagment
DRAM	**D**ynamic **R**andom **A**cess **M**emory
DRT	**D**ata **R**etention **T**ime
eDRAM	**e**mbedded **D**ynamic **R**andom **A**cess **M**emory
ECC	**E**rror **C**orrection **C**ode
EOR	**E**rror **O**n **R**ead
EOW	**E**rror **O**n **W**rite
ER	**E**rror **R**ate
ES	**E**rror **S**ignificance
EAS	**E**nergy **A**ware **S**cheduler
ETA	**E**rror **T**olerant **A**pplication
FA	**F**ull **A**dder
FIR	**F**inite **I**mpulse **R**esponse
GFP	**G**et **F**ree **P**age
HPC	**H**igh **P**erformance **C**omputing
HW	**H**ard**W**are
IoE	**I**nternet **o**f **E**verything
IoT	**I**nternet **o**f **T**hings
ISA	**I**nstruction **S**et **A**rchitecture
LSB	**L**east **S**ignificant **B**it
MA	**M**irror **A**dder
ML	**M**achine **L**earning
MMU	**M**emory **M**anagement **U**nit
MSB	**M**ost **S**ignificant **B**it
MSE	**M**ean **S**quared **E**rror
NBB	**N**egative **B**itline **B**oosting
OS	**O**perating **S**ystem
PDP	**P**ower **D**elay **P**roduct
PSNR	**P**eak **S**ignal to **N**oise **R**atio
PULP	**P**rocessor **U**ltra **L**ow **P**ower
QoS	**Q**uality **o**f **S**ervice
RM	**R**ead **M**argin

RTL	**R**egister **T**ransfer **L**evel
RPR	**R**educed **P**recision **F**ormat
SoC	**S**ystem **o**n **C**hip
SSIM	**S**tructural **SIM**ilarity
SNBB	**S**elective **N**egative **B**itline **B**oosting
SNR	**S**ignal to **N**oise **R**atio
SRAM	**S**tatic **R**andom **A**cess **M**emory
SW	**S**oft**W**are
TCG	**T**iny **C**ode **G**enerator
TS	**T**raining **S**et
VLSI	**V**ery **L**arge **S**cale **I**ntegration
VGG	**V**isual **G**eometry **G**roup
WM	**W**rite **M**argin

To my dearest affections, for always being by my side.
To myself for the constancy and commitment to achieve this goal, for trying and being successful.

Chapter 1

Introduction

1.1 Need for Low Power Circuit Design

In the past, during the so-called desktop PC era, the main goal of VLSI design was to develop systems capable of satisfying and optimizing real time processing requirements, computational speed and graphics quality in applications such as video compression, games and graphics. With the advancement of VLSI technology, digital systems have increased in complexity and three factors have come to convergence making energy efficiency the most important constraint:

- *Technology*. From a technological point of view, excessive energy consumption has become the limiting factor in integrating multiple transistors on a single chip or a multi-chip module, determining the end of the *Dennard scaling* law (Fig. 1.4).

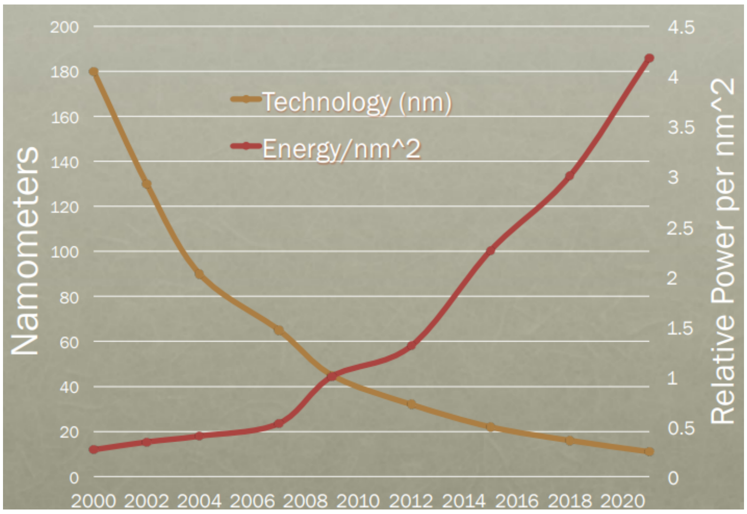


FIGURE 1.1: Dennard scaling and power consumption models. Source: Hennessy, 2018

The latter, also known as *MOSFET scaling* (1977), claims that as the transistors shrink in size, the power density stays constant; meaning indeed that the power consumption is proportional to the chip area. This law allowed chipmakers to increase clock speed of processors over years without increasing power. However, the MOSFET scaling, which held from 1977 to 1997, became fading between 1997 to 2006 until it collapsed rapidly in 2007 when the transistors became so small that the increased leakage started overheating the chip and preventing processors from clocking up further.

In these years there has been also a slowdown in *Moore's law* (Fig. 1.2), according to which the number of devices in a chip doubles every 18 months [Platt, 2018]. In facts, with the end of Dennard's law process technology scaling can continue to allow the duplication of the number of transistor for every generation, but without getting a significant improvement in switching speed and energy efficiency of transistors. As the number of transistors increased, also energy consumption raising followed. Technology scaling, while involving a reduction of supply voltage (0.7 scaling factor) and a reduction in the die area (0.5 scaling factor), started increasing parasitic capacities of about 43%, meaning that the power density increases by 40% with each generation [*Low Power Design in VLSI*].

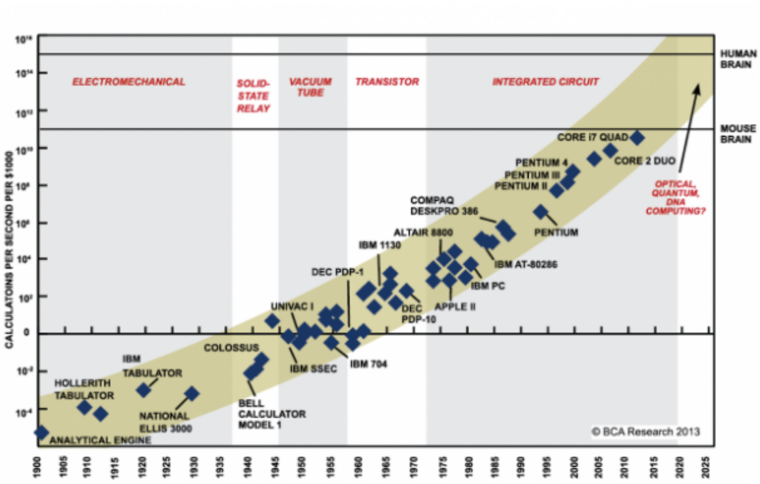


FIGURE 1.2: Moore's law. Source:

- *Architecture*. The limitations and inefficiencies in exploiting instruction level parallelism (dominant approach from 1982 to 2005) has led to the end of single-processor era. Moreover the *Amdahl's Law* (Fig. 1.3), which aims to predict the theoretical maximum speedup for programs using multiple processors, and its implications ended the 'easy' multicore era. Citing this law: *"the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude"*[Amdahl, 1967]. In [Esmaeilzadeh et al., 2013] it is also shown that, as the number of cores increases, constraints on power consumption can prevent all the cores from being powered at maximum speed, determining a fraction of cores which are always powered off (*dark silicon*). At 22 nm the 21% of a fixed-size chip must be "*dark*", and, with ITRS projections, at 8 nm, this number can grow up to more than 50%.

- *Application focus shift*. The transition from desktop PC to individual, mobile devices and ultrascale cloud computing determines the definition of new constraints. The advent of the Big Data era, IoT technologies, IoE and CPS have led to increasing demand for data processing, storage and transmission: modern digital systems are required to interact continuously with the external physical world, satisfying not only requirements on high performance capabilities but also specific constraints on energy and power consumption.

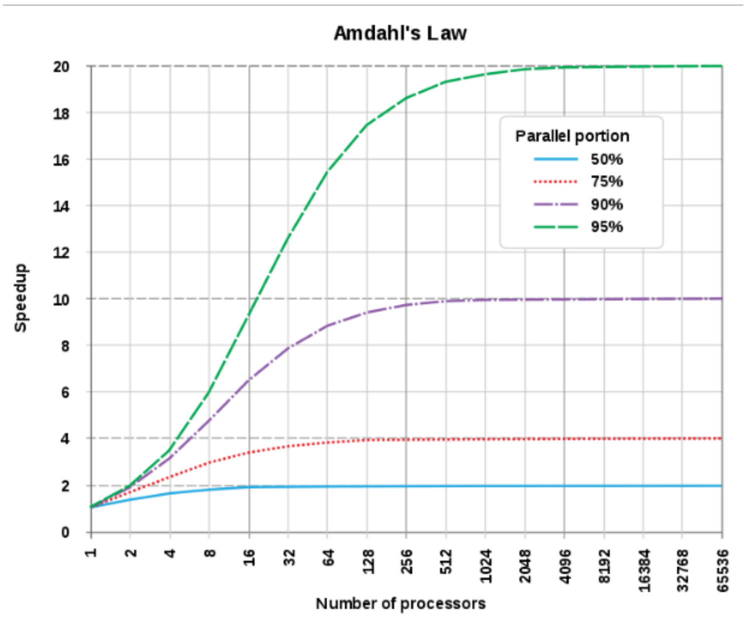


FIGURE 1.3: Amdahl's law. Source:

Power consumption therefore has become an essential constraint in portable devices, in order to take advantage of real time execution times having minimum possible requirements on weight, battery life and physical dimensions due to battery size. As an example, we can consider that almost 70% of users look for longer talk times and battery lasting times as a key feature for mobile phones [Natarajan et al., 2018]. Moreover, power consumption impacts significantly the design of VLSI circuits since users of mobile devices continue to demand:

- *mobility*: Nowadays consumers continue to request smaller and sleeker mobile devices. To support these features, high level silicon integration processes are required, which, in turn, determines high levels of leakage. Hence the need to find a strategy to reduce powr cinsumption and in particular leakage currents.
- *portability*: battery life is impacted by energy consumed per task; moreover a second order effect is that effective battery energy capacity is decreased by higher drawing current that causes IR drops in power supply voltage. To overcome this issue, more power/ground pins are required to reduce the resistance R and also thicker/wider on-chip metal wires or dedicated metal layers are necessary.
- *reliability*. Power dissipated as heat reduces speed and reliability since high power systems tend to run hot and the temperature can exacerbate several silicon failure mechanisms. More expensive packaging and cooling systems are indeed required.

1.2 Source of Power Dissipation

The sources of power dissipation in a CMOS device are expressed by the elements of the following equation:

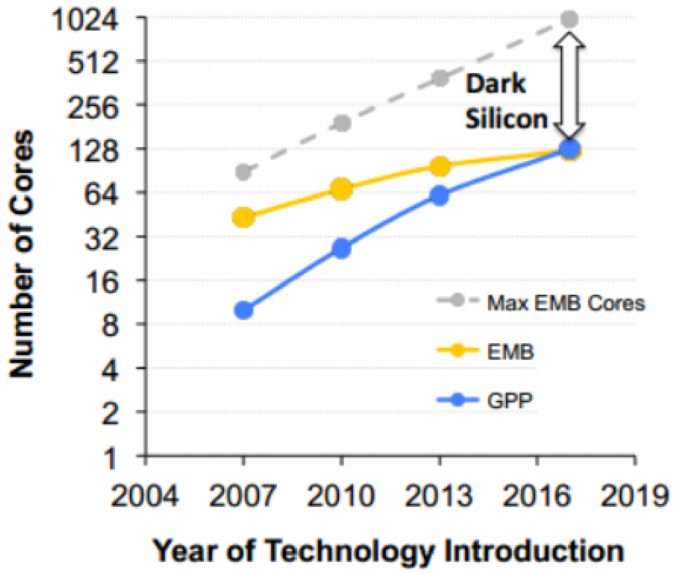


FIGURE 1.4: Dark silicon:end of multicore era. Source: Hardavellas et al., 2011

$$TotalPowerP = P_{dynamic} + P_{short-circuit} + P_{leakage} + P_{static}$$

In particular:

- *Dynamic power* or *switching power*:

$$P = \alpha C V^2 f$$

where P corresponds to the power, C is the effective switch capacitance which is a function of gate fan-out, wire length and transistor size, V is the supply voltage, f is the frequency of operation and α the switching activity factor. It corresponds to the power dissipated by charging/discharging the parasitic capacitors of each circuit node.

- $P_{short-circuit}$:

$$P = I_{short} V$$

which is produced by the direct path between supply rails during switching. In particular both pull-down (n-MOS) and pull-up (p-MOS) networks may be momentarily ON simultaneously leading to an impulse of short-circuit current.

- $P_{leakage}$:

$$P = I_{leakage} V$$

which is mainly determined by the fabrication technology. There are many sources that contribute to the leakage power, such as gate leakage, junction leakage, sub-threshold conduction. It occurs even when the system is idle or in standby mode. The leakage power, which is the dominant component in static energy consumption, was mainly neglected until 2005 when, in correspondence with the 65nm technology step, further scaling of threshold voltage has extremely increased the sub-threshold leakage currents (Fig. 1.5). It is expected that this kind of power will increase 32 times per device by 2020 [Roy and Prasad, 2009].

- P_{static}:

$$P = I_{static}V$$

which is determined by the constant current from V_{dd} the ground (standby power).

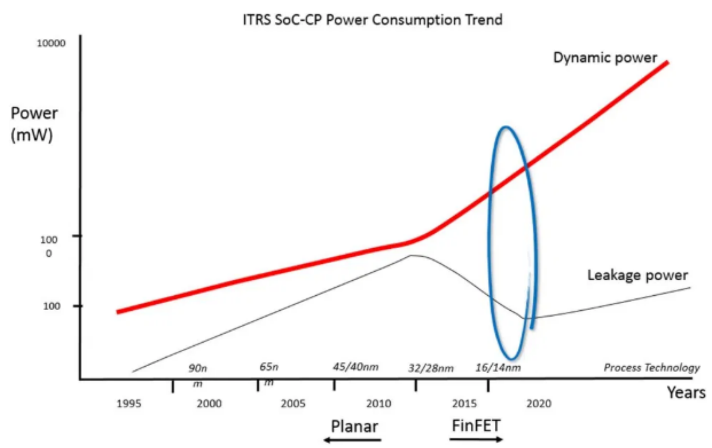


FIGURE 1.5: Power trends. Source: [Burns, 2016]

1.3 Approximate Computing

The problems caused by the sharp increase in power densities of SoC, due to the interruption of Dennard's downsizing law, prompted the search for new solutions. There are different strategies that can be applied for reducing power consumption at different levels of the VLSI design flow [Gupta and Padave, 2016, R., 2016] (Fig.1.7). These are listed below:

- *Operating System and Software Level*: approaches such as partitioning and compression algorithms. At Operating System level, power consumption can be reduced by designing an "'*energy-aware*"' scheduler (EAS). In this scenario, the scheduler takes decision relying on energy models of the resources. As an example in Linux kernel 5.4.0 the EAS, based on energy models of the CPU, is able to select an energy efficient CPU for each task [*Energy Aware Scheduling*] (Fig. 1.6).

  OS can also achieve energy efficiency by implementing *Dynamic Power Management* (DPM) of system resources allowing to reconfigure dynamically the system providing the requested services [*Low Power Principles*]. Another approach at system level can be code compression, which proposes basically to store programs in a compressed form and decompress them on-the-fly at execution time [Varadharajan and Nallasamy, 2017, Benini, Menichelli, and Olivieri, 2004]. An example of a simple code compression approach consists in the definition of a *dense* instruction set, characterized by a limited number of short instructions. This technique has been adopted by several commercial core processors such as ARM (Thumb ISA), MIPS and Xtensa.

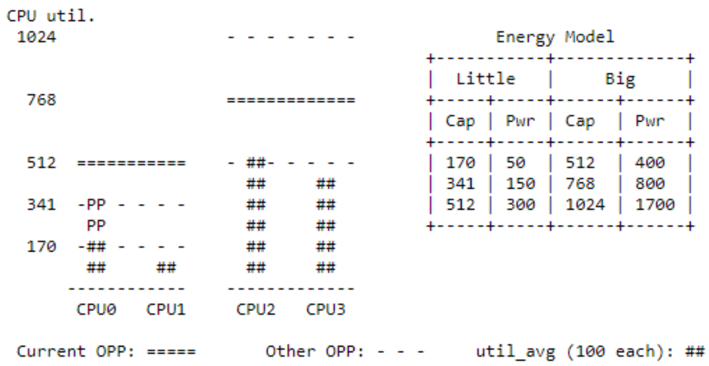


FIGURE 1.6: Power trends. Source: [*Energy Aware Scheduling*]

- *Architecture level*: techniques such as parallelism, pipelining, distributed processing and power management. In particular, parallelism and pipelining can optimize power consumption at the expense of area while maintaining the same throughput. By combining these two approaches it is possible to achieve a further reduction in power by aggressively reducing supply voltage.

- *Circuit/Logic level*: examples are voltage scaling, double edge triggering and transistor sizing. The latter in particular has a strong impact on circuit delay and power dissipation in combinational circuits. At logic level an example is represented by *adiabatic circuits* which use reversible logic to conserve energy. The implementation of these circuits is governed by two rules:

  1. a transistor must never be turned on when there is a potential voltage between drain and source;
  2. a transistor must never be turned off when current is flowing through it.

- *Technology Level*: techniques such as threshold reduction and multi-threshold (MT-CMOS) devices. The latter refers to the possibility of realizing transistors with different thresholds in a CMOS process, which allows to save energy up to 30% [Gupta and Padave, 2016]. Another technique is the usage of multiple supply voltages (Voltage Islands) that, by assigning different V_{dd} values to cells according to their timing criticality, allows to reduce both leakage and dynamic power. In particular, the basic idea is to group different supply voltages in a reduced number of voltage islands (each one with a single V_{dd}) avoiding complex power supply system and a large amount of level shifters.

In the past, across all low power approaches described above, computing platforms have always been designed following the principle of deterministic accuracy, for which every computational operation must be implemented deterministically and with full precision (exact computation). However, continuing to include deterministic accuracy in computation at all stages seems to be outperforming and dos not allow to solve the upcoming energy efficiency challenges; this stimulated the exploration of new directions in the design of modern digital systems.

Approximate Computing (AxC) is an emerging paradigm which proposes to reduce power consumption by relaxing the specifications on fully precise or completely deterministic operations. This is due to the fact that many applications, known as ETA (Error Tolerant Applications), are intrinsically resilient to errors and can produce outputs with a slightly shift in accuracy and a significant reduction in

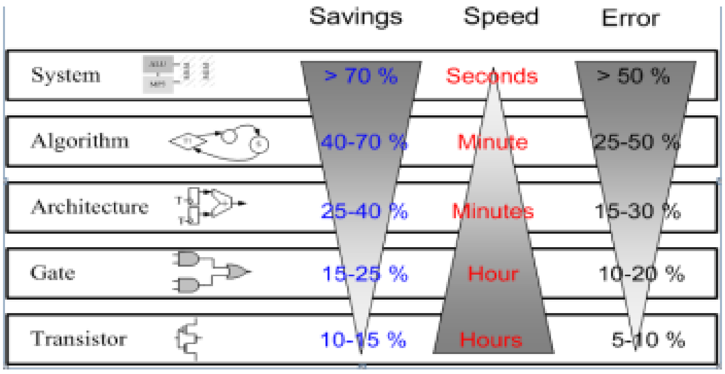


FIGURE 1.7: Low power strategies at different abstraction levels. Source: [Gupta and Padave, 2016]

computations. AxC therefore exploits the gap between the effective quality level required by applications and the one provided by the computer system in order to save energy.

Although it is an extremely promising approach, Approximate Computing is not a panacea. It is right to underline the need to accurately select where to apply AxC techniques in code and data portions. Suffice it to consider that a uncontrolled use of approximate circuits can lead to intolerable quality losses or applying approximate techniques in program control flow can lead to catastrophic events such as process or systems crashes. Finally, it is necessary to evaluate the impact of the approimate computation on output quality, in order to assess that the quality specifications are still met.

1.4 Contribution and thesis organization

The aim of this work is mainly to explore design and programming techniques for low-power microprocessor architectures based on Approximate Computing. Chapter 2 illustrates the promises and challenges of AxC, focusing on the key concepts (Section 2.1), on the motivations that led to the development of this paradigm and on quality metrics. Then it is presented a survey of Approximate Computing techniques, from programming languages (how to automatically find approximable code/-data and support of programming languages for approximate computing, Section 2.2.1), approximate ISA (Section 2.2.2), design automation methods developed for the approximation of a class of problems (*Functional Approximate Computing*, Section 2.2.6), design of approximate devices and components (*Approximate Computing ad hoc*: approximate memories, approximate adders, approximate multipliers. Section 2.2.5). The key concepts of AxC are then compared to those of another emerging paradigm: the *Transprecision Computation*, born in the context of *OPRECOMP*, a 4-year research project funded under the EU Horizon 2020 framework. Finally, the adoption of approximate computing strategies for machine learning algorithms and architecture is illustrated.

Particular emphasis in this work has been placed on approximate memories (SRAM, DRAM, eDRAM), considering that it is expected that the power consumption of memories will constitute more than 50% of the entire system power. The scope of designing an architecture with approximate memories is to reduce power consumption for storing part of the data, by managing critical data and non-critical (approximate) data separately; with this partitioning it is possible to save energy by relaxing design constraints and allowing errors on non-critical data. One of the final goals of this thesis is to provide the required tools to evaluate the impact of different levels of approximation on the target application, considering that quality output is not only impacted by the fault rate but it also depends on the application, its implementation and its representation of the data in memory.

The work starts from the software layer, implementing the support for Approximate Memories (AxM) in Linux Kernel (version 4.3 and 4.20) and allowing the OS to distinguish between exact memory and approximate memory. In particular Chapter 3 provides a description of the physical memory management in Linux OS kernel (Section 3.2.3); then the development of the extension to support approximate memory allocation is described (Section 3.3). This extension has been implemented and tested for three different of architectures:

- *Intel x86*, as a target in the group of High Performance Computing;
- *ARM*, as a target in the group of Embedded Systems architectures;
- *RISC-V*, as the emerging instruction set which is rising interest in the research and industrial communities.

In order to complete this step, it has been also necessary to implement a user-space library to allow applications to easily allocate run-time critical and non critical data in different memory areas. Finally, results are illustrated, in the form of allocation statistic messages provided by the kernel, discussing the characteristics of the implementation.

The next step of the work is represented by the development of *AppropinQuo*, a hardware emulator to reproduce the behavior of a system platforms with approximate memory. This part is illustrated in Chapter 4, the core of this work has been the implementation of models of the effects of approximate memory circuits and architectures, that depend on the internal structure and organization of the cells. The ability to emulate a complete platform, including CPU, peripherals and hardware-software interactions, is particularly important since it allows to execute the application as on the real board, reproducing the effects of errors on output. In fact, output quality is related not only to error rate but it also depends on the application, implementation and its data representation. The level of approximation instead is determined by key parameters such as the error rate and the number of bits that can be affected by errors (*looseness level*).

After the implementation of AxM support at OS level and of the emulator AppropinQuo, Chapter 5 describes the analysis and the implementation of different error tolerant applications modified for allocating non critical data structures in approximate memory. For these applications the impact of different levels of approximation on the output quality has been studied. The first application is a h264 encoder (Section 5.2). This work started by profiling memory usage and finding a strategy for selecting error tolerant data buffers. Then the modified application was run on the AppropinQuo emulator, for several combination of fault rates and fault masking at bit level (*looseness level*). The obtained results, showing the importance of exploring the relation between these parameters and output quality, are then analyzed. The second application is a digital filter (Section 5.3). In particular a 100-taps FIR filter program (configured as low-pass filter), working on audio signal, is implemented. The latter has been implemented to allocate tolerant data (internal tap registers and input and output buffers) on approximate memory while the FIR coefficients are kept exact. Results are provided in terms of SNR.

Eventually, Chapter 6, in the context of Transprecision Computing, describes the implementation of a fully combinational and reconfigurable Floating Point Unit, which can perform arithmetic operations on FP numbers with arbitrary length and structure (mantissa and exponent) up to 64 bits, included all formats defined by the IEEE-754 standard. In particular, the analysis of the resutls has been focused on reduced precisions formats, between 16 bit (*IEEE half-precision*) and 32 bit (*IEEE single-precision*), evaluating the impact on performance, reuired area, power consumption and propagation speed.

Chapter 2

Approximate Computing: State of the Art

2.1 Approximate Computing: main concepts

Energy savings has become a first-class constrain in the design of digital systems. Currently most of the digital computations are performed on mobile devices such as smart-phones or on large data centers (for example cloud computing systems), which are both sensitive to the topic of energy consumption. Concerning only the US data centers it is expected that the electricity consumption will increase from 61 millions kW in 2006 to 140 kW in 2020. In the era of nano-scaling, the conflict between increasing performance demands and energy savings represents a significant design challenge.

In this context Approximate Computing [Han and Orshansky, 2013] has become a viable approach to energy efficient design of modern digital systems.

Such an approach relies on the fact that many applications, known as Error Tolerant Applications (see section 2.2.4), can tolerate a certain degree of approximation in the output data without affecting the quality of results perceived by the user. The solutions offered by this paradigm are also supported by technology scaling factors, due to the growing statistical variability in process parameters which makes traditional design methodologies inefficient [Mastrandrea, Menichelli, and Olivieri, 2011].

Approximate Computing can be implemented through different paradigms as:

1. *Algorithmic level and Programming level Approximate Computing* [Palem, 2003], subsection: 2.2.1;

2. *Instruction level Approximate Computing* [Gupta et al., 2013; Venkataramani et al., 2012; Esmaeilzadeh et al., 2012], subsection: 2.2.2;

3. *Data-level Approximate Computing* [Frustaci et al., 2015b; Frustaci et al., 2016], section:2.2.3;

In literature several approximation methodologies are described even if two scenarios are dominating: approximate computing *ad hoc* and design automation (or *functional*) approximate computing. These approaches will be discussed in the following sections (2.2.5 and 2.2.6).

2.2 Strategies for Approximate Computing

2.2.1 Algorithmic and programming language Approximate Computing

Discovering data and operations that can be approximated is a crucial issue in every AC technique: in several cases it can turn out to be intuitive, as for example lower-order bits of signal processing applications but in other more complex cases it can require a deep dive analysis of application characteristics. Several works and techniques have been implemented in order to address this problem. In [Roy et al., 2014] the authors propose ASAC (*Automatic Sensitivity Analysis for Approximate Computing*), a framework which uses statistical methods to find relaxable data of applications in an automatic manner. The proposed approach, as shown in Fig. 2.1, consists in the following steps:

- collect application variables and the range of values that these ones can assume;
- use binary instrumentation to perturb variables and compute the new output;
- compare the new output to the exact one in order to measure the contribution of each variable;
- mark a variable as approximable or not according to the results of the previous step.

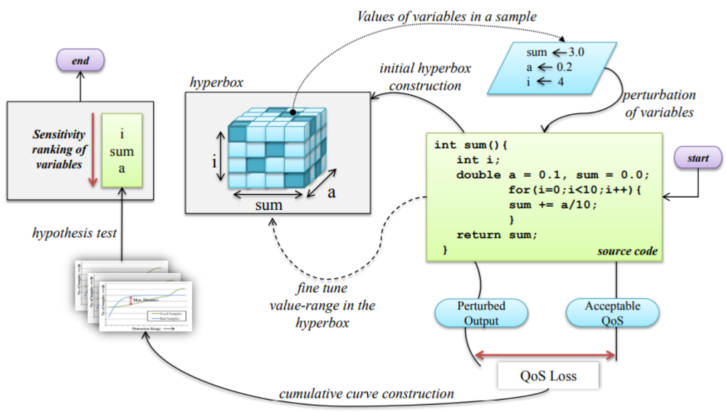


FIGURE 2.1: Overview of ASAC framework. Source: Roy et al., 2014

The benefits of this approach is that the programmer is not involved in the process of annotating relaxable program portions. However, a drawback is that a variable can be marked as approximable even if it is not, leading to errors.

In [Chippa et al., 2013] the ARC (*Application Resilience Characterization*) framework is described (Fig. 2.2). In this work, the authors firstly identify the application parts that could be potentially resilient, and so that could be approximated, and the sensitive parts; then they use approximation models, abstracting several AxC techniques, to characterize the resilient parts. In particular the proposed framework consists in the following steps:

- *resilience identification step*: identification of atomic kernels as innermost loop in which the application spends more than the 1% of its execution time. Random errors are injected in the kernel output variables, using Valgrind DBI tool. The kernel is considered *sensitive* if the output does not match quality constraints, otherwise it could be *resilient*.

- *resilience characterization step*: errors are injected in the kernel using Valgrind in order to quantify the kernel resilience. In particular, using specific models such as loop perforation, inexact arithmetic circuits and so on, a quality profile of the application is produced.

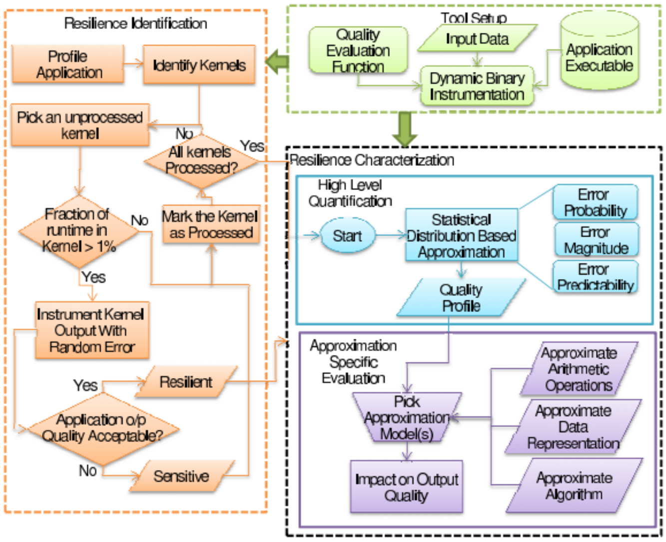


FIGURE 2.2: Overview of ARC framework. Source: Chippa et al., 2013

The distinction between non approximable portions of a program (critical data) and approximable ones can be used to introduce AxC support at programming language. In [Sampson et al., 2011] the authors present *EnerJ*, a Java extension which adds type qualifiers for approximate data types: *@Approx* for non critical data and *@Precise* for critical data (Fig. 2.3). By default every variable or construct is of type @Precise, so it is necessary to specify only approximate data types. An example is provided in the code section below.

```
@Approximable class FloatSet {
@Context float[] nums = ...;
float mean() {
float total = 0.0f;
for (int i = 0; i < nums.length; ++i)
  total += nums[i];
  return total / nums.length;
}
@Approx float mean APPROX() {
@Approx float total = 0.0f;
 for (int i = 0; i < nums.length; i += 2)
    total += nums[i];
return 2 * total / nums.length;
  }
}
```

This approach indeed, with respect to previous works that use annotations on blocks of code, allows programmers to explicitly specify the data flow from approximate data to precise ones, ensuring the construction of a safe programming model. The latter, in fact, is safe in that it always guarantees data correctness, unless an approximate annotation has been provided by the programmer. Moreover this approach eliminates the need for dynamic checking, reducing the runtime overhead and improving the energy savings.

Construct	Purpose
@Approx, @Precise, @Top	Type annotations: qualify any type in the program. (Default is @Precise.)
endorse(e)	Cast an approximate value to its precise equivalent.
@Approximable	Class annotation: allow a class to have both precise and approximate instances.
@Context	Type annotation: in approximable class definitions, the precision of the type depends on the precision of the enclosing object.
_APPROX	Method naming convention: this implementation of the method may be invoked when the receiver has approximate type.

FIGURE 2.3: Overview of EnerJ language extension. Source: Sampson et al., 2011

2.2.2 Instruction level Approximate Computing

Concerning the second issue, in [Esmaeilzadeh et al., 2012] the authors describe the requirements that an approximation-aware ISA should exhibit to support approximable code at programming language:

- AxC should be applied with **instruction granularity**, allowing to distinguish between exact (precise) instructions and approximate ones. For example, control flow variables like a loop increment one must be exact while the computation inside the loop could be approximated;
- for AxM support, the compiler should instruct the ISA to store data in exact or approximate memory banks;
- ISA should be flexible to transitioning data between approximate and exact storage;
- Approximation should be applied only where it is requested by the compiler, exact instructions indeed must respect traditional semantic rules;
- Approximation should be applied only to specific areas (for example memory addressing and indexing computation must be always exact).

According to these specifications, the authors produce approximate instructions and add them to the original Alpha ISA. As shown in Fig. 2.4 , the ISA extension contains approximate versions of integer load /store, floating-point arithmetic and load /store, integer arithmetic, bit-wise operation instructions. The format of approximate instructions is the same as that provided by the original (and exact) ISA, but the new ones cannot give any guarantees concerning the output values. In the paper a micro-architecture design, *Truffle*, supporting the proposed ISA extension is also described.

2.2.3 Data-level Approximate Computing

While algorithmic level and, less incisively, instruction level approximate computing are more aggressive and challenging from a technical point of view, data-level approaches can be of major relevance because most applications that are suitable for approximate computing utilize large amounts of dynamically allocated data memory (e.g. multi-media processing Olivieri, Mancuso, and Riedel, 2007; Bellotti et al., 2009]) and it is generally agreed that memory devices accounts by far for the largest

Group	Approximate Instruction
Integer load/store	LDx.a, STx.a
Integer arithmetic	ADD.a, CMPEQ.a, CMPLT.a, CMPLE.a, MUL.a, SUB.a
Logical and shift	AND.a, NAND.a, OR.a, XNOR.a NOR.a, XOR, CMOV.a, SLL.a, SRA.a, SRL.a
Floating point load/store	LDF.a, STF.a
Floating point operation	ADDF.a, CMPF.x, DIVF.a, MULF.a, SQRTF.a, SUBF.a, MOV.a, CMOV.a, MOVFI.a, MOVIF.a

FIGURE 2.4: ISA extension for AxC support. Source: Esmaeilzadeh et al., 2012

parts of static power consumption in modern ICs [Frustaci et al., 2016; Pilo et al., 2013]. Approximate memories are an example of Approximate computing at data-level (2.3). In order to open the way for reliable data-level approximate application development, it is required an evaluation environment to simulate the actual output of the application when processing true inaccurate data. Inaccuracy can occur at different degrees in different memory segments, can be caused by single random events or can be correlated to read and write accesses, can be differently distributed in the bytes of the memory word. The main concepts dealing with Approximate Memories are described in details in section 2.3.

2.2.4 ETAs - Error Tolerant Applications

Error Tolerant Applications (ETAs), Fig. 2.5, are defined as applications that can accept a certain amount of errors during computation without impacting the validity of their output. There may be several factors, as shown in Fig. 2.6, for which an application can be tolerant to errors:

- *perceptive limitations*: applications interacting with human senses do not need fully precise computation, they can accept imprecisions in their results due to human brain ability of "filling in" missing information. As an example, we can consider phone lines: these ones are not able to carry sound perfectly, nevertheless the transmitted information is sufficient for human perception.
- *redundant input data*;
- *noisy inputs*, for example sensors;
- *probabilistic estimates* as outputs, for example machine learning applications.

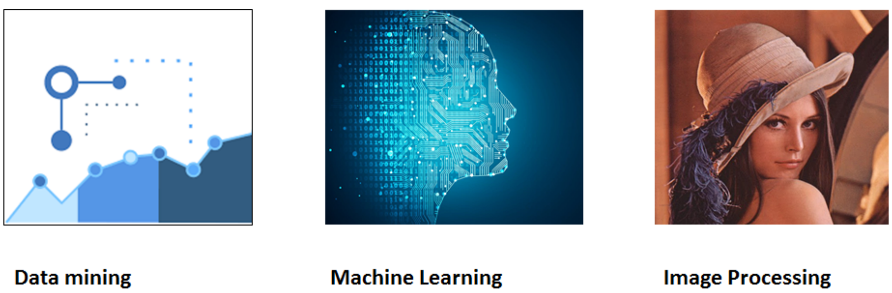


FIGURE 2.5: Examples of ETAs

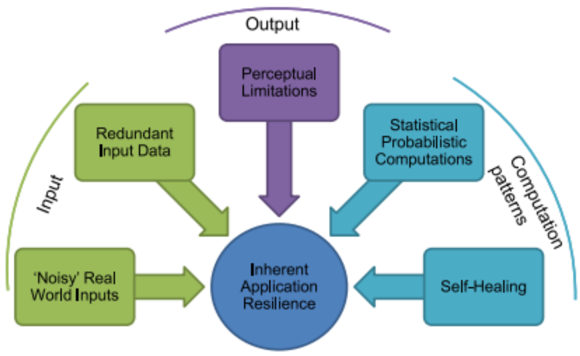


FIGURE 2.6: Possible sources of application error resilience .Source: Chippa et al., 2013

Multimedia applications are an important class of ETAs: they process data that are already affected by errors/noise and, as said before due to limitations of human senses, can introduce approximations on their outputs (e.g. lossy compression algorithms). Moreover, they tend to require large amounts of memory for storing their buffers and data structures. Another example of ETAs is represented by closed loop control applications; again, they process data that are affected by noise (sensor reading) and produce outputs to actuators affected by physical tolerances and inaccuracies.

2.2.5 Approximate Computing *ad hoc*

This approach consists in employing specific (*ad hoc*) methods for the approximation of a system or of a specific component. In this case a lot of knowledge of the system or the component in question is required; moreover the adopted approximation method can rarely be applied to another system or component. Examples of *ad hoc* AxC include arithmetic circuits such as approximate adders 2.2.5 and approximate multipliers [Masadeh, Hasan, and Tahar, 2018], approximate memories 2.3, approximate dividers [Chen et al., 2016], pipeline circuits [] etc.

Approximate Adders

In [Gupta et al., 2011] *IMPACT*, IMPrecise adders for low-power Approximate ComPuTing, are described. The authors propose three different implementations of approximate FA cells (Fig.:2.7) by simplifying the Mirror Adder (MA) circuit and ensuring minimal errors in the FA truth table. The proposed approximate units not only have few transistors, but also the internal node capacitances are much reduced. Simplifying the architecture allows reducing power consumption in two different ways:

1. smaller transistor count leads to an inherent reduction in switched capacitances and leakage;

2. complexity reduction results into shorter critical paths, facilitating voltage scaling without any timing-induced errors.

These simplified FA cells can be used to implement several approximate multibit adders, as building blocks of DSP systems. In order to obtain a reasonable output quality, the approximate FA cells are used only in the LSBs while accurate FA cells

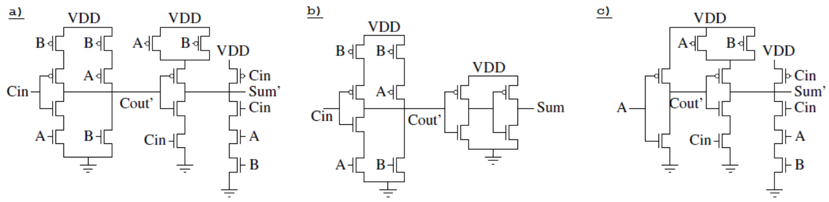


FIGURE 2.7: a) simplified MA, b) approximation 1, c) approximation 2. Source:Gupta et al., 2011

are used for the MSBs. In general, the proposed approach can be employed to any adder structure that use FA cells as basic building block as, for example, the approximate tree multipliers, which are extensively used in DSP systems. In order to evaluate the efficacy of the proposed approach, the authors use these imprecise units to design architectures for video and image compression algorithms. The results show that it is possible to obtain power savings of up to 60% and area savings of up to 37% with an insignificant loss in output quality, when compared to the actual implementations.

In [Yang et al., 2013] the authors describe the design of three XOR/XNOR-based approximated adders, (*AXAs*), comparing them to an exact full adder in terms of energy consumption, delay, area and PDP. Fig. 2.8 shows an accurate full adder based on 4T XNOR gates and composed by 10 transistors; from the schematic of this accurate adder, applying a transistor reduction procedure, the AXAs adders are designed. The first AXA (Fig. 2.9) is composed by 8 transistors; in this design the XOR is implemented using an inverter and two pass transistors connected to the input signals (X and Y). The output *Sum* and the carry *Cout* are correct for 4 of the total 8 input combinations. Fig. 2.10 and Fig. 2.11 show respectively AXA2 and AXA3 implementations. The former is composed by a 4 transistor XNOR gate and a pass transistor, for a total of 6 transistors; the output *Sum* is correct for 4 of the total 8 input combinations while the *Cout* is correct for all input combinations. AXA3 has 8 transistors in total, adding two transistor in a pass transistor configuration in order to improve the accuracy of the output *Sum*. In this way, *Sum* output is exact for 6 input combinations while ,*Cout* is correct for all input combinations. The comparison between the three approximate adders and the exact one are illustrated in Fig. 2.12. Summarizing, AXA1 yields the best performance having the shortest delay, AXA2 shows the best results in terms of area while AXA3 is the most power efficient design.

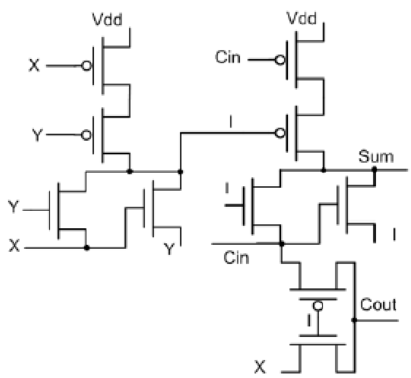


FIGURE 2.8: Design of exact full adder, 10 transistors. Source: Yang et al., 2013

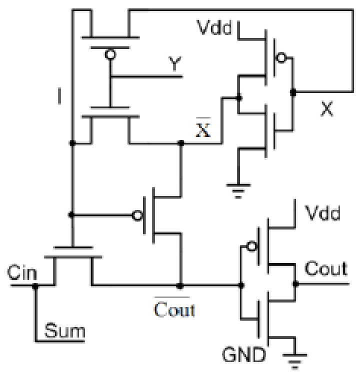


FIGURE 2.9: Design of AXA1, 8 transistors. Source: Yang et al., 2013

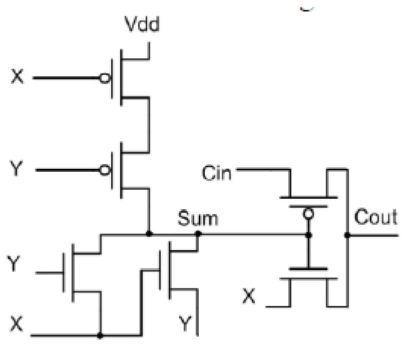


FIGURE 2.10: Design of AXA2, 6 transistors. Source: Yang et al., 2013

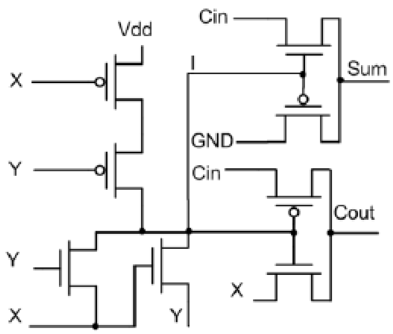


FIGURE 2.11: Design of AXA3, 8 transistors .Source: Yang et al., 2013

Approximate Multipliers

In [Masadeh, Hasan, and Tahar, 2018] the authors propose a methodology to design and evaluate the accuracy and the circuit characteristics of different approximate multipliers, allowing to select the most suitable circuit for a specific application. The design of these multipliers is based essentially on three different decisions:

- the type of approximate Full Adder employed to build the approximate multiplier;
- the architecture of the multiplier (e.g. array or tree);

Metric	AXA1	AXA2	AXA3	ACA [7]	Improvement (%) for AXA1, AXA2, AXA3		
Transistor Count	8	6	8	10	20.00	40.00	20.00
Static power (nW)	72.82	19.33	30.77	56.02	-29.99	65.45	45.07
Dynamic power (uW)	3.872	3.448	3.234	4.657	15.22	25.07	30.57
Delay for Sum (ps)	0	20.16	61.82	35.98	100.0	43.96	-71.8
Delay for C_{out} (ps)	60.7	254.9	159.7	253.9	76.09	-0.39	37.09

FIGURE 2.12: Comparison between AXA1, AXA2, AXA3 and exact full adder .Source: Yang et al., 2013

- how approximate and exact sub-modules are placed in the target multiplier module.

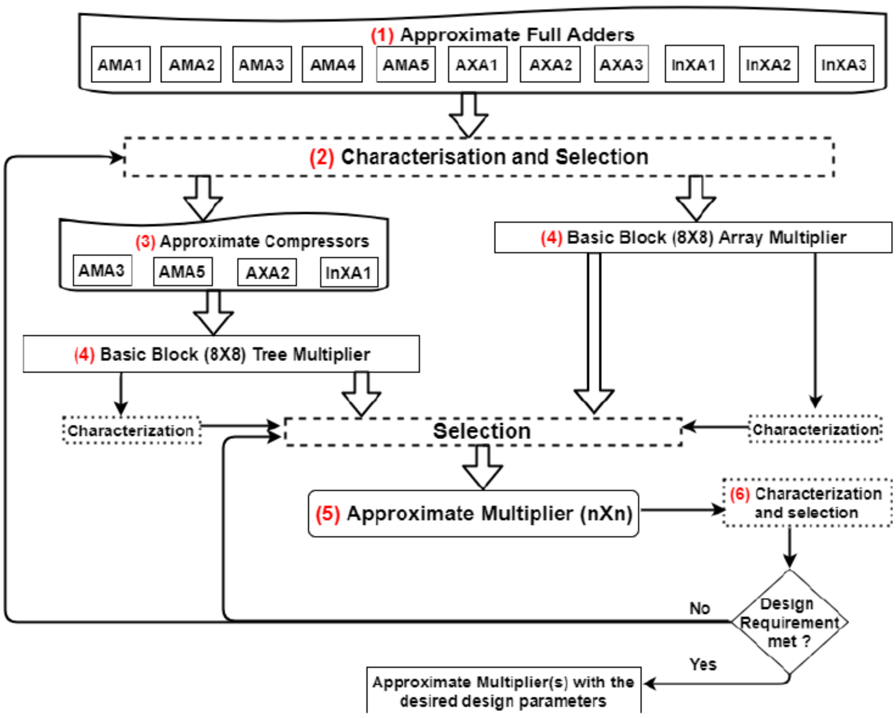


FIGURE 2.13: Overview of approximate multipliers comparison. Source: Masadeh, Hasan, and Tahar, 2018

The methodology is illustrated in Fig. 2.13 and it is composed by the following steps:

1. Building a library of approximate FAs: the authors consider five approximate mirror adders (AMA1, AMA2, AMA3, AMA4 and AMA5), three approximate XOR/XNOR based full adders (AXA1, AXA2 and AXA3), three inexact adder cells (InXA1, InXA2 and InXA3);

2. Characterization and early space reduction: the different approximate FAs are characterized in terms of power, area, latency and quality (Fig.: 2.14);

FA Type	Size (A)	Power(nw) (P)	Delay(ps) (D)	# of Error Cases (E)	PDP(fJ)
Exact FA	28	763.3	244	0	186.25
AMA1 (M1)	20	612	195	2	119.34
AMA2 (M2)	14	561.1	366	2	205.36
AMA3 (M3)	11	558.1	360	3	200.92
AMA4 (M4)	15	587.1	196	3	115.07
AMA5 (m5)	8	412.1	150	4	61.82
AXA1 (X1)	8	676.2	1155	4	781
AXA2 (X2)	6	358.7	838	4	300.59
AXA3 (X3)	8	396.5	1467	2	582
InXA1 (In1)	6	410	740	2	303.4
InXA2 (In2)	8	355.1	832	2	295.44
InXA3 (In3)	6	648	767	2	753.5

FIGURE 2.14: Overview of different approximate FA properties. Source: Masadeh, Hasan, and Tahar, 2018

3. Building a library of approximate compressors;
4. Building approximate multipliers basic blocks: approximate FAs and compressors are used to design respectively *8x8 array* and *tree* based multipliers. These will constitute the basic blocks for designing higher-order multipliers (e.g 16x16);
5. Designing target approximate multipliers: the basic modules described above are employed to build higher-order multipliers;
6. Selection of design corners: a subset of sample points is selected considering the quality requirements of the given application.

An image blending application is used by the authors to evaluate in MATLAB the proposed multiplier designs in terms of SNR and PDP. The results are then compared to 24 different designs reported in [Jiang et al., 2016]. The proposed multipliers shows a better PDP reduction.

Algorithmic Noise Tolerance and Reduced Precision Redundancy techniques

Algorithmic Noise-Tolerance (ANT) is an architectural level AxC technique based on the scaling of supply voltage below the critical value V_{dd} (Voltage Over-Scaling); it can be applied both to arithmetic and DSP circuits, allowing them to operate with a scaled V_{dd} without reducing the original throughput of the system [Hegde and Shanbhag, 1999; Shim and Shambhag, 2003]. According to this technique, the original circuit, called *Main DSP* (MDSP), is coupled with an Error Control (EC) block, in charge of limiting the impact of timing-errors introduced by the lowered supply voltage V_{dd}. In this scenario, even if the *Main DSP* operates with a reduced supply voltage, only a low percentage of the total paths does not satisfy the timing constraints.

Reduced Precision Redundancy (RPR) [Shim, Sridhara, and Shanbhag, 2004] is a particular instance of ANT approach, according to which the Error Control block contains a reduced-bitwidth replica of the MDSP (Fig. 2.15). Several works in literature propose effective implementations of RPR, in which the replica is usually designed using ad-hoc procedures and error impact analysis is carried out statistically.

The latter implies that very simplified assumptions are made on data distribution, and important aspects as input temporal correlation are not considered.

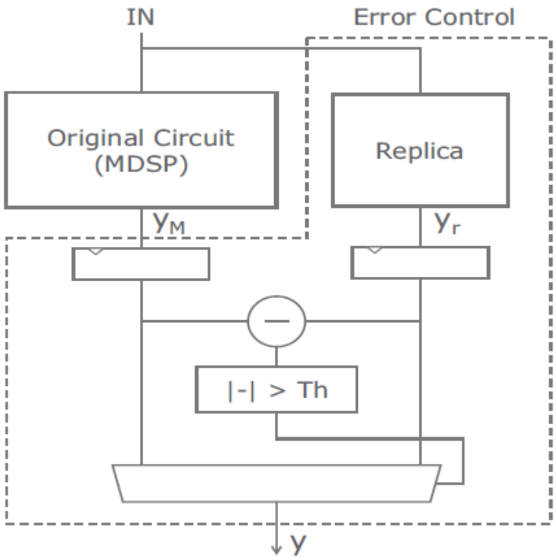


FIGURE 2.15: Reduced Precision Redundancy ANT Block Diagram.
Source: Pagliari et al., 2015

In [Pagliari et al., 2015], the authors propose a new generalized approach to RPR-ANT, by building a design tool able to automatically add RPR architectures to existing gate-level netlists of fixed-point arithmetic and DSP circuits. The proposed framework uses accurate circuit models of real standard-cell libraries improving accuracy of energy saving and timing degradation estimation, further it takes into account input dependencies, considering temporal correlation and non-trivial distributions. Moreover the tool is application-agnostic, meaning that it allows to apply the RPR technique to circuits that had never been considered before; despite this, according to authors it can still reach comparable results with respect to ad-hoc approaches.

2.2.6 Design automation Approximate Computing or *functional approximation*

This second approach proposes to implement an approximated function instead of the original, reducing key parameters such as power consumption and providing acceptable quality (acceptable errors). With respect to the previous approach, functional approximation provides a procedure that can be applied to all problem instances of a given category. All approximation methods based on this approach have to address two crucial aspects:

- Determine how the approximated function can be obtained from the original one. Concerning this issue, generally a heuristic procedure is applied to the original and exact function (hardware or software); the procedure is then repeated iteratively in order to improve the approximated function at each iteration, verifying at the same time that the new (approximated) implementation still satisfies functional and non-functional requirements.

- Determine how to assess the quality of the candidate approximated function (*relaxed equivalence checking*). To address this problem usually a TS (Training

Set) is applied to the approximated function and the corresponding error is measured. In any case, this approach can be applied with increased difficulty when the function to be approximated is complex and only a very small error can be accepted, due to the fact that the TS need to be too large. For complex systems it is necessary to calculate the distance from the exact implementation by defining suitable metrics and checking that the approximated implementation is equivalent within some bounds, with respect to the chosen metric. To address this aspect, in [Holık et al., 2016] the authors implement relaxed equivalence checking algorithms, based on formal verification. In [Vasicek and Sekanina, 2016] the authors illustrate a circuit approximation method in which the error is expressed in terms of Hamming distance between the output produced by the approximated circuit and the output produced by the exact one. In particular, to perform equivalence checking of two combinatorial circuits, determining the Hamming distance between the truth table of both circuits, the author propose to use an auxiliary circuit, as shown in Fig 2.16.

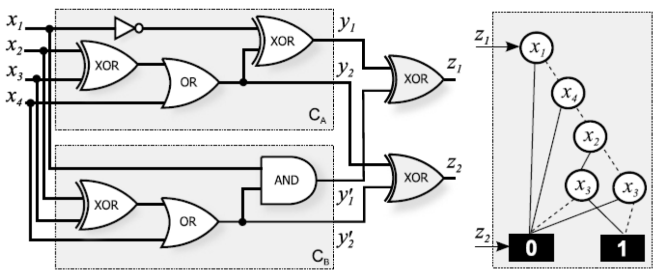


FIGURE 2.16: Determining the Hamming distance of two combinational circuits using a Binary Decision Diagrams (BDD). Source : Vasicek and Sekanina, 2016

Examples of functional approximation are ABACUS [Nepal et al., 2014], SALSA [Venkataramani et al., 2012], SASIMI [Venkataramani, Roy, and Raghunathan, 2013].

ABACUS

In [Nepal et al., 2014] the authors present ABACUS, a methodology for automatically generating approximate designs starting directly from their behavioral register transfer level (RTL) descriptions, with the idea of obtaining a wider range of possible approximations. This methodology involves, at first, the creation of an abstract syntax tree (AST) from the behavioral RTL description of a circuit; then, in order to create acceptable approximate designs, ABACUS applies variant operators to the generated AST. These operators include simplifications of data types, approximations of arithmetic operations, variable to constant substitutions, loop transformations and transformations of arithmetic expressions. The transformations made by ABACUS are not limited to a particular circuit but are global in nature since the behavioral RTL descriptions capture the algorithmic structure of the circuits. Consequently ABACUS can be applied to arbitrary circuits without needing to know the application domain. The authors integrate ABACUS in a traditional ASIC / FPGA design flows, as shown in Fig. 2.17.

As said before the behavioral register transfer level (RTL) code (Design Files) represents the input of ABACUS which produces several approximate code variants, which are then synthesized at RTL/behavioral level. A simulator is used to

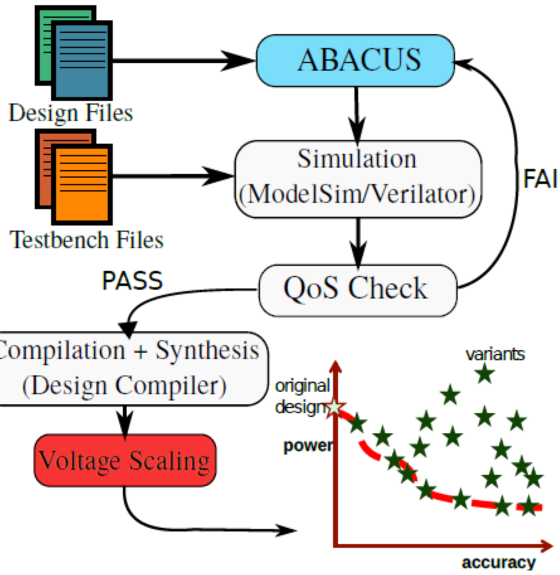


FIGURE 2.17: Integration of ABACUS in a traditional design flow.
Source : Nepal et al., 2014

verify the functional accuracy of each approximate variant. Finally, each variant is compiled and synthesized only if it passes the quality specifications check (QoS check). The produced design is compared with the original one in terms of timing: if there is a further gain in timing slack in the approximate design (since the critical path is reduced), voltage scaling is applied until the slack gain is eliminated. In this way, it is possible to save further power without impacting the design accuracy.

The proposed methodology has been evaluated on four realistic benchmarks from three different domains (machine learning, signal processing, computer vision). The results show that the approximated design variants generated by ABACUS allow to save power up to 40%.

2.2.7 Approximate Computing metrics

Performance metrics

Several metrics have been introduced to assess the reliability of approximate circuits and quantify errors in approximate designs:

- **Error rate (ER)** or error frequency: it is defined as the fraction of incorrect outputs out of a total number of inputs in an approximate circuit Breuer, 2004.
- **Error significance (ES)**: it corresponds to the degree of error severity due to the approximate operation of a circuit:
  - the numerical deviation of an incorrect output from a correct one;
  - the Hamming distance of the vectors;
  - the maximum error magnitude of circuit outputs.

Quality metrics

In order to compare output from approximate computation with the one obtained from exact computation, trading-off quality loss and energy savings, quality metrics for AxC are defined. The choice of the error metric to be adopted is application-specific and even within a single application, different metrics can be applied. These metrics indeed are not mutually exclusive and can be used together to evaluate applications. Examples of these quality metrics are:

- Relative difference/error from standard output;
- *MSE (Mean Squared Error)*: it measures the average squared difference between the estimated values and the actual value. It is defined as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(X_i - Y_i)^n$$

  where n represents the number of data points; Xi the observed values and Yi the predicted values. For model quality it represents the difference between the individual's original fitness function and the output of the approximate model.

- *Pixel difference*: it is used to calculate the distortion between two images on the basis of their pixelwise differences. Examples of applications to which this quality metric can be applied are: particle filter (Rodinia), volume rendering, Gaussian smoothing, mean filter, dynamic range compression, edge detection, raster image manipulation etc.
- *PSNR (Peak Signal to Noise Ratio)*: it represents the ratio between a signal's maximum power and the power of the signal's noise. Because many signals have a very wide dynamic range, PSNR is generally expressed in terms of the logarithmic decibel scale. Assuming to have a noise-free image X, the PSNR is defined as:

$$PSNR = 10\log_{10}\frac{MAX_I}{MSE(X,Y)}$$

  where X represents the reference image, Y the test image and MAX_I is the maximum signal value that exists, the maximum possible pixel value of the image. For example if the pixels are represented using 8bits per sample, MAX is 255.

- *PSNRB*: it is used for measuring the quality of images which consists of blocking artifacts. This metric includes the PSNR and BEF (Blocking Effect Factor) which measures the blockiness of images.
- *SNR (Signal to Noise Ratio)*: this metric is computed as the ratio of the power of a signal to the power of background noise and it is usually expressed in decibels.

$$SNRdB = 10\log_{10}\frac{P}{N}$$

  where P represents the signal power and N the noise power.

- *SSIM (Structural Similarity Index)*: this metric is used to measure quality by capturing the similarity between two images. SSIM measures and computes the product between three aspects of similarity: Luminance, contrast and structure.
- Classification/clustering accuracy;
- Correct/incorrect decisions;

For many applications several quality metrics can be used to evaluate the quality loss introduced by the approximation, as an example for k-means clustering applications both clustering accuracy and mean centroid distance can be employed as metrics.

2.3 Approximate Memories

In modern digital sstems memory represents a significant contribution to overall system power consumption [Liu et al., 2012a]. This is mainly going in parallel with the increasing performance of computing platforms, which put under stress memory bandwidth and capacity. Applications as deep learning, high definition multimedia, 3D graphic, contribute to the demand for such systems, with added constraints on energy consumption.

Techniques for reducing energy consumption in SRAM and DRAM memories have been proposed in many works. A class of very promising approaches, generally called approximate memory techniques, is based on allowing controlled occurrence of errors in memory cells [Weis et al., 2018].

Approximate memories are memory circuits where cells are subject to hardware errors (bit flips) with controlled probability. In a wide sense the behavior of these circuits is not different from standard memories, since both are affected by errors. However, in approximate memories the occurrence of errors is allowed by design and traded off to reduce power consumption. The important difference between approximate and standard memory reside in the order of magnitude of error rate and its consequences: in approximate memories errors become frequent and are not negligible to software applications.

2.3.1 Approximate Memory Circuits and Architectures

Approximate memory circuits have been proposed in research papers since some years. By relaxing requirements on data retention and faults, many circuits and architectures have demonstrated to significantly reduce power consumption, for both SRAM and DRAM technologies.

Approximate SRAM

SRAM approximate memories are designed with aggressive supply voltage scaling. In SRAM bitcells, read and write errors are caused by low read margin (RM) and write margin (WM). Since process variations affect RM and WM in opposite directions, the corner defines which is the critical margin (i.e. the slow–fast (SF) corner makes the bitcell write critical, the fast–slow corner makes it read critical). Under voltage scaling, WM and RM are degraded, increasing read and write BERs. The degradation is in general abrupt (BER increases exponentially at lower voltages), but techniques have been proposed to make such degradation graceful.

In [Frustaci et al., 2015b; Frustaci et al., 2016] the authors propose, in the context of ETAs, approximate SRAM circuits where voltage scaling is applied to bit cells, saving energy at the expanse of read/write errors. ETAs, due to their nature, can tolerate a more aggressive voltage scaling and so a greater BER with respect to error free applications. Anyway, when the supply voltage of the SRAM circuit scales down, the BER increases exponentially at voltage values under the minimum operating voltages. Starting from the consideration that the impact of errors is different for different bit positions, the authors explore, along with voltage scaling, multiple bit-level techniques, which enable dynamic management of the energy-quality tradeoff. The most important of these techniques are:

- **Multiple** V_{dd}. Example it of this techniques is a dual V_{dd} circuit, in which the LSBs are powered with a lower V_{dd} while the other bits keep the nominal V_{dd}.

- **Bit-dropping**: It is a bit-level technique which consists in completely disabling some memory bitlines. The approach showed to be interesting since cells can be completely powered off or even omitted []. The dropped bitlines correspond to a certain number of LSBs in each word, since the impact of errors is exponentially lower for smaller bit weights. This can be paired with the consideration that in many applications, such as machine learning, big data and multimedia, the quality is defined essentially by the MSBs. For SRAM memory cells the precharge circuit of the selected LSBs is disabled during read and write operations. This approach is quite different from the traditional dual V_{dd} scheme where the supply voltage of both precharge circuits and bit cells of the selected columns is reduced to a lower value. In [Frustaci et al., 2015b; Frustaci et al., 2016] the implementation of a bit dropping precharge circuit is proposed: the drop signal, (corresponding to transistors M1 and M2 in Fig. 2.18), connects the bitline of the approximated cell to ground, eliminating the dynamic energy.

- **Selective negative bitline boosting (SNBB)**. Negative bitline boosting (NBB) is an example of write assist technique, which is, in other words, a technique that aims at improving the energy quality tradeoff in the write critical corner of the cell [Frustaci et al., 2016]. By setting the bitline voltage to a negative voltage instead of ground(writing a strong 0 to the corresponding cell) it is possible in fact to improve the cell behavior during the write operation. Selective negative bitline boosting applies this approach only to a reduce number of bitcells (the ones corresponding to MSBs), improving the quality of the MSBs in the write operations but reducing the cost of NBB techniques. From a circuital point of view, only two additional transistors are required in the precharge circuit (Fig. 2.19), in order to connect the bitline to a negative voltage or to ground.

Results on some application (i.e. H.264 hardware video decoder) were produced by modeling the architecture in Matlab and the SRAM bitcell failures were injected according to the measures on test chips.

In [Cho et al., 2011] a dynamically reconfigurable SRAM array is suggested. The proposed solution uses a dual voltage architecture where nominal voltage is applied to cells storing higher order bits while a reduced voltage is applied to cells storing low-order bits. The number of bits with under voltage power supply is reconfigurable at run-time to change the error characteristic. Results are produced by storing static images in the array of cells and measuring the quality degradation induced by injected bit-flips, demonstrating that it is possible to achieve 45% savings in memory power (with only a small reduction, about 10%, in the image quality).

In [Shoushtari, BanaiyanMofrad, and Dutt, 2015] SRAM approximate caches are explored . The work focuses on relaxing the guard-bands required for masking the effects of manufacturing variations as a way of reducing leakage energy of SRAM caches. The proposed approach requires a modification to both traditional hardware and software components (Fig. 2.21). Errors are allowed by exploiting the tolerance

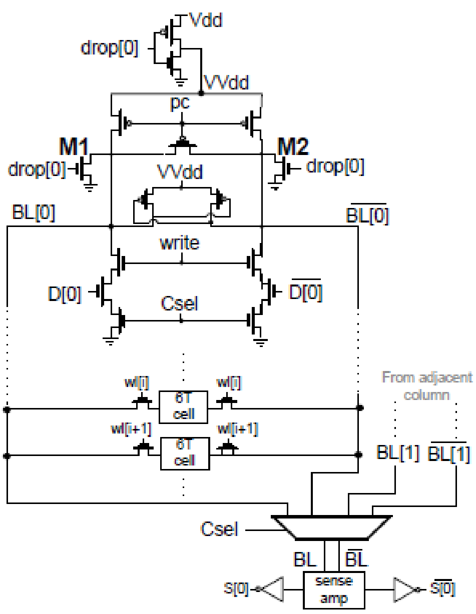


FIGURE 2.18: SRAM bit dropping precharge circuit. Source: Frustaci et al., 2016

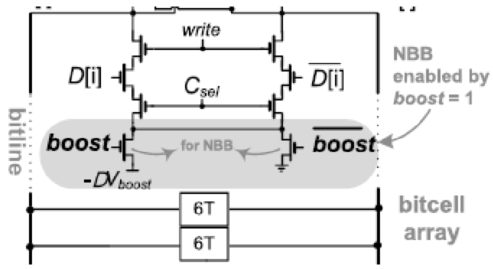


FIGURE 2.19: Sram SNBB precharge circuit. Source: Frustaci et al., 2016

of specific applications and energy savings up to 74% (in leakage) are claimed. Results where produced using the *gem5* simulator [Binkert et al., 2011] by modifying the cache architecture, however the details of the models are not given.

Approximate DRAM

Main memory in modern systems is composed of DRAM cells, that store data as charge on a capacitor. Due to leakage currents, the charge must be periodically restored by a refresh operation, which is usually performed in the background by dedicated hardware units (DRAM controllers). This operation degrades performance, but also wastes energy (for example, when the system is in standby mode, it can reach up to 50% of total power consumption [Liu et al., 2012a]), a drawback which is expected to worsen as DRAMs scale to higher capacities and densities. In fact, in exact DRAMs refresh time interval is set according to the worst case access-statistics of the most leaky cells. Commercial DRAM modules, for example, have a worst case retention time of 64ms determined by the leakiest cells in the entire array. This high refresh rate, which guarantees a storage without errors at the expense of power consumption, could not be required in some applications. As regards eDRAM/DRAMs,

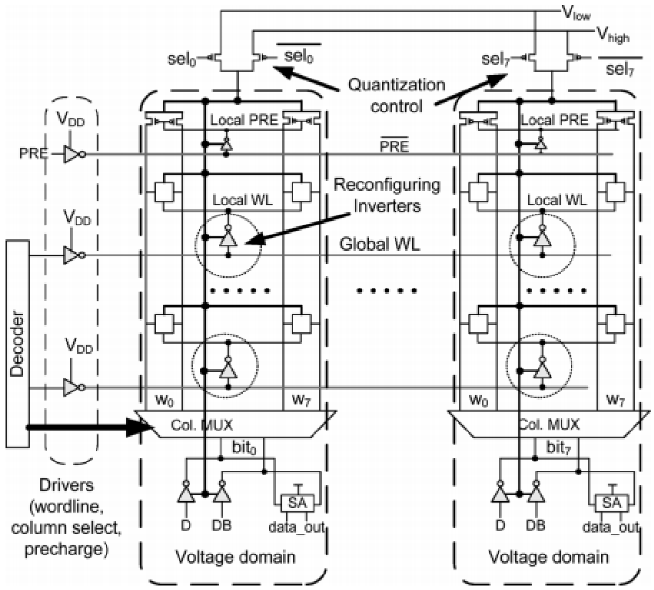


FIGURE 2.20: Architecture of dual V_{dd} memory array. Source : Cho et al., 2011

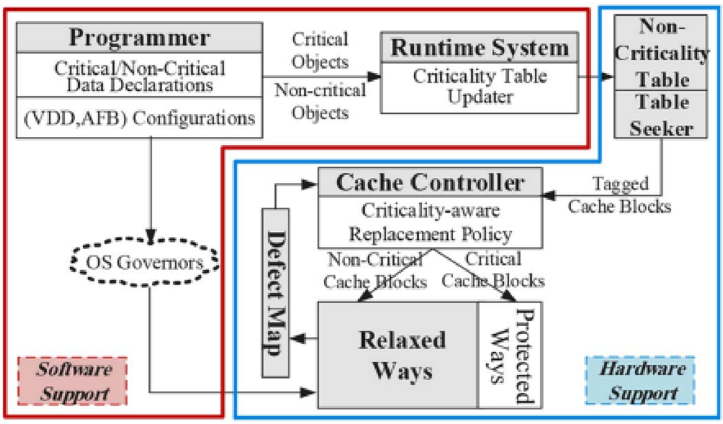


FIGURE 2.21: Overview of HW/SW components for approximated caches. Source : Shoushtari, BanaiyanMofrad, and Dutt, 2015

several techniques to reduce refresh rate have been proposed [Liu et al., 2012a; Liu et al., 2012b; Teman et al., 2015; Raha et al., 2017; Nguyen et al., 2018].

In [Liu et al., 2012a] the authors propose *RAIDR* (*Retention-Aware Intelligent DRAM Refresh*) to partition DRAM rows in different *bins* and apply consequently different refresh rates. In particular, using knowledge of cell retention time, the memory controller uses bloom filters (Fig. 2.22) to classify the rows into the different bins: rows containing leaky cells are refreshed at normal rate, while most rows are refreshed with reduced rate (256, 128 or 64 ms) . This approach does not require modifications in DRAM memory cells but it needs only a little change in the DRAM controller. Evaluated on a 8 core system with 32bit DRAM, RAIDR allows to achieve a refresh rate reduction of about 76% with a DRAM power reduction of 16.1% and a performance improvement of about 8%.

In [Liu et al., 2012b] the idea of partitioning critical and non-critical data is explored. An application-level technique named *Flikker* enables software developers

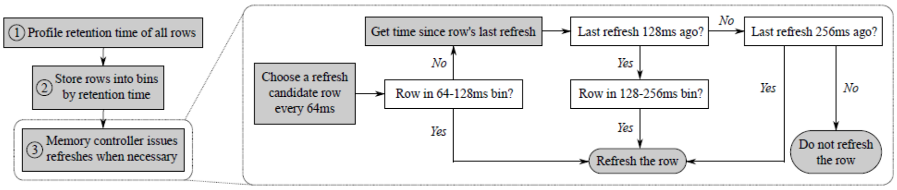


FIGURE 2.22: RAIDR implementation. Source : Liu et al., 2012a

to specify critical and non-critical data in programs, that are then allocated in separate parts of memory (Fig. 2.23). With respect to the previous work, *RAIDR*, the application of a different refresh rate does not depend on the cell retention time but on data. In particular, regular refresh rate is applied to the portion of exact memory containing critical data, while the portion containing non-critical data is refreshed at lower rates. In order to implement the technique, some changes on hardware architecture and on software support are introduced: software requires the OS to be aware of exact and approximate DRAM banks, providing a way to allocate non-critical data in the approximate banks. Analytical models are used to evaluate power savings, showing that this approach can save about 20%-25% of power consumed by the memory system in mobile applications.

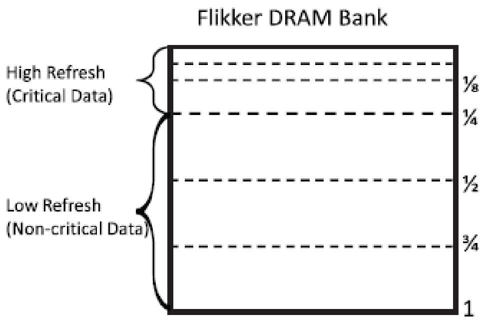


FIGURE 2.23: Proposed DRAM partitioning according to refresh rate.
Source : Liu et al., 2012b

An extension of *Flikker* and *RAIDR* is proposed in *Sparkk* [Lucas et al., 2014], where several refresh periods can be applied on different bits depending on their importance. This work is inspired by the consideration that not all bits within a memory area are critical in the same way and that memory systems, in order to satisfy requirements on bandwidth and capacity, are composed by multiple DRAM banks. The authors propose to have multiple CS signals (one for every DRAM bank instead of one CS shared between all DRAM banks) in order to allow several DRAM chips of a rank to have different refresh periods for the same row as shown in Fig.2.24. To support the Sparkk model, it is required to implement a small hardware unit to control the different refresh rates. The proposed approach has been evaluated simulating the impact of approximate memory on image data, using PSNR as quality metric. In particular, *Sparkk* provides better results than *Flikker*, having a PSNR improvement of about 10dB. Moreover, with respect to *Flikker*, the approach proposed by *Sparkk* is flexible, allowing to get, at the same time, different and configurable quality levels for approximate memory. This can be useful, for example, to implement several approximate memory areas and run different applications in parallel, each one with different requirements on quality level.

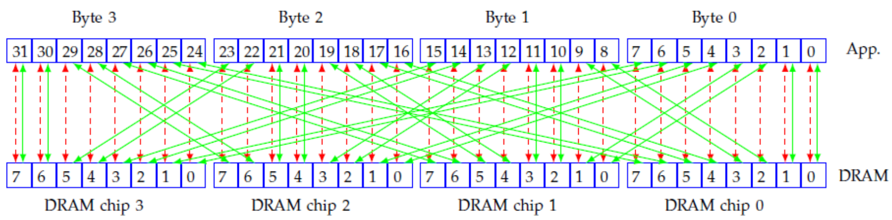


FIGURE 2.24: Proposed mapping of bits of 4 DRAM chips. Source : Lucas et al., 2014

In [Teman et al., 2015] the authors abandon the worst case design paradigm, showing the benefits that can be achieved by relaxing refresh time interval at the expense of increasing error rates. In particular, tests on 8 chips of GC-eDRAMS show that, admitting an error rate of 10^{-3} and relaxing refresh rate from 11ms (worst case retention time) to 24 ms, 55% of energy can be saved; while an error rate of 10^{-2} guarantees energy savings up to 75%.

In [Raha et al., 2017], after an experimental characterization of memory errors as a function of the DRAM refresh-rate, the authors propose a methodology for constructing a quality configurable approximate DRAM system. The core idea is to refresh DRAM with a single but reduced rate, characterizing portions of the memory array and splitting them in several *quality bins* (Fig. 2.25), based on the frequency, location and nature of bit errors in each physical page. During program execution, non-critical data can be allocated to *bins* sorted in descending order of quality. Experiments were performed on a FPGA board where a soft-processor (Nios II) and a DDR3 memory controller (UniPHY) are synthesized. The setup included the use of the lightweight operating system SYNCC/OS-II for memory management and task creation. A reduction in DRAM refresh power of up to 73% on average is shown. However, the paper proposes to use the quality bins just in a descending order, ensuring that lower quality bins are always used as last resource. The work does not explore the possibility of selecting the *quality bins* at program level, depending on data.

Quality Bins	Strategy 1	Strategy 2	Strategy 3	Strategy 4
	Word errors/page	Bit errors/page	BWm /page	Bit errors/page
qbin0	0	0	0	0
qbin1	1	1	0 - 0.1	1
qbin2	2	2	0.1 - 0.25	2
qbin3	3-10	3-10	0.25 - 0.5	3-10
qbin4	11-15	11-15	0.5-1	11+
qbin5	16-20	16-20	1 - 1.25	1
qbin6	21-25	21-25	1.25 - 1.5	2
qbin7	26-30	26-30	1.5 - 1.75	3
qbin8	31-35	31-35	1.75 - 2	4
qbin9	36+	36+	>2	5

FIGURE 2.25: Proposed quality bins. Source : Raha et al., 2017

The specific use of approximate DRAM architectures is studied in [Nguyen et al., 2018] for deep learning applications, since they are tolerant to the presence of errors in data. In particular, DRAM organization is modified to support the control of the refresh rate according to the significance of stored data. Simulations were performed injecting errors at algorithm level, using random bit flips with uniform distribution, without reference to the internal structure of DRAM cells. Results with GoogleLeNet and VGG16 show that this approach allows to reduce power consumption up to 70%, with a negligible impact on the accuracy of the classification process.

In [Widmer, Bonetti, and Burg, 2019] the authors propose an FPGA based framework that accurately reproduce errors in eDRAMs. The goal is to provide a way for the evaluation of error resilience in embedded applications that use eDRAMs working with sub-critical refresh rate under the DRT. As illustrated by the block diagram in Fig. 2.26, the eDRAM emulator is composed by:

- a control logic;
- a storage block composed by a set of SRAMs where data and additional information are stored. The latter correspond to a data timestamp (Write TS), stored every time a write operation is performed in the bitcell; an event table containing one entry for each bit in the DRT map and finally a pointer to this table. The last two elements are programmed before the emulation is started. In particular, the DRT map can be obtained from silicon measurements or statistics.

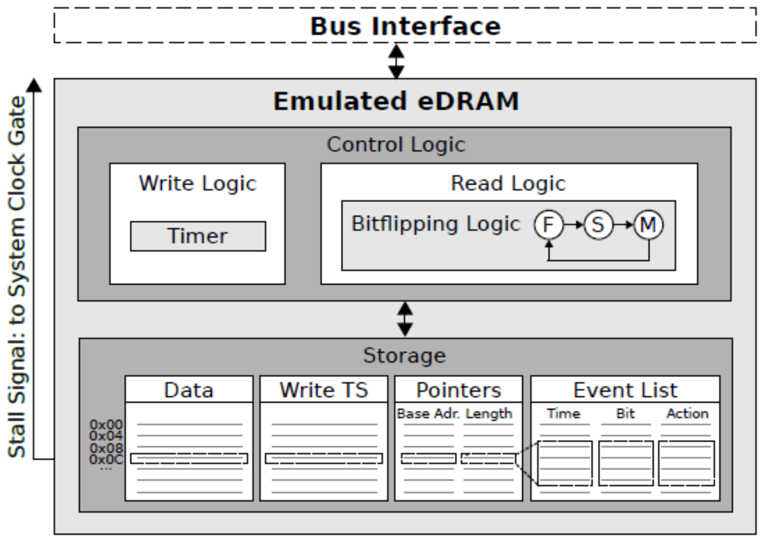


FIGURE 2.26: eDRAM emulator: block diagram. Source: Widmer, Bonetti, and Burg, 2019

Every time a write operation is performed, a write timestamp is stored. Errors are injected during the read operations: the bit flip happens only if the delay between the read and write time exceeds the value stored in the corresponding entry in the DRT map. This eDRAM emulator is then integrated in an embedded system built using the Xilinx Vivado 2018b design tool, composed of a MicroBlaze softCPU and a typical memory size of 1MB of eDRAM. Six different benchmarks have been analyzed in order to evaluate the impact on quality produced by the refresh rate relaxation.

Results (Fig. 2.27) show that the number of accesses to exact memory is significantly reduced, reducing energy consumption, with only a negligible impact on quality.

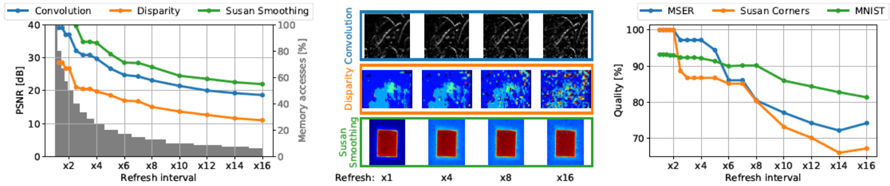


FIGURE 2.27: Benchmarks output quality for refuced refresh rate. Source: Widmer, Bonetti, and Burg, 2019

2.4 Transprecision Computing

As we said, Approximate Computing is a technique which proposes to relax the specifications on precise computation allowing digital systems to introduce errors implied by imprecise hardware/software, and trading off quality, in terms of computational accuracy, for energy consumption or speed. *Transprecison Computing* instead, claims that low power embedded systems should be designed to deliver the required precision for computation. In particular *Transprecison Computing* allows to get a fine-grained control (Fig. 2.28) over approximation in space and time, meaning where and when (e.g. using multiple feedback control loops in both hardware and software in order to follow the required constraints on the precision of output results).

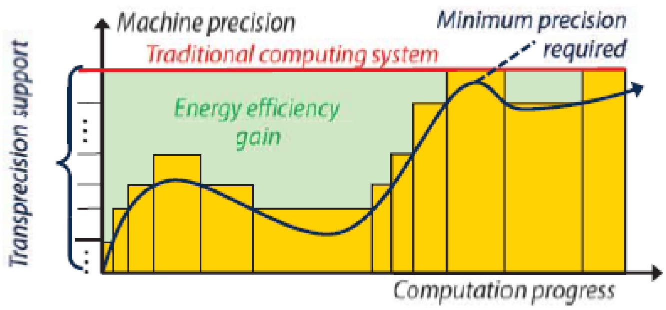


FIGURE 2.28: Transprecision Computing paradigm. Source : Malossi et al., 2018

The first complete Transprecision Computing framework has been developed in the context of the H2020 European project OPRECOMP, in particular, in order to show the benefits of this new approach, two demonstrator systems have been developed ([Malossi et al., 2018]):

- *mW Demonstrator*: this is a technology demonstrator for transprecision units. To validate the approach the PULP (*Parallel Ultra Low Power*, [Rossi et al., 2015]) platform is used, extended with transprecision computing units such as accelerators and memory infrastructure.
- *kW Demonstrator*: this is a functional and scalability demonstrator for the software stack. In this case the base platform is one node of a HPC system, extended with transprecision capabilities.

2.5 Approximate Computing and Machine Learning

In recent years there has been a growing interest in artificial intelligence, due to the noteworthy progress made in the fields of machine learning (ML) and neural networks. In particular, using ML algorithms, computers can find solutions by learning from a set of data (training set). Compared to the general purpose CPUs, GPUs have been found to efficiently execute ML algorithms, due to the availability of multiple simple cores and to the capability of supporting massive parallelism at thread level. However, despite the ability to support intensive calculations necessary in the training phase, GPUs cannot be currently adopted for the inference phase on embedded devices with energy constraints and IoT devices, where the resources available are limited and the energy budget available is often in sub-Watt range.

This section presents the opportunities and challenges in building hardware architectures for ML using the *Approximate Computing*, since this paradigm could allow to decrease the overall system energy/power consumption without impacting significantly the prediction accuracy.

In [Shafique et al., 2017a] the authors propose a methodology to guide the implementation of energy-efficient accelerators for Machine-Learning, especially in the context of Convolutional Deep Neural Networks (DNNs) where quality-/energy-configurable approximate modules can be employed inside the accelerator datapath, allowing a high degree of adaptivity. The implementation of DNNs particularly fit the use of approximate computational modules since the applications for DNNs are intrinsically resilient to errors. This is essentially due to two reasons:

1. the input vectors are always processed in the same way, despite the intrinsic complexity of the classification application changes considerably;
2. the training process is an iterative process, so it can be interrupted when an adequate level of accuracy is reached.

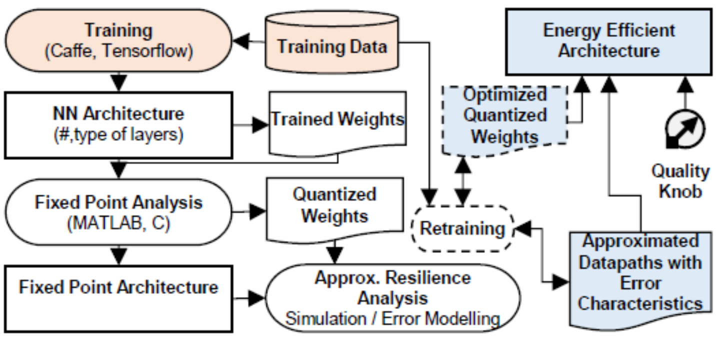


FIGURE 2.29: Methodology for designing energy-effcient and adaptive neural network accelerator-based architectures for Machine Learning. Source: Shafique et al., 2017a

Fig. 2.29 illustrates the proposed methodology, which can be summarized in the following steps:

- *Training*: ML libraries as *TensorFlows* and *Caffe* can be used for customizing pre-trained NN for a specific application.
- *Fixed Point Analysis*: this step is necessary to select an adequate fixed-point format (Qmn, m-bit for the integer and n-bit for the fractional part). It can be performed for example using Matlab or a C application.

- *Approximation Resilience Analysis*: this steps consists in the evaluation of the quantized weights and the architecture in order to explore their resilience properties. It is necessary in fact to select a particular approximate configuration in order to build an energy efficient accelerator without reducing the prediction accuracy. Moreover since the error of the approximate modules depends also on the data distribution, a preliminary analysis can be performed on the training data to guide the approximation process. The resilience analysis can be performed through MonteCarlo simulations or using analytical models of approximate modules. As the DNN layers and the possible number of approximate configurations increase, the authors propose also the implementation of a multi-objective evolutionary algorithm to automatically develop energy-efficient and adaptive accelerators. In particular the CPG approach is used for the evaluation of the candidate approximate circuits: if the circuit is not complex, it is possible to evaluate its responses for all input combinations and determine the prediction error, otherwise for more complex circuits the evaluation is performed using a TS and estimating the resulting error.

- *Quality Knobs*: these ones allow to control the accuracy-power tradeoff, selecting the appropriate approximate hardware modules according to the application requirements. Therefore they are required for the implementation of hardware accelerators that have to be adaptive.

- *Re-Training*: this step could be performed to update the quantized weights and/or improve the classification accuracy alleviating the accuracy loss introduced by the approximation.

In Chen et al., 2017 the authors propose DRE, a framework for the evaluation of data resilience in CNNs. The ultimate goal of this work is to employ the proposed framework to assess the feasibility of applying AxM techniques to a given CNN, since memory accesses and data transmissions are crucial aspects in the implementation of hardware CNNs. In this work approximation models are used to abstract several AxM techniques in order to make quick evaluations; in particular a wide range of AxM techniques can be modeled as random errors which are uniformly injected into resilient bits. For off-chip memory reduction instead the AxM techniques are based on data bit-width scaling [Tian et al., 2015].

Fig. 2.30 illustrates the DRE framework, built on the *Caffe* platform (the *Tool Setup* box in the figure).

The framework is composed by the following four modules:

1. *Single Layer Analysis*: this module is used to analyze and evaluate the neurons numerical representation requirement and the weights of each layer. The flow is as follows (Fig. 2.31):

    - A resilient subset of data is selected and transformed in the chosen numerical representation, in order to be able to observe the impact on the accuracy of the network.
    - If the obtained prediction accuracy is acceptable, the bit-width of the chosen numerical representation is reduced by bit truncation. Then the previous step is repeated.
    - if the drop in accuracy is not acceptable, the numerical representation requirement of this data subset corresponds to the previous chosen bit width.

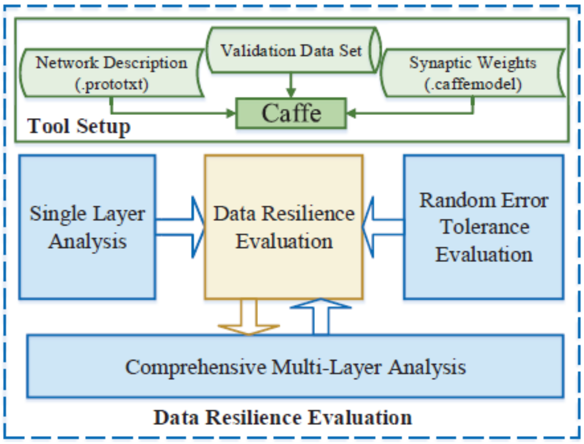


FIGURE 2.30: Overview of DRE. Source : Chen et al., 2017

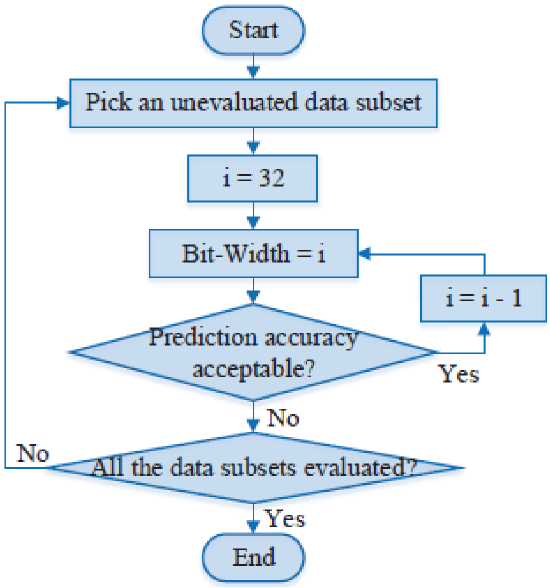


FIGURE 2.31: Flow of Single Layer Analysis. Source: Chen et al., 2017

Through these passages, the numerical representation requirements of neurons and weights are obtained at each CNN layer.

2. *Comprehensive Multi-Layer Analysis*: this module provides an analysis of multi layers, in order to derive a complete numerical representation scheme for all data subsets

3. *Random Error Tolerance Evaluation*. The adopted strategy is the same as the one employed in single layer analysis with the difference that, in addition to bit truncations, random errors with a given probability are injected into specific resilient data subsets.

4. *Data Resilience Characterization*: it gives periodical evaluations of the three modules, producing a concluding evaluation of data resilience for a given CNN.

As case study, the proposed DRE framework has been applied to four prevalent CNNs demonstrating that a high degree of data resilience exists in these networks.

Simulation results show that for off-chip memory accesses, it is possible to employ bit-width scaling in order to reduce the amount of data transfer from off-chip memory to on-chip memory. The authors demonstrate that on average the data volume can be reduced by 80.38%, with a 2.69% loss in the prediction accuracy. For AxM with random errors, all the synaptic weights can be stored in the approximate part when the error rate is less than 10^{-4}. When the error rate is fixed at 10^{-3}, 3 MSBs must be protected while other LSBs can be stored in the approximate part.

In [Nguyen et al., 2018] an approximate DRAM architecture for deep learning applications is illustrated. The authors propose to store data in a transposed manner so that:

- data bits are distributed according to their significances (e.g. all the first MSBs of data[0], data[1],..., data[7], are transposed and stored in the memory location at addressed 0, i.e. data[0]) as shown in Fig. 2.32;
- different refresh rate are used depending on the significance of the row.

In particular Fig. 2.32.a shows the typical data storage scheme while Fig. 2.32.b represents the proposed approximate data storage scheme where data are stored in a transposed manner. This approximate memory structure requires a minor change inside the DRAM memory controller: a *bit transpose unit* (as shown in Fig. 2.32.c) which is responsible for the data format conversion.

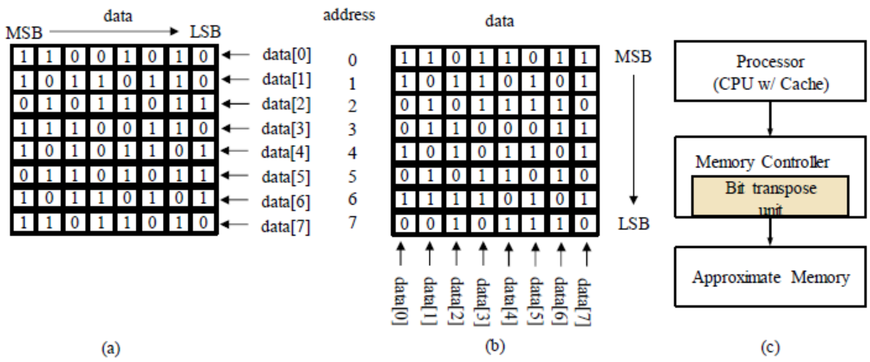


FIGURE 2.32: Approximate memory architecture; (a) conventional data storage scheme, (b) approximate data storage scheme, (c) system architecture to support approximate memory access. Source: Nguyen et al., 2018

The authors propose to use this architecture for deep learning applications, which in general access 32-bit FP data. According to IEEE-754 standard, each floating-point data has:

- 1 sign bit;
- 8 exponent bits;
- 23 mantissa bits

The bits corresponding to the sign and the exponent are critical data due to the fact that the presence of errors in this bits may change significantly the data value. The 23 mantissa bits can be stored in memory rows refreshed at slower rate. The refresh rate decreases as the row number increases, meaning that that the LSBs are stored in the rows with higher error (Fig. 2.33).

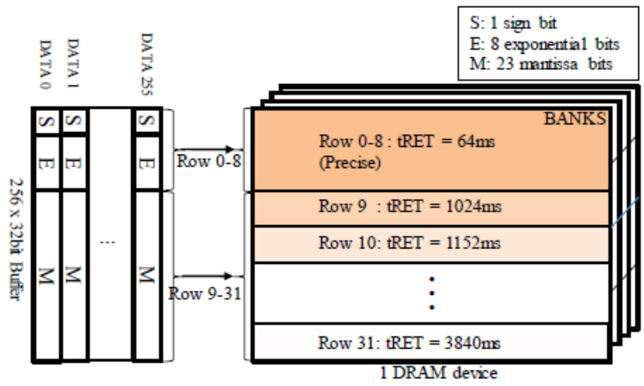


FIGURE 2.33: Row-level refresh scheme for approximate DRAM. Source: Nguyen et al., 2018

The simulation results obtained with state-of-the-art networks (GoogLeNet and VGG-16) show that the power consumption due to refresh rate can be reduced by 69.68%, with a negligible degradation of the classification accuracy.

Chapter 3

Approximate Memory Support in Linux OS

3.1 Introduction

As described in the previous chapter, Approximate Memories are part of the wider research topic regarding Approximate Computing and error tolerant applications, in which errors in computation are allowed at different levels (data level, instruction level, algorithmic level). In general these errors are the result of circuital or architectural techniques (i.e. voltage scaling, refresh rate reduction) which trade off energy savings for the occurrence of errors in data processing. The ability to support approximate memory in the OS is required by many proposed techniques which try to save energy by raising memory fault probability, but the requirements at OS level have never been described and an actual implementation has never been proposed. In this chapter an analysis of the requirements and a description of the implementation of approximate memory management is provided. The proposed approach allows Linux kernel to be aware of exact (normal) and approximate physical memories, managing them as a whole for the common part (e.g. optimization algorithms, page reuse) but distinguishing them in term of allocation requests and page pools management.

In particular the introduction of approximate memory management allows at application level to distinguish between *critical data*, that must be stored in exact memory, and *non-critical data*, that can be stored in approximate memory.

Critical data are identified as all program data that cannot accept any level of corruption without significantly impact the functionality of the application itself as, for example, variables used in conditional constructs and responsible for control flow, state variables, pointers. Typically, the memory allocated by the OS for applications is divided into four parts:

- Code (.text section);
- Static global variables;
- Stack memory;
- Heap memory.

Code must always be considered critical and should never be allowed to be corrupted. Stack memory contains local variables, parameters, return addresses (e.g. return address of function calls) and register file copies. Due to the variable nature of this data (some local variables could be tolerant to errors but other data contents are not), in this work we did not consider the management of an approximate stack memory.

Memory for static global variables and the heap memory can instead contain both critical data and non-critical data, depending on the application. Both memory spaces are allocated with the same function calls at OS level, the only difference being that the first is allocated and initialized automatically by the OS while the second is allocated on request at user program level.

In order to support the management of two different memories (exact and approximate), the Linux kernel has been extended by adding a new memory area, called 'ZONE_APPROXIMATE' and a new dynamic allocator, which has been implemented to ensure that only approximate data are allocated in this portion of memory. Before going into the details of the implementation, an overview of the Linux Memory Management is provided.

3.2 Linux Memory Management

3.2.1 Virtual Memory and Address Spaces

All recent operating systems support virtual memory, a memory management technique that allows a process to have its own address space and to protect its memory from corruption by other process. This means that the logical addresses seen by a user-level program do not correspond directly to the physical addresses seen at hardware level and that every process, considering itself the only one to have access to the system resources, has its own virtual address space potentially larger than available physical memory. During program execution, the processor accesses main memory to load instructions (*fetch phase*) or to load/save data from/in a memory location. Virtual memory represents an abstraction layer between the memory requested by an application and the MMU (Memory Management Unit). In a virtual memory system indeed all the addresses at program level are virtual and these are translated into physical addresses by the processor through (*page tables*) managed by the operating system. This abstraction offers several advantages:

- concurrent execution of several programs, without address space conflicts;
- execution of applications whose address space is larger than available physical memory;
- each process can run even if code is only partially loaded in memory;
- fast relocation of programs and de-fragmentation of de-allocated physical memory;
- user level code is machine-independent, meaning that it is not necessary to know the physical memory organization of a specific architecture to write a program.

To support this memory management capability, Linux implements *paging*: virtual memory is divided into identical blocks called *pages* and RAM is equally divided into blocks of the same size called *page frames*. Therefore Linux internally manages the following types of addresses:

- *User-level virtual addresses*: the regular addresses seen at the user-space by a program. A virtual address consists of an offset and a *PFN* (virtual page frame number); when it is necessary to access memory the processor extracts the PFN and accesses the requested location by applying the correct offset to the physical page. These addresses can be 32 or 64 bits depending on the architecture.

- *Physical addresses*, i.e. the addresses used by processor and memory. They can be 32 or 64 bits.
- *Bus addresses*, i.e. the addresses used by the memory and the buses of the peripherals.
- *kernel logical addresses*, corresponding to the kernel address space. These addresses map a portion of RAM and they are often handled as physical addresses. A call to *kmalloc* returns a kernel logical address.
- *kernel virtual addresses*. These addresses are similar to kernel logical addresses as they are mapped to physical addresses but unlike the previous ones, the mapping is not 1:1. All logical addresses are kernel virtual addresses but the viceversa is not true. A call to *vmalloc* returns a kernel virtual address.

3.2.2 Low Memory and High Memory

The difference between kernel logical and virtual addresses emerges significantly in 32-bit architectures, where it is possible to address up to 4GB of memory. For example in x86 32-bit architecture the 4GB of virtual memory are typically divided in user space and kernel space. In the default configuration, 1 GB, the one mapped in the lower part of memory, contains kernel data structures and kernel code *(kernel space)* while the remaining 3GB are reserved for *user space*. The kernel can directly manage only the memory that is mapped in its virtual address space (1GB); consequently x86-based Linux systems can operate with a maximum of 1GB of physical memory (kernel virtual address space minus the kernel space reserved for kernel code itself.) If physical memory is equal or less than 1GB, then physical memory is mapped directly into the kernel virtual address space and these addresses, for which the mapping is 1:1, correspond to kernel logical addresses. In architectures where physical memory is more than 1GB, to overcome the limitation of having to map all physical memory in the address space of 1GB, the kernel uses 128 MB of its own virtual address space to perform a temporary mapping between virtual and physical addresses in order to access to all available physical memory. In these architectures kernel virtual address space is divided into two regions: *Low Memory* or *Lowmem* and *High Memory* or *Highmem*. The Lowmem corresponds to the portion of memory allocated in the first 896 MB and it contains the data the kernel needs to access more frequently. Being directly mapped into kernel address space (kernel logical addresses), this region can be accessed immediately by the kernel. Highmem corresponds instead to the portion of physical memory higher than the address of 896 MB; this region cannot be directly mapped into kernel address space but it is temporarily mapped in kernel virtual space when the kernel needs to access it (virtual addresses). This temporary mapping of data from highmem to kernel address space is performed through calls to *kmap, kunmap, kmap atomic, kunmap atomic*. In general, data allocated in high memory correspond to data that the kernel accesses occasionally (including for example page cache, page tables and process memory). In 64-bit architectures this separation of 1GB/3GB between kernel space and user space is not needed: as shown in Fig. 3.1 the address space in these case is large enough (512 GB or more) to allow a separation between kernel space and user space and to map all the available physical memory in the kernel address space.

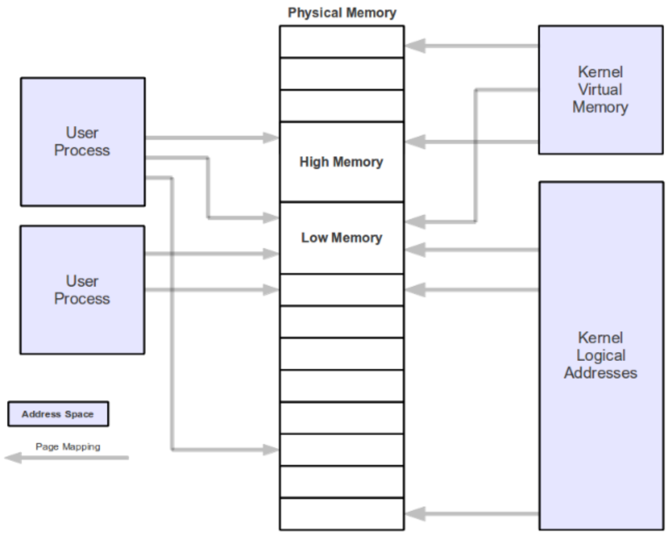


FIGURE 3.1: Kernel address space and User process address space

3.2.3 Physical Memory

Starting from Linux 2.5, kernel supports *NUMA* (*Non Uniform Memory Access*) architectures. The latter are designed for multiprocessor systems in which CPUs and RAM memory chips are grouped into *"local"* nodes (usually each node contains a CPU and a few RAM chips). The access time to a specific node depends on the distance between the node to be accessed and the CPU that wants to access it: each CPU accesses its own local node faster than other nodes local to other processors. Each node therefore corresponds to a memory bank and it is described, in the kernel source files, by a struct *pglist* data (this is also valid for *UMA*, *Uniform Memory Access*, architectures where a single node is present and the access time to memory is uniform).

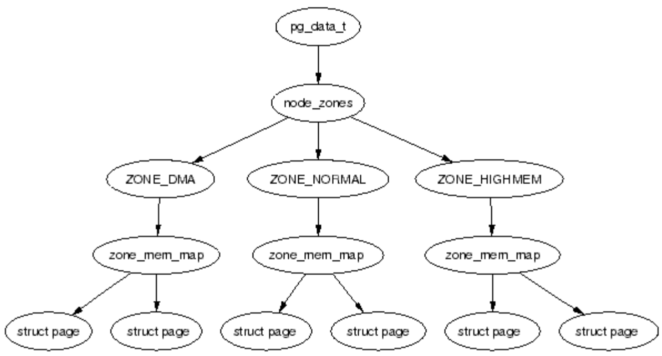


FIGURE 3.2: Nodes, Zones and Pages. Source:[Gorman, 2004]

Each node is divided into memory ranges, called *zones* which are used by the Linux Kernel to group pages having similar properties (Fig.3.2). This partitioning has no physical relevance, but allows the kernel to keep track of pages and overcomes hardware limitations:

- some hardware devices can perform DMA (Direct Memory Access) only at certain memory addresses.

- some architectures can address an amount of physical memory greater than the virtual addressing space. As showed above, it involves that not all physical memory available is mapped in the kernel address space .

Internally, five primary zones are defined:

1. **ZONE DMA**: pages within this zone can be used by DMA. As shown in Table 3.1 with some examples, the range of physical memory reserved for this area varies according to the architecture.
2. **ZONE DMA32**: this zone is like ZONE DMA but it is the only that can be accessed by some 32 bit devices.
3. **ZONE NORMAL**: pages within ZONE NORMAL are directly mapped by the kernel. Many of the operations performed by the kernel can take place only in this zone, which consequently is the most critical from the performance point of view.
4. **ZONE HIGHMEM**: this zone corresponds to high memory and it is not directly mapped by the kernel. In x86 architectures this area allows the kernel to address memory over 900MB, performing special mappings (via the page tables) for each page the kernel needs to access.
5. **ZONE MOVABLE**: Unlike memory zones described previously, zone movable does not have a specific physical range but pages within this zone come from other memory zones. The scope of this virtual memory zone is to avoid memory fragmentation.
6. **ZONE DEVICE**: this zone is used to distinguish the pages belonging to *'device memory'*, having an allocation mechanism different from the standard one. The *'device memory'* has indeed different characteristics compared to RAM memory in terms of lifetime and performance.

TABLE 3.1: DMA zone, physical ranges

Architecture	*Physical limit*
PARISC, IA64, SPARC	< 4G
S390	< 2G
ALPHA	Unlimited or 0-16MB
I386, X86 and other architectures	< 16MB

Linux memory zones usage and layout can change according to the architecture on which they are implemented: for example, in architectures where a DMA device supports transfers over all addressable memory, the DMA zone can be empty and DMA operations can be performed on pages belonging to ZONE NORMAL. Unlike zone NORMAL and zone MOVABLE that are always present in the Linux OS, the other zones can be enabled or not through appropriate *#ifdef* commands in the Kernel configuration files. Each memory zone is described by a struct *zone* declared in *<linux/mmzone.h>* source file; the fields of this structure allow to keep track of some data such as the number of free pages (*unsigned long free_pages*), the *spinlock* to protect the zone from concurrent access, the zone name (*char * name*, initialized during the boot phase in <*page alloc.c*>), and other useful data for statistics concerning the usage of the pages. An interesting field concerns the *watermarks*, used by the kernel as a benchmark to check the right memory occupancy. For each zone three watermarks are defined: *pages_min*, *pages_low*, *pages_high*. If the available memory is low, the kernel swap daemon, *kswap*, starts to free pages and if the pressure is high the release mechanism takes place in synchronous way. The three watermarks are used therefore to monitor this pressure (Fig. 3.3). In particular:

- *pages_low*: when the *pages_low* number of free pages is reached, *kswapd* is awakened by the allocator, *buddy allocator*, to start freeing pages. The value of *pages_low* is two times the default value of *pages_min*.
- *pages_min*: when *pages_min* mark is reached, the allocator performs the work of *kswapd* synchronously; sometimes this is referred as the *direct-reclaim* path.
- *pages_high*: when the *kswapd* manager has been woken up to start freeing pages, the daemon does not take into account the zone to be balanced when*pages_high* are free. Once the watermark has been reached, *kswapd* will go back to sleep. The default setting for *pages_high* is three times the value of *pages_min*.

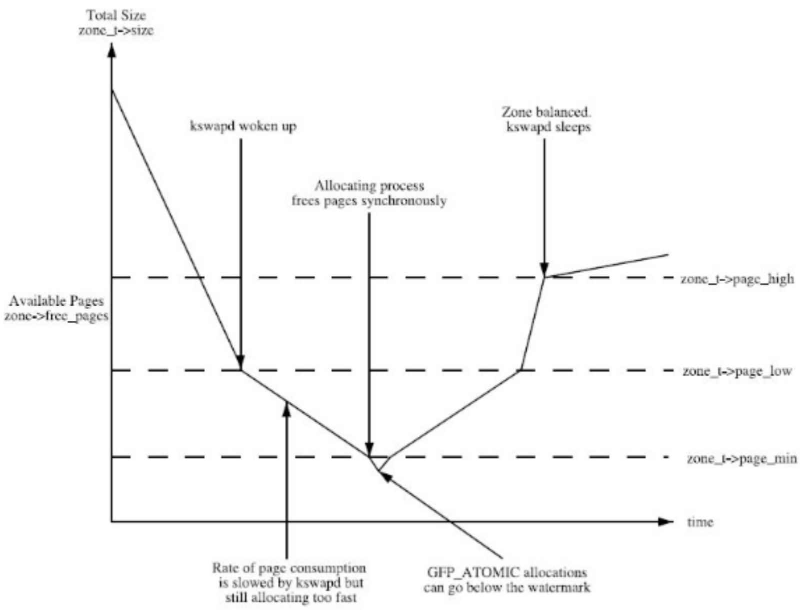


FIGURE 3.3: Zone watermarks

As physical memory is partitioned into zones to group pages with the same property, pages are in turn the basic unit of memory. Each page, whose dimensions vary according to the architecture (on x86 architectures each page is 4KB), is described by a struct *page*, defined in <*linux/mm_types.h*>.

```
struct page {
 // First double word block //
 unsigned long flags;
 // Atomic flags, some possibly updated asynchronously//
 union {
 struct address space *mapping;
 // If low bit clear, points to inode address space, or NULL. If
 page mapped as anonymous memory,
```

```
        low bit is set, and it points to anon vma object to see PAGE
    MAPPING ANoN below.
        void *s mem; // slab first object //;
        // Second double word //
        struct {
        union {
        pgoff t index; // our offset within mapping.//
        void *freelist; // sl[aoub] first free object//
        };
```

This structure allows to keep track of the current state of a page even if a way to determine which task is using a specific page is not implemented (if the page is a *pagecache*, than the *rmap* structures allow to go back to who is mapping the page in question). The page descriptor is needed to allow the kernel to distinguish between page frames that contain pages belonging to user-level processes and frames that contain kernel code and structures. Moreover the Linux kernel must be able to determine dynamically which page frames are free (a page frame has to be considered free if it does not contain free data, consequently it is not free if it contains user-level process data, kernel data, buffered data of a device driver and so on). The struct fields are organized in blocks of double word, in order to perform atomic double word operations on some parts of the struct. For example, as shown below, the first double word block is composed of *page flags* and a * *struct mapping* of address space type. In the following text box the list of page flags, declared in <*linux/page_flags.h*>, is reported. These flags are used to describe the state of the page (for example if a page is present, if it is reserved, if it has been accessed, if an error has occurred and so on).

```
        enum pageflags{
        PG_locked, // Page is locked. Don't touch. //
        PG_error,
        PG_referenced,
        PG_uptodate,
        PG_dirty,
        PG_lru,
        PG_active,
        PG_slab,
        PG_owner_priv 1, // owner use. If pagecache, fs may use//
        PG_arch 1,
        PG_reserved,
        PG_private, // If pagecache, has fs-private data //
        PG_private_2,// If pagecache, has fs aux data //
        PG_writeback,// Page is under writeback //
        PG_head, // A head page //
        PG_swapcache,// Swap page: swp entry t in private //
        PG_mappedtodisk,// Has blocks allocated on-disk //
        PG_reclaim,// To be reclaimed asap //
        PG_swapbacked,// Page is backed by RAM/swap //
        PG_unevictable,// Page is 'unevictable' //
        #ifdef CONFIG_MMU
        PG_mlocked,// Page is vma mlocked //
        #endif
        #ifdef CONFIG_ARCH_USES_PG_UNCACHED
        PG_uncached, // Page has been mapped as uncached //
        #endif
        #ifdef CONFIG_MEMORY_FAILURE
        PG_hwpoison, // hardware poisoned page. Don't touch //
        #endif
        #if defined(CONFIG_IDLE PAGE_TRACKING) && defined(CONFIG_64BIT)
        PG_young,
```

```
        PG_idle,
#endif
        NR_PAGEFLAGS,
```

3.2.4 Kernel Memory Allocators

It is possible to distinguish several dynamic memory allocators within the Linux Kernel. Some of the main allocation mechanisms are described below, these allocators will be also described in the following paragraphs:

- Page-level allocation (*Buddy System*);
- Contiguous memory allocation (*kmalloc*);
- Non-contiguous memory allocation (*vmalloc*).

Page-level allocator (Buddy System algorithm)

All interfaces provided by the kernel to allocate memory are based on a low level algorithm with page size granularity, called *Binary Buddy Allocator*. According to this algorithm, physical memory is divided into power of two size blocks, when a block of the requested size is not available, a larger block is split in two half (*buddies*) and the process is iteratively repeated until a block of the requested size is produced (see Fig. 3.4). When a block is then released, the Kernel checks if also the *"buddy"* block is free; in that case the two blocks are merged again.

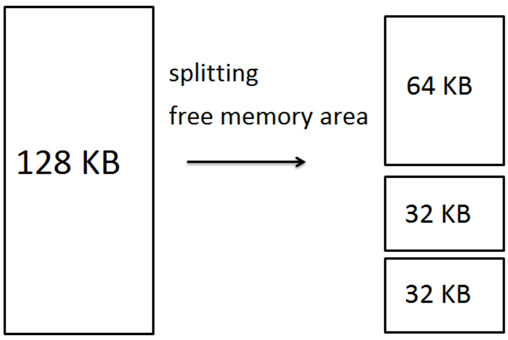


FIGURE 3.4: Buddy system allocator

In the Linux kernel, the buddy allocator core routine receives as parameter, among others, a bit mask (called *gfp mask*) which is a set of GFP flags (*Get Free Page flags*) that allow to direct the allocator behavior. Three types of GFP flags are defined in the <*include/linux/gfp.h*> source file:

- *'zone modifiers'* flags: these flags allow to specify the zone for the allocation, indicating that the kernel should choose memory from the requested zone if possible. The allocator considers the zone specified in gfp mask as an indication and, in some cases depending on actual memory utilization and balancing policies, the request could be satisfied selecting pages belonging to hierarchically higher zones.

```
    #define GFP_DMA          0x01u
    #define GFP_HIGHMEM}     0x02u
    #define GFP_DMA32 }      0x04u
    #define GFP_MOVABLE}     0x08u
    // gpz zone modifiers //
```

```
#define GFP_DMA (( force_gfp_t) GFP_DMA)
#define GFP_HIGHMEM (( force_gfp_t) GFP_HIGHMEM)
#define GFP_DMA32 (( force_gfp_t) GFP_DMA32)
#define GFP_MOVABLE (( force gfp t) GFP_MOVABLE) // Page is
movable //
#define GFP_ZONEMASK (GFP_DMA GFP_HIGHMEM GFP_DMA32
GFP_MOVABLE)
```

- *'action modifiers'* flags: these flags are used to determine (and change) the behavior of the Virtual Memory and establish what the calling process can do.

```
#define GFP_WAIT (( force_gfp_t) GFP_WAIT) // Can wait and
reschedule //
#define GFP_HIGH (( force_gfp_t) GFP_HIGH) // Should access
emergency pools? //
#define GFP_IO (( force_gfp_t) GFP_IO) // Can start physical
IO? //
#define GFP_FS (( force_gfp_t) GFP_FS) // Can call down to
low-level FS//
#define GFP_COLD (( force_gfp_t) GFP_COLD) // Cache-cold page
 required//
#define GFP_NOWARN (( force_gfp_t) GFP_NOWARN) // Suppress
page allocation failure warning//
#define GFP_REPEAT (( force_gfp_t) GFP_REPEAT) // See above//
#define GFP_NOFAIL (( force_gfp_t) GFP_NOFAIL) // See above
//
#define GFP_NORETRY (( force_gfp_t) GFP_NORETRY) // See above
 //
#define GFP_MEMALLOC (( force_gfp_t) GFP_MEMALLOC) //Allow
access to emergency reserves//
#define GFP_COMP (( force_gfp_t) GFP_COMP) // Add compound
page metadata //
#define GFP_ZERO (( force_gfp_t) GFP_ZERO) // Return zeroed
page on success //
#define GFP_NOMEMALLOC (( force_gfp_t) GFP_NOMEMALLOC) // Don
't use emergency reserves.
// This takes precedence over the * GFP_MEMALLOC flag if both
 are set//
#define GFP_HARDWALL (( force_gfp_t) GFP_HARDWALL) //Enforce
hardwall cpuset mem allocs//
#define GFP_THISNODE (( force_gfp_t) GFP_THISNODE)// No
fallback, no policies //
#define GFP_RECLAIMABLE (( force_gfp_t) GFP_RECLAIMABLE) //
Page is reclaimable //
#define GFP_NOACCOUNT (( force_gfp_t) GFP_NOACCOUNT) // Don't
 account to kmemcg //
#define GFP_NOTRACK (( force_gfp_t) GFP_NOTRACK) // Don't
track with kmemcheck //
#define GFP_NO_KSWAPD (( force_gfp_t) GFP_NO_KSWAPD)
#define GFP_OTHER_NODE (( force_gfp_t) GFP_OTHER_NODE) // on
behalf of other node //
#define GFP_WRITE (( force_gfp_t) GFP_WRITE) // Allocator
intends to dirty page
```

- The third set of flags is composed by the combination of some of the *action modifiers* flags defined previously. Some of the *action modifiers* flags in fact are too low-level to be used individually and consequently it becomes difficult to determine the correct flag combination for each instance. To overcome this problem, Linux provides high level flag combinations, as shown, for exmple,in the box below.

```
#define GFP_NOWAIT (GFP_ATOMIC & GFP_HIGH)
//GFP_ATOMIC means both !wait ( GFP_WAIT not set) and use
emergency pool //
#define GFP_ATOMIC ( GFP_HIGH)
#define GFP_NOIO ( GFP_WAIT)
#define GFP_NOFS ( GFP_WAIT | GFP_IO)
#define GFP_KERNEL ( GFP_WAIT | GFP_IO| GFP_FS)
#define GFP_TEMPORARY ( GFP_WAIT| GFP_IO| GFP_FS |
GFP_RECLAIMABLE)
#define GFP_USER ( GFP_WAIT | GFP_IO | GFP_FS | GFP_HARDWALL)
#define GFP_HIGHUSER (GFP_USER | GFP_HIGHMEM)
#define GFP_HIGHUSER MOVABLE (GFP_HIGHUSER | GFP_MOVABLE)
#define GFP_IOFS ( GFP_IO | GFP_FS)
#define GFP_TRANSHUGE (GFP_HIGHUSER_MOVABLE | GFP_COMP)
```

As described previously, these flags are always passed as parameters to all the allocation functions inside the kernel. Regardless of which API is used, the *alloc_pages_nodemask* function, defined in <*linux/mm/page_alloc.c*>, represents the heart of the buddy allocator. This function, which is never called directly, analyzes the zone selected for the allocation and checks if this zone is suitable for allocating the number of requested pages. If the selected zone cannot support this request, the allocator chooses another zone according to a *fallback* mechanism. The zone order for the fallback is established at boot time; usually the HIGHMEM zone falls back into the NORMAL zone, which in turn falls back within the DMA zone. When the number of free pages reaches the *pages_low* watermark, the *kswapd* manager is activated to release memory. Finally, once the zone has been selected, the *buffered_rmqueque* function (defined in *page_alloc.c*) is called to allocate the requested page block or to split a larger block satisfying the allocation request. The *alloc_pages_nodemask* returns a pointer to the struct *page* of the first allocated page.

The API *free_pages_ok*, defined in <*linux/mm/page_alloc.c*> and never called directly, implements the buddy system strategy to release pages. This function receives two key parameters: the struct pointer of the first page of the block to be released and the logarithmic dimension (order) of the block in question.

Continuous Memory Allocator Kmalloc

Kmalloc is the main method for obtaining memory from Kernel with a granularity smaller than page size (byte granularity). In particular, it allows the allocation of memory blocks with physically contiguous addresses and arbitrary length up to a maximum of 128KB.

```
void * kmalloc (size t_size , gfp_t flags) {
  struct kmem cache *s;
  void *ret;
  if (unlikely(size >KMALLOC_MAX_CACHE_SIZE))
    return kmalloc_large(size , flags);
  s = kmalloc_slab(size , flags);
  if (unlikely(ZERO _OR_NULL_PTR(s)))
    return s;
  ret = slab_alloc(s, flags , RET IP );
  trace_kmalloc( RET_IP , ret , size , s->size , flags);
  kasan_kmalloc(s, ret , size);
  return ret;
  }
```

This function is defined in <linux/slab.h> and receives two parameters:

- The *size*, which is the size in bytes of the object to be allocated (similarly to what happens for the malloc at userspace level);
- A *flag* or a *GFP type flag* mask, to specify the zone and/or the modes of allocation. The two most common flags passed to this allocation function are the GFP_ATOMIC flag and the GFP_KERNEL flag. The first is used when the allocation has high priority and consequently the caller cannot be put to sleep, but it must be served in short time. For example, an interrupt handler that requires memory must use this flag to avoid going into sleep or to avoid I/O operations. Since, in this case, the kernel cannot stop the caller and try to free up enough memory to satisfy the allocation request, a call with GFP_ATOMIC flag set has a lower probability of success than one that does not use this flag. The GFP_KERNEL flag specifies a normal kernel allocation but, with respect to the previous case, when a call to kmalloc is carried out with this flag, the caller can go to sleep; consequently it is necessary to use the GFP_KERNEL only when it is safe to do so. The kernel makes use of the ability to put the caller on sleep to free up memory, if it were necessary. A call to the kmalloc with this flag therefore has considerable chance of success.

In case of success, the kmalloc returns a *void* pointer to a memory block of the required size. In order t release memory allocated via kmalloc the kfree function is used, defined in *<linuxslab.h>*.

```
void * kfree (const void *x) {
  struct page *page;
  void *object = (void *)x;
  trace kfree( RET IP , x);
  if (unlikely(ZERO OR NULL PTR(x)))
    return;
  page = virt to head page(x);
  if (unlikely(!PageSlab(page))) {
    BUG ON(!PageCompound(page));
    kfree hook(x);
    __free_kmem_pages(page, compound order(page));
    return;
    }
  slab_free(page->slab cache, page, object, RET IP );
  }
```

Non Contiguous Memory Allocator vmalloc

If the kmalloc ensures that the pages allocated are contiguous from the point of view of both physical and virtual addresses, the vmalloc allows the allocation of pages which are contiguous only in the virtual address space.

```
void *  vmalloc {
return __vmalloc_node(size,1,gfp_mask,prot,-1,
                      __builtin_return_address(0))
  }
```

This function, defined in *<linux/vmalloc.h>* and declared in *<linux/mm/vmalloc.c>*, takes as input parameter only *size*, which corresponds to the amount of memory requested in bytes, and returns a *void** pointer to the memory allocated and mapped in kernel virtual address space. In case of error, a NULL pointer is returned. The range of addresses that can be allocated via the vmalloc is delimited by the labels VMALLOC_START and VMALLOC_END, defined in <*linux/arch/asm/pgtable.h*>. Generally this range corresponds to a reduced memory area; for example on the 32-bit x86 architecture, where the kernel address space is usually 1GB, the memory area that can be allocated by vmalloc is 128MB (Fig. 3.5). The memory allocated via *vmalloc* (or *__vmalloc*) is released through the *vfree* function, defined in <*linux/mm/vmalloc.c*>.

```
void * vfree {
BUG_ON(in interrupt());
kmemleak_free(addr);
__vunmap(addr, 1);
  }
```

This function receives as input parameter the address *addr*, starting from which the previously allocated memory is freed; if *addr* is NULL no operation is performed.

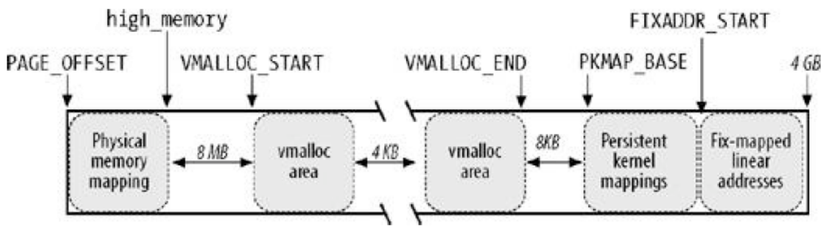


FIGURE 3.5: The linear address interval starting from PAGE_OFFSET. Source: [Bovet, 2005]

3.3 Development of approximate memory management in Linux Kernel

3.3.1 Kernel compile-time configuration menu

Two Linux kernel version have been extended with the support for approximate memory management, namely:

- Linux kernel 4.3.3 for x86 and ARM architectures (released in December 2015);
- Linux Kernel 4.20 for RISC-V (32 and 64 bits) architectures (released in 2019).

The Linux kernel presents a compile-time configuration, where it is possible to expose features and declare dependencies. One of the most common user interfaces to this configuration is *menuconfig*, a menu interface invoked through the Makefiles every time the *"make menuconfig"* command is launched. This interface presents a series of entries, each of which corresponds to a configuration option. In particular each entry has its own dependencies, which are visible only if the parent option is enabled; moreover for each item there is a short text guide to illustrate what the option in question entails and when it is convenient to set it. To introduce the capability of enabling the kernel extension concerning approximate memory support and declare dependencies, the Kconfig file in Linux/arch/<xxx> (i.e. Linux/arch/x86/Kconfig for x86 architectures) of the selected architecture has been edited adding the option to **Enable ZONE APPROXIMATE**.

```
config ZONE_APPROXIMATE
bool 'Enable approximate memory'
default n
——help——
/*This option enables an approximate area of memory ("APPROXIMATE_ZONE")
  :for every memory allocation request,based on the 'GFP_APPROXIMATE'
  flag, kernel selects the APPROXIMATE_ZONE. If APPROXIMATE_ZONE is not
  enabled, kernel selects NORMAL_ZONE.//
```

As shown in the previous text box, for the **Enable ZONE APPROXIMATE** item, the following attributes have been declared:

- *type definition*: bool. Each configuration option must have a type, which could be: *bool*, *tristate*, *string*, *hex*, *int*.
- default value: *n* indicates that this attribute is not enabled in kernel for the current option.
- *Help*: brief description.

In this way, all code for approximate memory support is implemented under the *#ifconfig* ZONE_APPROXIMATE switch, to enable approximate memory support in Linux it will be sufficient to set this option from the menuconfig menu (Fig.3.6).

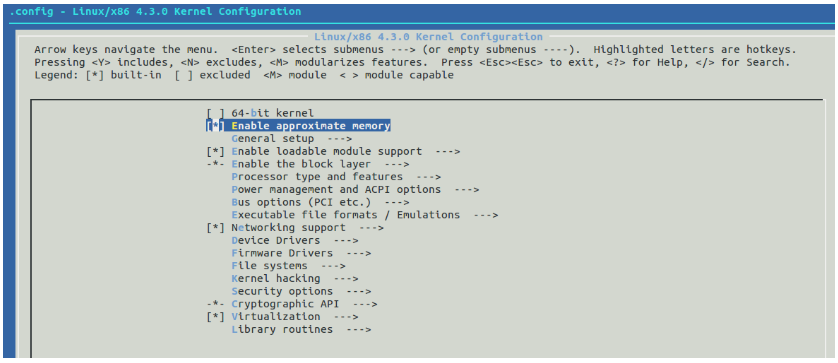


FIGURE 3.6: Example of menuconfig menu for x86 architecture

The details of kernel configuration for x86, ARM and RISC-V approximate memory support will be provided in the following sections.

3.3.2 Creation of ZONE_APPROXIMATE on 32-bit architectures

To identify the memory zones within the system, Linux uses numeric *enum* constants, defined in the header file <*include / linux / mmzone.h*>.

```
enum zone_type {
#ifdef CONFIG_ZONE_DMA
ZONE_DMA,
#endif
#ifdef CONFIG_ZONE_DMA32
ZONE_DMA32,
#endif
ZONE_NORMAL,
#ifdef CONFIG_HIGHMEM
ZONE_HIGHMEM,
#endif
ZONE_MOVABLE,
#ifdef CONFIG_ZONE_DEVICE
ZONE_DEVICE,
```

```
#endif
__MAX_NR_ZONES
};
```

In this list, an item for the ZONE_APPROXIMATE has been added; from now on, if the support for the memory approximate is enabled, the approximate memory zone is visible and identifiable by the kernel code.

```
enum zone_type {
#ifdef CONFIG_ZONE_DMA
ZONE_DMA,
#endif
#ifdef CONFIG_ZONE_DMA32
ZONE_DMA32,
#endif
ZONE_NORMAL,
#ifdef CONFIG_HIGHMEM
ZONE_HIGHMEM,
#endif
ZONE_MOVABLE,
#ifdef CONFIG_ZONE_APPROXIMATE
//ZONE_APPROXIMATE is used to isolate critical data, that must be
precise, from non-critical data,that can be approximate.
//Kernel allocates non-critical data in ZONE_APPROXIMATE.
ZONE_APPROXIMATE,
#endif
#ifdef CONFIG_ZONE_DEVICE
ZONE_DEVICE,
#endif
__MAX_NR_ZONES
};
```

The ZONE_APPROXIMATE has been defined as the last memory zone in the list on purpose, because of kernel *fallback* mechanism. According to this policy, when an allocation request is scheduled, Linux kernel checks if the selected zone is suitable to satisfy the request (e.g. there is enough space); if the zone is not suitable, the kernel allocator falls back to a hierarchically higher zone. In other words, when an allocation call will requests the ZONE_APPROXIMATE region, if memory pages in this zone were not available, the request would be satisfied by one of the hierarchically higher zones (e.g. ZONE_NORMAL). This would result in storing approximate data in exact memory and cancel possible energy savings, but the functionality of the application would not be compromised. Moreover, ZONE_APPROXIMATE must be the last one in hierarchy also if we consider allocation requests for exact memory (ZONE_NORMAL or ZONE_DMA). In case of a normal (i.e. exact) allocation request, if physical memory pages in ZONE_NORMAL could not satisfy the request, the kernel must never select ZONE_APPROXIMATE pages as alternative. In this way critical data, that must be exact, will never be stored in approximate memory.

In addition to a numeric constant, each zone is also identified by a string corresponding to the zone name. In <*linux/mm/page_alloc.c*> file, the array *zone_names* of *MAX_NR_ZONES* char pointer elements is defined.

```
static char * const zone_names[MAX_NR_ZONES] = {
#ifdef CONFIG_ZONE_DMA
"DMA",
#endif
#ifdef CONFIG_ZONE_DMA32
"DMA32",
#endif
"Normal",
#ifdef CONFIG_HIGHMEM
```

```
    "HighMem",
    #endif
    "Movable",
    #ifdef CONFIG_ZONE_APPROXIMATE
    "Approximate",
    #endif
    #ifdef CONFIG_ZONE_DEVICE
    "Device",
    #endif
    };
```

In order to properly associate each zone to the corresponding name, the order in which strings are declared must match the order in which the numeric constants are associated to the zones.

Finally, another important step is required to complete the creation of ZONE_APPROXIMATE: the association of a physical address range to this zone. This step, as well as the layout of each memory zone, is architecture dependent, so there may be different sizes and layouts for approximate memory zones, depending on the architecture.

It should be noticed that, as specified in <include/linux/pageflags.h>, on 32-bit architectures it is possible to enable at the same time up to four memory zones (including ZONE_NORMAL and ZONE_MOVABLE, that are always present). This is due to the fact that the ZONE_SHIFT macro, used in <include/linux/gfp.h> to build a zone table for the identification of all the activated memory zones, has to be less than 2 on 32-bits architectures. This in an important limitation and it is due to maintain an efficient implementation of the whole OS memory management. As we will see, it has been overcome in 64-bit architectures.

```
#if MAX_NR_ZONES < 2
#define ZONES_SHIFT 0
#elif MAX_NR_ZONES <= 2
#define ZONES_SHIFT 1
#elif MAX_NR_ZONES <= 4
#define ZONES_SHIFT 2
#else
#error ZONES_SHIFT //too many zones configured adjust calculation
#endif
```

The description of architecture-specific approximate memory zone support (x86, ARM and RISC-V) is addressed in the next subsections.

ZONE_APPROXIMATE on x86 architectures

The layout of Linux memory zones in 32-bits x86 architectures is illustrated in Table 3.2.

TABLE 3.2: 32-bit x86 architecture memory layout

Zone	*Description*	*Physical Memory*
ZONE_DMA	DMA pages	<16M
ZONE_NORMAL	pages that could be normally addressed	896M
ZONE_HIGHMEM	pages dynamically mapped	> 896M

It can be observed that, in this architecture, there are four memory zones that are enabled by default. Since, as said before, on all 32-bit architectures it is possible to enable only up to four memory zones at the same time and since ZONE_NORMAL and ZONE_MOVABLE must always be present, in order to enable ZONE_APPROXIMATE, ZONE_HIGHMEM has been disabled. Specifically, it has been chosen to disable high memory support preferring to keep active the memory zone reserved for DMA transfers. This operation is performed in the configuration step through the *menuconfig* option; in particular from the *Processor type and features* menu, the item *High Memory Support* has been set to *OFF*. By disabling high memory, also PAE (*Physical Adress Extension*) support is automatically set to off; this feature is used in some x86 processors to address more physical memory than the 4GB limit of 32-bit architectures.

ZONE_APPROXIMATE is mapped in the remaining memory space, usually assigned completely to ZONE_NORMAL. The implementation that has been chosen consists in splitting in half the memory space, in order to have the first 440MB of exact memory and the following 440MB of approximated memory, as shown in Table 3.3.

TABLE 3.3: 32-bit x86 memory layout with ZONE_APPROXIMATE

Zone	*Description*	*Physical Memory*
ZONE_DMA	DMA pages	<16M
ZONE_NORMAL	pages that could be normally addressed	16 - 456M
ZONE_APPROXIMATE	pages for non critical data	456 - 896M

To implement this scheme, the following source files have been modified:

- <arch/x86/mm/init.c>
- <mm/nobootmem.c>
- <include/linux/bootmem.h>
- <arch/x86/setup/kernel.c>
- <mm/page_alloc.c>

In <arch/x86/mm/init.c> the *__init zone_sizes_init* function is defined. This routine allows to associate to each memory zone the upper bound of its physical memory range. Specifically, the array *max_zone_pfns* of *NR_MAX_ZONES* (with NR_MAX_ZONES equal to 4) elements is instantiated. For each element of the array, indexed with the constants zone type previously defined, a *pfn* variable, corresponding to the upper bound of the zone, is assigned.

```
  void __init zone_sizes_init(void){
  unsigned long max_zone_pfns[MAX_NR_ZONES];
  memset(max_zone_pfns, 0, sizeof(max_zone_pfns));
#ifdef CONFIG_ZONE_DMA
  max_zone_pfns[ZONE_DMA] = min(MAX_DMA_PFN, max_low_pfn);
#endif
#ifdef CONFIG_ZONE_DMA32
  max_zone_pfns[ZONE_DMA32] = min(MAX_DMA32_PFN, max_low_pfn);
#endif
  max_zone_pfns[ZONE_NORMAL] = max_normal_pfn;
#ifdef CONFIG_ZONE_APPROXIMATE
  max_zone_pfns[ZONE_APPROXIMATE] = max_low_pfn;
#endif
ifdef CONFIG_HIGHMEM
  max_zone_pfns[ZONE_HIGHMEM] = max_pfn;
#endif
  free_area_init_nodes(max_zone_pfns);
}
```

All PFN (*Page Frame Number*)-type variables are declared in the <linux /mm/nobootmem.c> and <linux/bootmem.h> files. These variables are used inside the Linux Kernel to store physical addresses in page units, defined generally as blocks of 4KB size.

Considering the order in which the zone type constants are defined in *mmzone.h*, the third element of the array is associated to the upper bound of zone approximate (*max_zone_pfn*). The latter corresponds to end of the low memory and, before the introduction of approximate memory support, constituted the upper bound of zone normal. Since, as mentioned previously, the approximate region must be derived from the physical memory range associated to ZONE_NORMAL, a new variable *max_normal_pfn*, initialized within the file <arch/x86/kernel/setup.c>, has been defined in the files <linux/mm/nobootmem.c> and <linux/bootmem.h>. In particular for a 32-bit x86 architecture, the variable *max_low_pfn* is initialized by calling the *find_low_pfn_range*function; once the value of *max_low_pfn* has been determined, the variable *max_normal_pfn* is also initialized. If the support for the approximate memory is enabled then *max_normal_pfn* is set to one half of *max_low_pfn*, otherwise, being present only the zone normal, the two variables coincide.

```
#ifdef CONFIG_X86_32
  // max_low_pfn get updated here
  find_low_pfn_range();
#else
  check_x2apic();
  // How many end-of-memory variables you have, grandma! //
  // need this before calling reserve_initrd //
  if (max_pfn > (1UL<<(32 - PAGE\_SHIFT)))
    max_low\_pfn = e820_end_of_low_ram_pfn();
  else
    max_low\_pfn = max_pfn;
  high_memory = (void *)__va(max_pfn * PAGE_SIZE - 1) + 1;
#endif
#ifdef CONFIG_ZONE_APPROXIMATE
  max_normal_pfn=(max_low_pfn)/2
#endif
```

Once the elements of max_zone_pfns have been initialized, the array is passed to the *free_area_init_nodes* function, (defined in <mm/page alloc.c>), in order to determine the lower bounds (identified by the array *arch_zone_lowest_possible_pfn*) of physical memory ranges associated to each zone. First it is necessary to compute the bounds of the zone identified by the numeric constant 0: the lower bound is get by calling the function *find_min_pfn_with_active_regions*, while the upper bound corresponds to the first element of the array *max_zone_pfn*. For the following zones, the interval is computed by computing, for the i-th zone:

- as upper bound the i-th element of the array *max_zone_pfns*;
- as lower bound the element (i-1) of the array *arch_highest_possible_pfn*, corresponding to the upper bound of the previous zone.

If the i-th zone corresponds to ZONE_MOVABLE, since the it is a fictitious zone and it doesn't have its own range of physical memory, the process is skipped. Therefore in case of ZONE_APPROXIMATE, considering that it must be the last memory zone, its bounds are defined in the following way:

- the lower bound corresponds to the upper bound of ZONE_NORMAL (i-nd element of *arch_highest_possible_pfn*);

- the upper bound corresponds to max_low_pfn, as said previously.

```
void __init free_area_init_nodes(unsigned long *max_zone_pfn)
{
  unsigned long start_pfn, end_pfn;
  int i, nid;
  //Record where the zone boundaries are
  memset(arch_zone_lowest_possible_pfn, 0,
        sizeof(arch_zone_lowest_possible_pfn));
  memset(arch_zone_highest_possible_pfn, 0,
        sizeof(arch_zone_highest_possible_pfn));
  arch_zone_lowest_possible_pfn[0] = find_min_pfn_with_active_regions();
  arch_zone_highest_possible_pfn[0] = max_zone_pfn[0];
  for (i = 1; i < MAX_NR_ZONES; i++) {
    if (i == ZONE_MOVABLE)
      continue;
#ifdef CONFIG_ZONE_APPROXIMATE
    if((i-1)==ZONE_MOVABLE)
      arch_zone_lowest_possible_pfn[i] =
      arch_zone_highest_possible_pfn[i-2];
    else
      arch_zone_lowest_possible_pfn[i] =
      arch_zone_highest_possible_pfn[i-1];
#else
    arch_zone_lowest_possible_pfn[i] =
    arch_zone_highest_possible_pfn[i-1];
#endif
    arch_zone_highest_possible_pfn[i] =
    max(max_zone_pfn[i], arch_zone_lowest_possible_pfn[i]);
    pr_info("zone:%d max_zone_pfn:%lu \n", i, max_zone_pfn[i]);
    pr_info("zone:%d arch_zone_lowest_possible_pfn%lu \n", i,
    arch_zone_lowest_possible_pfn[i]);
  }
```

This final operation completes the creation and configuration of an approximate memory zone on 32-bits x86 architectures. As for the other memory zones, the approximate one now has its own physical memory range and it can be identified in the kernel by the string name or by the corresponding constant zone type. As first test, after a system boot of the modified kernel, it is possible to examine the output of the kernel *ring buffer* (Fig. 3.7) in order to check RAM memory mappings and zone ranges for x86. These logs come from the *dmesg* command and show that ZONE_APPROXIMATE has been properly created: the new zone is the last memory zone, after ZONE_DMA and ZONE_NORMAL, and it is mapped in physical memory at range 0x1bf8a000-0x37f13fff.

More information concerning ZONE_APPROXIMATE can be obtained through the *cat /proc/zoneinfo* command (Fig. 3.8).

For each zone, the memory node (in Fig. 3.8 only node 0 is present because the target architecture is not NUMA) and some statistics concerning memory zone pages are reported. On zone approximate 114570 pages are present; considering that every page on a 32-bit x86 architecture has a size of 4K, it is possible to obtain another confirmation that 440 MB of RAM has been assigned to the approximate zone.

ZONE_APPROXIMATE on ARM architectures

As said previously, also for ARM architectures ZONE_HIGHMEM has been disabled.

```
cat /proc/dmesg
...
895MB LOWMEM available
mapped low ram:  0 - 37f14000
low ram:  0 - 37f14000
Zone ranges:
DMA [mem 0x0000000000001000-
0x0000000000ffffff]
Normal [mem 0x0000000001000000-
0x000000001bf89fff]
Approximate [mem
0x000000001bf8a000-
0x0000000037f13fff]
...
```

FIGURE 3.7: Output of *dmesg* command

```
cat /proc/zoneinfo
...
Node 0, zone Approximate
pages free 113570
spanned 114570
present 114570
nr_dirtied 0
nr_written 0
...
```

FIGURE 3.8: Output of *cat /proc/zoneinfo* command

Mapping between memory zones and physical address ranges is done in the <arch/arm/mm/init.c> source file. In particular, in the *__init zone_sizes_init* function the array *max_zone_pfns* of MAX_NR_ZONES elements is defined; each element of the array is indexed by the zone constant identifier and stores the upper bound of the corresponding memory zone. The lower bound of a zone corresponds to the upper bound of the previous zone, while for the first zone it corresponds to the memory start address of the specific ARM architecture.

```
void __init zone_sizes_init(void)
{
  unsigned long max_zone_pfns[MAX_NR_ZONES];

  memset(max_zone_pfns, 0, sizeof(max_zone_pfns));

#ifdef CONFIG_ZONE_DMA
  max_zone_pfns[ZONE_DMA]     = min(MAX_DMA_PFN, max_low_pfn);
#endif
#ifdef CONFIG_ZONE_DMA32
  max_zone_pfns[ZONE_DMA32]  = min(MAX_DMA32_PFN, max_low_pfn);
#endif
  max_zone_pfns[ZONE_NORMAL]   = max_approx_pfn;

#ifdef CONFIG_ZONE_APPROXIMATE
  max_zone_pfns[ZONE_APPROXIMATE] = max_low_pfn;
#endif

```

```
#ifdef CONFIG_HIGHMEM
  max_zone_pfns[ZONE_HIGHMEM] = max_pfn;
#endif
```

In order to implement the mapping of ZONE_APPROXIMATE, the *max_approx_pfn* variable has been introduced; this one corresponds to the end of ZONE_NORMAL and so to the start address of the approximate memory region.

- DEVICE TREE FOR APPROXIMATE MEMORY

On ARM architectures, for all new SoCs from 2012, it has become mandatory to describe the hardware components of the system in a data structure called *device tree*. In particular, during the boot process, a 'Device Tree Blob' (DTB) file is loaded into memory by the bootloader and passed to the Linux kernel. This DTB file is a tree data structure containing nodes that describe the system hardware layout to the Linux kernel, allowing for platform-specific code to be moved out of kernel sources and replaced with generic code that can parse the DTB and configure the entire system as required (Fig. 3.9).

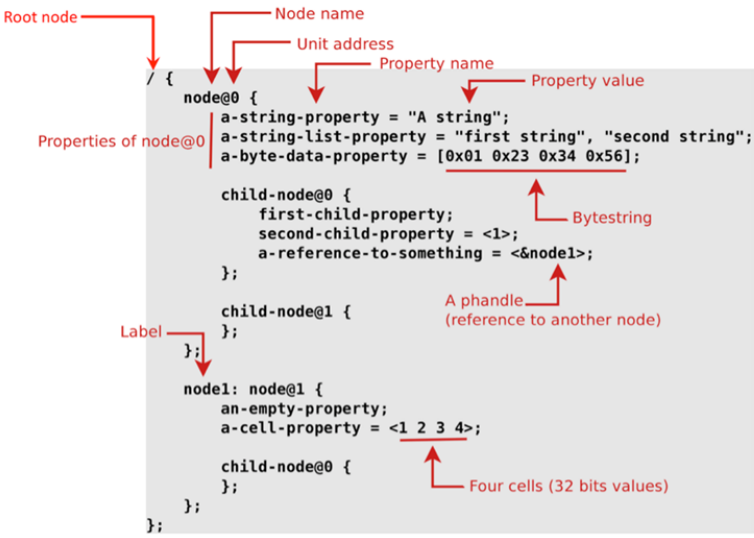


FIGURE 3.9: Device tree structure

For ARM architectures all device tree sources are located at the path <arch/arm/boot/dts/...>. Each physical device is indeed described inside the device tree, in particular it is represented as a node and all its properties are defined under that node.

The */memory* node provides information about addresses and size of physical memory. This node is usually filled or updated by the bootloader, depending on the actual memory configuration of the given platform. The memory layout is therefore described by this node, which presents two attributes:

1. *baseaddrX* that indicates the base address of the defined memory bank;

2. *sizeX* that corresponds to the size of the defined memory bank.

In order to support approximate memory management in ARM architectures, a new DTS file (from which the DTB is generated) has been defined; this file is specific for each ARM platform and a special node for approximate memory has been added (Fig.3.10). In particular, this node called *approx_mem*, collects the information about the physical address range and the size of approximate memory.

```
approx_mem {
device_type = memory  reg =
<(baseaddr1)  (size1)
(baseaddr2)  (size2)
...
(baseaddrN)  (sizeN)>;
};
```

FIGURE 3.10: Example of approximate memory node in DTB file

- ARM INTERRUPT VECTORS

Another critical step for supporting adding the approximate memory region on ARM architecture is its interference with the initialization of ARM interrupt vectors, carried out very early in ARM system boot through the *devicemaps_init()* routine (source file <arch/arm/mm/mmu.c>).

In order to boot the primary core, the kernel allocates a single 4KB page as vector page, mapping it to the location of ARM exception vectors at virtual address 0xFFFF0000 or 0x00000000. When this step is completed, the *trap_init* function copies the exception vector table, exception stubs, and helpers from *entry-arm.S* into the vector page.

In particular, the allocation of the ARM vectors page is performed by the *early_alloc* function allocator. This allocator cannot exclude approximate memory, since it does not allow to specify the memory zone where the allocation should be satisfied. In order to ensure that the vectors page is never allocated in approximate memory, the implementation of a new *early_alloc* is required. The new *early_alloc* uses the *memblock* interface (this topic will be discussed in section 3.3.3) and it allows to explicitly set an address limit (*approx_limit* variable) for the allocation request, in order to exclude approximate memory. This approx_limit, which corresponds to a physical address, must be consistent with the start of approximate memory indicated in the DTS file.

```c
#ifdef CONFIG_ZONE_APPROXIMATE
static void __init *early_alloc_aligned(unsigned long sz, unsigned long
    align)
{
  void *ptr = __va(memblock_alloc_base(sz, align, approx_limit));
  memset(ptr, 0, sz);
  return ptr;
}
```

ARM Versatile Express (Vexpress) Cortex A9 has been chosen as the architecture for performing tests. These tests were run on the emulation platform AppropinQuo (see Chapter 4).

Fig. 3.11 shows the memory map of Vexpress Cortex A9, RAM memory can be present from 0x60000000 to 0x80000000 and from 0x84000000 to 0xA000000. For the tests it was chosen to a configuration of 128MB+128MB of RAM: the first 128MB part is exact, starting from address 0x60000000 to address 0x67FFFFFF and the second 128MB part, from address 0x68000000 to 0x6FFFFFFF, corresponds to approximate memory. This memory map was used to configure the emulator and also to set the corresponding kernel *dts* file. Fig. 3.12 (left) shows the statistics for the ZONE_NORMAL region, obtained through the *zoneinfo* system command. The region contains exact memory and it is composed of 32768 pages; considering that every page in the ARM architecture is 4KB large, it confirms the availability of

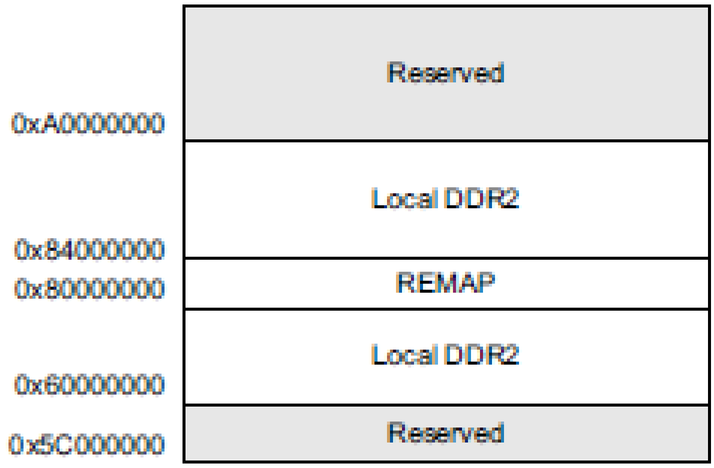


FIGURE 3.11: Vexpress Cortex A9 board memory map (extract)

128MB of exact memory. The start_pfn number indicates the start address of exact memory in physical pages ($393219 \times 4096 = 0x60000000$). Fig. 3.12 (right) shows the same statistics for the ZONE_APPROXIMATE. The approximate area has 32768 pages; again this confirms that the approximate region area is 128MB large. The start_pfn number indicates the start address of approximate at address 0x68000000 ($425984 \times 4096 = 0x68000000$).

```
cat /proc/zoneinfo
Node 0, zone Normal
pages free 28036
spanned 32768
present 32768
min 167
low 208
high 250
scanned 0
...
start_pfn:  393216
```

```
cat /proc/zoneinfo
...
Node 0, zone Approximate
pages free 32768
spanned 32768
present 32768
min 180
low 225
high 270
scanned 0
...
start_pfn:  425984
```

FIGURE 3.12: On left: kernel boot logs. On right: zone_approximate statistics

ZONE_APPROXIMATE on RISC-V 32-bit architectures

The RISC-V processor support has been introduced in Linux kernel starting from version 4.15. The mainline kernel supports the *VIRT-IO* board and the *HiFive Unleashed* board, which features the SiFive Freedom U540 SoC, 8GB DDR4, 32MB quad SPI, and micro-SD card.

The mapping between memory zones and physical ranges is done in the <arch/riscv/mm/init.c> source file. In particular, as for ARM architectures, the *__init zone_sizes_init* function defines the array *max_zone_pfns* of MAX_NR_ZONES elements. Again, each element of the array is indexed by the zone constant identifier and associated to a *pfn* variable, storing the upper bound of the corresponding memory zone. The lower bound of the zone corresponds to the upper bound of the previous zone, while for the first zone it is defined as the DRAM start address which is, for RISC-V boards, the address 0x8000000.

```
static void __init zone_sizes_init(void)
{
  unsigned long max_zone_pfns[MAX_NR_ZONES] = { 0, };

#ifdef CONFIG_ZONE_DMA32
  max_zone_pfns[ZONE_DMA32] = PFN_DOWN(min(4UL * SZ_1G, max_low_pfn));
#endif
#ifdef CONFIG_ZONE_APPROXIMATE
    max_low_pfn = max_low_pfn / 4096;
    #define start_pfn_normal_riscv     0x80000
    #define max_normal_riscv (((max_low_pfn - start_pfn_normal_riscv)/2) +
     start_pfn_normal_riscv)
    max_zone_pfns[ZONE_NORMAL] = max_normal_riscv;
    max_zone_pfns[ZONE_APPROXIMATE] = max_low_pfn;
#else
  max_zone_pfns[ZONE_NORMAL] = max_low_pfn;
#endif

  free_area_init_nodes(max_zone_pfns);
}
```

On RISC-V 32-bits architecture, only the ZONE_NORMAL is enabled by default; ZONE_DMA32 instead should be enabled, as mentioned in the kernel configuration files, only on RISC-V 64-bit architectures. In order add *ZONE_APPROXIMATE* to the memory map, the *start_pfn_normal_riscv* variable has been introduced; this variable corresponds to the offset, with respect to the DRAM start address, from which the approximate memory should be mapped. The boot messages, showing the RISC-V memory mapping, are illustrated in Fig. 3.13.

```
Zone ranges:  Normal [mem 0x0000000081000000-0x0000000087ffffff]
Approximate [mem 0x0000000088000000-0x000000008fffffff]
Movable zone start for each node
Early memory node ranges
node 0:  [mem 0x0000000080200000-0x000000008fffffff]
```

FIGURE 3.13: RISC-V Boot messages

3.3.3 Approximate Memory and Early Boot Allocators

Linux early boot allocators are an important function of the kernel and are used during the boot process in order to allocate data structures in the initial phase of system startup, before the main allocators are instantiated. For ARM architectures, the initialization of all physical zones, including ZONE_APPROXIMATE, takes place in *bootmem_init* function. This routine determines the limits of all available physical memory (PFN limits) and sets up the early memory management subsystem. After this process, pages allocated by the boot allocator are freed and physical zone limits, including ZONE_APPROXIMATE, are determined.

At start-up Linux kernel gains access to all physical memory available in the system. Before memory zone allocator is set up and running, it can be necessary to preallocate some initial memory areas for kernel data structures and system-wide use, taking them from available RAM. To address this requirement, a special allocator called *bootmem allocator* or *memblock allocator*, is used. The initialization of this early allocator is architecture dependent and it is set up in *setup_arch* routine (Fig. 3.14).

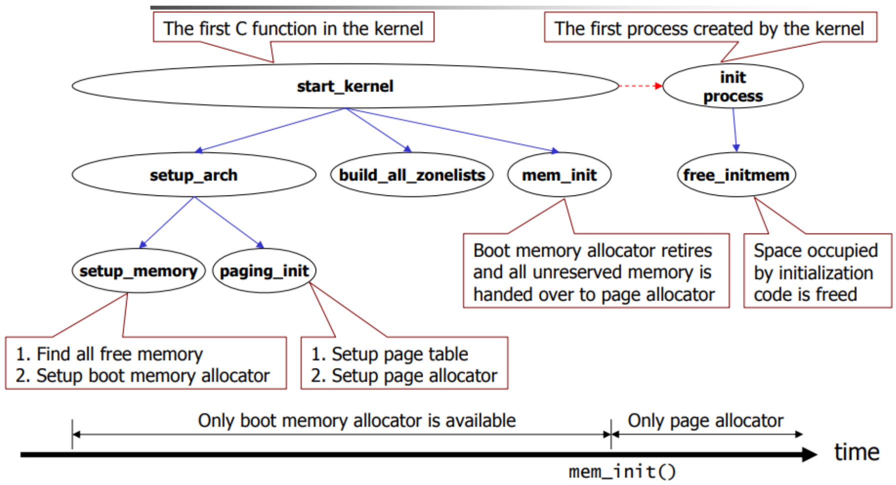


FIGURE 3.14: Overview of memory allocators. Source:[Liu, 2010]

Once the boot memory management is available, it can allocate areas from low memory (memory directly mapped in Kernel space), with page granularity. To keep track of reserved and free pages, the bootmem allocator uses a bitmap: each bit in this bitmap represents one page and its index in the bitmap represents the page frame number. In particular a value of '0' in any bit in the bitmap indicates that the corresponding page is free while a value of '1' indicates that the page is in use. This bitmap, representing all pages available to the bootmem allocator, is created by *init_bootmem_core()*; in particular this function takes as input parameter the address beyond end of kernel to make sure it doesn't overwrite kernel text or data. This bitmap is used to manage only *low memory*. Later in the startup code, the bootmem allocator bitmap is used to determine which memory pages are in use. The corresponding page structures are marked as reserved. After paging is enabled, the bootmem allocator is not needed and so the allocator bitmap is freed. The early allocator is used only at boot time to reserve and to allocate pages for internal kernel use. For example, *page tables* are built from this pool of physical memory pages, allowing the MMU to be turned on and Linux kernel to switch to virtual memory management.

The whole mechanism requires that the kernel must be aware of approximate memory in the early boot phases: approximate memory must be visible in order to properly instantiate paging and main allocators, but must not be used for kernel data structures, which contains critical data that cannot be sukject to any form of corruption. In order to exclude physical memory pages mapped as approximate from early allocation, the algorithm of the *memblock* interface has been modified. A *memblock* is a structure that stores information of physical memory regions reserved by Linux kernel during the early bootstrap stage. In particular, *memblock* manages two regions: *memblock.memory* and *memblock.reserved*. All available physical memory is added to the *memblock.memory* region and each time data structures are allocated, their portion of memory is added to the *memblock.reserved* region.

The core function of this early allocation is *memblock_virt_alloc_internal*, which then calls *memblock_find_in_range_node* (Fig.3.16).

This routine receives as parameters, among others, the requested memory size and the lower and upper bounds of the physical region where the memory block will be allocated. At first, the allocation starts from the lower address bound and the following allocation requests will proceed to lowest available address starting from

```
+-------------------------+   +-------------------------+
|        memblock         |   |       Array of the      |
|  _____________________  |   |     memblock_region     |
| |        memory       | |   |                         |
| |     memblock_type   |-|-->| [start1, end1)          |
| |_____________________| |   |                         |
|                         |   | [start2, end2)          |
|                         |   |                         |
|                         |   | [start3, end3)          |
|                         |   |                         |
|                         |   +-------------------------+
|                         |
|  _____________________  |   +-------------------------+
| |       reserved      | |   |       Array of the      |
| |     memblock_type   |-|-->|     memblock_region     |
| |_____________________| |   |                         |
|                         |   |  [start1, end1)         |
|                         |   |                         |
|                         |   |  [start2, end2)         |
|                         |   |                         |
+-------------------------+   +-------------------------+
```

FIGURE 3.15: Memblock memory allocation

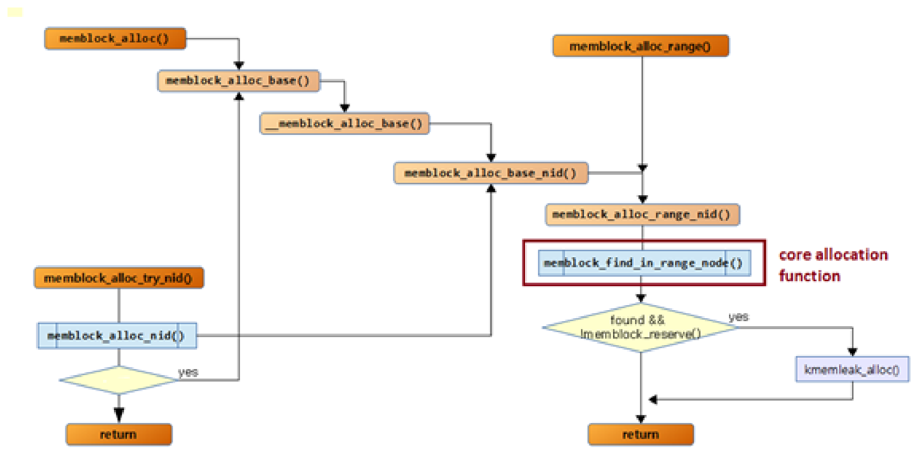


FIGURE 3.16: Memblock allocator function tree

the lower bound. The upper bound instead corresponds to the end of the candidate physical memory range and, if it coincides with the MEMBLOCK_ALLOC_ACCESSIBLE, it is set to the value of the global parameter *memblock_current_limit*, which is set to the end of *low memory* region, forcing early boot allocation within the *low memory* region, that is the only region the kernel can directly access.

In order to include and support the presence of approximate memory, the algorithm has been modified for computing *memblock_current_limit* (Fig.3.17), forcing it to be always below the lower limit of the approximate memory physical region. In this way it is ensured that the *bootmem* allocator gets free pages only from exact physical memory.

In order to check the correctness of the new code, it is necessary to analyze the messages produced by kernel during the boot phase. Considering the zoneinfo messages analyzed before, particular importance comes from information regarding '*present*' and '*managed*' lines. The latter corresponds to pages managed by the buddy system (the main allocator); they are computed as the number of present pages minus the number of reserved pages, including those allocated by the bootmem allocator.

managed pages = *present pages* - *reserved pages*

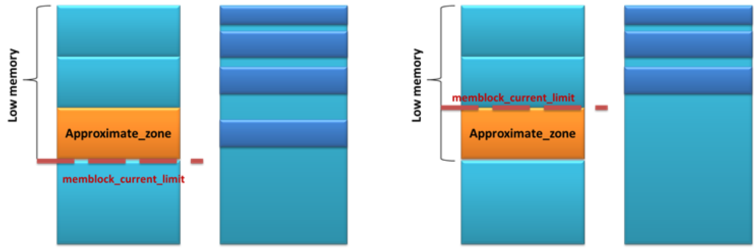


FIGURE 3.17: Memblock current limit on architectures with ZONE_APPROXIMATE

Fig. 3.18 shows, for example, the zoneinfo messages for ARM Vexpress architecture: since the number of present pages always matches the number of managed pages, during the boot phase no pages belonging to the zone_approximate were allocated.

```
cat /proc/zoneinfo
...
Node 0, zone Approximate
pages free 32768
min 180
low 225
high 270
scanned 0
spanned 32768
present 32768
managed 32768
...
```

FIGURE 3.18: Output of *cat /proc/zoneinfo* command

3.4 Allocation in ZONE_APPROXIMATE

In this section the implementation of a custom allocator to dynamically allocate non-critical data on ZONE_APPROXIMATE is described.

3.4.1 Approximate GFP Flags

All internal functions provided by the kernel to allocate memory pages get a mask of *gfp_flag*s as parameter, which allows to drive the behavior of the allocator. In order to be able to specify ZONE_APPROXIMATE as favorite zone for allocation, it is necessary to define a new *gfp_flag*, that we called GFP_APPROXIMATE: when this flag is set, the kernel tries to allocate memory using ZONE_APPROXIMATE pages.

```
//Plain integer GFP bitmasks. Do not use this directly.
#define ___GFP_DMA 0x01u
#define ___GFP_HIGHMEM 0x02u
#define ___GFP_DMA32 0x04u
#define ___GFP_MOVABLE 0x08u
#define ___GFP_WAIT 0x10u
#define ___GFP_HIGH 0x20u
#define ___GFP_IO 0x40u
#define ___GFP_FS 0x80u
```

```
#define ___GFP_COLD 0x100u
#define ___GFP_NOWARN 0x200u
#define ___GFP_REPEAT 0x400u
#define ___GFP_NOFAIL 0x800u
#define ___GFP_NORETRY 0x1000u
#define ___GFP_MEMALLOC 0x2000u
#define ___GFP_COMP 0x4000u
#define ___GFP_ZERO 0x8000u
#define ___GFP_NOMEMALLOC 0x10000u
#define ___GFP_HARDWALL 0x20000u
#define ___GFP_THISNODE 0x40000u
#define ___GFP_RECLAIMABLE 0x80000u
#define ___GFP_NOACCOUNT 0x100000u
#define ___GFP_NOTRACK 0x200000u
#define ___GFP_NO_KSWAPD 0x400000u
#define ___GFP_OTHER_NODE 0x800000u
#define ___GFP_WRITE 0x1000000u
#ifdef CONFIG_ZONE_APPROXIMATE
#define ___GFP_APPROXIMATE 0x2000000u
#endif
```

As shown in the text box, at first a ___GFP_APPROXIMATE bit-mask, corresponding to the integer value 0x2000000u, is defined. This mask should not be called directly by the kernel code and should not be used in any conditional constructs. Consequently, within the *gfp.h* file, a bitwise __GFP_APPROXIMATE is defined by casting with a type gfp_t the bitmask ___GFP_APPROXIMATE defined previously.

```
#define __GFP_DMA ((__force gfp_t)___GFP_DMA)
#define __GFP_HIGHMEM ((__force gfp_t)___GFP_HIGHMEM)
#define __GFP_DMA32 ((__force gfp_t)___GFP_DMA32)
#define __GFP_MOVABLE ((__force gfp_t)___GFP_MOVABLE) /* Page is
movable */
#ifdef CONFIG_ZONE_APPROXIMATE
#define __GFP_APPROXIMATE ((__force gfp_t)___GFP_APPROXIMATE)
#endif
#define GFP_ZONEMASK (__GFP_DMA|__GFP_HIGHMEM|__GFP_DMA32|__GFP_MOVABLE)
```

Finally, starting from __GFP_APPROXIMATE, the flag GFP_APPROXIMATE is defined. This zone modifier, as anticipated, is used to select the approximate area as target for the allocation request.

```
#ifdef CONFIG_ZONE_APPROXIMATE
#define GFP_APPROXIMATE __GFP_APPROXIMATE
#endif
```

The introduction of the GFP bitmask for the approximate zone involves a redefinition of the value of __GFP_BITS_SHIFT, a constant required to define the __GFP_BITS_MASK, used universally in kernel source files.

```
#ifdef CONFIG_ZONE_APPROXIMATE
#define __GFP_BITS_SHIFT 26
#else
#define __GFP_BITS_SHIFT 25 /* Room for N __GFP_FOO bits */
#endif
#define __GFP_BITS_MASK ((__force gfp_t)((1 << __GFP_BITS_SHIFT) - 1))
```

As the bitmask __GFP_APPROXIMATE corresponds the bit 25 in a 32-bit word, if approximate memory is enabled then the GFP_BITS_SHIFT should be set to 26. As mentioned above, the GFP flags are always passed as parameter to the functions that manage page level allocation; in particular all the kernel APIs concerning the allocation mechanism propagate these flags up to the core function *alloc_pages_nodemask*, defined in <linux/mm/page_alloc.c>.

```
__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
struct zonelist *zonelist, nodemask_t *nodemask)
{
struct zoneref *preferred_zoneref;
struct page *page = NULL;
unsigned int cpuset_mems_cookie;
int alloc_flags = ALLOC_WMARK_LOW|ALLOC_CPUSET|ALLOC_FAIR;
gfp_t alloc_mask; //The gfp_t that was actually used for allocation
struct alloc_context ac = {
.high_zoneidx = gfp_zone(gfp_mask),
.nodemask = nodemask,
.migratetype = gfpflags_to_migratetype(gfp_mask),
};
[...]
```

This function performs the initialization of the *alloc_context* type struct *ac*, which stores the main information for the management of the allocation mechanism. In particular, the field *high_zoneidx*, corresponding to the zone chosen to satisfy the allocation request, is initialized using the return value of the *gfp_zone* function. The latter in fact, depending on the*gfp_bit_mask* that receives as a parameter, identify the memory zone where the allocation should be performed. In order to select the approximate zone, the implementation of this function, defined in <include/linux/gfp.h>, must be modified.

```
  static inline enum zone_type gfp_zone(gfp_t flags)
  {
    enum zone_type z;
    if((flags & GFP_APPROXIMATE)==0) {
      int bit = (__force int) (flags & GFP_ZONEMASK);
        z = (GFP_ZONE_TABLE >> (bit * ZONES_SHIFT)) &
              ((1 << ZONES_SHIFT) - 1);
        VM_BUG_ON((GFP_ZONE_BAD >> bit) & 1);
    }
    else
      z= ZONE_APPROXIMATE;
  return z;
}
```

This function checks the result of the AND operation between the gfp flag *flags*, received as parameter, and the GFP_APPROXIMATE flag: if the result is 0, the requested zone is not the approximate one and the identification of the zone for allocation is done by reading two bits in the GFP_ZONE_TABLE, after scanning a number of positions equal to the product between the ZONES_SHIFT and a constant derived from *flags*. Otherwise, if the resulting value is 1, then the zone requested for the allocation can be only ZONE_APPROXIMATE. This mechanism, based on the definition of the GFP_APPROXIMATE flag, allows to allocate pages in approximate memory only on request: if the Kernel APIs which manage the allocation requests do not receive the GFP_APPROXIMATE flag, the zone approximate will never be selected.

Alloc Fair policy

In order to schedule the allocation requests, Linux defaults to the *fair allocation* policy. According to this policy, kernel tries to balance allocation requests by interleaving them between enabled zones, avoiding that a zone is saturated before other zones. Leaving the original policy, requests for ZONE_APPROXIMATE memory could be diverted to ZONE_NORMAL or ZONE_DMA even before ZONE_APPROXIMATE is full, resulting in sub-optimal allocation strategy. In order to properly handle allocation requests for ZONE_APPROXIMATE, kernel allocation fair policy should be disabled for ZONE_APPROXIMATE (i.e. when GFP_APPROXIMATE flag is set).

```
int alloc_flags = ALLOC_WMARK_LOW|ALLOC_CPUSET|ALLOC_FAIR;
gfp_t alloc_mask;
struct alloc_context ac = {
.high_zoneidx = gfp_zone(gfp_mask),
.nodemask = nodemask,
.migratetype = gfpflags_to_migratetype(gfp_mask),
};
if((gfp_mask & GFP_APPROXIMATE) != 0)
  alloc_flags&= ~ALLOC_FAIR;
```

This point can be accomplished in the *alloc_page_nodemask* function, when the allocation request is processed. In particular, as shown in the box above, the mask *alloc_flags* is modified by disabling the corresponding ALLOC_FAIR flag, responsible for the activation of the policy. The definition of GFP_APPROXIMATE flag and the changes to the fallback mechanism and fair allocation policy allow to block all allocations of ZONE_APPROXIMATE pages apart from explicit requests: i.e. if the kernel allocator routines do not get GFP_APPROXIMATE flag explicitly set as parameter, ZONE_APPROXIMATE will never be selected as memory zone to satisfy the allocation request.

3.4.2 User level approximate memory allocation

Physical pages within ZONE_APPROXIMATE can be allocated only on request, meaning that the Linux kernel can select the ZONE_APPROXIMATE to satisfy the allocation request only when the GFP_APPROXIMATE flag is set (cfr 3.4.1). Consequently, in order to request pages belonging to the approximate memory zone, it is necessary to find a system call that takes a GFP flag of type *zone modifier* as input parameter and that propagates this flag into the allocation API down to the page allocator core function (*alloc_pages_nodemask*). There are two functions inside the kernel that allows to allocate memory specifying directly a *gfp* flag: the *kmalloc* function and the *vmalloc* function. Since pages allocated by *kmalloc* are contiguous not only in the virtual address space but also in the physical address space (see Section 3.2.4), an allocation mechanism based on *vmalloc*, which allows to obtain pages that are contiguous only in the virtual address space, has been implemented. Moreover, as this function can be only directly invoked within the kernel source files, it is necessary to call it in a new kernel space routine, which then could be exported to user space level allowing user space applications to request pages in ZONE_APPROXIMATE. As for the allocation, a similar mechanism must be implemented to release pages allocated dynamically in the approximate areas.

Summing up, in order to support memory allocation in ZONE_APPROXIMATE at user program level, the following items should be addressed:

1. Implementation, within the kernel, of a memory allocation system call based on *vmalloc* in order to specify, through the flag GFP_APPROXIMATE, that the allocation must take place in the approximate memory zone;

2. encapsulation of the allocation mechanism in a library function that can be invoked by user space applications;

3. implementation, within the kernel, of a deallocation system call based on *vfree* in order to release the approximate pages allocated with the function described in 1;

4. encapsulation of the de-allocation mechanism in a library function that can be invoked by user space applications.

3.4.3 Implementation of the device /dev/approxmem

As said before, memory pages allocated by *vmalloc* are mapped into kernel virtual addresses, which do not cover memory valid for user-space application. The most efficient way to overcome this limitation is to remap the virtual addresses returned by *vmalloc* in the address space of the user mode process. To address this, the allocation mechanism for approximate memory has been implemented inside a new kernel device called */dev/approxmem*, implemented as a kernel module. The device will keep an image of all the approximate memory requests in the system and at the same time the same device (Fig. 3.19) will be used to exchange information between the user space application and the kernel module (i.e. *approxmem.ko*) running in kernel space. The */dev/approxmem* device will export system calls to allocate and free approximate memory, implementing point 1) and 3) of the previous list.

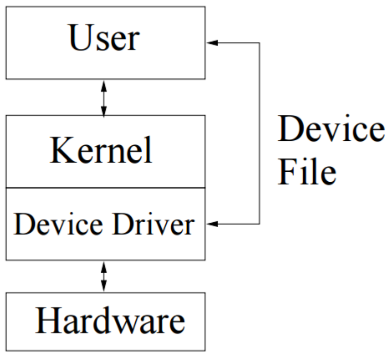


FIGURE 3.19: Device driver interaction

By managing the approximate memory as a device, it is possible to define additional operations (apart from alloc and free requests) as the *mmap*, which will be used to remap the virtual addresses returned by the *vmalloc* in user space addresses. The steps required to implement the */dev/approxmem* are shown below:

1. ***init* and *exit* functions**
   Each device driver has two basic entry points: an *init* function for kernel module initialization, and an *exit* function responsible for freeing the system resources required by the driver. Concerning the former, for built-in drivers, the ones automatically loaded by the kernel at boot time, the *init* is used only at initialization time and it can be later discarded, de-allocating the memory area reserved for it. This is not true for loadable modules, which are not automatically loaded by the kernel but they can be loaded by the user at run-time.

```
static int __init approxmem_init(void)

```

   Concerning the latter in particular, the *init* function is invoked through the *insmod* or *modprob* command. This function performs the following operations:

   • *Major number dynamic allocation*. Traditionally the major number identifies the driver associated with the device. Modern Linux kernels allow more drivers to share the same major number even if most of the devices are structured according to the principle *major one driver*. The driver for *approxmem* device is included in the second category. Several device major numbers are statically assigned to the most common devices (the list of these devices, and the corresponding kernel source tree, can be found in the <*documentation/devices.txt*> file). As a result, the major number of the device approxmem could be added directly to this file, choosing a number that has not yet been assigned to another device. This strategy is usually recommended when the device will be used only by the user who created it because, if the driver is shared by more users, conflicts could arise from choosing a random number. The major number for the device approxmem is then created dynamically, using the function *register_chrdev(unsigned int major, const char *name, const struct file operations * fops)*. In particular if the first parameter is 0, the function allocates dynamically the major number and returns the allocated number.

```
    approxmem_major = register_chrdev(0, "approxmem", &
  approxmem_fops);

```

There is also a *minor number*, used by the kernel to identify the specific device; for the approxmem it has been statically set to 0. Assuming that the major number 248 has be assigned to the */dev/approxmem*, the device is identified inside the kernel by the pair of numbers (248,0).

- *Class Device Registration* Each device class defines a type of device, specifying a set of semantics and a programming interface to which the devices, belonging to this class, must be compliant. A device driver represents indeed the implementation of this programming interface for a given device on a given bus. To register the class, the function *create_class* is used.

```
    approxmem_major = register_chrdev(0, "approxmem", &
  approxmem_fops);

```

- *Device Registration*. The last step is the device registration. This operation is performed calling the function *struct device * device_create (struct class * class, struct device * parent, dev_t devt, void * drvdata, const char * fmt)*; the latter creates and register the device through the virtual Linux filesystem *sysfs*, which exports the device driver from kernel space to user space.

```
    approxmem_device = device_create(approxmem_class, NULL,
  MKDEV(
                              approxmem_major, 0), NULL, "
  approxmem");

```

This function returns the pointer to the device struct in *sysfs* and creates the *dev* file for the approxmem device. If the initialization was successful, the messages shown in Fig. 3.20 can be obtained using the dmsg command.

The exit function is executed when the kernel unloads the module using the *rmmod* command.

```
    static void __exit approxmem_exit(void)

```

This function releases the resources allocated previously performing, in reverse order, the operations complementary to those carried out by the initialization function. In particular, the steps that must be executed are:

- *device removal*:

```
        device_destroy(approxmem_class, MKDEV(approxmem_major, 0));

```

- *device class unregistration:*

```
        class_unregister(approxmem_class);

```

- *device class removal*

```
        class_destroy(approxmem_class);

```

- *major number unregistration*

```
    unregister_chrdev(approxmem_major, "approxmem");

```

2. ***file_operations* struct**
All devices are represented inside the Linux Kernel by a *file_operations* struct, used to manage all the operations that a driver can perform on the device. The struct, defined in <*linux/fs.h*>, is an array of pointers to callback functions: each field of the structure corresponds to the entry point of a function defined by the driver to perform a specific operation. The aim of these operations is mainly to invoke system calls (this is why these functions are called *open, mmap, read, write* .. etc.). Comparing it with objects oriented programming languages, it is possible to consider the device *file_operations* as an *object* and the entry points defined in the struct, as the *methods* that operate on it. For the device driver *approxmem* the struct *approxmem_fops* has been defined as shown in the box below.

```
static struct file_operations approxmem_fops={
.open = approxmem_open,
.mmap = approxmem_mmap,
.unlocked_ioctl = approxmem_ioctl,
.release= approxmem_release,
};
```

Operations defined for approxmem device are:

- *approxmem_open*

```
static int approxmem_open(struct inode *inodep, struct file
                            *filep)
```

  This function is invoked every time the device is opened. The parameters received by this function are a pointer to the inode struct *inodep* and a pointer to the struct file *filep*. Both the structures are defined in <*linux/fs.h*>; in particular the struct inode is used internally by the kernel to represent a device file on the disk while the struct file represents the descriptor of an open file. It is possible to have more struct files representing several descriptors opened on a single file, but these refer all to a single structure inode. The latter contains a great deal of information on the file;

there are two fields in particular that are of interest to the driver: the *dev_t* field *i_rdev*, which contains the actual device number, and the struct field *cdev * i_CDEV*, corresponding to the structure used by the kernel to represent char devices. The struct file instead represents an open file, it is created by the kernel when the *open* function is invoked and then it is passed to every function that operates on it. When all instances of the files are closed, the kernel releases the structure. The *approxmem_open* prints only the message '*Ready for approximate memory allocation*', indicating that it is possible to proceed with the allocation in the approximate zone.

- *approxmem_mmap*

```
static int approxmem_mmap(struct file *filp, struct
                           vm_area_struct *vma)
```

The scope of the *approxmem_mmap* function is to associate a range of user-space addresses to the device, making sure that when an application reads or writes at these addresses the device is accessed. This function is called every time the system call *mmap* is invoked on the device and it receives as parameters a pointer to a struct file *filp* and a pointer to a struct *vma* (at user space level one of the parameters received by the *mmap* is a file descriptor, all the devices in Linux are in fact accessed as files). Specifically, the struct file *filp*, as said before, represents the open file associated with the device while the struct *vma* contains the information concerning the range of virtual addresses used to access the device. The first operation performed in the *approxmem_mmap* function is to call *vmalloc*, passing the *GFP_APPROXIMATE* and GFP_USER flags, indicating that the allocation must take place in the ZONE_APPROXIMATE and that the allocated pages must be accessible to a user-space application. After the allocation performed by vmalloc, the *approxmem_mmap* needs to remap, building a new page table, the addresses returned by vmalloc in the range of the user-space addresses associated to the *approxmem* device. This operation is performed through the *remap_pfn_range* function and it is repeated until all the requested memory has been allocated.

```
static int approxmem_mmap(struct file *filp, struct
   vm_area_struct *vma)
{
  size_t size = vma->vm_end - vma->vm_start;
  unsigned long start= vma->vm_start;
  unsigned long pfn;
  unsigned long ret;
  void * vmalloc_app;

  vmalloc_addr=__vmalloc(size,GFP_APPROXIMATE | GFP_USER,
   PAGE_SHARED);
  printk("addr : %x\n", (unsigned int)vmalloc_addr);
  vmalloc_app=vmalloc_addr;
    if (vmalloc_app!= NULL)  {

    while(size>0) {
      pfn=vmalloc_to_pfn(vmalloc_app);
      if((ret=remap_pfn_range(vma,start, pfn,PAGE_SIZE,
   PAGE_SHARED))<0){
         return ret;
         }
      start+= PAGE_SIZE;
      vmalloc_app+=PAGE_SIZE;
```

```
      size-=PAGE_SIZE;
      }

    return 0;
    }

  else
  return -ENOMEM;
}
```

If the value returned by vmalloc is NULL, the operation of remapping does not proceed and an error message (corresponding to the value of -ENOMEM) is printed.

- *approxmem_ioctl*

```
static long approxmem_ioctl(struct file* filep, unsigned int cmd,
                                        unsigned long arg)
```

Several device drivers need to perform generic control operations on the device they are managing and these operations can be supported through the *ioctl* function (input/output control), which then invokes the *ioctl* system call. The *ioctl* function is usually called with three parameters: the pointer to the struct file *filp*, to identify the device file on which the operation needs to be performed, an unsigned int *cmd*, corresponding to the ioctl number that indicates the command, and finally a third optional parameter *arg* of type unsigned long (in this way it can be converted in any type of data with a cast). The *approxmem_ioctl* function implements a switch which selects the correct behavior that the device driver *approxmem* has to perform, depending on the value of the *cmd* parameter. If *cmd* is 0, the operation to be performed is the APPROXFREE_IOCTL; in this case the parameter *arg* corresponds to a pointer to the memory address on which the *vfree* is called, releasing all the approximate pages allocated previously. If *cmd* is 1, the operation to be performed is the APPROXGETADDREE_IOCTL; in this case the *arg* parameter is a long used to store the value returned by *vmalloc*. In particular this data will be used by the user space library function *approx_malloc* (see Section 3.4.4) to build a linked list in which the virtual address returned by *vmalloc* and the corresponding user space address returned by the *mmap* will be progressively stored.

- *approxmem_release*

```
static int approxmem_release(struct inode *inodep, struct file *
                                        filep)
```

This function has the opposite role of the *approxmem_open* and therefore it is invoked whenever the device is closed by a user space application. As the *approxmem_open*, the received parameters are a pointer to the struct *inodep* and a pointer to the struct *filep*. The only operation performed by the *approxmem_release* is the printing of the message '*device approxmem closed*' in order to indicate that the device file approxmem has been properly closed.

3. **Build the *approxmem* module** The building phase produces the following files:

   - *approxmem.o* object file;

- approxmem.ko LKM (Loadable Kernel Module) file.

The module can be finally loaded into the kernel by launching the command *sudo insmod approxmem.ko*. This kernel module for the approxmem device can be accessed only with superuser permissions, so any program that wants to interface with it has to be launched with *sudo* command. In order to make sure that the *approxmem* module is able to be accessed also by a particular user or group of users, it is necessary to change the module permissions protecting the file system. To address this, the kernel *udev* rules are used. For each device present in the system, there is a file of rules (the file has the extension *.rules*) which is read from the *udev* daemon at system boot time; these rules are then saved in memory. A *.rules* file for the device *approxmem* has been created; in this file the rules for read and write permissions, even for a user or group user, are specified. To produce the *approxmem.rules* file is necessary:

- Identify the sysfs entry for the device, launching from the *sys* folder the command *find. -name 'approxmem'*;
- Identify the KERNEL and FILESYSTEM values to write the *.rules* file related to the device. The *udevadm* command is used for this purpose;
- Write the *99-approxmem.rules* file in the */etc/udev/rules.d* directory. The content of the file is shown in the box below.

```
#Rules file for the approxmem device driver
KERNEL == "approxmem", SUBSYSTEM == "approxmem class",
MODE = "0666"
```

As a test, the command: *ls -l / dev / approxmem* can be launched. It is observed that now also a user and a user group have the permissions to access the device

```
Initializing approxmem device
approxmem device registered with major number 248
approxmem class registered correctly
approxmem device class registered correctly
```

FIGURE 3.20: Creation of device approxmem

3.4.4 Approximate Memory Library: approx_malloc and approx_free

approx_malloc

The *approx_malloc* function allows a user space application to request the dynamic allocation of approximate memory, in a similar way as a *malloc* call.

```
void * approx_malloc(unsigned long size)
{
void* vmaddr;
void *ptr;
int fd = open('/dev/approxmem', O_RDWR);
if (fd < 0){
  printf('Failed to open the device...%s \n',strerror(errno));
}
ptr = mmap(0, size, PROT_READ | PROT_WRITE, MAP_SHARED,fd,0);
//printf('ptr is :%x\n', (unsigned int)ptr);
if(ptr == 0) {
  fprintf(stderr,'no memory available! \n');
```

```
    return NULL;
  }
  else {
    vmaddr = (void*)ioctl(fd,APPROXGETADDRESS_IOCTL,0);
    add_list(&table, ptr, vmaddr,size);
    printf('user:%x virtual:%x \n', (unsigned int)ptr, (unsigned int)
          vmaddr);
  print_list(table);
  return ptr;
  }
close(fd);
};
```

As shown in the text box above the function has only one input parameter, corresponding to the size of memory to be allocated. The allocation request is performed through the *mmap* function, which receives as parameter the size of memory to be allocated and the flags to specify the protection and the type of mapping to be performed in the address space of the calling process. In particular mmap is called with PROT_READ and PROT_WRITE flags, as it is required that the mapped memory is accessible for read and write operations, while the MAP_SHARED flag indicates that the type of mapping must be able to be shared by the other processes mapped on the same file. Once invoked, the *mmap* calls the *file operation mmap*, which is specific for the file identified by the parameter *fd*. In this case *fd* corresponds to the file descriptor of */dev/approxmem*, therefore the *mmmap_approxmem* function is invoked. If successful, this function returns the pointer to the area mapped in the address space of the running process. Furthermore the *vmaddr* pointer, returned by *vmalloc* and retrieved by the following APPROXGETADDRESS_IOCTL, and the *ptr* pointer corresponding to the area mapped by *mmap* are stored in a linked list to maintain the correspondence between the kernel virtual address and the mmap address of the approximate memory just returned. In case of error, if the value returned by mmap is equal to 0, the message '*no memory available!*' is printed.

approx_free

The *approx_free* function allows a user space application to release memory previously allocated in the approximate zone. This function receives as inputs two parameters: a pointer to the memory to be released (mapped in the process address space) and the size of memory to be de-allocated. In particular the operation of de-allocating memory requires to:

- Remove all mappings for the pages mapped in the address space of the process, in particular those in the *addr* + *size* range;
- release the pages allocated by *vmalloc* via *vfree*.

The first operation is performed by calling the syscall *munmap*, which receives as input the same parameters of *approx_free* function. This system call removes the mappings for the specified range within the process address space; otherwise the memory region mapped in the address space would be automatically released only at the end of the process (closing the file descriptor *fd* instead does not release the mapping). Concerning the second operation, it is necessary to identify the kernel virtual address corresponding to the user space *addr* passed as input parameter to the *approx_free* function. In order to transparently implement this operation, the information in the linked list is scanned: when the *addr* pointer matches a value in the list, the corresponding kernel virtual address is retrieved and then passed to the

APPROXFREE_IOCTL. If the APPROXFREE_IOCTL return value is negative, an error message is printed, otherwise the fields corresponding to the kernel virtual address and the corresponding user space address, are removed from the list and the operation ends successfully.

```
void approx_free(void *addr)
{
  int ret_ioctl;
  addr_table_t **table_app;
  long vaddr_free;
  print_list(table);
  int fd = open('/dev/approxmem', O_RDWR);
  if (fd < 0){
    printf('Failed to open the device...%s \n",strerror(errno));
  }
  if(addr!= NULL) {
   munmap(addr, size);
    printf('ADDR:%x \n', (unsigned int)addr);
    table_app =search_list(&table, addr);
    if(table_app!= NULL) {
      vaddr_free=(long)((*table_app)->vmalloc_addr);
      printf("VADDR:%x \n', (unsigned int)vaddr_free);
      ret_ioctl = ioctl(fd, APPROXFREE_IOCTL, vaddr_free);
      if(ret_ioctl <0)
        printf('ERROR ioctl');
      else remove_list(table_app);
    }
  }
close(fd);
}
```

3.4.5 Initial verification

In order to verify the whole allocation system for approximate memory (user space library, kernel device interface, kernel internal calls and policies), a testbench application has been written. In particular this application calls the *approx_malloc* library function in order to request, for three consecutive times, a block of 1000 memory pages (about 4MB) from ZONE_APPROXIMATE.

```
  #include"a_malloc.h"
  #include<stdio.h>
  #include<stdlib.h>
  int main()
  {
    int i;
    int j;
    int k;
    int *ptr[3];
    for( i=0; i<3 ;i++) {
      ptr[i] = (int *)approx_malloc(4096*1000);
      if (ptr[i] == 0)
      {
        fprintf(stderr,'ERROR: Out of memory\n');
        return 1;
      }
      for (k=0 ; k<((4096*1000)/sizeof(int)); k++) {
        ptr[i][k] = 2614;
      }
      getchar();
    }
    for( j = 0; j < 3; j++) {
```

```
        approx_free(ptr[j]);
        getchar();
    }
    return 0;
}
```

After verifying that the pointers returned by the *approx_malloc* are valid, the next step is to force the writing of a numeric constant on all bytes making sure that all allocated pages have write permission. Finally the *approx_free* function is invoked to release approximate memory. This application has been run in x86, ARM and RISC-V architectures, but only the results concerning the x86 architecture are illustrated below. During the program execution it is possible to obtain the following prints, corresponding to the three allocation performed by the *approx_malloc*. In particular for each allocation request the linked list under construction is printed, showing the user space address *user*, returned by the mmap, and the corresponding kernel virtual address *virtual*, returned by *vmalloc* (Fig. 3.21).

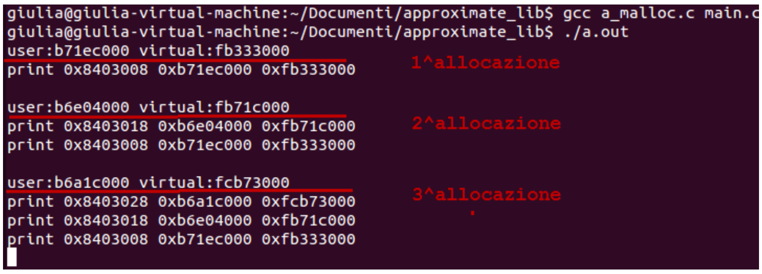


FIGURE 3.21: Bulding the linked list

As shown in the application code, each allocation request is divided from the other by the **getchar()** function, in this way the program execution remains blocked until a keyboard character is entered. After the third allocation request, before entering another character and invoking the *approx_free*, it is possible to open another terminal and launch the command **sudo cat/proc/vmallocinfo**, obtaining the messages shown in Fig.3.22 [1].

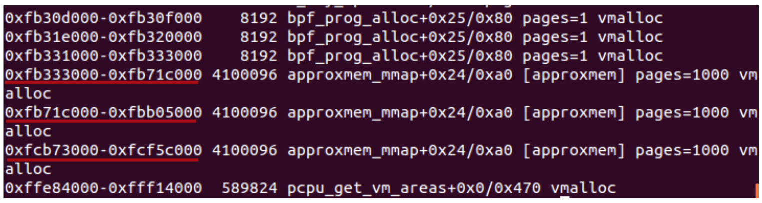
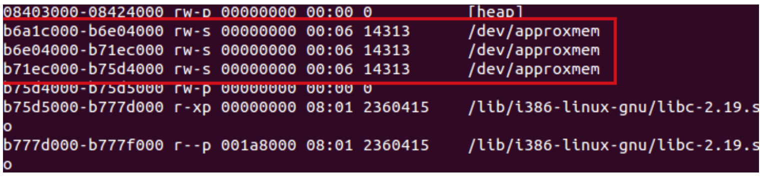


FIGURE 3.22: *vmallocinfo* messages on x86 architecture

These messages confirms that the *vmalloc* allocates for three consecutive times 1000 pages; the columns on the left print the ranges of virtual addresses where the allocated memory has been mapped. In particular, it can be observed that the virtual addresses printed in the linked list during the program execution are included in the ranges of virtual addresses printed by *vmallocinfo*. As a further test, using again the second terminal opened during the testbench execution, we can launch the command: **cat /proc/pid/maps** (Fig.3.23). By doing so, it is possible to visualize the ranges of user-space virtual addresses in which the running program has been mapped. In particular, there are three memory regions within the user address space

[1]these messages are produced running the application on an x86 architecture with approximate memory model using the ApprpinQuo emulator (see Chapter 3)

of the process that have been mapped by the device *approxmem* using the *mmap* function. All of them have been mapped with read and write permissions. Again the user space addresses printed in the linked list are included in the ranges of virtual addresses mapped by */dev/approxmem* confirming that the allocation mechanism is working properly.



```
08403000-08424000 rw-p 00000000 00:00 0          [heap]
b6a1c000-b6e04000 rw-s 00000000 00:06 14313      /dev/approxmem
b6e04000-b71ec000 rw-s 00000000 00:06 14313      /dev/approxmem
b71ec000-b75d4000 rw-s 00000000 00:06 14313      /dev/approxmem
b75d4000-b75d5000 rw-p 00000000 00:00 0
b75d5000-b777d000 r-xp 00000000 08:01 2360415    /lib/i386-linux-gnu/libc-2.19.s
o
b777d000-b777f000 r--p 001a8000 08:01 2360415    /lib/i386-linux-gnu/libc-2.19.s
o
```

FIGURE 3.23: Messages of cat / proc / pid / maps

Using the *zoneinfo* system command the information printed in Fig. 3.24 are obtained, showing some statistics about ZONE_APPROXIMATE before the allocation request. Before the allocation request the ZONE_APPROXIMATE has 114570 free pages. Fig. 3.25 shows the same statistics after the first allocation request. It can be seen now that ZONE_APPROXIMATE has 113570 free pages confirming that the *approx_malloc* allocated exactly 1000 pages.

```
cat /proc/zoneinfo
...
Node 0, zone Approximate
pages free 114570
spanned 114570
present 114570
nr_dirtied 0
nr_written 0
...
```

FIGURE 3.24: ZONE_APPROXIMATE statistics after boot

```
cat /proc/zoneinfo
...
Node 0, zone Approximate
pages free 113570
spanned 114570
present 114570
nr_dirtied 0
nr_written 0
...
```

FIGURE 3.25: ZONE_APPROXIMATE statistics after *approx_malloc* call

In order to assess stability the OS has been extensively tested with allocation benchmarks programs on different architectures (x86, ARM, RISC-V), filling the whole ZONE_APPROXIMATE page set. As expected, from that point on further requests of ZONE_APPROXIMATE pages caused allocations in ZONE_NORMAL, confirming that the fallback mechanism is working properly.

3.5 Quality Aware Approximate Memory Zones in Linux OS

3.5.1 Introduction and 64-bit implementation potentials

As previously said in Chapter 1, in Raha et al., 2015; Raha et al., 2017 the authors introduce a methodology for designing a quality aware approximate memory system based on DRAM. The core idea is to refresh DRAM with a single but reduced rate, characterizing portions of the memory array and splitting them in several *quality bins*, based on the frequency, location and nature of bit errors in each physical page. During program execution, non-critical data can be allocated to bins sorted in descending order of quality. The setup included the use of the lightweight operating system μC/OS-II for memory management and task creation. However, the paper proposes to use the *quality bins* in a descending order, ensuring that lower quality bins (i.e. having a higher level of approximation) are always used as last resource. The work does not explore the possibility of selecting the *quality bins* at program level, depending on data. In this section the notion of quality bins is taken and applied to approximate memory, implementing their management in the Linux Kernel OS.

In the previous paragraphs the approximate memory support on 32-bit Linux OS is described. Approximate memory management has been integrated in the kernel memory management, relying on the internal concept of Linux *physical zone*. In this way the Linux kernel is aware of exact physical memory pages (grouped in ZONE_DMA, ZONE_NORMAL and ZONE_MOVABLE) and approximate physical memory pages (grouped in a new ZONE_APPROXIMATE), managing them as a whole for the common part (e.g. optimization algorithms, page reuse, de-fragmentation) but distinguishing them in terms of allocation requests and page pools management.

Compared to the first implementation, the extension of approximate memory support to 64-bit architectures has led to new potential ideas, that were precluded due to the limits of the 32-bit memory management. In particular, the 64-bit kernel can manage larger memory sizes, but also has the ability to support up to 8 memory zones (32-bit kernel is limited to four zones, including the always active ZONE_NORMAL and ZONE_MOVABLE). In this way it is possible to insert the instantiation and management of up to four approximate zones, each one corresponding to physical memory pages with different levels of approximation. In this scenario applications could then allocate approximate memory for their data structures selecting between different levels of approximation, depending on the requirements on output quality. This could allow to design an architecture where approximate physical memory, instead of being composed of a unit intercepting a single point in the energy-quality tradeoff curve, can be split into multiple banks trading off levels of approximation and energy savings. Moreover the potential of having quality aware memory zones opens the way to further investigations, tailoring allocations of approximate memory depending on, for example:

- different sensitivity of output quality to errors in input data structures;
- variable-time output quality requirements;
- requirements of different applications in a multitasking environment.

The 64-bit could run in architectures containing several physical memory banks with different levels of approximation. In this context, the term *level of approximation* of a memory will be used in order to classify different approximate memories. This definition is related to error rate (i.e. higher error rate corresponds to higher level of approximation), but also, depending on approximate memory circuits, to the weight of bits affected by errors (i.e. on equal conditions, a memory with approximate cells limited to the least significant bits of a word has a lower level of approximation Frustaci et al., 2016). The multiple levels of approximation could be realized, for example, using several DRAM banks with different refresh rates, lower than required by specifications.

In particular, the extension to 64-bit architectures makes the following contributions:

- introduction of approximate memory support larger memory sizes;
- introduction of the capability of configuring up to four approximate memory zones (in addition to standard Linux memory zones), where each of these zones corresponds to a physical memory with a certain level of approximation;
- implementation of an internal data allocation scheme, capable of handling separately the allocation requests in quality aware approximate zones;
- development of a user space data allocation mechanism and support library.

3.5.2 Approximate memory zones on 64-bit architectures

Introducing multiple approximate memory zones within the Linux OS is mainly architecture independent, while a reduced number of modifications (as the definition of the memory map) is required in architecture dependent source files. For the first implementation, RISC-V 64-bit architecture has been selected as target.

The architecture independent part includes the creation of new memory zones and the implementation of the corresponding data allocation policy. Both should be consistent with the requirements of the approximate memory management already defined in subsection 3.3.2. It should be noticed that, due to the Linux kernel memory management implementation, on 32-bit architectures it is possible to define and create up to 4 memory zones, while on 64-bit architectures this limit is extended up to 8. Considering that ZONE_NORMAL and ZONE_MOVABLE are always enabled and required, while ZONE_DMA and ZONE_DMA32 could be enabled depending on architecture requirements for managing DMA devices, on 64-bit architectures, with the current implementation, it is possible to create up to 4 zones for approximate memory (that have been called ZONE_APPROXIMATEx, $x = 1...4$). The rationale behind these multiple zones is that each ZONE_APPROXIMATEx is filled with pages backed by physical memories with different, and *decreasing*, level of approximation (i.e. the approximate memory zone with the lowest index corresponds to memory with the highest level of approximation).

As seen on 32 bit architectures, defining an order is important since it has an impact on internal allocation policies. The former organization is compliant with the fallback mechanism of the Linux OS, which is activated if a memory zone is not able to satisfy an allocation request. In other words, if an allocation request for memory with a certain level of approximation cannot be satisfied (e.g. because the requested size is not available), the allocator will fallback to a hierarchically higher approximate zone, characterized with a lower level of approximation, up to the exact zones (ZONE_NORMAL, ZONE_DMA, etc.).

The mapping of the layout of approximate memory zones into physical RAM is architecture dependent. A function inside the kernel computes the available physical memory; this information is then used in the initialization phase to group physical memory pages into memory zones. Each zone must be characterized by its *start* *pfn* and *end pfn* (page frame number), corresponding to the physical address bounds of each memory (expressed in as $page\_number = physical\_address/page\_size$). On the 64-bit RISC-V architecture two memory zones are present by default: ZONE_DMA32 and ZONE_NORMAL. To introduce quality aware approximate zones, the physical memory has been partitioned into five parts by modifying the *__init zone_sizes_init* function in <arch/riscv/mm/init.c>. The first memory zone corresponds to exact memory and it will be used for ZONE_DMA32 and ZONE_NORMAL; the others are used for approximate memory zones (1 to 4). The *pfn* bounds of these zones are assigned statically, depending on the number of quality aware memory zones that are present (Fig. 3.26).

```c
#ifdef CONFIG_ZONE_APPROXIMATE
#define start_pfn_normal_riscv        0x80000
#define max_normal_riscv  (((max_low_pfn - start_pfn_normal_riscv)/2) + start_pfn_normal_riscv)
#endif

static void __init zone_sizes_init(void)
{
  unsigned long max_zone_pfns[MAX_NR_ZONES] = { 0, };

#ifdef CONFIG_ZONE_DMA32
  max_zone_pfns[ZONE_DMA32] = max_dma_pfn_riscv;
#endif

#if defined CONFIG_ZONE_APPROXIMATE && (!defined (CONFIG_ZONE_APPROXIMATE_2))
    max_zone_pfns[ZONE_NORMAL] = max_normal_riscv;
    max_zone_pfns[ZONE_APPROXIMATE] = max_low_pfn;

#elif defined (CONFIG_ZONE_APPROXIMATE) && defined (CONFIG_ZONE_APPROXIMATE_2) && (!defined (CONFIG_ZONE_APPROXIMATE_3))
  #define max_pfn_approx2 (((max_low_pfn - max_normal_riscv)/2) + max_normal_riscv)
  max_zone_pfns[ZONE_NORMAL] = max_normal_riscv;
  max_zone_pfns[ZONE_APPROXIMATE_2] = max_pfn_approx2;
  max_zone_pfns[ZONE_APPROXIMATE] = max_low_pfn;

#elif defined (CONFIG_ZONE_APPROXIMATE) && defined (CONFIG_ZONE_APPROXIMATE_2) && defined (CONFIG_ZONE_APPROXIMATE_3) && (!defined (CONFIG_ZONE_APPROXIMATE_4))
  #define max_pfn_approx2 (((max_low_pfn - max_normal_riscv)/2) + max_normal_riscv)
  #define max_pfn_approx3 (((max_low_pfn - max_pfn_approx2)/2) + max_pfn_approx2)
  max_zone_pfns[ZONE_NORMAL] = max_normal_riscv;
  max_zone_pfns[ZONE_APPROXIMATE_3] = max_pfn_approx2;
  max_zone_pfns[ZONE_APPROXIMATE_2] = max_pfn_approx3;
  max_zone_pfns[ZONE_APPROXIMATE] = max_low_pfn;

#elif defined (CONFIG_ZONE_APPROXIMATE) && defined (CONFIG_ZONE_APPROXIMATE_2) && defined (CONFIG_ZONE_APPROXIMATE_3) && defined (CONFIG_ZONE_APPROXIMATE_4)
  #define max_pfn_approx2 (((max_low_pfn - max_normal_riscv)/2) + max_normal_riscv)
  #define max_pfn_approx3 (((max_low_pfn - max_pfn_approx2)/2) + max_pfn_approx2)
  #define max_pfn_approx4 (((max_pfn_approx2 - max_normal_riscv)/2) + max_normal_riscv)
  max_zone_pfns[ZONE_NORMAL] = max_normal_riscv;
  max_zone_pfns[ZONE_APPROXIMATE_4] = max_pfn_approx4;
```

```
  max_zone_pfns[ZONE_APPROXIMATE_3] = max_pfn_approx2;
  max_zone_pfns[ZONE_APPROXIMATE_2] = max_pfn_approx3;
  max_zone_pfns[ZONE_APPROXIMATE] = max_low_pfn;
#else
  max_zone_pfns[ZONE_NORMAL] = max_low_pfn;
#endif

  free_area_init_nodes(max_zone_pfns);
```

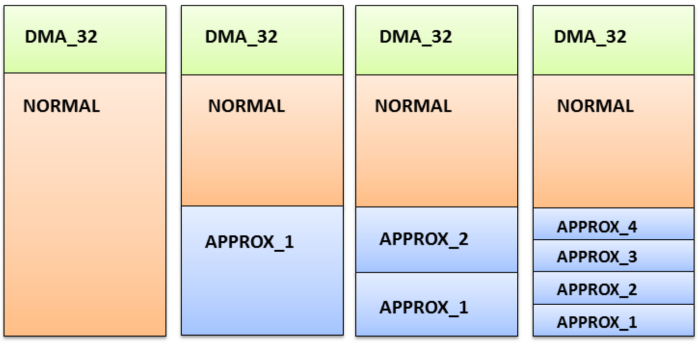


FIGURE 3.26: Configuration of physical memory layout

Moreover, the *start pfn* of the highest enabled ZONE_APPROXIMATE is used as the current limit inside the allocation algorithm of the Linux *bootmem allocator*. This implementation ensures that pages belonging to ZONE_APPROXIMATE*x* are never selected by the kernel to initialize page allocators data structures.

```
#ifdef CONFIG_ZONE_APPROXIMATE
#include <linux/types.h>
phys_addr_t get_approx_current_limit(phys_addr_t limit) {
  phys_addr_t approx_limit = max_normal_riscv * 4096;
  if (limit > approx_limit)
    limit = approx_limit;
  return limit;
}
#endif
```

3.5.3 Data Allocation

The Linux OS manages each allocation request using the set of *GFP flags*, in order to drive the allocation algorithm. These flags are used, among others, to define which memory zone should be selected for the current request. However, the zone requested by the flag is not completely binding since there are policies (fallbacks) that allow to go back in the memory zone hierarchy (e.g. in case a memory zone is full).

To correctly manage allocation requests in multiple approximate memory zones, it was necessary to define new *GFP flags*, one for each approximate memory zone (*GFP_APPROXIMATEx*, $x = 1...4$).

```
//#ifdef CONFIG_ZONE_APPROXIMATE
#define ___GFP_APPROXIMATE          0x800000u//0x800000u
//#endif
//#ifdef CONFIG_ZONE_APPROXIMATE_2
#define ___GFP_APPROXIMATE_2        0x1000000u
//#endif
//#ifdef CONFIG_ZONE_APPROXIMATE_3
#define ___GFP_APPROXIMATE_3        0x2000000u
//#endif
```

```
//#ifdef CONFIG_ZONE_APPROXIMATE_4
#define ___GFP_APPROXIMATE_4        0x4000000u
```

According to the requirements described in the previous sections, the priorities and fallback mechanism were configured to ensures that:

- the allocation in approximate zones can only take place on explicit request;
- the memory zone hierarchy guarantees that the fallback mechanism will always move from a higher to a lower level of approximation.

To allow user space applications to request data allocation in different approximate memory zones, a new */dev/approxmem* device and a new *approx library* have been implemented.

approx library for multiple approximate memory zone

The new *approx_malloc* function takes as input parameters the *size* of data that should be allocated and the *level* of approximation required. The *level* parameter is an internal allocation flag which is propagated inside the kernel and then it is associated to the GFP flag of the corresponding approximate memory zone.

```
void * approx_malloc(unsigned long size, unsigned long level)
```

- **approxmem dev for multiple approximate memory zone**

The main differences with respect to the implementation described in 3.4.3 deal with the addition of a new IOCTL command, *APPROX_GET_FLAGZONE_IOCTL*, and the implementation of the *approx_mmap* function, as shown in the code boxes below. Concerning the first, it allows to map the flag for the selection of the approximate memory zone from the user space approx library to the */dev/approxmem* device inside the kernel. In particular the *arg* parameter of *APPROX_GET_FLAGZONE_IOCTL* corresponds to the internal allocation flag in the *approx_malloc* and it is used for initializing a variable *gfp_flag*.

```
static long approxmem_ioctl(struct file* filep, unsigned int cmd, unsigned
    long arg) {
  switch(cmd){
    case APPROXFREE_IOCTL:
      printk("vaddress free:%lx\n", (long)arg);
      vfree((const void*)arg);
    break;

    case APPROXGETADDRESS_IOCTL:
      if(copy_to_user((void*)arg,&vmalloc_addr,sizeof(void*)))
        return -EFAULT;
    break;

    case APPROX_GET_FLAGZONE_IOCTL:
      gfp_flag = arg;
    break;

    default:
    return -ENOTTY;
    }

  return 0;
  }
```

The *gfp_flag* is used in the *approx_mmap* function to choose the *GFP_FLAG* for the allocation request, and so to determine the correct approximate memory zone.

```
static int approxmem_mmap(struct file *filp , struct vm_area_struct *vma)
{
  [...]
  if(gfp_flag == 1)
    vmalloc_addr=__vmalloc(size ,GFP_APPROXIMATE | GFP_USER, PAGE_SHARED);
  else if(gfp_flag == 2)
    vmalloc_addr=__vmalloc(size ,GFP_APPROXIMATE_2 | GFP_USER, PAGE_SHARED)
    ;
  else if(gfp_flag == 3)
    vmalloc_addr=__vmalloc(size ,GFP_APPROXIMATE_3 | GFP_USER, PAGE_SHARED)
    ;
  else if(gfp_flag == 4)
    vmalloc_addr=__vmalloc(size ,GFP_APPROXIMATE_4 | GFP_USER, PAGE_SHARED)
    ;
  [...]
```

3.5.4 Initial verification of the implementation

In order to perform an initial verification of the implementation, two approximate memory zones have been enabled on a 64-bit RISC-V architecture. In particular, the new kernel can to boot on a RISC-V SiFiveU platform emulated in AppropinQuo (see 4), with 256MB RAM memory.

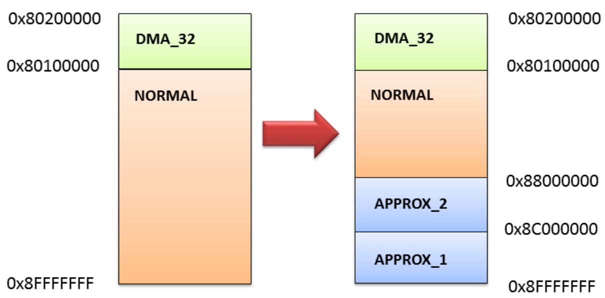


FIGURE 3.27: Configuration of physical memory layout on RISC-V SiFiveU

The 256MB of RAM are partitioned into 128MB of exact RAM, 64MB of approximate memory (level 1 of approximation), 64MB of approximate memory (level 2 of approximation), as illustrated in Fig. 3.27. The Kernel boot messages are shown in Fig.3.28: as expected, in addition to the ZONE_NORMAL and the DMA_ZONE32 two approximate memory zones are present, each one of 64 MB.

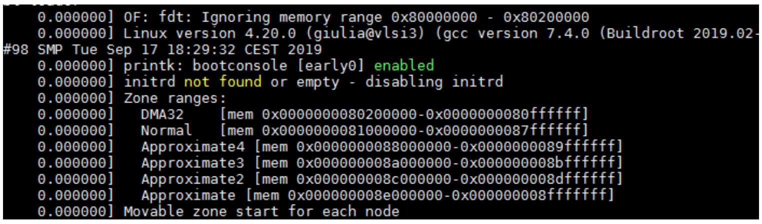
3.5.5 Verification and allocation tests

In order to verify the correctness of the multiple approximate memory zones allocator and of the new approximate memory library, a new testbench application has

```
...
Zone ranges:
DMA32 [mem 0x0000000080200000-0x0000000080ffffff]
Normal [mem 0x0000000081000000-0x0000000087ffffff]
Approximate2 [mem 0x0000000088000000-0x000000008bffffff]
Approximate [mem 0x000000008c000000-0x000000008fffffff]
Movable zone start for each node
Early memory node ranges
node 0:  [mem 0x0000000080200000-0x000000008fffffff]
...
```

FIGURE 3.28: Boot messages printing the physical memory layout

been written. The test is executed in AppropinQuo on a RISC-V64 platform with 512MB of RAM and 4 approximate memory zones enabled. The 512MB of RAM is split in half: the first half corresponds to exact memory, the second one to the approximate regions.



```
    0.000000] OF: fdt: Ignoring memory range 0x80000000 - 0x80200000
    0.000000] Linux version 4.20.0 (giulia@vlsi3) (gcc version 7.4.0 (Buildroot 2019.02-
#98 SMP Tue Sep 17 18:29:32 CEST 2019
    0.000000] printk: bootconsole [early0] enabled
    0.000000] initrd not found or empty - disabling initrd
    0.000000] Zone ranges:
    0.000000]   DMA32    [mem 0x0000000080200000-0x0000000080ffffff]
    0.000000]   Normal   [mem 0x0000000081000000-0x0000000087ffffff]
    0.000000]   Approximate4 [mem 0x0000000088000000-0x0000000089ffffff]
    0.000000]   Approximate3 [mem 0x000000008a000000-0x000000008bffffff]
    0.000000]   Approximate2 [mem 0x000000008c000000-0x000000008dffffff]
    0.000000]   Approximate [mem 0x000000008e000000-0x000000008fffffff]
    0.000000] Movable zone start for each node
```

FIGURE 3.29: Kernel boot messages for RISCV 64 platform with 4 approximate memory zones

Fig. 3.29 shows the kernel boot messages concerning the mapping of the low memory and the creation of the four approximate memory zones, each one of 64MB. The structure of the testbench, as shown in the text box below, is the same as the one described in 3.4.5: the application first requests 1000 pages from ZONE_APPROXIMATE (*approx_malloc(4096*1000, 1)*), then it requests the same number of pages from *ZONE_APPROXIMATE2*, *ZONE_APPROXIMATE3* and *ZONE_APPROXIMATE4*.

```c
#include"a_malloc_multiple.h"
#include<stdio.h>


  int main()
  {
    int i;
    int j;
    char *ptr[10];

//TEST ZONE APPROXIMATE 1
    for( i=0; i<3 ;i++) {
    ptr[i]= (char *)approx_malloc(4096*1000, 1);

    if (ptr[i]== 0)
    {
      fprintf(stderr,"ERROR: Out of memory\n");
      return 1;
    }

    //printf("virtual: %x \n", (unsigned int)ptr[i]);
    for (j=0 ; j<(4096*1000); j++) {
      //printf("virtual: %x \n", (unsigned int)&ptr[i][j]);
      ptr[i][j]+= 26;

```

```
    }
      //*ptr[i]=25;
    printf(" %d \n", *ptr[i]);
    printf("virtual: %p \n", ptr[i]);


    getchar();

    }

  for(j=0; j<3 ;j++)
    approx_free(ptr[j]);

//TEST ZONE APPROXIMATE 2
for( i=0; i<3 ;i++) {
    ptr[i]= (char *)approx_malloc(4096*1000, 2);

    if (ptr[i]== 0)
    {
      fprintf(stderr,"ERROR: Out of memory\n");
      return 1;
    }

    //printf("virtual: %x \n", (unsigned int)ptr[i]);
    for (j=0 ; j<(4096*1000); j++) {
      //printf("virtual: %x \n", (unsigned int)&ptr[i][j]);
      ptr[i][j]+= 26;

    }
      //*ptr[i]=25;
    printf(" %d \n", *ptr[i]);
    printf("virtual: %p \n", ptr[i]);


    getchar();

    }

  for(j=0; j<3 ;j++)
    approx_free(ptr[j]);

//TEST ZONE APPROXIMATE 3
      for( i=0; i<3 ;i++) {
    ptr[i]= (char *)approx_malloc(4096*1000, 3);

    if (ptr[i]== 0)
    {
      fprintf(stderr,"ERROR: Out of memory\n");
      return 1;
    }

    //printf("virtual: %x \n", (unsigned int)ptr[i]);
    for (j=0 ; j<(4096*1000); j++) {
      //printf("virtual: %x \n", (unsigned int)&ptr[i][j]);
      ptr[i][j]+= 26;

    }
      //*ptr[i]=25;
    printf(" %d \n", *ptr[i]);
    printf("virtual: %p \n", ptr[i]);


    getchar();
```

```
    }

  for(j=0; j<3 ;j++)
    approx_free(ptr[j]);

//TEST ZONE APPROXIMATE 4
      for( i=0; i<3 ;i++) {
    ptr[i]= (char *)approx_malloc(4096*1000, 4);

    if (ptr[i]== 0)
    {
      fprintf(stderr,"ERROR: Out of memory\n");
      return 1;
    }

    //printf("virtual: %x \n", (unsigned int)ptr[i]);
    for (j=0 ; j<(4096*1000); j++) {
      //printf("virtual: %x \n", (unsigned int)&ptr[i][j]);
      ptr[i][j]+= 26;

    }
      //*ptr[i]=25;
    printf(" %d \n", *ptr[i]);
    printf("virtual: %p \n", ptr[i]);


    getchar();

    }

  for(j=0; j<3 ;j++)
    approx_free(ptr[j]);

  return 0;

  }
```

Before the allocation request, each zone approximate has 64 MB of RAM. Fig. 3.30, 3.31, 3.32, 3.33, show the statistics for each memory zone after the first allocation request.

```
cat /proc/zoneinfo
...
Node 0, zone Approximate
pages free 15384
spanned 16384
present 16384
nr_dirtied 0
nr_written 0
...
```

FIGURE 3.30: ZONE_APPROXIMATE statistics after *approx_malloc* call

```
cat /proc/zoneinfo
...
Node 0, zone Approximate2
pages free 15384
spanned 16384
present 16384
nr_dirtied 0
nr_written 0
...
```

FIGURE 3.31: ZONE_APPROXIMATE2 statistics after *approx_malloc* call

```
cat /proc/zoneinfo
...
Node 0, zone Approximate3
pages free 15384
spanned 16384
present 16384
nr_dirtied 0
nr_written 0
...
```

FIGURE 3.32: ZONE_APPROXIMATE3 statistics after *approx_malloc* call

```
cat /proc/zoneinfo
...
Node 0, zone Approximate4
pages free 15384
spanned 16384
present 16384
nr_dirtied 0
nr_written 0
...
```

FIGURE 3.33: ZONE_APPROXIMATE4 statistics after *approx_malloc* call

Chapter 4

AppropinQuo, Full System Emulator for Approximate Memory Platforms

4.1 Introduction

This Chapter presents an emulation framework for hardware platforms with approximate memory units, called AppropinQuo. The specific characteristic of AppropinQuo is to reveal the effects, on the hardware platform and on software, of errors introduced by approximate memory circuits and architectures. The emulator allows to execute software code without any modification with respect to the target physical board, since it includes the CPU, the memory hierarchy and the peripherals, capturing as well software-hardware interactions and faults due to approximate memory units. The final scope is reproducing the effects of errors generated by approximate memory circuits, allowing to evaluate the impact (quality degradation) on the output produced by the software. In fact, output quality is related to error rate, but their relationship strongly depends on the application, the implementation and its data representation on physical memory.

As said previously in Chapter 2, the idea behind approximate memory circuits and approximate computing in general is to trade off energy consumption at the expense of computational accuracy and degradation of output quality. Memory is accounted for a large part of total power consumption in advanced architectures and it is supposed to increase as new memory hungry applications migrate toward the implementation on embedded systems (embedded machine learning, high definition video codecs, etc.). By relaxing design constraints regarding error probability on bit cells, researchers have proposed techniques that significantly reduce memory energy consumption. These techniques, which can be accounted in the general topic of approximate memory design, are implemented at circuit or architecture level, and are specific to the memory technology (i.e. SRAM or DRAM memories).

The effective introduction of approximate memory units in a hardware platform relies on the possibility of using them for allocating selected data structures in software applications. Although memory errors may degrade the quality of output results, the effects can be tolerated by ETAs. The output degradation can be measured in different ways, depending on the specific output (e.g SNR in case of a digital signal processing application), however its relationship with respect to approximate memory parameters (i.e. architectures, error rate, number and weight of affected bits) is not straightforward since it depends on many factors as the kind of application, the implementation details and how data are represented in memory by the compiler. The result is that it is possible to characterize the behavior of a complex, real world application with respect to the presence of errors introduced by approximate memories only by executing it on the target hardware platform (including approximate memories), or on an emulator that can model the complete platform.

AppropinQuo allows to run actual applications as on the physical platform, to expose the effects of specific approximate memory circuits and architectures on output quality and to vary their parameters (e.g error rate, number of affected bits, etc.). By exploring the approximate memory design space and its effects on the output of a software application, it is possible to characterize the application behavior, as a step toward the determination of the trade-off between saved energy and output quality (energy-quality tradeoff).

4.2 Related Works: Simulation environments for digital platforms

Simulation environments for embedded system platforms are considered fundamental tools to support the design and optimization flow, because the typical hardware/software interactions that are present in this field of application make the debug process, the characterization of performances and the optimization difficult. These tools allow a complete emulation of the physical platform, including instruction set architecture (ISA) emulation and hardware units emulation (e.g. memory, I/O units, timers, etc.).

They are needed, among others, during the first design phases since they provide the ability to explore different architectures and ideas, allowing to collect data regarding functionality, performance, energy consumption, with reduced costs and time compared to physical prototypes. This is true for functional and performance emulators [Bellard, 2005; Binkert et al., 2011; Jung, Weis, and Wehn, 2015; Burger and Austin, 1997], energy consumption emulators [Rethinagiri et al., 2014; Chandrasekar et al., 2012], faults emulators [Parasyris et al., 2014; Höller et al., 2015].

In [Jung et al., 2016] a simulation environment for investigation on approximate DRAM is presented. The simulator uses the gem5 framework [Binkert et al., 2011] as functional simulator and includes DRAMSys and DRAMPower [Jung, Weis, and Wehn, 2015; Chandrasekar et al., 2012]. Thanks to the addition of DRAMSys and DRAMPower, the emulation environment is capable of modeling data retention starting from memory physical parameters and producing at run-time power consumption data. The presented results demonstrate that, for the two cases taken as case study, refresh rate can be completely disabled with a negligible degradation on output quality. However, the tool is focused on technology parameters and variations, it is limited to DRAM models and no details are added with respect to the support for approximate techniques that rely on bit-weights or in general on architecture level techniques.

The contribution of this work is to fill the gap in hardware platform emulators, adding specific support for approximate memories, showing the valuable information that can be obtained. Even if emulators including faults in memory units have already been developed, the fault models are not specific to the area of approximate memories and many effects of approximate memory circuits and architectures cannot be emulated using general fault models. The models developed in AppropiQuo include the ability of emulating the effects of different approximate designs and implementations, which depend on the internal structure and organization of memory cells.

Previous works on approximate memory tend to use ad-hoc or limited solutions when results are produced, injecting bit flips at algorithmic level (without emulating the actual hardware platform) or using restricted modifications on existing emulators. While injecting bit flips at algorithmic level can be quite efficient in terms of emulation speed, if the hardware platform is not emulated all results that depend on the actual data allocation and implementation can be lost. This is particularly true for relatively advanced approximate memory strategies, as those using bit-weights to calibrate fault rates. Moreover, faults that depend on memory accesses (as typical for SRAMs, see Section 4.5.2) cannot be correctly emulated if the real memory access patterns are not reproduced.

The AppropinQuo emulator, based on *QEmu* [Bellard, 2005] (see Section 4.3) and supporting x86, ARM and RISC-V based platforms, allows to run real applications and operating system, to analyze application behavior and to expose the effects on output quality of different memory error rates and approximate techniques. By exploring the design space and its effects on output, a complete characterization of the application is possible, allowing the determination of the trade-off between the level of approximation and output quality. The main contribution of AppropinQuo indeed is to provide:

- a complete implementation for modeling approximate SRAM;
- DRAM models for approximate memories;
- bit dropping models for SRAM and DRAM.

4.3 QEmu Emulator

4.3.1 Main Concepts

QEmu, which is the acronym for *Quick EMUlator*, is a hosted hypervisor that allows to virtualize hardware and quickly emulate different processors. In particular two operating modes can be distinguished within QEmu:

- *User mode emulation* allows to run programs compiled for a given CPU on others CPUs, emulating only the instruction set of the processor and not the whole system. For example, it can be used for cross-compilation and cross-debugging. The main features of this mode (*application level virtualization*) are:
  - ability to convert generic Linux system calls, including many *ioctl*;
  - accurate signal handling to remap host signals into guest system;
- *Full system emulator* allows to emulate a complete system, usually a complete platform, including the processor and the various peripherals. The advantage of this approach, (*full platform virtualization*), is the possibility of running several operating systems at a time without restarting the host machine. The main features of this operating mode are:
  - use of a fully software MMU to ensure maximum portability;
  - emulation of various hardware devices;
  - optional use of in-kenel accelerators, such as the Kernel Loadable Modules;
  - Symmetic Multiprocessing (SMP) also on host systems with only one CPU.

For the present scopo of emulating the microprocessor architecture connected to approximate memory, the full-system approach has been used.

4.3.2 Dynamic Translation: Tiny Code Generator

Concerning the emulator, it is possible to distinguish always a *host*, corresponding to the host machine on which the simulator itself is running, and a *guest*, corresponding to the emulated system. In this work, the following setups have been used:

- *host*: (Linux workstation grade machine, x64 architecture)
- *guest*: (Linux 4.3, 4.19, 4.20 kernel verios, x86, ARM, RISC-V architectures).

The CPU target instructions, corresponding to the emulated guest processor, must be converted into the instructions of the host CPU: this operation, called *dynamic translation* in QEmu, is performed by the *Tiny Code Generator* (*TCG*). Specifically, blocks of target code, called *TB* (*Translated Blocks*), are translated in micro-operations (*TCG operations*), by the TCG front-end; these micro-operations correspond to an intermediate machine-independent notation (Fig.4.1). The scope of TCG is to remove the dependencies on the specific version of compiler in use. In fact, blocks of target CPU instructions are translated to RISC-like TCG instructions (*TCG microops*).

The second part of the dynamic translation consists of reconverting the *TCG miscoops* to instructions of the host CPU and this step is performed by the TCG backend. In the scope of QEmu developers, the TCG approach allows to obtain the best performance in terms of speed since it can translate blocks of target code rather than an interpretation of single instructions.

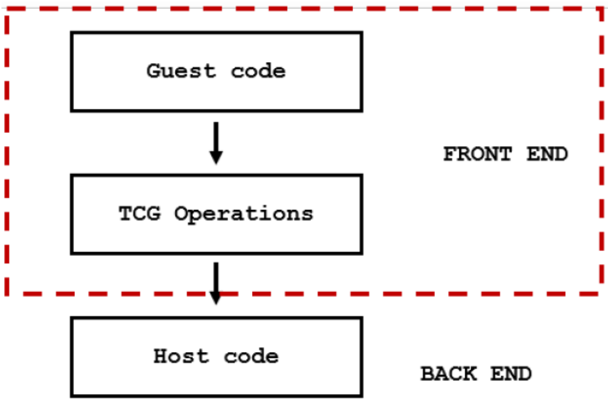


FIGURE 4.1: Tiny Code Generator

4.3.3 QEmu SoftMMU

In system emulation mode, QEmu supports a softMMU, which allows to translate a guest virtual address into a host virtual address for every memory access. Knowing the guest virtual address, the emulator can reach the corresponding virtual address of the host and this operation is divided into the following steps:

1. discover the guest physical address from the guest virtual address. This operation is performed by searching in a table (*phys_map*) and reading the physical offset (*phys_offset*) in the corresponding element;

2. discover the host virtual address by applying the obtained *phys_offset* to the host ram physical address (*phys_ram_base*).

To speed up this process QEmu uses a softMMU composed of two paths, by saving in a TLB table the offset to map a guest virtual address to host virtual address. In particular when a translation is required, QEmu looks first in the TLB table and if it finds a match in an entry of this table (hit on TLB), it uses it as an offset to obtain the host virtual address starting from the guest virtual address (Fig.4.2). Otherwise (miss on TLB) the emulator must consult the *phys_map* table, obtaining the offset and then filling the corresponding entry in the TLB table (slowpath) Fig.4.2.

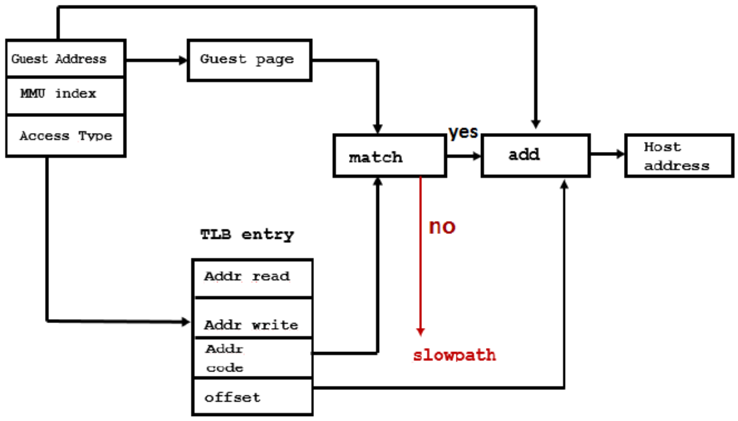


FIGURE 4.2: QEmu SoftMMU

4.4 Approximate Memory in AppropinQuo

4.4.1 QEmu Memory Management

In general, all guest RAM virtual pages used by QEmu at runtime are allocated in blocks getting it from RAM *AddressSpace* structure. The internal *ram_addr_t* type identify addresses belonging to this space (virtual addresses), while the *hwaddr_t* type identify addresses belonging to the physical address space (so read and write operations will use this type). Each page in physical space belongs to RAM (sytem *MemoryRegion*) or to I/O devices (a specific *MemoryRegion* for each device). In QEmu the *MemoryRegion* structure is responsible for managing the operation on the target emulated memory. In particular QEmu distinguishes between different types of *MemoryRegion*:

- regions for pages belonging to RAM;
- regions for pages belonging to the ROM;
- regions for pages mapped as I/O (MMIO). In this case another structure *MemoryRegionOps*, contained in each MemoryRegion structure, must be initialized. This additional structure contains the function pointers *read* and *write*, that provide the callback functions used for processing I/O operation on the emulated address space.

A special MemoryRegion, called *system_memory*, represents the all memory address space of the guest machine while the MemoryRegion *system_io* is used to emulate the operation of all memory mapped I/O devices. Both these structures are allocated by the *memory_map_init* function, called by *cpu_exec_init_all* directly in QEmu *main*.

```
static void memory_map_init(void)
{
    system_memory = g_malloc(sizeof(*system_memory));

    memory_region_init(system_memory, NULL, "system", UINT64_MAX);
    address_space_init(&address_space_memory, system_memory, "memory");

    system_io = g_malloc(sizeof(*system_io));
    memory_region_init_io(system_io, NULL, &unassigned_io_ops, NULL, "io",
                          65536);
    address_space_init(&address_space_io, system_io, "I/O");
}
```

The initialization of guest RAM instead is implemented in each machine specific file through calls to:

- *memory_region_init(MemoryRegion *mr, const char *name, uint64_t size)*, to initialize a Memory Region that can act as a container for other memory regions;
- *void memory_region_init_io(MemoryRegion *mr, const MemoryRegionOps *ops, void *opaque, const char *name, uint64_t size)*, to initialize an I/O memory region;
- *void memory_region_init_ram(MemoryRegion *mr,const char *name, uint64_t size)*, to initialize a RAM memory region.

In AppropinQuo specific units to model approximate memories have been developed inside the architecture; in particular the approximate memory model is implemented as a QEmu *MemoryRegion*, mapped in the I/O memory address space, receiving faults according to the error injection models. The reason to map approximate memory in I/O address space is to intercept read and write accesses at run time, qhich is required by some fault injection models.

As for the Linux Kernel (see **??**), the mapping of AxM into RAM memory region is specific for each architecture, so this operation is performed in the model of each architecture, in the same point where the allocation of RAM memory is implemented. This mapping will be briefly described in the following paragraphs.

Approximate memory mapping on PC PIIX, x86 architecture

```
FWCfgState *pc_memory_init(PCMachineState *pcms,
MemoryRegion *system_memory,
MemoryRegion *rom_memory,
MemoryRegion **ram_memory,
PcGuestInfo *guest_info)
{
  int linux_boot, i;
  MemoryRegion *ram, *option_rom_mr;
  MemoryRegion *ram_below_4g, *ram_above_4g;
  MemoryRegion *approxmem;
  FWCfgState *fw_cfg;
  MachineState *machine = MACHINE(pcms);
  assert(machine->ram_size == pcms->below_4g_mem_size +
  pcms->above_4g_mem_size);
```

```
linux_boot = (machine->kernel_filename != NULL);
ram = g_malloc(sizeof(*ram));
memory_region_allocate_system_memory(ram, NULL, "pc.ram",
                                     machine->ram_size*2);
*ram_memory = ram;
ram_below_4g = g_malloc(sizeof(*ram_below_4g));
memory_region_init_alias(ram_below_4g, NULL, "ram-below-4g", ram,
                         0, pcms->below_4g_mem_size);
memory_region_add_subregion(system_memory, 0, ram_below_4g);
approxmem =g_malloc(sizeof(*approxmem));
memory_region_init_io(approxmem, NULL, &approxmem_ops, NULL,
                      'approxmem',pcms->below_4g_mem_size);
printf("pcms: %dM \n", pcms->below_4g_mem_size>>20);
memory_region_add_subregion(system_memory,pcms->below_4g_mem_size,
                            approxmem);
approxmem_init(pcms->below_4g_mem_size);
e820_add_entry(0, (pcms->below_4g_mem_size)*2, E820_RAM);
```

It can be observed that RAM is allocated in the *pc_memory_init* function as a single memory region (*MemoryRegion * ram*), using aliases to address,within it, smaller portions of memory (*subregions*) (for example to distinguish between RAM above and below the 4G boundary). To introduce approximate memory in the architecture, another *MemoryRegion*, called *approximem*, has to be implemented and mapped as I/O. This operation is performed by calling the function *memory_region_init_io*, which receives as parameters the pointer to the memory region to be initialized, the structure with the callback functions (i.e. read and write) and the size of the region to be mapped. The RAM is then composed of two memory regions:

- MemoryRegion *ram*, corresponding to exact memory;
- MemoryRegion *approximem*, corresponding to approximate memory;

After the initialization of these regions, both are then added as a subregion to *system memory*, which, as said before, identifies all memory present in the system. Finally the *approxmem_init* is invoked to initialize the *approxmem* device.

It is possible to note that these two regions are initialized in order to have the same size, as declared correspondingly in the Linux kernel. The actual amount of RAM allocated is specified to QEmu and passed to Linux Kernel in the command line. In particular, the option *-m* is used to indicate the RAM size corresponding to the exact memory, while the option *-append memmap* specifies approximate memory size.

Since in this architecture memory size is configured by the BIOS with an automatic discovery procedure, approximate memory is not found because it is mapped as I/O. Therefore, during bootstrap, the BIOS finds only the exact portion of RAM (e820: BIOS-provided physical RAM map), later ram size is changed to indicate that another region is present (e820: user-defined physical RAM map), see Fig. 4.3. This is obtained passing the *memmap* command line to Linux Kernel.

It is important to observe that hiding the approximate memory region during the initial boot phase is a requirement of this architecture, since it could be otherwise used by the bootloader to allocate initial code and structures that are sensitive to errors (i.e. kernel code, initram filesystem, etc.).

Approximate memory mapping on Vexpress Cortex A9, ARM architecture

In the text box below the initialization of the RAM, and therefore also of the approximate memory region, for the Vexpress ARM platform is shown.

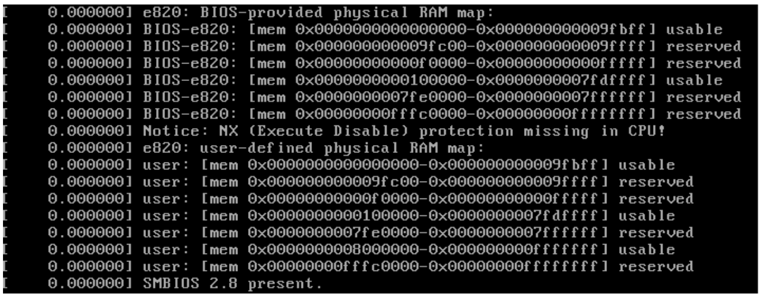


FIGURE 4.3: e820 Bios Memory Mapping passed to Linux Kernel

```c
static void a9_daughterboard_init(const VexpressMachineState *vms,
                                  ram_addr_t ram_size,
                                  const char *cpu_model,
                                  qemu_irq *pic)
{
    MemoryRegion *sysmem = get_system_memory();
    MemoryRegion *ram = g_new(MemoryRegion, 1);
    MemoryRegion *approxmem = g_new(MemoryRegion, 1);
    MemoryRegion *lowram = g_new(MemoryRegion, 1);
    ram_addr_t low_ram_size;

    if (!cpu_model) {
        cpu_model = "cortex-a9";
    }

    if (ram_size > 0x40000000) {
        /* 1GB is the maximum the address space permits */
        fprintf(stderr, "vexpress-a9: cannot model more than 1GB RAM\n");
        exit(1);
    }

    memory_region_allocate_system_memory(ram, NULL, "vexpress.highmem",
                                         ram_size);
    low_ram_size = ram_size;
    if (low_ram_size > 0x4000000) {
        low_ram_size = 0x4000000;
    }
    /* RAM is from 0x60000000 upwards. The bottom 64MB of the
     * address space should in theory be remappable to various
     * things including ROM or RAM; we always map the RAM there.
     */
    memory_region_init_alias(lowram, NULL, "vexpress.lowmem", ram, 0, low_ram_size);
    /*approxmem*/
    memory_region_init_io(approxmem, NULL, &approxmem_ops, NULL, "approxmem",approx_ram_size);

    memory_region_add_subregion(sysmem, 0x0, lowram);
    memory_region_add_subregion(sysmem, 0x60000000, ram);
    /*Map approximate memory*/
    memory_region_add_subregion(sysmem, 0x60000000 + ram_size, approxmem);
    approxmem_init(approx_ram_size);
```

The implementation is similar to the one already described for the PIIX x86 architecture; the main difference concerns how the information about the amount of approximate memory is propagated into the emulator. Since the boot mechanism is different and an equivalent to x86 BIOS is not present, it is not necessary to provide this information to the operating system via the *memmap* option. The memory map information, instead, is intrinsic in the *device tree* structure (passed to the emulator in the command line) and it is also specified in the configuration file for the *approxmem* device [1].

Approximate memory mapping on VirtIO, RISCV-32 architecture

The solution adopted to map approximate memory on RISC-V32 VirtIO board is the same used on ARM architecture, since even in this case the size of approximate memory is specified in the device tree file.

```
memory_region_init_ram(main_mem, NULL, "riscv_virt_board.ram",
                           machine->ram_size, &error_fatal);
memory_region_add_subregion(system_memory, memmap[VIRT_DRAM].base,
    main_mem);

/*Approxmem*/
memory_region_init_io(approxmem, NULL, &approxmem_ops, NULL, "approxmem",
    ram_size_approx);
memory_region_add_subregion(system_memory, memmap[VIRT_DRAM].base +
    ram_size, approxmem);
approxmem_init(ram_size_approx);
```

This was also repeated for the Si-FiveU board.

Multiple Approximate memories mapping on VirtIO, RISC-V64 architecture

As last implementation, it is illustrated the initialization of multiple approximate memory regions for the 64-bit RISC-V64 platform VIRT-IO.

```
static void riscv_virt_board_init(MachineState *machine)
{
[...]
      memory_region_init_ram(main_mem, NULL, "riscv_virt_board.ram",
                           machine->ram_size / 2 , &error_fatal); // /2
      memory_region_add_subregion(system_memory, memmap[VIRT_DRAM].base,
        main_mem);

    /*Approxmem*/

    while(approx_index < approx_mem_regions->num_approx_regions){
      size_approx = (approx_mem_regions->approx_mem[approx_index].
    approx_config.mem_size);
      if(approx_index == 0)
          approx_mem_regions->approx_mem[approx_index].approx_base =
    memmap[VIRT_DRAM].base + ram_size/2;
      else
          approx_mem_regions->approx_mem[approx_index].approx_base =
    approx_mem_regions->approx_mem[
                    approx_index - 1].approx_base + size_approx;
      approxmem_init( size_approx, &approx_mem_regions->approx_mem[
    approx_index], approx_mem_regions->       approx_mem[approx_index].
    approx_base, system_memory);
      approx_index++;
       }
```

In this case the information regarding the actual amount of physical memory is obtained directly from the *approxmem* configuration file. In particular, the mapping of the approximate memory regions is implemented as follows:

1. the size of the i-th approximate memory area (*ram_size* parameter) is obtained;

[1] The structure and the role of the *approxmem* configuration file is described in Section 4.4.2

2. the physical start address of each approximate memory zone is computed.

   - If the approximate memory region is the first one (*approx_index = 0*) the start address of this region is given by the base address of the RAM plus the size of the exact RAM;
   - For the next approximate memory zones, the start address is given by the start address of the previous approximate zone plus its size. Obviously the information specified in the configuration file must be consistent with what is implemented in the kernel.

3. Each approximate region is initialized through a specific *approxmem_init* function.

4.4.2 Approxmem device in AppropinQuo

The *approxmem* device is the actual model of approximate physical memory. As previously exposed, it is mapped in the QEmu I/O *MemoryRegion* and it is associated to the corresponding device in Linux Kernel. The implementation is completely architecture independent and it has been written in the *approxmem.c* file. This file has been added to the QEmu model library, in particular in the root directory where the files common to all architectures are collected.

In ordert to manage the *approxmem* devices it is required to define:

1. the *approximate_mem* structure, which is its descriptor.

```
typedef  struct approximate_mem{
MemoryRegion* iomem;
approxmem_state   approx_state;
approxmem_config  approx_config;
hwaddr approx_base;
}approximate_mem;

```

2. an *approximate_mem_regions* struct, .

```
struct approximate_mem_regions{
approximate_mem  approx_mem[MAX_APPROX_REGIONS];
int num_approx_regions;
}

```

   The scope of the struct is to store in the *approx_mem[MAX_APPROX_REGIONS]* array the pointers to the descriptors of all the approximate memory regions present in the architecture. The *num_approx_regions* contains the number of enabled approximate memory regions (as they can ben less than the maximum allowed). As said in the previous chapter, on 64 bit architectures Linux Kernel allows to declare up to 4 ZONE_APPROXIMATE, so *num_approx_regions* can range from 1 to 4. On 32 bits architectures it is possible manage one ZONE_APPROXIMATE, therefore *num_approx_regions* will be always set to 1.

3. a state strucTure for a single region, *approxmem_state*.

```
typedef struct approxmem_state {
void *approxmem_ptr;
int read_counter;
int write_counter;
```

```
QEMUTimer *timer_error_01;
QEMUTimer *timer_error_10;
}approxmem_state;

```

This structure is used to store the main information concerning the device state. In particular, the following fields are present:

- a *void * approxmem_ptr* pointer to refer to the actual memory array;
- an integer variable, *read_counter*, to count read accesses;
- an integer variable, *write_counter*, to count the write accesses.
- two pointers to QEmu timer units, *timer_error_10* and *timer_error_01*, used by some fault models (see Section 4.5.1).

4. approximate memory configuration struct, *approxmem_config*.

```
typedef struct approxmem_config{
char* name;
int error_access_enable;
int error_timing01_enable;
int error_timing10_enable;
uint looseness_mask;
access_fault_config access_f_config;
timing_fault_config timing_f_config;
uint32_t mem_size;
}approxmem_config;

```

The scope of *approxmem_config* structure is to store the configuration parameters of a single approximate memory region. Essentially, these parameters allow the user to setup the fault models, configuring the type of injected faults (*error on access*, Section 4.5.2, *DRAM fault orientation* model, Section 4.5.1, *bit dropping* Section 4.5.3), the fault rate and the number and position of bits affected (circuit and architectural level techniques concerning approximate memories work mainly at bit level). In particular, these parameters are:

- *name*, the name of approximate memory region;
- *error_access_enable*, to enable error on access model;
- *error_timing01_enable*, to enable DRAM fault orientation model for anti-cells;
- *error_timing10_enable*, to enable DRAM fault orientation model for true-cells;
- *access_f_config* a structure which allows to specify the fault rates for error on read and error on write;

  ```
  typedef struct access_fault_config{
  double p_fault_read;
  double p_fault_read_tx;
  double p_fault_write;
  }access_fault_config;

  ```

  p_fault_read is to the error probability for error on read operations, *p_fault_read_tx* indicates the error probability for transmission error on read operations and finally *p_fault_write* is used for the error probability of error on write operations.The meaning of these error rates will be clarified in Section 4.5.2.

- *timing_f_config* which allows to specify the fault rates for true-cells and anti-cells in DRAM fault orientation error models;

```
    typedef struct timing_fault_config{
    double p_fault01_per_bit;
    double p_fault10_per_bit;
    int n_errors;
    }timing_fault_config;

```

The meaning of these fault rates will be discussed in Section 4.5.1.

- *size*, to specify the size (in MB) of the approximate memory region.

The configuration struct is filled by parsing a .cfg configuration file received by AppropinQuo emulator through the command line option *-approx*. The option has been added on purpose to the standard QEmu options and it allows to specify the path to the configuration file for the approximate memory models. For example:

qemu-system-riscv64 -M virt -kernel $APPROX_ROOT/bbl -append 'root=/dev/vda ro console=ttyS0 mem=256' -d approx_log -D approx.log -nographic -m 256M -drive file='$APPROX_ROOT/rootfs.ext2',format=raw,id=hd0 -device virtio-blk-device,drive=hd0 -netdev user,id=net0 -device virtio-net-device,netdev=net0
-approx $APPROX_ROOT/config_approx.cfg

An example of configuration file is shown below:

```
# authenticator
#APPROXIMATION GROUP: approximate memory parameters
#timing_enable_01:Enable a single shot timer to inject faults on
    transaction 0->1
#timing_enable_10:Enable a single shot timer to inject faults on
    transaction 1->0
#mask:bitmask to protect most significant bits from fault-injection.
#mem_size: approximate memory size

[approximation]
timing_enable_01=1
timing_enable_10=0
access_enable=0
mask=0x7fffffff
mem_size=128

#ACCESS GROUP: error on access parameters
#p_fault_read: fault read probability [Errors/access]
#p_fault_read_tx: fault read_tx probability [Errors/access]
#p_fault_write: fault write probability [Errors/access]
#if p_fault_read = 1 --> bit dropping enable
[access]
p_fault_read= 1e-30
p_fault_read_tx=1e-30
p_fault_write=1e-30
#TIMING GROUP: timing error parameters
[timing]
# f_rate01_per_bit: fault rate per bit on transaction 0->1[Errors/(
    bit*s)]
# f_rate10_per_bit: fault rate per bit on transaction 1->0[Errors/(
    bit*s)]
f_rate01_per_bit=1e-5
```

```
f_rate10_per_bit=1e−1
n_errors=1000
```

If several approximate zones are enabled, each zone will have its own configuration file and a corresponding struct storing the state of the specific memory region. For example, when two approximate memory regions are present:

$APPROX_ROOT/qemu-system-riscv64 -M virt -kernel $APPROX_ROOT/bbl -append 'root=/dev/vda ro console=ttyS0' -nographic -m 256M
-drive file='$APPROX_ROOT/rootfs.ext2',format=raw,id=hd0
-device virtio-blk-device,drive=hd0
-netdev user,id=net0 -device virtio-net-device,netdev=net0
-approx $APPROX_ROOT/config_approx2.cfg
-approx $APPROX_ROOT/config_approx.cfg -d approx_log -D approx.log

The order the configuration files are passed to the emulator, defines internally the zone they are used for. Hence, the first is used for the zone with the lowest level of approximation (in this case *approx2*) and the second for the one with the highest level.

5. *approximate memory region callbacks*
   Being mapped as memory I/O, approximate *MemoryRegion*s present a *.read* and *.write* callbacks, that are invoked every time the CPU reads from or writes to locations belonging to approximate memory. These callbacks are defined in the *MemoryRegionOps* structure, associated to the I/O memory when it is initialized by the *memory_region_init_io()* function. Moreover, this structure is used to specify the minimum and maximum access sizes and the endianness of the device.

```
const MemoryRegionOps approxmem_ops= {
  .read=  (void *)read_approxmem,
  .write= (void *)write_approxmem,
  .impl = {
      .min_access_size = 1,
      .max_access_size = 4,
  },
  .endianness = DEVICE_LITTLE_ENDIAN,
};
```

Finally, a mechanism for logging approximate memory accesses was implemented, adding the an *APPROX* severity log to those already present in QEmu. These log messages allow to trace accesses to approximate memory regions and faults injection. The APPROX logs are enabled with the following command line options:

-d approx_log -D approx.log

where -d enables output log messages and -D specify the file name (*approx.log*).

4.5 Error injection models for approximate memories

4.5.1 DRAM orientation dependent models

DRAM memory cells use a single transistor and a single capacitor to store a bit, represented as charge on the capacitor. Lowering the refresh rate of DRAMs determines that the charge loss induced by leakage current will proceed until discharge. The effects on bit value depend on the DRAM circuit architecture, and will be discussed briefly.

In DRAM, single cells are organized in arrays (memory banks) and are connected to an equalizer and a sense amplifier (Fig. 4.4).

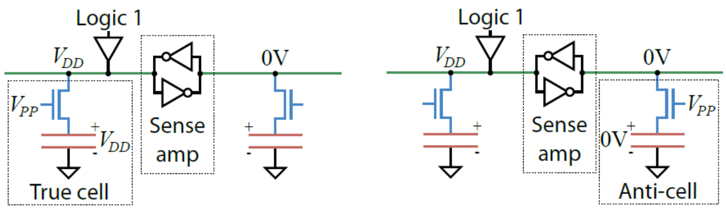


FIGURE 4.4: DRAM true cell and anti cell. Source:Liu et al., 2013

Being differential, every sense amplifier is connected to two bitlines in order to determine whether the charge of one of them should be interpreted as logical 0 or 1: when a bitline is activated, the other holds the reference precharge voltage ($V_{DD}/2$). The sense amplifier architecture, which is a specific manufacturer design choice, determines the DRAM cells orientation. In particular, the following implementations are adopted:

- *true-cells*: cells store a logical value of '1' as V_{DD} and a logical value of '0' as $0V$;
- *anti-cells*: cells store a logical value of '0' as V_{DD} and a logical value of '1' as $0V$;
- *mixed-cells*: a combination of both true-cells and anti-cells.

When lowering refresh rate, the corresponding charge loss appears, at logic level, as a bit flip, whose orientation depends on the internal DRAM array structure. In particular, the following errors can emerge and are implemented in our model:

- *true-cell error model*: when a cell loses charge, a '1' to '0' bit flip is observed;
- *anti-cell error model*: when a cell loses charge, a '0' to '1' bit flip is observed;
- *mixed-cell error model*: both '1' to '0' and '0' to '1' bit flip occurs in the array.

The orientation of DRAM cells is specific of each design and usually not known. The capability of emulating DRAM faults according to its internal structure becomes indeed fundamental to correctly reproduce the effects on the bit cell and consequently on data at software level. In our model, each bit flip direction is characterized by an error probability, in particular fault rates p_01, p_10 are defined, respectively, for the emulation of true-cell and anti-cell error effects (as shown in the previous section, the values of these error probabilities are specified in the approximate memory configuration file). In case of mixed cells, both error rate can be specfied.

The probability distribution is assumed uniform in the array of cells, as showed in many works (e.g. [Mathew et al., 2018]), and expressed as an error rate ($errors/(bit \times s)$). In particular, after a time interval has elapsed (implemented as a QEmu single-shot internal timer and whose value is determined by the corresponding error probability), a callback is invoked, depending on the type of cell. The initialization of the two timers has been implemented in the *approxmem_init* function. Going into details, the *timer_new_ns* function is used to create a new QEMUTimer object of *virtual clock* type (meaning that the timer is active only when the simulation is running) and to associate it to the corresponding callback. The *timer_mod_ns* function is used instead in order to arm the timer for the first time.

```
/*Approximate memory init*/
void approxmem_init(unsigned int size) {
    qemu_log_mask(APPROX_MEM, "approximate memory init call!\n");
    approx_state.approxmem_ptr=g_malloc(size);
    approx_state.read_counter=0;
    approx_state.write_counter=0;
    if(approx_config.error_timing01_enable){
      qemu_log_mask(APPROX_MEM, "Initialize timer error 0->1\n");
      approx_state.timer_error_01=timer_new_ns(QEMU_CLOCK_VIRTUAL,(QEMUTimerCB*)error_generator_01,NULL);
      timer_mod_ns(approx_state.timer_error_01,1000);
      }
    if(approx_config.error_timing10_enable){
      qemu_log_mask(APPROX_MEM, "Initialize timer error 1->0\n");
      approx_state.timer_error_10=timer_new_ns(QEMU_CLOCK_VIRTUAL,(QEMUTimerCB*)error_generator_10,NULL);
      timer_mod_ns(approx_state.timer_error_10,1000);
      }
    }
```

Depending on which model is enabled (true-cell or anti-cell), the callback generating the corresponding fault is executed, while, in case of a mixed cells architecture, both callbacks are invoked. The box below shows the implementation for the anti-cell callback.

```
static void error_generator_01(void)
{
    qemu_log_mask(APPROX_MEM, "approximate memory:error generator 0->1 timer callback!\n");
    uint8_t *approx_array=(uint8_t*)approx_state.approxmem_ptr;
    uint64_t time_expire;
    int c_faults;
    float time_f;
    hwaddr address;
    uint64_t val;
    uint64_t val2;

    time_f= -1e9*(logf(1.0 - (float) rand() / (RAND_MAX + 1.0)) / TIMER_F_RATE_01);
    time_expire=time_f;

    for(c_faults = 0; c_faults < approx_config.timing_f_config.n_errors; c_faults++){
      address= (hwaddr) ((rand_32() % MEMORY_SIZE) & 0xFFFFFFFC);

    val= (*(uint32_t*)&approx_array[address]);
    val2 = val ^ (approx_config.looseness_mask&(1<<(rand()%32)));
    *(uint32_t*)&approx_array[address]= val2;
    qemu_log_mask(APPROX_MEM, "val:%llx val2:%llx val^val2:%llx \n", val, val2, val^val2);
    }

    timer_mod_ns(approx_state.timer_error_01,qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL) +time_expire);
    //qemu_log_mask(APPROX_MEM,"time expire:%llu %f \n", time_expire, time_f);
    //qemu_log_mask(APPROX_MEM," error generator:%llu \n", qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL) +time_expire);
    }
```

Since the model is integrated with the *looseness level* model, faults occur only if the bit selected to be flipped has a '1' in the corresponding bit position of the *looseness* *mask* (see Section 4.5.4). At the end of the routine, the timer is re-armed using as expire time a value followin a Poisson distrubution depending on fault rate.

4.5.2 SRAM models

SRAM approximate memories are designed with aggressive supply voltage scaling. In SRAM bitcells, read and write errors are caused by low read margin (RM) and write margin (WM) [Itoh and Horiguchi, 2009]. Since process variations affect RM and WM in opposite directions, the corner defines which is the critical margin (i.e. the slow–fast (SF) corner makes the bitcell write critical, the fast–slow corner makes it read critical). Under voltage scaling, WM and RM are degraded, increasing read and write bit error rates (BERs). The degradation is in general abrupt (BER increases exponentially at lower voltages), but techniques have been proposed to make such degradation graceful [Frustaci et al., 2014].

Given this behavior, the fault injection model implements an error on access mechanism, which happens when a cell is activated to perform a read or write operation. Depending on the access, we distinguish three kind of errors:

- *Error on write (EOW)*: introduced during a write operation, the bit stored in the cell is flipped with respect to the bit coming from the data bus;
- *Destructive error on read (EOR)*: introduced during a read operation, the bit stored in the cell is flipped and passed to the data bus (both cell and data bus contain the corrupted bit);
- *Non-destructive error on read (EOR_TX)*: introduced during a read operation, the bit stored in the cell is not corrupted during the operation, but it is flipped when passed to the data bus.

For each one of these access errors, a uniform probability distribution in the array of cells is assumed, expressed as *errors/access*.

This fault mechanism is implemented inside the *approx_mem* memory region access callbacks, specified in the corresponding QEmu *MemoryRegionOps*. As said before, this structure contains two function pointers, *.read* and *.write*, which process the I/O operations on the emulated memory: each time a read or write operation is requested to the approximate memory region, the corresponding callback is invoked. The fault mechanism for destructive and non-destructive *error on read* is implemented in the read callback while the fault injection mechanism for *error on write* is implemented in the write callback.

As for the DRAM error orientation models, due to the integration with the *looseness level* model, faults occur only if the bit selected to be flipped has a '1' in the corresponding bit position of the *looseness mask* (see Section 4.5.4).

Error on read

Two types of error on read have been implemented:

1. *destructive error on read*: this type of error is referred as *destructive*, meaning that in result of a read operation, the value stored in the cell is corrupted. This means that the corrupted value will be read again in case of further read operations.
2. *transmission error on read*: this type of error is referred as *non-destructive*, meaning that in result of a read operation, a wrong value is read from the bit cell but

the stored value is preserved. Therefore, subsequent read operations will get the correct value.

These two types of error are assumed statistically independent (Poisson stochastic process). Consequently, two different independent extractions are performed in order to determine if errors should occur in the memory cell. As well, during the same read access, both types of read access errors can occur (destructive and non-destructive).

```
r_prob = rand_32() % PF_RAND_R;
if(r_prob == 0)
error_on_read = 1;
r_prob= rand_32() % PF_RAND_TX
if(r_prob == 0)
error_tx= 1;
```

The box above illustrates the stochastic extraction concerning the two types of error; in particular, *P_FAULT_R* and *P_FAULT_TX* indicate respectively the fault rates for destructive error on read and read transmission errors. In this case, the probability respective probability is given by *1/FAULT_RATE_x*.

Inside the routine that implements the error model, debug printouts have also been inserted (Fig. 4.5) in order to display, during the execution of an application, the actual value that is read and to print an error message any time a fault is injected.

```
approxmem read:78d58e4 size:4 val:a36
approxmem read:78d58e8 size:4 val:a36
approxmem read:78d58ec size:4 val:a34
Warning: an error occurred! val:a34 is different from a36
Error_on_read:1 Error_tx:0
approxmem read:78d58f0 size:4 val:a36
approxmem read:78d58f4 size:4 val:a36
approxmem read:78d58f8 size:4 val:a36
```

FIGURE 4.5: Error on Read debug messages produced during execution

Error on write

When a write operation is performed, just one type of error can occur. The *error on write* is always a permanent error: in case of fault, a wrong value is written in the bit cell, corrupting the data. As for read operations, a stochastic extraction with probability *1/PF_RAND_W* is performed to determine if during the access the error should be introduced.

```
w_prob=rand_32() % PF_RAND_W;
if(w_prob==0)
error_on_write=1;
```

4.5.3 Bit dropping fault model

Bit dropping is a bit-level technique which consists in completely disabling some memory bitlines. It has been shown to be interesting since cells can be completely powered off or even omitted [Frustaci et al., 2016; Yang et al., 2016; Frustaci et al., 2015a]. The dropped bitlines are always the LSBs in a data word, since the impact of errors is exponentially lower for smaller bit weights. This can be paired with the consideration that in many applications, such as machine learning, big data and multimedia, the quality is defined essentially by the MSBs. The technique is independent of technology and can be applied to both SRAM and DRAM memory circuits [Yang et al., 2016].

For SRAM memory cells the precharge circuit of the selected LSBs is disabled during read and write operations. This approach is quite different from the traditional dual V_{DD} scheme where the supply voltage of both precharge circuits and bitcells of the selected columns is reduced to a lower value. In [Frustaci et al., 2016; Frustaci et al., 2015a] an implementation of SRAM bit dropping precharge circuit is proposed: the drop signal, (corresponding to transistors M1 and M2 in Fig. 4.6), connects the bitline of the dropped cell permanently to ground, eliminating dynamic energy consumption.

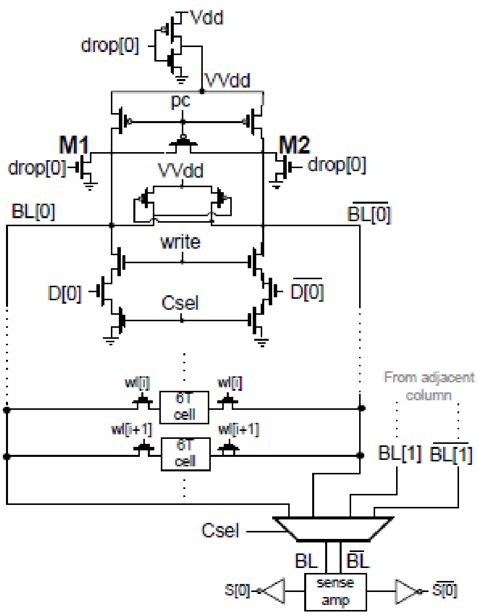


FIGURE 4.6: SRAM precharge circuit for bit-dropping technique. Source:Frustaci et al., 2015a

In DRAM memories instead, the refresh operation is completely disabled on dropped bitlines. In [Jung et al., 2015] the authors demonstrate that, for dedicated applications, the refresh operation can be switched off with a negligible impact on the application performance.

In our model, bit dropping is implemented as follows:

- in SRAMs, when a word in memory is read or written and the bit dropping is enabled, a given subset of bits is always set to '0'.
- in DRAMs, when a word in memory is read or written and the bit dropping is enabled, a given subset of bits is set to '0' or '1' depending on memory cell orientation distribution.

This fault injection mechanism is implemented in the read and write callbacks of the *approx_mem* device: every time a read or write operation is performed in the approximate memory, if the bit dropping error model is enabled, a specified number of bits is always set to '0' or '1', depending on memory technology. The number of dropped bits is defined by the *looseness mask*, a configurable parameter described in the following section.

4.5.4 Memory looseness level and fault models

Bit level approximate techniques have been introduced in order to exploit the exponential weight that bits assume in data words. Effectively, the approach introduces a new level of freedom in the approximate memory design space, that can be explored in search of better trade-offs.

As said in the previous section, a fault in a cell that is part of the most significant bits (MSBs) has a larger impact on the stored value, with respect to a fault occurring in one of the least significant bits (LSBs). In [Frustaci et al., 2016] selective voltage scaling is proposed in order to modulate error rate, at the cost of an increment in circuit complexity; in [Lucas et al., 2014] DRAM banks are reorganized and refresh rate modulated in order to obtain a similar effect. Considering results, the technique appears effective but the actual implementation is dependent on the microprocessor ISA and its data representation and organization in memory.

In order to support the emulation of bit level techniques, the concept of *looseness level* and *looseness mask* have been inserted in AppropinQuo. The looseness level (i.e. the number of bit that are affected by errors in a data word) is implemented by introducing a 32-bit configurable mask (constant for the whole memory array) that is applied to every 32-bit word in memory (Fig.4.7). Its scope is the selection of bits affected by faults (i.e. the MSBs). The bits inside a word are not affected by faults when the corresponding bit in the looseness mask is set to zero (i.e. with a looseness mask set to 0x0FFFFFFF, the 4 MSBs are exact, while the 28 LSBs are affected by faults). The structure of the looseness mask allows to effectively tune approximation at bit level for 32-bit, 16-bit and 8-bit data.

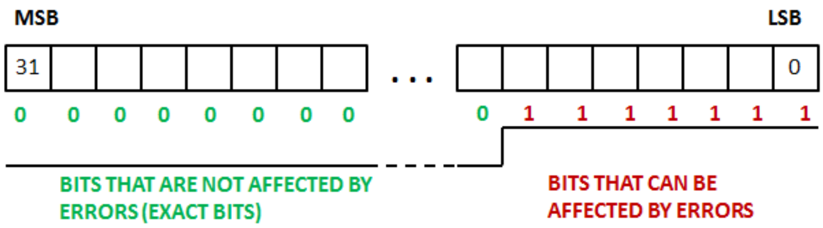


FIGURE 4.7: Example of Looseness Mask on Big Endian architecture

It should be underlined that having a single looseness mask defined for the whole approximate memory array means having a single looseness level for the array. This is mainly due to the consideration that practical implementations are limited to this configuration, since it already is has a large impact in the design of the whole memory layout. Another consideration specific to this model is that MSBs and LSBs in memory are not uniquely defined but depend on microprocessor ISA (i.e. data endianess) and also data size (i.e. in a 32 bit memory word 8 bit data are packed in groups of four and stored in the same word). The choice of defining the memory *looseness level* using a 32 bit *looseness mask* provide the flexibility required by the above mentioned cases.

4.6 Quality aware selective ECC for approximate DRAM and model

This section describes an innovative technique that can be applied to approximate DRAMs under reduced refresh rate. It allows to trim error rate at word-level, while still performing the refresh operation at the same rate for all cells. The number of bits that are protected is configurable and depends on output quality degradation that can be accepted by the application.

4.6.1 Bit dropping for LSBs, bit reuse and selective ECC

The proposed approach results from the exploration of the relation between output quality and BER on the LSBs and MSBs. LSBs in a data word can be dropped and set to a constant value (i.e. 0) with a marginal impact on output quality degradation. It is a technique that is proposed since it achieves energy savings with a simple circuit implementation (bit cells are powered off or even omitted).

Previous works have proposed to use selective ECC in SRAMs to reduce errors in MSBs.

1. by enlarging memory words as in classical ECC memory systems (i.e. 32bit memory word are expanded to 36bit, introducing 4bit ECC) [Lee et al., 2013];
2. by reusing LSB dropped bits [Frustaci et al., 2015a].

The contribute of the work is

1. to design selective ECC specific for approximate DRAM memory systems;
2. to tailor selective ECC to the specific application, by first analyzing its output quality degradation related to bit error rate, looseness level and dropped bits.

4.6.2 Quality aware selective ECC

The idea of quality aware selective ECC consists of a two step process. First, an application is analyzed in order to find the desired tradeoff between output quality and approximate memory parameters (i.e. error rate, level of approximation, dropped bit); then an error correcting code is chosen in order to reduce error rate in a specific portion of data bits. In order to avoid increasing memory requirements with additional ECC bit, bit dropping and reuse is always considered for the additional check bits required by ECC.

ECC codes for approximate memories

In order to reduce hardware complexity, (n,k) SEC (single error correcting) Hamming codes were considered. In this notation, k indicates the number of protected bits (data bits), while n is the code length, including additional check bits. We note that Hamming codes can provide also error detection (e.g. double error detection, typically), but for our scope error detection is not used: in case of detected errors, program execution would continue as for undetected errors in approximate memory. Table 4.1 summarizes the most common SEC Hamming codes. We note that, as a general rule, increasing the number of data bits k produces more efficient codes, since the rate k/n increases. However, larger k are effective at very small error rates (as is common in exact memories). In approximate memories typical error rates are much larger (i.e. from 10^4 to $10^{-2} errors/(bit \times s)$ [Stazi et al., 2019]) and, as consequence, shorter codes are desirable since enlarging n increases the probability of multiple errors within the same word, which cannot be corrected.

As described in section 4.5.4 Looseness Level represents the concept of having a certain number of exact MSBs in an approximate data word. As an example, Table

TABLE 4.1: List of Hamming codes

#Check bits (n-k)	#Total bits (n)	#Data bits (k)	Name	Rate
2	3	1	Hamming(3,1)	1/3
3	7	4	Hamming(7,4)	4/7
4	15	11	Hamming(15,11)	11/15
5	31	26	Hamming(31,26)	26/31
6	63	57	Hamming(63,57)	57/63

TABLE 4.2: FIR, output SNR [dB]

Looseness Level	Fault rate [$errors/(bit \times s)$]			
	10^{-1}	10^{-2}	10^{-3}	10^{-4}
12 MSBs	70.5	83.4	93.5	104.2
8 MSBs	46.5	59.6	69.3	80.3
4 MSBs	22.6	35.3	45.5	56.4
1 MSB	4.6	17.2	27.6	38.2

# of dropped bits			
4 LSBs	8 LSBs	12 LSBs	16 LSBs
134.7	122.4	106.1	82.2

4.2 (left) reports results obtained on a 32 bit integer FIR filter, showing how Looseness level (i.e. the number of exact MSBs) can impact output SNR.

Instead of using exact DRAM cells for MSBs, the idea is to use a single, and slower, refresh rate for all cells, while using SEC ECC in order to reduce error rate in MSBs. In this way, MSBs are still affected by errors, but their error rate is reduced with respect to LSB cells.

4.6.3 Impact of bit dropping and bit reuse

Table 4.2 (right) reports results obtained on the same 32 bit integer FIR filter, showing how bit dropping (i.e. powering them off and reading them as '0') impacts output SNR. As already confirmed in literature, output SNR is only slightly dependent on LSBs. Instead of powering them off, these LSBs can be effectively reused as checkbits for the MSBs, without requiring additional bits.

4.6.4 Implementation

Given the list of Hamming codes in Table 4.1, it appears that the most suitable for our application are Hamming (3,1),(7,4) and (15,11). This choice depends on two factors: first we assume to protect single 32 bit words in memory, in order to reduce the impact on read/write speed; infact, protecting with a single code larger data would require multiple read/write accesses on the entire data. Secondly, given the relatively high bit error rate of approximate memories, longer SEC codes tend to fail due to the rising probability of multiple errors.

Fig. 4.8 shows the formats considered for 32 bit data, where k MSBs are protected by SEC ECC, $32 - n$ bits are left unprotected and $n - k$ dropped and reused as checkbits. Assuming a uniform error probability p_e for each bit, expressed as $errors/(bit \times s)$, the probability of having i errors in a set of n bits is:

$$P_e(n,i) = \binom{n}{i} p_e^i (1 - p_e)^{n-i};$$

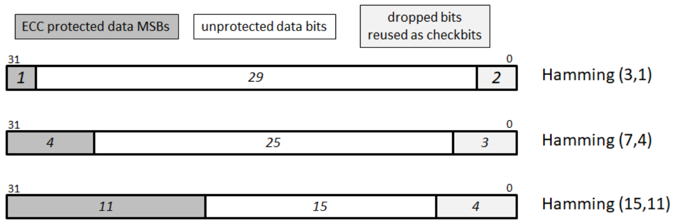


FIGURE 4.8: 32 bit ECC data format in approximate memory

TABLE 4.3: BER for 32 bit data in approximate memory

Hamming (3,1) word		
ECC prot. 1 bit	unprot. 29 bit	drop 2 bit
9.42E-03	1.00E-01	-
9.93E-05	1.00E-02	-
9.99E-07	1.00E-03	-
1.00E-08	1.00E-04	-
1.00E-10	1.00E-05	-

Hamming (7,4) word		
ECC prot. 4 bit	unprot. 25 bit	drop 3 bit
2.29E-02	1.00E-01	-
2.90E-04	1.00E-02	-
2.99E-06	1.00E-03	-
3.00E-08	1.00E-04	-
3.00E-10	1.00E-05	-

Hamming (15,11) word		
ECC prot. 11 bit	unprot. 15 bit	drop 4 bit
3.92E-02	1.00E-01	-
6.45E-04	1.00E-02	-
6.94E-06	1.00E-03	-
6.99E-08	1.00E-04	-
7.00E-10	1.00E-05	-

Considering the SEC ECC code, protected bits will contain errors for $i \geq 2$; hence:

$$Pecc_e(n) = \sum_{i=2}^{n} P_e(n,i) = \sum_{i=2}^{n} \binom{n}{i} p_e^i (1-p_e)^{n-i};$$

In order to get a measure of the improvement, we can find the equivalent error rate peq_e, considered as the error rate that n bits (without ECC) should have to produce the same $Pecc_e(n)$.

$$Peq_e(n) = \sum_{i=1}^{n} P_e(n,i) = 1 - \sum_{i=0}^{0} P_e(n,i) = 1 - (1 - peq_e)^n;$$

Equivalent bit error rate peq_e for ECC protected bits can be obtained with $Peq_e(n) = Pecc_e(n)$:

$$peq_e = 1 - \sqrt[n]{1 - Pecc_e(n)};$$

Assuming 32 bit data stored in approximate memory, Fig. 4.8 resumes how selective ECC could be applied using Hamming (3,1), (7,4) and (15,11) codes. The most appropriate choice depends on the application; for example, according to Table 4.2, a range from 8 to 12 protected MSBs results in an output SNR between 60dB to 93dB, while dropping 4 LSBs does not significantly impacts SNR. In this case Hamming (15,11) seems the most suitable choice.

Table 4.3 reports effective error rates that would result on data considering the previous Hamming codes. It shows that MSBs protected by SEC codes expose and equivalent BER significantly lower than unprotected bits. According to the previous example, Hamming (15,11) and a BER of $10^{-}3$ on cells produces an equivalent BER of $6.94 \times 10^{-}6$ on MSBs.

Due to the relatively high error rates in approximate memories, SEC codes reduce but do not eliminate errors on MSBs. This is completely acceptable since the scope is to improve output quality by reducing the impact of faults, but they are still tolerated by the application.

Future works will implement the technique in simulation models and apply it to error tolerant applications, allowing the characterization and the comparison with respect to previous techniques.

4.7 Verification of fault models

In order to evaluate the correctness of the developed fault-injection models, a suite of configurable benchmarks has been implemented. They are resumed in the following list:

- *bench_access*, an application that stresses fault injection models related to error on accesses (Section: 4.7.1);
- *bench_spontaneous_error*, an application that stresses fault injection models related to the DRAM orientation fault models (Section:4.7.2);
- *bench_dropping*, an application to test the bit dropping models (Section: 4.7.3).

4.7.1 Error on access models verification

The *bench_access* application allows to evaluate the fault injection mechanism related to memory accesses. The algorithm implemented is the following:

1. an array of n integers is allocated in an approximate memory buffer (x is a parameter configured by the user from the command line);
2. all the n integer locations are initialized to 0 (write access in approximate memory);
3. all the samples integer are read (reading in approximate memory);
4. the value is read exactly one time and then stored in a local variable: if it is different from zero, an error occurred and the errors counter is incremented.

Table 4.4 shows the obtained results. The benchmmark was run varying the array size n and the error probabilities (EOR, EOW, EOR_TX, see Sections 4.5.2); the looseness mask was kept constant and set to 0x00FFFFFF.

It is possible, for example, to consider the first row of the table: 10^5 integers have been allocated in approximate memory, setting all fault rates to 10^{-3} $[errors/access]$ (meaning statistically one error every 1000 accesses). The "*#errors*" column of the table shows that 227 errors occured when the application runs with this setup. The result confirms the correctness of the implementation since, according to this specific configuration, about 300 errors should occur. However considering that the looseness mask protects the most significant 8 bits, the expected number of errors is $300 \times 0.75 = 225$. The same considerations hold for the first three rows. Considering now the fourth row, where only EOR (destructive error) are enabled, the benchmark reports 78 errors occurrence. This result is again correct because the expected errors are $100 \times 0.75 = 75$. Therefore the application and the obtained results allow to validate the correctness of the implemented fault models.

4.7.2 DRAM orientation model verification

The *bench_spontaneous_error* application allows to evaluate the fault injection related to the DRAM orientation model. The algorithm implemented is the following:

TABLE 4.4: *bench_access* results, fixed Looseness Level

Looseness Level	EOR Fault (rate)	EOR_tx fault rate	EOW Fault rate	# errors	# integers
0X00FFFFFF	10^{-3}	10^{-3}	10^{-3}	227	10^5
0X00FFFFFF	10^{-3}	10^{-3}	10^{-3}	22	10^4
0X00FFFFFF	10^{-3}	10^{-3}	10^{-3}	2	10^3
0X00FFFFFF	10^{-3}	0	0	78	10^5
0X00FFFFFF	10^{-3}	0	0	10	10^4
0X00FFFFFF	10^{-3}	0	0	2	10^3
0X00FFFFFF	10^{-2}	0	0	735	10^5
0X00FFFFFF	10^{-2}	0	0	69	10^4
0X00FFFFFF	10^{-2}	0	0	8	10^3
0X00FFFFFF	10^{-1}	0	0	7502	10^5
0X00FFFFFF	10^{-1}	0	0	764	10^4
0X00FFFFFF	10^{-1}	0	0	72	10^3

1. an array of n integers is allocated in an approximate memory buffer. The user specifies from command line the number n and a delay t between a write and a read operation;
2. all the n elements of the array are initialized to 0xFFFFFFFF (write access in approximate memory);
3. a *wait(t)* function is invoked, suspending the execution until the specified time interval is elapsed;
4. at the end of the time interval all the elements od the array are read (read access in approximate memory);
5. each bit read as zero indicate the occurrence of errors and the errors counter is incremented.

The scope of the application is to run with only the DRAM orientation model enabled (otherwise read and write accesses in approximate memory would inject further errors). Table 4.5 shows the obtained results. The benchmark was launched varying the number n and the error rate (*fault_rate_10*, i.e error due to leakage in DRAM true cells); the *looseness mask* is kept constant and set to 0x00FFFFFF.

Considering the *looseness mask* set to 0x00FFFFFF expected #errors is:

$$\#errors = 8 \times array\_size \times t \times fault\_rate \times 0.75;$$

Also in this case the obtained results allow to validate the correctness of the implemented fault models.

4.7.3 Bit dropping model verification

The *bit_dropping* application allows to evaluate the fault injection model related to memory bit dropping. The algorithm implemented is similar to the one implemented in the *bench_access* application. The application has been executed varying

TABLE 4.5: *bench_spontaneous_error* results (Wait time $t = 1000ms$)

Array size (byte) Level	Fault Rate [Errors/bit*s]	# errors
1MB	10^{-6}	6
10MB	10^{-6}	58
100MB	10^{-6}	603
1MB	10^{-3}	5980
10MB	10^{-3}	60353
100MB	10^{-3}	598322

the *looseness level* parameter in the model, which in this case indicates the number of dropped bits.

TABLE 4.6: *bench_dropping* results

Looseness Level	Peak signal[dB]	PSNR (dB)
0x0000000F	186.64	167.69
0x0000001F	186.64	161.45
0x0000003F	186.64	155.69
0x0000007F	186.64	149.56
0x000000FF	186.64	143.21
0x00000FFF	186.64	119.71

Table 4.6 shows the obtained results. In particular, the third column is filled with the computed PSNR (section 2.2.7) while the second column illustrates the Peak Signal (the reference for PSNR). It corresponds to the maximum value that the signal can assume, therefore it is obtained when no bits are dropped (looseness Level = 0x00000000). The Peak signal has been computed in the following way:

$$
\begin{aligned}
PeakSignal[db] &= 10\log(MAX\_INT^2) \\
&= 20\log(MAX\_INT) = 20\log(2^31) \\
&= 186.64
\end{aligned}
$$

It is possible to consider, as an example, the data in row 1: the obtained PSNR is 167.69 dB when the Looseness Level is set 0x0000000F (all bytes are protected except for the most significant byte). This is exactly what expected since, from the theory of truncation error, truncating $n = 4$ bits correspond to a rms error of of 18.89 db:

$$RMS\_trunc\_error[db] = 10\log((n-1)(2n-1)/6)$$

Chapter 5

Exploiting approximate memory in applications and results

5.1 Introduction

This chapter represents the final step of this work, where eventually it was possible to write and execute real world applications and exploit approximate memory for data structures. Since we run on top of Linux OS, every application that has already been written and run on it can potentially be modified in order to allocate approximate memory for some of its data structures, with the sole limitation that this data must be error tolerant. The interest was addressed toward high memory consuming application and toward signal processing applications, that typically manifest tolerance to errors in their input data.

Once the applications were ready, they were run on an emulated approximate system, meaning an emulated hardware platform containing approximate memories (AppropinQuo) plus a running Linux operating system with approximate memory management support. This setup allows to evaluate the impact of the different levels and strategies of approximation (i.e. emulating different approximate memory circuit by varying fault rate, error injection models, looseness level) and allows to mark the dependency between the approximate hardware configuration and degradation of output quality of the application.

This exploration and characterization is important since it is not possible to easily predict how output quality is degraded considering only approximate memory parameters. This is due to the fact that results do not depend only on the level of approximation but also depend on the specific target application and, as important remark, on the implementation of the application and how it is translated by the compilation toolchain into machine code and machine data structures.

In particular, this chapter presents the results of two different works and investigations:

- section 5.2 presents the study, the implementation and analysis of the impact of approximate memory on a H.264 software video encoder. After a phase of study, the code was internally modified in order to allocate selected data buffers in approximate memory. After compilation and link with the approximate alloc library, it was executed into an Intel x86 approximate platform (Linux AxM OS + AppropinQuo emulator). Results showing degradation in output video frames were produced, using approximate SRAMs and approximate DRAMs.

- section 5.3 presents the implementation of a software FIR filter with buffered input and output data. It represents a classical real-time application which receives a stream of data, apply filtering and send it to the output device. The specific input data that were processed are, actually, audio signals that can be listened to before and after filtering. Application performance were characterized in terms of output SNR.

- section 5.4 try to perform an analysis in a more complex case study, considering the concept of *quality aware approximate memory zones* which will be defined later. The application is still a digital FIR filter working on audio signals. This case study opened the way to the exploration of a new concept: the presence of memory zones with different level of approximation and the allocation of non-critical data structures of error tolerant applications depending on sensitivity to errors and desired output quality.

5.2 Impact of Approximate Memory on a H.264 Software Video Encoder

The contribution of this work is to propose a strategy for selecting error tolerant data structures and then allocate them in approximate memory, in order to extensively study the impact on the quality of H.264 video streams. Since the level of approximation (and hence power savings) is dependent on the amount of errors that the application requirements can tolerate, the results allow to discover a relationship between video output quality and hardware fault rate, which is the final metric to guide the relaxation of hardware design constraints to save power (energy quality tradeoff Huang, Lach, and Robins, 2012).

The activity has been developed in three phases which correspond to the following steps:

1. *Analysis of the x264 application*. First of all the x264 source code (Section 5.2.1) was analyzed in order to distinguish between the portions of code that could be approximated and the ones that have to be kept exact (see Section 5.2.2).

2. *Code modification (introduction of AxM allocations) and verification*. Once non-critical data have been identified, allocation on approximate memory are introduced in the x264 code (Section 5.2.2).

3. *Evaluation and results*. This phase was performed running the approximate x264 encoder on top of Linux in the *AppropinQuo* emulator, using different input video samples (both standard resolution and full HD videos) (see Section 5.2.3).

5.2.1 H.264 video encoding and the x264 encoder

H.264 (Fig.5.1), or MPEG-4 AVC is a video compression format developed especially for use in high definition video systems. One of the main goals of H.264 is the capability of providing good video quality at substantially lower bit rates than previous standards at the cost of additional computational complexity. Such complexity has been accompanied by the advancements in process technology that enabled the diffusion of high-performance (embedded) multimedia hardware platforms.

Due to its characteristics and flexibility, it is currently adopted in many video applications, ranging from HDTV broadcast to HD consumer products, portable video systems, including smartphones and internet streaming. Moreover it constitutes the core of many web video services, such as Youtube, Facebook, Vimeo, and Hulu and it is widely used by television broadcasters and ISPs. Because of its widespread use

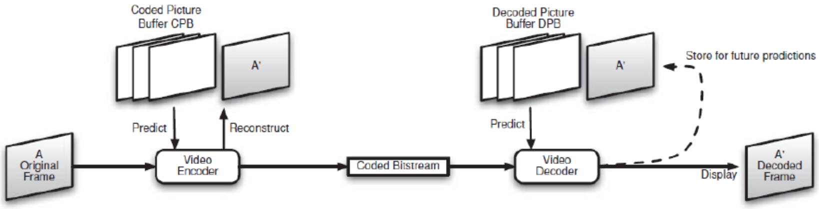


FIGURE 5.1: H.264 high level coding/decoding scheme

and computational requirements, advanced platforms for its efficient implementation, in terms of cost, power, quality, have been proposed. Research has been conducted on processing units and memory subsystems [Asma, Jarray, and Abdelkrim, 2017].

H.264 Encoder

In order to produce a compressed H.264 bitstream, the H.264 encoder perform the following operations:

1. *Prediction*. it generates block prediction by *motion estimation*. The video frame is processed by the encoder in unit of a *macroblock*[1]. Based on the previously coded data, the encoder makes a prediction of the current block from both the current frame, *intra-prediction*, and from the other frames that have already been coded, *inter-prediction* (Fig.5.2). Finally the encoder subtract this prediction from the macroblock forming a *residual*.

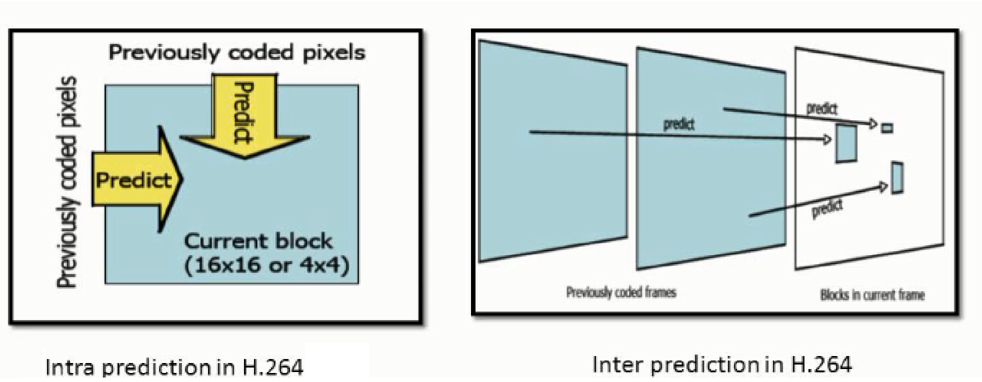


FIGURE 5.2: H.264 inter-frame and intra-frame prediction

2. *Transformation and quantization*. Using a 4x4 or 8x8 integer operation[2], the encoder transforms a block of residual samples: it converts the difference between the true value and the prediction into a set of coefficients. Each of these coefficients represents a weighting value for a basis pattern; combining together these coefficients it is possible to re-build the block of residual samples.

3. *Bitstream encoding*. the compressed bitstream is obtained encoding the data produced by the video coding process. In particular these data, called *syntax elements*, include:

[1]16x16 displayed pixels
[2]an approximate form of the DCT

- quantized transform coefficients;
- information used by the decoder to re-create the prediction;
- information dialing with the compression tools used during encoding and also the structure of the compressed data;
- information concerning the full video sequence.

H.264 Decoder

The decoding process is performed in three steps:

1. *Bitstream decoding*. After receiving the H.264 bitstream, *syntax elements* are decoded, extracting the information required to reverse the coding process (e.g. prediction information, quantized transform coefficients, etc.) and re-create the original video sequence.
2. *Rescaling and inverse transform*. The second step is to re-scale the quantized transform coefficients, by multiplying each coefficient for an integer value restoring the original scale. Then an inverse transform is applied to recreate each block of residual data: combining all these blocks together, the residual *macroblock* is restored.
3. *Reconstruction*. The decoder creates an identical prediction for each *macroblock*, adding this prediction to the decoded residual in order to reconstruct the decoded *macroblock*.

H.264 data fault resilience

Data fault resilience of H.264 algorithm has already been studied [Rehman et al., 2011; Shafique et al., 2017b], however, the approaches consider unwanted and random faults due to unreliable hardware platforms (smaller feature size of transistors, lower threshold voltage, and tighter noise margins render the modern multimedia platforms more susceptible to soft errors). These faults (manifested as spurious bit flips) can be characterized in terms of statistical probability, but cannot be controlled at data level (i.e. allowed only on data that are more tolerant to errors).

During this work, as typical in approximate computing, faults are intentionally allowed on selected data structures and with controlled and higher probability than the former works.

The *x264* software video encoder

The x264 encoder is a free software library and application for encoding video streams into H.264 [Merritt and Vanam, 2006]; it can be downloaded directly from VideoLAN official website [*VideoLan*] either as binary executable or as tarball collection of source code files. Due to its availability it has become one of the most widely used H.264 encoder in free and commercial applications, as well as in recent research works [De Cock et al., 2016].

An example of x264 command line is shown below:
./x264 –o output/test.264 input/bus_qcif_15fps.y4m

During the video processing, the *x264* prints on the terminal some useful information concerning the encoding process as it can be observed in Fig.5.3, where some information (e.g. video resolution, number of encoded frames) are highlighted.

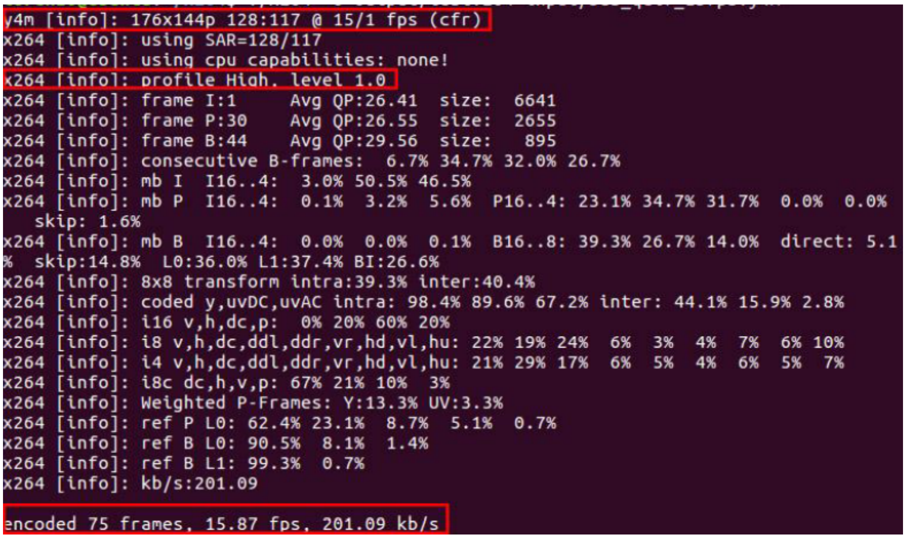


FIGURE 5.3: x264 encoding information

TABLE 5.1: Heap memory usage

video resolution	x264 option (preset)	peak heap[MB]	peak usefulheap [MB]
176x144	medium	15.6	15.4
704x576	veryfast	57.2	49.6
1920x1080	ultrafast	90.1	77.8
1920x1080	superfast	216.0	192.1
1920x1080	veryfast	269.0	238.6

Analysis of *x264* heap memory usage

Heap memory usage in *x264* has been characterized for different video resolutions and encoding options. Heap memory is commonly used by applications for the dynamic allocation of large memory buffers during data processing, which, for the x264 encoder and in general for ETAs, are good candidates to approximate memory storage.

It is expected larger memory requirements for higher resolution video, but also for different encoding options. Encoding options in *x264* set a tradeoff between encoding speed and output quality (considering the same bitrate) and are another source of increasing memory requirements. For practical use these options are grouped in presets ranging from high-speed/low-quality (*ultrafast* preset) to extremely low-peed/high-quality (*slow* preset).

Table 5.1 reports memory usage for different input video resolutions and encoding options, showing the expected dependency on them. *Peak heap* represents memory peak allocation while *useful heap* is the actual memory used for application data; the difference being memory consumed by allocation size rounding and administrative byte associated with each allocation. We note that not all heap can be allocated in approximate memory, since part of its data, typically called *critical data*, could be not tolerant to errors. A strategy for selecting candidates for approximate memory allocations is then required, and it is described in the following section.

```
88.69% (250,198,526B) (heap allocation functions) malloc/new/new[], -alloc-fns,
etc.
->88.03% (248,347,652B): x264_malloc
| ->70.33% (198,419,648B): x264_frame_new
| | ->70.33% (198,419,648B): x264_frame_pop_unused
| | ->41.92% (118,246,016B): x264_encoder_encode
| | | ->41.92% (118,246,016B):encode_frame
| | | ->41.92% (118,246,016B): main
| | |
| | ->12.18% (34,360,128B): x264_encoder_open_152
| | | ->12.18% (34,360,128B):main
| | |
| | ->12.18% (34,360,128B): x264_encoder_encode
| | |
| | ->04.06% (11,453,376B): x264_encoder_encode
| |
| ->08.82% (24,883,200B):x264_encoder_open_152
| | ->08.82% (24,883,200B):main
| |
| ->04.47% (12,603,136B): x264_macroblock_cache_allocate
...
| ->04.41% (12,441,668B): x264_encoder_open_152
...
```

FIGURE 5.4: Memory allocation profiling: Massif output

5.2.2 Approximate memory data allocation for the *x264* encoder

In order to select candidate data structures for approximate memory allocation, the *x264* memory usage traces have been analyzed during execution (memory profiling). All called functions were traced with respect to heap memory allocation and then analyzed in order to determined which data can be classified as non-critical for program execution.

The profiling has been performed using the *Valgrind* debug and profiling suite [Nethercote and Seward, 2007] and in particular the heap profiler tool called *Massif*. An example of command line for analyzing encoder allocation is shown below: *valgrind –tool=massif –time-unit=ms –detailed-freq=1 – massif-out-file=massif_out ./x264 –o output/test.264 input/ bus_qcif_15fps.y4m*

In Fig. 5.4 is reported an extract (peak memory sample) of *Massif* output for the encoding of a 1920x1080 resolution video and *veryfast* option setting. The following analysis is valid for other preset options and resolutions since, apart from absolute memory usage, relative percentages remain similar.

From the profiling reported it is possible to deduce that the total amount of *useful heap* memory, since the x264 application starts, is about 239MB. The largest part of heap memory allocation is indeed handled by function *x264_malloc*, which covers about 88.03% of total allocated heap memory. The function *x264_frame_new*, which calls *x264_malloc*, covers 70.33% of heap allocations.

The next step involved the analysis of source code in order to identify the actual data allocated by these functions. First the *x264_malloc* is analyzed, since it is responsible of the allocation of a large amount of heap memory. This function is implemented in <*common/common.c*> source file and it is used to allocate heap memory aligning data to 64 bytes.

```
void *x264_malloc( int i_size ){
uint8_t *align_buf = NULL;
#if HAVE_MALLOC_H
[. . .]
```

```
align_buf = memalign( NATIVE_ALIGN, i_size );
#else
// allocation code before approx_malloc's use
// uint8_t *buf = malloc( i_size + (NATIVE_ALIGN-1) + sizeof(void **) );
// allocation on approximate memory
uint8_t *buf = approx_malloc( i_size + (NATIVE_ALIGN-1) + sizeof(void
  **) );
if( buf ) {
align_buf = buf + (NATIVE_ALIGN-1) + sizeof(void **);
align_buf -= (intptr_t) align_buf & (NATIVE_ALIGN-1);
*( (void **) ( align_buf - sizeof(void **) ) ) = buf;
}
#endif
if( !align_buf )
x264_log( NULL, X264_LOG_ERROR, 'malloc of size %d failed\n', i_size );
return align_buf; }
```

By this analysis it is possible to discover that this function is too generic, handling also allocation of critical data structures. It could be classified as critical in *x264*, for example, data regarding encoder behavior, frames analysis, color space bits depth setting and encoding bitrate control. These data are critical because they are responsible of program control flow, which cannot be altered randomly by faults without completely compromising the encoding algorithm.

The second candidate is the *x264_frame_new* function, which is implemented in <*common/frame.c*>; it requests about 64% of *x264_malloc* allocated heap, moving these allocations to approximate memory would result indeed in reducing more than one half of the requirements for exact memory.

The analysis of the function *x264_frame_new* revealed that this routine is used to create and allocate frames for encoding or decoding the video, in the form of *frame structures* called *x264_frame_t* (defined in <*common/frame.h*>).

```
typedef struct x264_frame{
  /* */
  uint8_t *base; /* Base pointer for all malloced data in this frame. */
  int i_poc;
  int i_delta_poc[2];
  int i_type;
  int i_forced_type;
  int i_qpplus1;
  int64_t i_pts;
  int64_t i_dts;
  int64_t i_reordered_pts;
  int64_t i_duration; /* in SPS time_scale units (i.e 2 * timebase units)
    used for vfr */
  float f_duration; /* in seconds */
  [. . .]
  /* for unrestricted mv we allocate more data than needed
  * allocated data are stored in buffer */
  pixel *buffer[4];
  pixel *buffer_fld[4];
  pixel *buffer_lowres[4];
  [. . .]
  #if HAVE_OPENCL
  x264_frame_opencl_t opencl;
  #endif
  } x264_frame_t;
```

For each of these frames, the code allocates a heap space large enough to contain the whole picture buffer and other information, depending on encoder options. In particular the *x264_frame_t* structure, among others, stores data concerning frame encoding options, colors space information, buffers for frame pixels, motion vector buffers and rate control; some of this information is involved in the encoding control flow and must be still kept exact. Conversely, buffers for frame pixels are optimal candidates for approximate memory, because introducing errors in them does not alter program execution flow. Further analysis showed that image pixels are grouped into three different buffers, containing the pixel value for each color component, each 1-byte large (8-bit per pixel). In particular these buffers are:

- *pixel *buffer[4]*: contains image pixel data to be encoded;
- *pixel *buffer_fld[4]*: contains image pixel data when using fields, instead of frames, for interlaced encoding;
- *pixel *buffer_lowres[4]*: contains image pixel data of a low resolution version of reference frame. This is used to increase speed during stream decoding.

Each of these buffers is composed of three elements: one for each colorspace component plus an extra one that is never used during encoding, but it is used for data alignment and memory access efficiency. This analysis regarding internal data representation was revealed to be important for the optimization of approximate memory techniques, as will be discussed in the following section.

Once candidate buffers for approximate memory allocation were identified, changing the code in order to move them to approximate memory was straightforward, since in the target platform it is completely managed by the OS Stazi et al., 2017. The only ad-hoc coding was required since *x264* forces internally address aligment on some memory requests, by the definition od a *PREALLOC* macro. This was repeated for approximate memory alloc calls, introducing the macros APPROXMALLOC and APPROXFREE. These macros use *approx_malloc* and *approx_free* function calls (see Section 3.4.4) jointly with a data alignment mechanism. This manual alignment was implemented reusing and adapting the code of *PREALLOC*, implemented in the *x264_frame_new* function, and of *x264_malloc* function. In particular, the alignment of allocation size is taken from *PREALLOC* implementation while the allocation and alignment of the buffer from the *x264_malloc* function.

```
[. . .]
#ifdef APPROX
#define APPROXMALLOC(align_buf,size)\
do{\
size_t resize = ALIGN(size, NATIVE_ALIGN);\
uint8_t *buf = approx_malloc( resize + (NATIVE_ALIGN-1) + sizeof(void
  **) );\
if( buf ){\
align_buf = buf + (NATIVE_ALIGN-1) + sizeof(void **);\
align_buf -= (intptr_t) align_buf & (NATIVE_ALIGN-1);\
*( (void **) ( align_buf - sizeof(void **) ) ) = buf;\
}\
}while(0)
#define APPROXFREE(p) ( approx_free( *( ( ( void **) p ) - 1 ) ) )
#endif
[. . .]
```

As can be in the tex box above, in the APPROXMALLOC macro the original *malloc* call has been replaced by *approx_malloc* but the alignment commands are the same of *x264_malloc* function. APPROXFREE has the same implementation of *x264_free* where the *free* routine is replaced with an *approx_free* call.

An extract of the new *x264_frame_new* implementation is reported in the text box below.

```
static x264_frame_t *x264_frame_new( x264_t *h, int b_fdec ){
x264_frame_t *frame;
[. . .]
if( i_csp == X264_CSP_NV12 || i_csp == X264_CSP_NV16 ){
int chroma_padv = i_padv >> (i_csp == X264_CSP_NV12);
int chroma_plane_size = (frame->i_stride[1] * (frame->i_lines[1] + 2*
  chroma_padv));
#ifdef APPROX
APPROXMALLOC( frame->buffer[1], chroma_plane_size * sizeof(pixel) );
#else
PREALLOC( frame->buffer[1], chroma_plane_size * sizeof(pixel) );
#endif
[. . .]
}
for( int p = 0; p < luma_plane_count; p++ ){
int luma_plane_size = align_plane_size( frame->i_stride[p] * (frame->
  i_lines[p] + 2*i_padv), disalign );
if( h->param.analyse.i_subpel_refine && b_fdec ){
[...]
```

5.2.3 Experimental setup

The *x264* encoder, modified to allocate selected data buffers in approximate memory, was compiled and executed in the *AppropinQuo* emulator, running Linux kernel version 4.3 with support for approximate memory management and built for an Intel x86 architecture.

Input test files were selected from the Xiph.org Video Test Media (derf's collection) Montgomery, 1994. Given the large number of choices available, input video samples for test were selected with different resolutions, color and characterized by moving and still parts. A list of them is present in Table 5.2.

TABLE 5.2: Test videos from derf's collection

name	resolution	lenght [frames]
ducks_take_off	1080p	500
dinner	1080p	950
crowd_run	1080p	500
blue_sky	1080p	217
bus	CIF(176x144)	75
claire	QCIF	494
flower	CIF	250

The tests were executed configuring the approximate memory model in *AppropinQuo* for DRAM memories using slower refresh rate [Liu et al., 2012a; Raha et al., 2017] and for SRAM memories considering voltage scaling. Different fault rates (error probability) and bit level error masking (looseness level) were also explored.

As for DRAM, the range of fault rates was chosen according to a refresh rate increase ranging from 8x (256ms) up to 400x (25s), while bit level error masking allows to take into account more advanced approximate techniques that distinguish between bit weights (the quality of the user experience in multimedia application is mainly defined by the most significant bits [Kwon et al., 2012; Gong et al., 2012; Chang, Mohapatra, and Roy, 2011]).

Results are provided in terms of user perceived video quality, comparing original and coded frames. In particular, as quality metric, Peak signal-to-noise ratio (PSNR) was used, defined as the ratio between the maximum pixel value and rms of corrupting noise that affects the fidelity of its representation [Winkler and Mohandas, 2008].

All tests were executed with the *x264 veryfast* preset option, since this setting provides a good balance between encoding processing time and quality.

In order to produce reference values, first the original *x264* encoder, with buffers allocated in exact memory, was run. As quality metric, peak signal-to-noise ratio (PSNR) was used, defined as the ratio between the maximum pixel value and rms of corrupting noise that affects the fidelity of its representation Winkler and Mohandas, 2008.

5.2.4 Impact on output using approximate DRAM and power saving considerations

The results illustrated in this subsection are referred to HD videos (1980x1080 resolution) selected from the Xiph.org Video Test Media (derf's collection) Montgomery, 1994 with the following features:

- overall number of frames: 500;
- frame per seconds [fps]: 50;
- duration [s]: 10;
- input file format:YUV4MPEG (*.y4m*) . This format is used as a raw, color-sensitive video format before compression; in particular it stores a sequence of uncompressed *YCbCr* images that make up the video frame by frame.

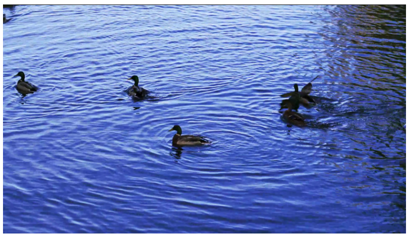
.

The global PSNR mean value, obtained on output videos using exact compression was 29.69 dB. This value should be considered an upper bound to evaluate the *x264* performances with the present settings. Tests were executed considering an approximate DRAM composed by true-cells, varying fault rate and bit level error masking (looseness level).

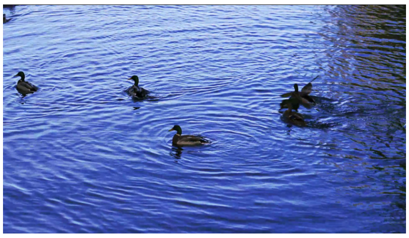
Table 5.4 shows the results of the same decoding using approximate memory. Global PSNR values are reported for each fault rate/looseness level combination. Fig. 5.5 and Fig. 5.6 provide a visual result of user perceived video quality, comparing the original frame (Fig. 5.5a and Fig. 5.6a) and the same frame using different approximate memory error rates and looseness levels. In particular, Fig. 5.5 refers to an error rate of 10^{-4} and three different looseness levels (0x0F0F0F0F, 0x1F1F1F1F, 0x3F3F3F3F). For this fault rate, a looseness level set to 0x3F3F3F3F (figure 5.5d) (which allows errors on the six LSBs of each 8-bit pixel data), still produces an output visually very close to the original frame. Lowering the looseness level does not have a significant impact on output while implying a larger number of exact bit cells. Fig. 5.6 is obtained for a fault rate set to 10^{-3}; in this case we can see that, for higher looseness levels (i.e. 0x3F3F3F3F, Fig. 5.6d), differences in output quality are starting to be noticeable.

It is possible to observe that for a fault rate of 10^{-3} $errors/(bit \times s)$ and a looseness mask set to 0x0F0F0F0F (i.e. error allowed on four LSBs), PSNR is 29.13 dB, or about 0.5 dB under the exact case, confirming good tolerance to errors. The table shows also that, with the same fault rate, all masks more protective than 0x0F0F0F0F

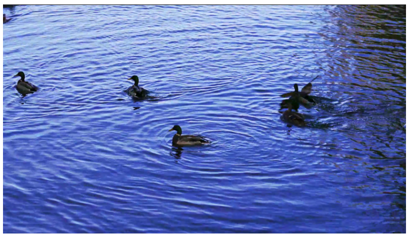
(A) Exact frame



(B) Looseness Mask 0x0F0F0F0F



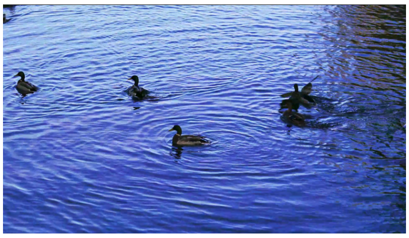
(C) Looseness Mask 0x1F1F1F1F



(D) Looseness Mask 0x3F3F3F3F

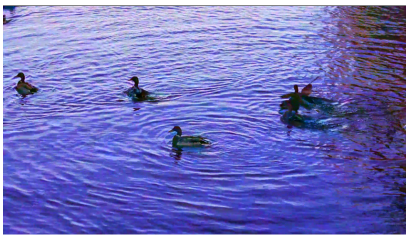
FIGURE 5.5: x264, output frame with different Looseness Levels and fault rate 10^{-4} $[errors/(bit \times s)]$

(A) Exact frame



(B) Looseness Mask 0x0F0F0F0F

(C) Looseness Mask 0x1F1F1F1F



(D) Looseness Mask 0x3F3F3F3F

FIGURE 5.6: x264, output frame with different Looseness Levels and fault rate 10^{-3} $[errors/(bit \times s)]$

TABLE 5.3: Video Output PSNR [dB]

Looseness mask	Fault rate [$errors/(bit \times s)$]		
	10^{-2}	10^{-3}	10^{-4}
0x3F3F3F3F	19.97	25.18	28.84
0x1F1F1F1F	24.47	28.01	29.43
0x0F0F0F0F	27.35	29.13	29.59
0x07070707	28.96	29.52	29.63
0x03030303	29.47	29.61	29.64
0x01010101	29.61	29.64	29.64

TABLE 5.4: x264, video output PSNR [dB] for approximate DRAM (true cells)

Looseness mask	Fault rate [$errors/(bit \times s)$]					Bit dropping
	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	
0xFFFFFFFF	10.87	16.02	22.98	27.99	29.41	_
0x7F7F7F7F	15.30	20.34	26.90	29.25	29.60	_
0x3F3F3F3F	19.97	25.18	28.84	29.56	29.63	_
0x1F1F1F1F	24.47	28.01	29.43	29.62	29.65	_
0x0F0F0F0F	27.35	29.13	29.59	29.64	29.65	24.14
0x07070707	28.96	29.52	29.63	29.65	29.65	26.30
0x03030303	29.47	29.61	29.64	29.65	29.65	28.32
0x01010101	29.61	29.64	29.64	29.65	29.65	29.34

(i.e 0x07...07, 0x03...03, 0x01...01) produce very close outputs, but would result in larger energy consumption (since they imply a larger portion of exact bits). Fig. 5.6 shows the visible effects, for 0x0F0F0F0F and 0x3F3F3F3F bit masks, on a portion of a frame.

Simulations with fault rate set to 10^{-2} $errors/(bit \times s)$, which is the worst case tested, illustrate that the 0x0F0F0F0F mask produces a PSNR value about 2 dB under the exact case, resulting in more visible effects of corruption on the output. Fig. 5.7 plots output PSNR for an extended fault rate range.

Power saving considerations

Actual power saving related to the application of our test cases can be extracted assuming as reference the results showed in [Raha et al., 2017]. Refresh power is dependent on refresh rate, if we assume a 10^{-3} error rate, a 60x increase in refresh period can be allowed. A looseness mask set to 0x0F0F0F0F means that half the cells must be exact while the other half can be approximate memory. In these tests, data structures selected to be allocated in approximate memory are about 60% of total data, resulting in a system where, globally, about 30% are approximate memory cells while 70% are exact memory cells.

According to this partition, and considering only refresh power, it is possible to expect a normalized refresh power in the range of 0.3-0.5 [Raha et al., 2017] with respect to the original exact implementation.

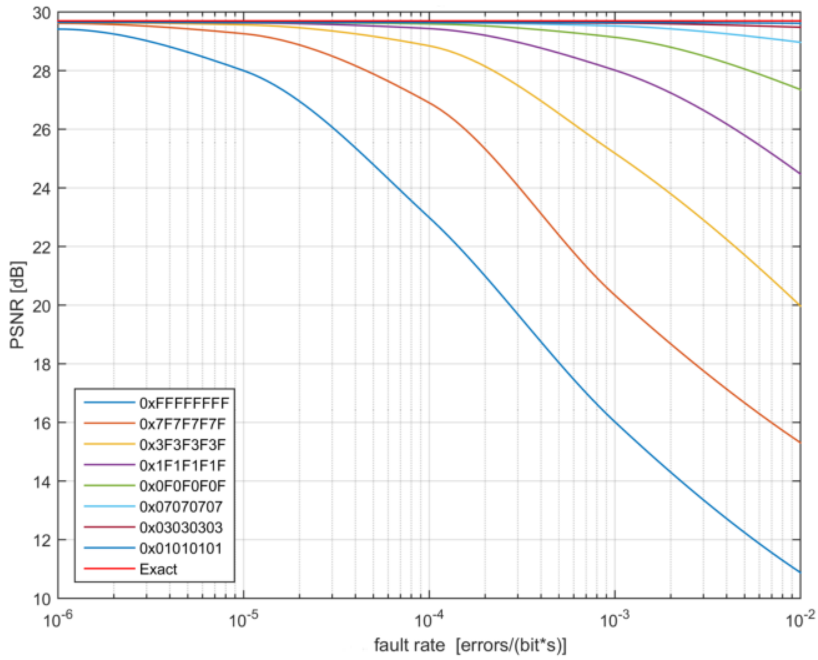


FIGURE 5.7: Video Output PSNR graph [dB]

5.2.5 Impact on output using approximate SRAM

The results illustrated in this section are referred to low resolution videos (176x144 resolution), selected again from the Xiph.org Video Test Media (derf's collection) Montgomery, 1994 with the following features:

- overall number of frames: 75;
- frame per seconds [fps]: 15;
- duration [s]: 5;
- file format:YUV4MPEG.

.

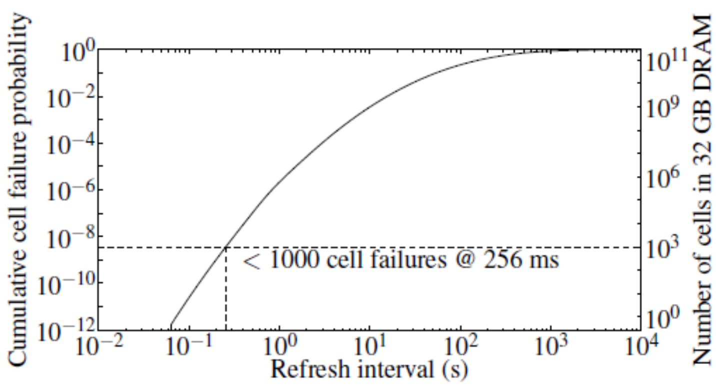


FIGURE 5.8: DRAM cell retention time distribution. Source:Liu et al., 2012a

TABLE 5.5: x264, video output PSNR [dB] for approximate SRAM (error on access)

Looseness mask	**Fault rate** [$errors/access$]				
	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}
0xFFFFFFFF	38.65	47.58	56.75	63.48	64.41
0x7F7F7F7F	44.17	53.92	61.70	64.41	64.41
0x3F3F3F3F	50.14	59.58	63.92	64.41	64.42
0x1F1F1F1F	56.09	62.67	64.34	64.42	64.42
0x0F0F0F0F	61.79	64.11	64.41	64.43	64.44

Tests were executed considering an approximate SRAM, where errors occurs both on read (destructive and non-destructive) and write accesses. As for the previous case, the original *x264* encoding was first run obtaining a global PSNR value of 64.46 dB for the exact algorithm.

Table 5.5 shows the global PSNR for several fault rate/looseness level combinations; in particular simulations were performed setting the fault rates (EOW, EOR, EOR_TX) to the same value just to reduce the number of cases presented in the table.

The effects of these PSNR values can be compared visually by looking at output frames: Fig. 5.9 shows the same frame obtained with exact compression (top left), and with three different fault rates using the higher looseness level (i.e all bits cells are affected by errors). As expected, the output quality for 10^{-6} (top right) is not much different from the exact one. This consideration can also be extended to the case of 10^{-4} (bottom left), which presents only minor artifacts despite it is more than 15dB under the exact PSNR. Finally the output frame obtained with a fault rate of 10^{-2} (bottom right) is the only one that clearly exhibits artifacts due to approximate memory.

5.2.6 Considerations on the results and possible future analysis

In the previuos section it was we presented an analysis of the *x264* software video encoder and the impact of using approximate memory for storing its error tolerant data structures. The work started by profiling memory usage and finding a strategy for selecting error tolerant data buffers. After that the modified application was run on the *AppropinQuo* emulator, for several combination of fault rates (derived from actual refresh rate reduction strategies) and fault masking at bit level (looseness level).

Results show the importance of exploring the relation between these parameters and output quality. For example, leaving some of the MSBs exact demonstrated to be an effective way of allowing error probabilities up to 100x higher with the same output quality and an estimated refresh period increase in the order of 60x.

Since leaving exact a portion of memory cells reduces global energy savings, this knowledge is also fundamental in order to drive research on hardware techniques specifically tailored to the application, revealing the tradeoff between designing more aggressive approximate circuits and the number of bit cells that must be kept exact.

Further works can consider better allocation strategies in order to increase the

FIGURE 5.9: x264, output frame coded with exact (top left) and approximate SRAM (0xFFFFFFFF looseness mask), fault rate 10^{-6} (top right), 10^{-4} (bottom left) and 10^{-2} (bottom right) [*errors*/*access*].

fraction of data allocated in approximate memory and more advanced DRAM architectures for embedded systems, as DRAMs chips with integrated ECC units. Another important aspect is a more accurate quantification of power savings, which could be obtained by integrating a power consumption model in the approximate DRAM memory model.

5.3 Study of the impact of approximate memory on a digital FIR filter design

In this section the impact of different approximate memory configurations on a signal processing application (digital FIR filter) is shown. The digital FIR filter consists of 100 taps implemented using 32-bit integer arithmetic. The FIR structure is shown in Fig. 5.10; it is possible to note the presence of two buffers,input and output, whose dimensions can be configured by user.

Since this is a generic implementation, audio signals were selected as input (extracted from uncompressed WAV file, 44100 Hz cample rate, single-channel and 16-bit samples. During operation the FIR application performs the following operations:

- copy the samples in the input buffer, until it is full;
- perform filtering;
- copy the output buffer in the output file until it is empty;
- repeat from step 1 until end of input file.

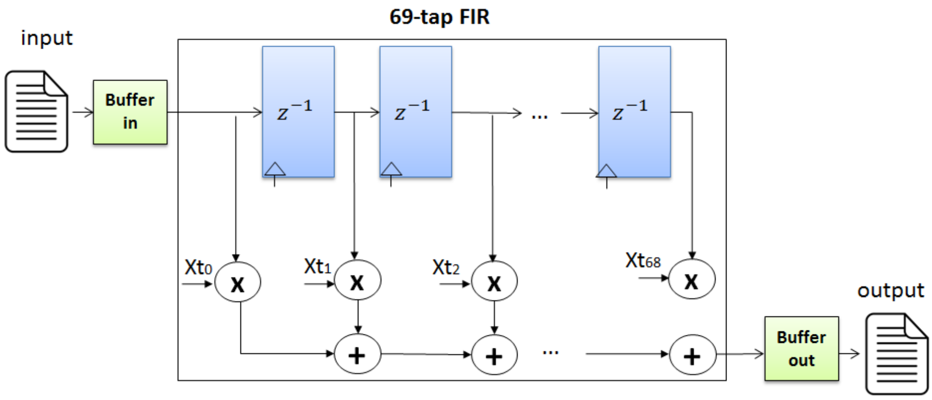


FIGURE 5.10: Digital FIR architecture

The input buffer, output buffer and internal tap registers are allocated in approximate memory, for a total of about 200KByte of memory space. The filter coefficients instead, being very sensitive to errors (even small variations produce large movement in the position of poles), are considered critical and consequently allocated in exact memory. The text box below lists the FIRConfig function, responsible for the configuration of the filter, including the allocation of buffers.

```
DATA_TYPE* FIRconfig(char* filename, unsigned int* ord, DATA_TYPE** IR,
    unsigned int* size_input,
DATA_TYPE** buffer_in, DATA_TYPE** buffer_out){

  FILE* fp;
  char insize[1024];
  char order[1024];
  char coeffbuffer[1024];

  int f;
  DATA_TYPE* regfile;
  DATA_TYPE* impulsive_r;
  DATA_TYPE* buffer1;
  DATA_TYPE* buffer2;
  unsigned int fir_order, inbuff_size;

  if((fp=fopen(filename, "r"))==NULL){
    return NULL;
  }

  fgets(insize, sizeof(insize), fp);
  inbuff_size=strtol(insize, NULL,10);

  fgets(order,sizeof(order), fp);
  fir_order=strtol(order, NULL,10);

  *ord = fir_order;

  //Buffers allocation
  buffer1 = approx_malloc(KBYTE*inbuff_size);
  buffer2 = approx_malloc((KBYTE*inbuff_size)+(fir_order -1)*sizeof(
    DATA_TYPE));
  regfile = approx_malloc(sizeof(DATA_TYPE)*fir_order);
  memset(regfile, 0, sizeof(DATA_TYPE)*fir_order);

  impulsive_r = malloc(sizeof(DATA_TYPE)*fir_order);

```

```
  for(f=0;f<fir_order;f++){
    fgets(coeffbuffer, sizeof(coeffbuffer), fp);
    sscanf(coeffbuffer, "%d", &impulsive_r[f]);
  }

  fclose(fp);

  *size_input=inbuff_size*KBYTE;
  *IR=impulsive_r;
  *buffer_in=buffer1;
  *buffer_out=buffer2;

  s_of_outbuff=((KBYTE)*inbuff_size)+(fir_order -1)*sizeof(DATA_TYPE);

  return regfile;
}
```

For the set of simulations, the following configuration was chosen:

- Data type (for coefficients and audio samples): integer, i.e. fixed point;
- FIR filter with 100 taps, integer coefficients generated with MATLAB [Shampine and Reichelt, 1997], low-pass filter);
- input and output buffer size: 100KB each;
- input .wav audio files: 44100Hz sample rate, 16-bit samples, single channel.

The tests were run by executing the application inside *AppropinQuo* (on top of Linux OS) using the following command line:

.fir -fir FIRp -out output -in input

where *FIRp* indicates the file containing the coefficients and output and input correspond respectively to the output and input files. The quality metric for this application is the output SNR; it is measured considering noise as the difference between the output of the exact filter and the output of the approximate filter.

As a common consideration on the value of minimum required SNR, this strictly depend on application. Since, in this case, the filtering is applied to an audio signal, a value from $60dB$ to $90dB$ could be considered of interest by most applications.

In the following, the results concerning the impact on output using approximate DRAM (Section 5.3) and approximate SRAM (Sections 5.3, 5.3) are shown. Several simulations were executed by varying error rate and *looseness level*. In particular, to assess the influence of each key parameter (error rate and looseness level) on the SNR two groups of simulations were performed:

- simulations with a constant looseness level and variable fault rate to evaluate the impact of the fault rate on the SNR;
- simulations with variable looseness level and constant fault rate to evaluate the variation of the SNR in response to the variation of the looseness level.

Impact on output using approximate DRAM

Fig. 5.11 shows the output SNR, with respect to the exact case, for an hardware platform using approximate DRAM memory. In the hypothesis of a memory circuit consisting of anti-cells, different error rates and looseness levels are explored. As reference, an error rate of about 10^{-3} is obtained in case of a 60x increase in refresh period [Raha et al., 2017]. The figure also shows that looseness level can be used in order to rise SNR orthogonally to error rate, at the cost of keeping some bits exact. As expected in this application, each exact bit as an impact of about $6dB$ on SNR.

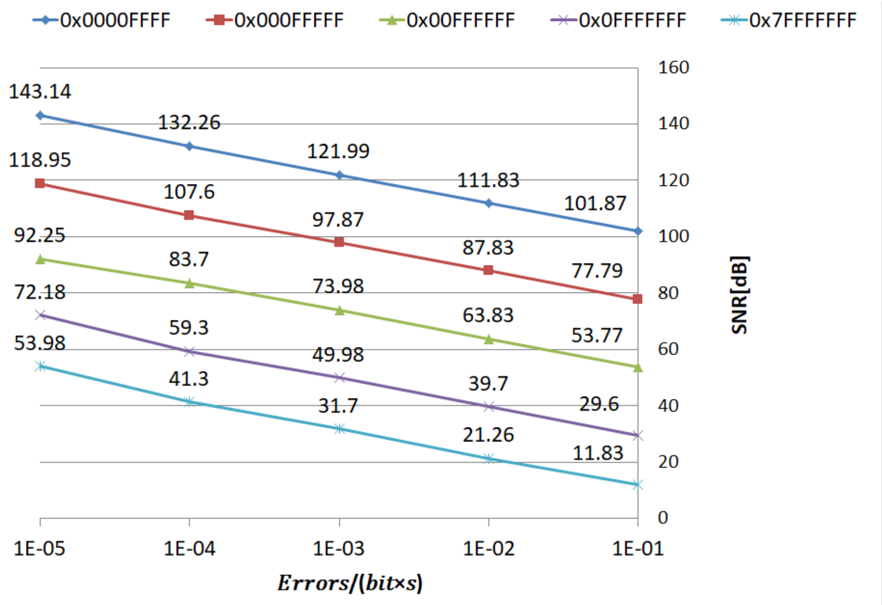


FIGURE 5.11: FIR, output SNR [dB] for approximate DRAM (anti cells)

Impact on output using approximate SRAM

Table 5.6 reports the output SNR in case of an approximate SRAM memory. The SRAM parameters are explored in the corners, i.e. cells affected only by:

1. error on write (EOW);
2. destructive error on read (EOR);
3. non destructive error (EOR_TX).

In this case, due to the access pattern of the application, it is possible to note that the EOW and the EOR_TX cases produce higher SNR than the EOR case. Again, by modulating looseness level, higher SNR values can be obtained for the same error rate.

Impact on output using approximate SRAM with bit dropping

Table 5.7 reports the output SNR in case of the bit dropping technique applied to an approximate SRAM. The first column lists the value of the SNR obtained by keeping 28 bits exact and dropping the first four LSBs. Further columns were obtained by increasing the number of dropped bits four at a time. The results confirm that LSB dropping is a valid approach in case of high BER (as can happen in case of V_{DD} scaling at voltages below the minimum operating voltage Frustaci et al., 2016), since it completely eliminates the energy associated with dropped bitlines.

TABLE 5.6: FIR, output SNR [dB] for SRAM

	Looseness Level	Fault rate [$errors/(bit \times s)$]			
		10^{-1}	10^{-2}	10^{-3}	10^{-4}
EOW	0x0000FFFF	94.6	107.3	117.6	127.3
EOR		90.6	104.2	114.9	125.0
EORnd		94.5	107.2	117.5	127.5
EOW	0x000FFFFF	70.5	83.4	93.5	104.2
EOR		66.4	80.3	90.9	101.4
EORnd		74.2	84.0	94.1	103.7
EOW	0x00FFFFFF	46.5	59.6	69.3	80.3
EOR		42.6	56.3	66.8	77.5
EORnd		45.9	59.8	69.9	79.9
EOW	0x0FFFFFFF	22.6	35.3	45.5	56.4
EOR		18.9	32.9	42.8	53.4
EORnd		25.2	33.0	45.8	56.0
EOW	0x7FFFFFFF	4.6	17.2	27.6	38.2
EOR		1.0	14.8	24.9	35.5
EORnd		6.2	17.5	27.8	37.8

TABLE 5.7: FIR, SNR [dB] for SRAM bit dropping

# of dropped LSBs						
4 bits	**8 bits**	**12 bits**	**16 bits**	**20 bits**	**24 bits**	**28 bits**
134.7	122.4	106.1	82.2	52.9	28.2	4.0

5.4 Quality aware approximate memories, an example application on digital FIR filtering

In this section it is presented a first analysis of the application of quality aware memory allocation, applied to digital FIR filter application, working on audio signals. We chose again the software FIR filter, set to 69taps and implemented in C language using 32 bit integer arithmetic described in Section 5.3. The application has been implemented in order to allocate the input and output buffers (indicated in green in Fig. 5.10) and the tap registers (showed in blue) in different approximate memory zones. They are respectively ZONE_APPROXIMATE1 and ZONE_APPROXIMATE2 and they have distinct approximate levels (ZONE_APPROXIMATE1 has an higher level of approximation (i.e. higher error rate) than ZONE_APPROXIMATE2). This choice is due to the fact that, in the implementation, the input and output buffer locations are accessed less frequently with respect to the tap registers. The text box below show the relevant code regarding the allocatio of the above mentioned data structures.

```
/*Buffer allocation*/
/*Allocation in ZONE_APPROXIMATE */
  buffer1 = approx_malloc(KBYTE*inbuff_size, 1);
/*Allocation in ZONE_APPROXIMATE */
  buffer2 = approx_malloc((KBYTE*inbuff_size)+(fir_order -1)*sizeof(
    DATA_TYPE), 1);
```

```
/*Allocation in ZONE_APPROXIMATE 2 */
 regfile = approx_malloc(sizeof(DATA_TYPE)*fir_order, 2);
 memset(regfile, 0, sizeof(DATA_TYPE)*fir_order);

 impulsive_r = malloc(sizeof(DATA_TYPE)*fir_order);
```

Again, the application was executed on top of Linux Kernel in *AppropinQuo*, configured in order to the following architecture:

- RISC-V 64-bit CPU, for which the kernel was compiled;
- RISC-V SiFiveU platform, with 256MB RAM memory;
- the 256MB RAM are partitioned into 128MB exact RAM, 64MB approximate memory (level 1), 64MB approximate memory (level 2);
- destructive error on read (EOR), destructive error on write (EOW).

Table 5.8 reports the relevant data. The input and output buffers size is 100Kbyte (they contain 25,000 32-bit samples), while the tap registers size is 276 bytes (for 69 32-bit registers). Access tracing reveals that, as expected, the tap registers array is accessed about two orders of magnitude more than the input and output buffers.

Different combinations of levels of approximation for ZONE_APPROXIMATE1 and ZONE_APPROXIMATE2 have been analyzed. Considering the order showed in Fig. 3.26, fault rate of ZONE_APPROXIMATE1 will be higher or equal than that of ZONE_APPROXIMATE2. A starting point is when fault rates are equal, since it corresponds to the case of having just one ZONE_APPROXIMATE for all non-critical data.

Table 5.9 and Table 5.10 show the results considering the two opposite corners of an approximate SRAMs, EOR and EOW. As quality metric, we used SNR, measured considering noise as the difference between the output of the exact filter and the output of the approximate filter. The diagonal values correspond to the case of a single ZONE_APPROXIMATE for all approximate data; on a row, moving from the diagonal to the adiacent element, reveals that if tap registers are allocated in memory with a fault rate 10 times lower, a gain of 7 to 8 dB in SNR is obtained for the EOR case. This gain is quite repeatable across all cases, while further reducing fault rate of factors of 100, 1000, etc. produces minor advantages. Table 5.10 shows how the same concept is valid in the EOW corner, but, since in this case application tap register are read are about twice than write accesses, SNR gain is 6 to 7 dB.

This study opens a new point of research, the presence of more that one level of approximation in memory and possibility of optimize the allocation of non-critical data for a specific implementation. In this case study the allocation was performed manually, but further investigations could go in the direction of an automatic strategy for the allocation of data, looking for an optimal point in the quality-energy tradeoff curve.

Future works will target more complex applications requiring larger memory size, exploring the use of multiple approximate zones. Automatic allocation strategies will also be considered as a way of reaching more significant savings.

TABLE 5.8: FIR, access count on approximate data structures

	buffer_in	**buffer_out**	**tap regs**
size [bytes]	100,000	100,000	276
#read/location	196	196	10,035,200
#write/location	196	196	5,017,601
#total_reads	4,900,000	4,900,000	692,428,800
#total_writes	4,900,000	4,900,000	346,214,469

TABLE 5.9: FIR, output SNR [dB] for SRAM, EOR

	Fault rate (buffers) [$errors/access$]	**Fault rate** (taps) [$errors/access$]				
		10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
EOR	10^{-2}	32.9	37.52	43.64	44.17	44.19
EOR	10^{-3}	_	42.8	50.92	53.57	53.96
EOR	10^{-4}	_	_	53.4	60.85	63.84
EOR	10^{-5}	_	_	_	63.09	71.01
EOR	10^{-6}	_	_	_	_	72.92

TABLE 5.10: FIR, output SNR [dB] for SRAM, EOW

	Fault rate (buffers) [$errors/access$]	**Fault rate** (taps) [$errors/access$]				
		10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
EOW	10^{-2}	35.3	41.18	43.94	44.12	44.1
EOW	10^{-3}	_	45.5	52.17	53.34	53.79
EOW	10^{-4}	_	_	56.4	61.58	63.19
EOW	10^{-5}	_	_	_	65.69	69.84
EOW	10^{-6}	_	_	_	_	78.11

Chapter 6

Synthesis Time Reconfigurable Floating Point Unit for Transprecision Computing

6.1 Introduction and previous works

In this chapter the design and the implementation of a fully combinatorial floating point unit (FPU), using VHDL hardware description language, is presented. The FPU can be reconfigured at synthesis time in order to use an arbitrary number of bits for the mantissa and exponent, and it can be implemented in order to support all IEEE-754 compliant FP formats but also non-standard FP formats, exploring the tradeoff between precision (mantissa field), dynamic range (exponent field) and hardware physical resource requirements.

This work is inspired by the consideration that, in modern low power embedded systems, the execution of floating point operations represents a significant contribution to energy consumption (up to 50% of the energy consumed by CPU). In [Tagliavini et al., 2018] experimental results show that 30% of the energy consumption is due to FP operations and an additional 20% is caused by moving FP operands from memory and registers and viceversa. Several works try to overcome the limitations of fixed-format FP types: for example in [Bailey et al., 2002; Fousse et al., 2007] multi-precision arithmetic software libraries for performing computations on numbers with arbitrary precision are proposed. In particular, these libraries (*APREC* and *MPFR*) are mainly used where a high dynamic range is required and they cannot be adapted for the exploration of FP types with less than 32 bits length. In fact since these libraries use an entire machine word for the exponent, they can't reproduce the behavior of FPUs with reduced precision formats since the tuning the dynamic range is not possible. The *Softload* library [Hauser, 1996] implements all the IEEE-754 FP types, allowing to accurate emulate the operations performed by a HW FPU. This library can be easily extended to support FP types with arbitrary precision. The drawback of this implementation is that the program execution are very slow since all the computation are performed in software. In [Tagliavini et al., 2018] the authors present a transprecision FPU capable of handling 8-bit and 16-bit operations in addition to the IEEE-754 compliant formats (Fig. 6.1). In this scenario, the adoption of multiple FP formats, with a tunable number of bits for the mantissa and the exponent fields, is very interesting for reducing energy consumption and, simplifying the circuit, area and propagation delay. Adopting multiple FP formats on the same platform complies with the concept of *transprecision computing*, since it allows fine-grained control of approximation while meeting the required constraints on the precision of output results.

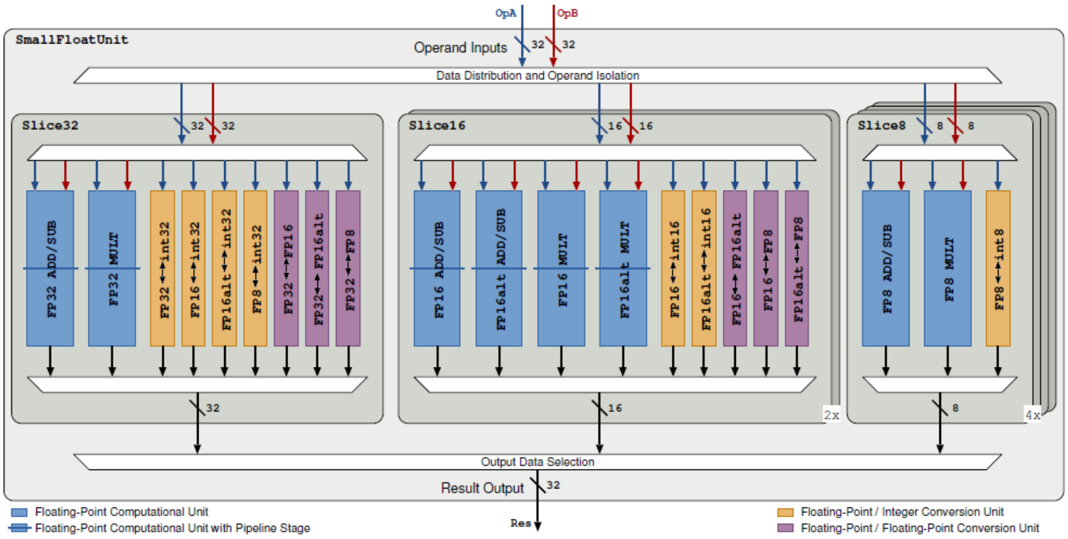


FIGURE 6.1: Block diagram of the FPU hardware datapath. Source:[Tagliavini et al., 2018]

6.2 Floating Point representation, IEEE-754 standard

The IEEE floating point standard IEEE-754 [Kahan, 1996] is a technical standard for floating-point computation, originally established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).

The IEEE-754 standard defines:

- a basic and an extended format;
- representation of special numbers: zero, negative and positive infinity, and NaN (Not a Number);
- precision, monotony and identity requirements for conversions between decimal numbers and binary floating point numbers;
- five types of exceptions and their management;
- operations of sum, subtraction, product, division, square root, comparison, rest, integer to FP, FP to integer conversion and conversion between different floating point formats;
- four types of rounding;
- rounding accuracy.

$$X = (-1^s) \times 1.m_b \times b^e \tag{6.1}$$

According to Equation 6.1 a floating point number consists of three fields: a sign bit (s), a biased exponent(e) and a mantissa (m).

Depending on the number of bits associated to the mantissa m and the exponent e, IEEE-754 standard defines several representation formats, that differ on dynamic range of representable numbers (determined by the exponent) and on precision (determined by the mantissa). In particular, indicating with k the total number of bits of the floating point number and with p the number of bits of the mantissa including the *hidden bit* (implicit bit), the FP number has:

- 1 bit for sign determination;
- (p-1) bits dedicated to the mantissa;
- (k-p) bits dedicated to the exponent;

IEEE-754 defines *half precision* floating-point format with 1 sign bit, 5-bit exponent, and 11-bit mantissa, *single precision* format with 1 sign bit, 8-bit exponent, and 23-bit mantissa and *double precision* format with 1 sign bit, 11-bit exponent, and 52-bit mantissa (Fig. 6.2). The advantage of floating-point over fixed-point is to extend the range of numbers that can be represented with the same number of bits, providing robustness to overflows.

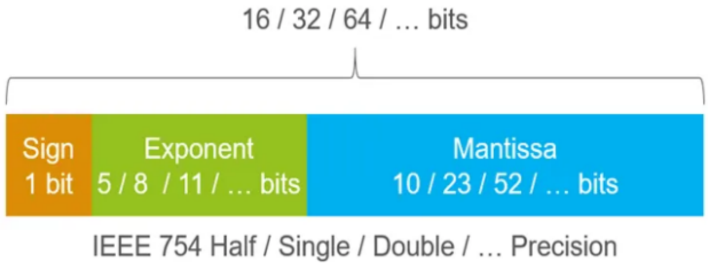


FIGURE 6.2: IEEE 754 Precision formats

In the floating point representation there is not a one-to-one correspondence between real numbers and floating point numbers, meaning that a real number x can be represented by different triplet, e.g. $sign_1, exponent_1, mantissa_1$ .. $sign_n, exponent_n, mantissa_n$ while a single triplet represent only one real number. To overcome this limitation, two different formats have been defined, based on the *biased exponent* concept[1]:

- normalized number: $e \neq 0$, $1 <= m < 2$;
- denormalized number: $e \neq 0$, $0 <= m < 1$.

6.3 Design of the reconfigurable Floating Point Unit

The primary goal of this work is the design and implementation of a fully combinatorial and reconfigurable FPU using VHDL hardware description language at Register Transfer Level (RTL). The unit allows to perform operations on floating point numbers in any format with a maximum length of 64 bits and with an arbitrary number of bits dedicated to the exponent and to the mantissa fields. In this way, it is possible to support floating point operations fully compliant with those defined by the IEEE-754 standard (*double-precision*, *single-precision* and *half-precision*) and operations with non standard precision formats.

The FPU has been designed as reconfigurable at synthesis time by declaring the length of all signals and variables as function of two generic types, m and e, used respectively for defining the length of the mantissa and the exponent.

The hardware architecture has been designed trying to satisfy the following targets:

- reduced area occupation;

[1]a biased exponent is obtained by adding a constant (*bias*) to the exponent to make the range of the exponent non negative.

- low power consumption.

The FPU is fully combinatorial, in order to be inserted in a CPU 1-cycle pipeline stage. The implemented operations are:

- sum;
- subtraction;
- multiplication;
- conversion from floating point to integer;
- conversion from integer to floating point.

The HW description of the FPU architecture has been divided into sub-blocks and the *Floating_Point_Unit_core* block represents the top unit. It is illustrated in the following section.

6.3.1 Top unit Floating_Point_Unit_core

Fig. 6.3 represents the *Floating_Point_Unit_core* internal diagram. The FPU core can be divided into three different part: on top there are the blocks that receive and process the input signals (*operand1* and *operand2*).

The FPU does not support operations between denormalized numbers (exponent seto to zero and mantissa different from 0), because they would led to a larger use of logic and therefore of area occupation in exchange for a marginal increase in accuracy. The approach is indeed the same one used in the VFP of ARM processors: all denormalized numbers in input to the FPU are directly approximated to 0. The check on the two operands is performed by the block *flush_to_zero* (Fig.6.3): if the inputs are denormalized then the block replaces the operands with bits composed by all 0, with the exception of the most significant bit (which corresponds to the bit sign). In the middle layer there are the blocks responsible of the computation. In particular the main blocks are:

- *adder*: it performs the algebraic sum of the two input operands;
- *right shifter*: it is necessary to align the mantissa of the two operands when performing addition or subtraction;
- *multiplier*: performs the multiplication between the mantissa of the two input operands when the operation to be performed is a multiplication, otherwise the output at the multiplier will be 0;
- *normalizer*: it is used to correctly handle the hidden bit, ensuring the the first bit different from 0 corresponds to the hidden bit.

Finally the bottom part is composed by all blocks (e.g. trunk, round, etc.) that prepare the output result.

Fig. 6.4 shows the FPU external interface.

Specifically, the FPU input signals are:

- *operand1 [(m+e) - 0]*: bit vector, first operand
- *operand2 [(m+e) - 0]*: bit vector, second operand

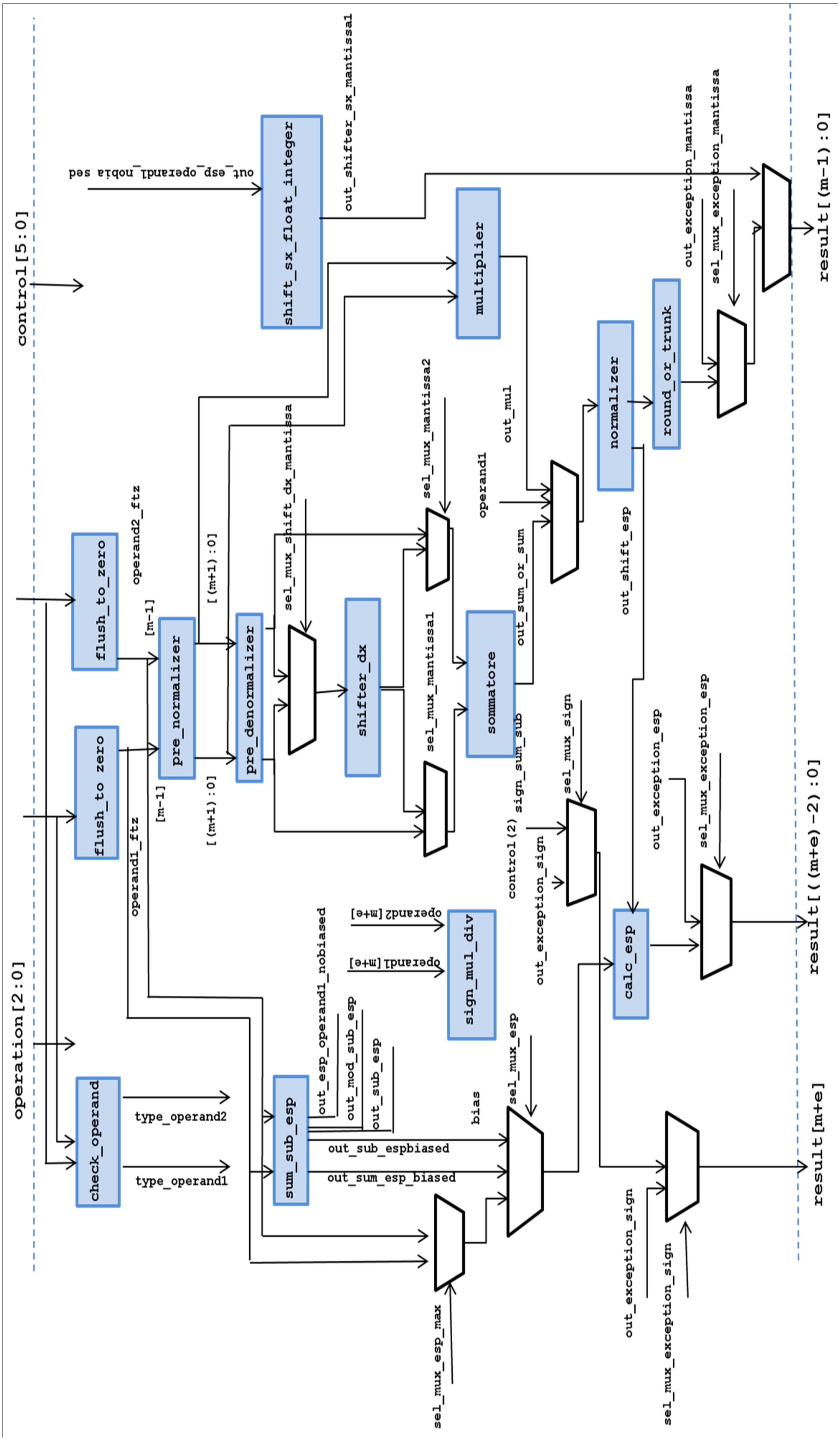


FIGURE 6.3: Floating Point Unit Core architecture

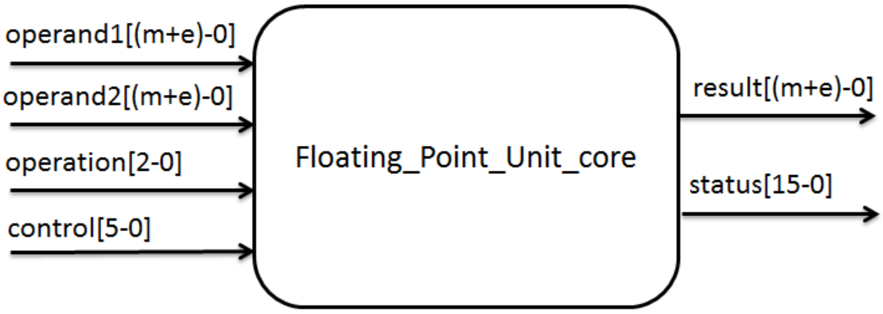


FIGURE 6.4: External interface of FPU core

TABLE 6.1: Operation codes

code	Operation
000	sum
001	subtraction
010	multiplication
011	_
100	integer to float
101	float to integer
110	_
111	_

- *operation [0-2]*; bit vector, it encodes the type of operation requested (Table 6.1);
- *control [5-0]*, bit vector which configures the unit behavior during computation. It is used for passing sign during integer to float conversion and it determines whether exceptions are enabled on the status port. In particular:
  - *control[0]*: unused;
  - *control[1]*: '1' exceptions are enabled on the output port, '0' exceptions are disabled;
  - *control[2]*: '1' attributes sign '-' to the result of integer-float conversion, '0' attributes sign '+' to the result of integer-float conversion.
  - *control[3]*: unused;
  - *control[4]*: unused;
  - *control[5]*: unused;

The output signals are:

- *result [(m+e) - 0]*, bit vector which provides the result of the performed operation. Again, its length depends on the number of digits for exponent and mantissa (e and m parameters).
- *status [15 - 0]*, bit vector which contains information regarding occurred exceptions. In particular:
  - *status [0]*: flag_invalid_operation. '1': operation is invalid, '0' operation is valid.
  - *status [1]*: flag_inexact_result. '1': rounding error or underflow error, '0' result is exact.

- *status [2]*: unused. Default: 0.
- *status [3]*: flag_overflow.'1' overflow occurs, '0' no overflow.
- *status [4]*: flag_underflow.'1' underflow occurs, '0' no underflow.
- *status [5]*: '1' result is zero. '0' result is different from 0.
- *status [6]*: '1' operand1 has a negative sign. '0' operand1 doesn't have a negative sign.
- *status [7]*: '1' operand1 is 0. '0' operand1 is different from 0.
- *status [8]*: '1' operand2 is 0. '0' operand2 is different from 0.
- *status [9]*: operand1 is ∞. '0' operand1 is different from ∞.
- *status [10]*: operand2 is ∞. '0' operand2 is different from ∞.
- *status [11]*: operand1 is sNaN. '0' operand1 is different from sNaN.
- *status [12]*: operand2 is sNaN. '0' operand2 is different from sNaN.
- *status [13]*: operand1 is qNaN. '0' operand1 is different from qNaN.
- *status [14]*: operand2 is qNaN. '0' operand2 is different from qNaN.
- *status [15]*: unused. Default: 0.

The FPU supports the following exceptions:

- *inexact result*: A result is inexact when its value cannot be represented by the floating point format in use. It is known indeed that between two successive numbers of a given interval there is a gap of representation, if the result falls just into that gap, it will be rounded, causing a certain amount of information to be lost. The inexact exception affects both m and e.
- *invalid operation*: the result is set to *qNaN*. This exception can be triggered in the following cases:
  - one of the two operands is equal to *sNaN* (signaling NaN).
  - conversion from float to integer if one of the operands is equal to NaN, ∞or if it has to be converted in an integer which exceeds the intervals of representable numbers.
  - the operand is not in the target format.
  - Adding plus infinity to minus infinity, subtracting an infinity from itself: $+\infty-(+\infty)$,$+\infty+(-\infty)$, $-\infty+(+\infty)$, $-\infty-(-\infty)$;
  - Multiplying infinity by zero: (+/- 0) x (+/- ∞);
  - Dividing zero by zero, or dividing infinity by infinity: (+/- 0) / (+/- 0); (∞) / (∞).
- *underflow*: the result of the operation can not be represented in a normalized format since it is too small.
- *overflow*: the result of the operation can not be represented in a normalized format since it is too large. This happens, for example, adding the largest representable number to itself.

When a case of underflow is detected, the result is set to 0, when instead an overflow occurs, the result is set to infinity. As far as the exception of an invalid operation is concerned, the result is set to *NaN*.

TABLE 6.2: List of analyzed formats

#bit sign	#bit exponent	#bit mantissa	Total bits
1	5	10	16 (IEEE half precision)
1	6	11	18
1	5	12	18
1	6	13	20
1	5	14	20
1	7	16	24
1	6	17	24
1	8	23	32 (IEEE single precision)
1	10	37	48
1	9	38	48
1	11	52	64 (IEEE double precision)

6.4 Experimental Results

6.4.1 Testing

The testing phase, performed immediately after the FPU design, represented an important step in order to assure that the computational unit operates according to its design specifications and produces the correct results. To verify the behavior of the designed FPU core for a variety of inputs, a testbench has been built. The testbench inserts input test vectors, automatically generated by a C program, into the FPU and then compares the results processed by the computational core with the output produced by the C program (using hardware FP). All validation steps were performed simulating the FPU at behavioral level using *Modelsim SE 10.1c*.

6.4.2 Synthesis Setup

After testing at behavioral level, the FPU was synthesized considering as target a Xilinx Kintex-7 FPGA (device xc7k325tfbv900-2) using *Xilinx Vivado Design Suite*. Since the unit is reconfigurable, multiple implementations were produced after having defined of the FP formats of interest.

Considering a FP number of predefined length, the partition of bits between the mantissa and the exponent fields has an impact on the represented numbers, enforcing a trade-off between dynamic range and precision; in particular the number of bits in the exponent field affects the range of numbers that can be represented while in the mantissa field sets the precision of the represented number.

The point of interest in this work was centered on reduced precision non-standard formats from a minimum of 16 bits and under 32 bits; this choice is supported by the consideration that IEEE *single precision* format (32-bit) is often not necessary for applications in embedded domains while IEEE *half precision* can be affected by serious underflow/overflow problems.

Maintaining a proportion between mantissa and exponent bits similar to the IEEE-754 standard formats, reduced precision formats of interest have been determined. Table 6.2 lists all formats taken under investigation, as we can see there are both standard and non-standard FP formats.

Since the FPU is fully combinatorial, it does not have an input clock signal. As timing constraint for the project, a *virtual clock*, which is not connected to any design object, was used. The virtual clock constraint was extensively used to force the optimizations during synthesis, in order to obtain the maximum speed for each configuration (FP format). In order to obtain data that reflect the effective proportions in gates number and area occupation, the synthesis was configured with *hardware DSP blocks* disabled. In this way the synthesizer can use only logic gates blocks.

This setup was required in order to compare the resources required by different FP formats, which otherwise would have been implemented using DSP blocks.

6.4.3 Results

Number of gates and resources

Table 6.3 shows the results gathered from the *Utilization report*. This report has been produced after the implementation of the FPU core at 40 MHz clock speed (25 ns is the minimum period obtained for the single precision FP format) and it collects data regarding number of LUTs and slices used in the unit. It can be seen that the reduced precision FP formats allow to significantly limit hardware resources (Fig. 6.5).

TABLE 6.3: Resources @ 40MHz clock with DSP disabled

Precision	**#slice**	**#LUT**	**% resources rel. to single precision**
half	191	624	37.9%
m11_e6	198	664	39.2%
m12_e5	209	730	41.5%
m13_e6	238	796	47.2%
m14_e5	243	807	48.2%
m16_e7	299	1037	59.3%
m17_e6	238	1114	67.0%
single	504	1787	100%

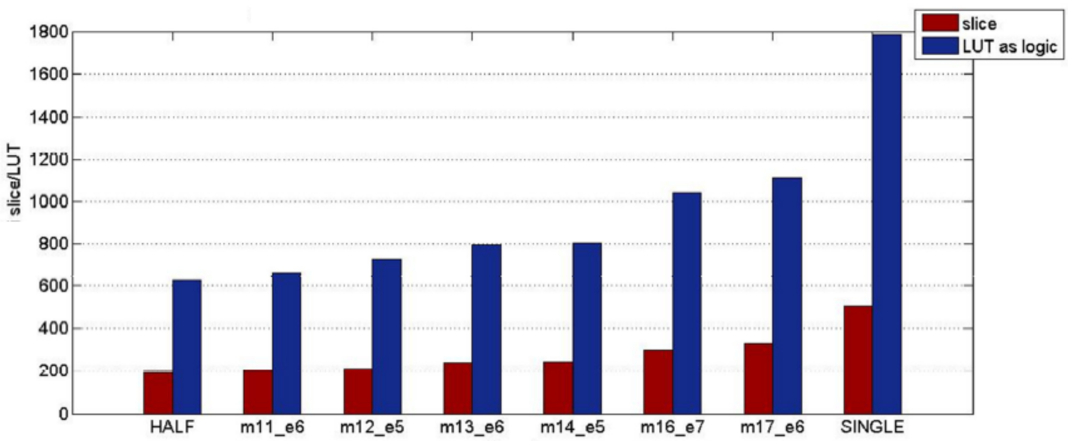


FIGURE 6.5: Resources with DSP disabled @40MHz

It is possible to observe that the occupation of resources increases as the total number of bits required by each precision grows. Starting from half precision, whose requirements are about 38% with respect to single precision, the range of FP precisions under interest result in takes between about 39% and 67% with respect to single precision resource count. Considering instead FP formats with the same number of bit, but differently distributed between mantissa and exponent, there is a slightly lager number of LUTs required for the FP precisions that reserve more bits for the mantissa than to the exponent.

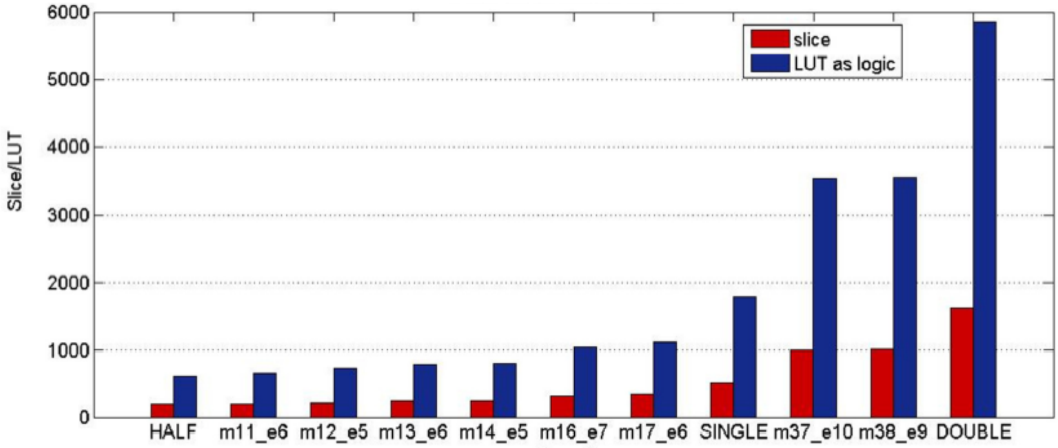


FIGURE 6.6: Resources with DSP disabled @40MHz: from half to double precision

Fig. 6.6 shows the same results extended to double precision.

Propagation delay and speed

Propagation delay results, extrapolated from the *Timing Summary Report* produced after each implementation, are illustrated in Table 6.4.

TABLE 6.4: Propagation delay for reduced precision FP formats

Precision	Propagation delay [ns]
half	20
m11_e6	20
m12_e5	20
m13_e6	20
m14_e5	20
m16_e7	21
m17_e6	21
single	25
m37_e10	25
m38_e9	25
double	29

We can see that, for half precision and for FP precisions between 18 and 20 bits, propagation delay is reported constant at 20 ns, that is 25% better than single precision. Single precision presents a propagation speed of 25ns, which limits the operating frequency under 40 MHz; finally, implementing for double precision introduces an increase in propagation time of about 45% with respect to the best case.

Power consumption

Data illustrated in Table 6.5 are taken from the *Power Report* after the implementation step at 40MHz (25ns).

TABLE 6.5: Power consumption @40MHz

Precision	Total on chip power [W]	Dynamic [W]	Static [W]
half	0.167	0.008	0.158
m11_e6	0.167	0.009	0.158
m12_e5	0.168	0.010	0.158
m13_e6	0.169	0.010	0.158
m14_e5	0.169	0.011	0.158
m16_e7	0.171	0.013	0.158
m17_e6	0.172	0.013	0.158
single	0.177	0.019	0.158

The power consumption has been estimated through the high-level power models of Vivado. The obtained results reveal a very high and constant leakage power consumption for the whole chip, while the dynamic power consumption has a significant dependency on the actual resource count (i.e. the number of LUT/slices required by the FPU). It is possible to see that reduced FP precisions consume up to a half with respect of single precision power consumption. However, this estimate are not completely suitable for our purposes, since leakage power reduction cannot be estimated.

6.5 Conclusion and Future works

This chapter presented the design of a FPU, which can be synthesized with arbitrary precision FP formats. It has been shown that a FPU with reduced precision is a good solution for low-power and low-cost microprocessor systems. The savings in terms of resource occupation, for the analyzed formats, range from about 38% for the m17_e6 format up to 63% for the m11_e6 format with respect to single precision. Moreover, reducing precision has also considerably decreased propagation delay. In particular, the propagation delay on reduced-precision implementations was about 20 ns, with a gain of about 25% and 45% with respect to single and double precision formats.

In future works, synthesizing the FPU on ASIC will allow more accurate estimation of area occupation and speed gain and will also add an estimate and a comparison on power consumption, which did not appear to be reliable using a FPGA as a target.

Chapter 7

Conclusion

Reducing power consumption in digital architectures gained a prominent role in research since almost two decades, especially when the shrinking of physical devices allowed by technology started to rise important design issues regarding the increased power density. The problem has been further amplified by application requirements, demanding increasingly amounts of processing power and memory size (e.g. high definition multimedia, high speed communication, big data applications) and working environment factor (e.g portable and battery operated systems).

Based on a deep dive analysis on the State of the Art, it can be concluded that Approximate Computing and Transprecision computing are promising approaches to achieve power reduction, by relaxing design requirements on computational accuracy and allowing controlled errors to be introduced during processing. These paradigms can be considered as a real low-power digital design approaches, since they break the established tradeoff between performance and power consumption, reducing the second without impacting the first.

The aim of this research has been to explore design and programming techniques for low-power microprocessor architectures based on the Approximate Computing and Transprecision Computing paradigms. After studying the current State of the Art, with particular focus on techniques dealing with approximate memories, the work evolved toward four major directions:

1. the introduction of approximate memory management within the Linux Kernel and the implementation of a custom library for approximate memory data allocation in user space applications;

2. the development of an emulator for the exploration and characterization of microprocessor platforms with approximate memory units;

3. the implementation and analysis of the impact of approximate data allocation on Error Tolerant Applications;

4. the implementation of a fully reconfigurable FPU for low power applications, according to the Transprecision Computing paradigm.

The results obtained in each of those four fields are summarized below.

7.1 Approximate Memory management within the Linux Kernel

The first contribution of this research has been the introduction, at the operating system level, of approximate memory management for 32-bit and 64-bit architectures. An analysis of the current State-of-the-Art revealed that the ability to support approximate memory in the OS is required by many proposed techniques which try to save energy by raising memory fault probability, but the requirements at OS level have never been described and an actual implementation has never been proposed. The solution implemented in this research activity makes the following contributions:

- implementation of a mechanism that allows the Linux Kernel to be aware of exact (normal) and approximate physical memories, managing them as a whole for the common part but distinguishing them in term of allocation requests and page pools management;
- implementation of an interface library that allows user-space applications to straightforwardly request the dynamic allocation of data in approximate memory;
- implementation on 64 bit architectures of a mechanism to allocate approximate data in separate memory zones according to quality requirements. The potential of having quality aware memory zones opens the way to further investigations, tailoring allocations of approximate memory depending on, for example:
  - different sensitivity of output quality to errors in input data structures;
  - variable-time output quality requirements;
  - requirements of different applications in a multitasking environment.

  In this context the following features have been developed:

  - the capability of configuring up to four approximate memory zones (in addition to standard Linux memory zones), where each zone corresponds to physical memory with a certain level of approximation;
  - an internal data allocation scheme, capable of handling separately the allocation requests in quality aware approximate zones;
  - a user space data allocation mechanism and support library. Applications can select the level of approximation of their data structures, trading off more efficiently the approximation level of data (and, hence, energy consumption) output quality.

The extended Linux kernel has been built and extensively tested on different 32/64 bits hardware architectures (x86, ARM, Risc-V) showing the correctness of the implementation and of the fallback allocation policies.

A possible future work will be the design of a hardware platform with approximated DRAM cells (e.g. DRAM banks with reduced refresh rate). Booting the kernel on such platform would expose ZONE_APPROXIMATE memory pages to real hardware faults allowing to experimentally validate the whole technique against power consumption reduction.

The *oz745 dev* platform by Omnitek (Fig. 7.1) has been identified as the board to address this goal. The *oz745* mounts the *xc7z045ffg900-3* Xilinx Zinq 7000 SoC, consisting of an ARM Cortex–A9 dual core processor and an FPGA. The peculiarity of this board is that multiple DRAM memories are available on board: a memory that is interfaced directly with the processor and a memory which is interfaced with the FPGA. The idea is to implement approximate memory on the memory mapped on the FPGA, by configuring the registers of its DRAM controller to reduce the refresh rate under the nominal value.

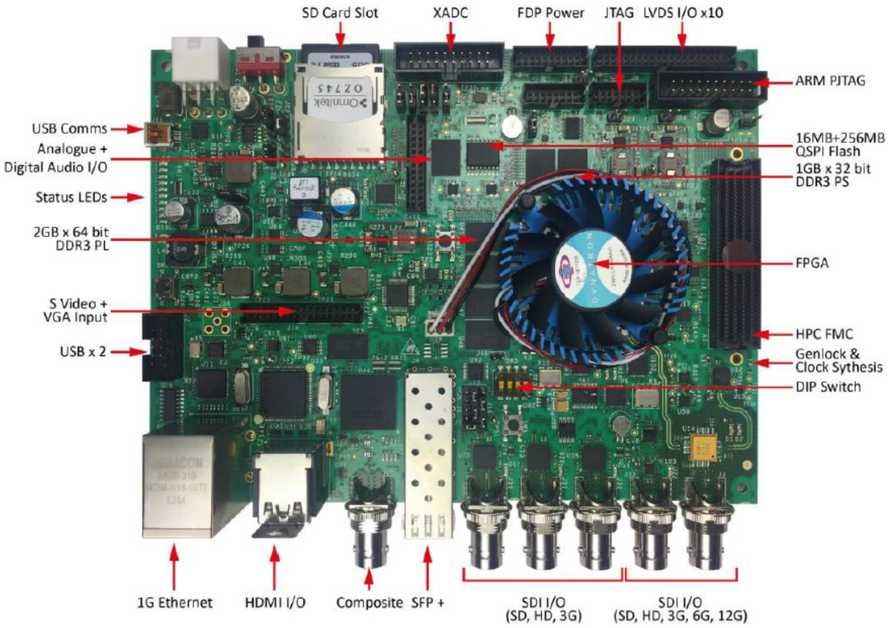


FIGURE 7.1: Omnitek board where approximate DRAM cells could be introduced

7.2 Models and emulator for microprocessor platforms with approximate memory

The second contribution has been the development of an emulator for embedded platforms with approximate memory. Error injection models for approximate memory have been implemented, deriving them from approximate memory circuits proposed in literature for DRAMs and SRAMs. They include the ability of emulating the effects of different approximate designs and implementations, which depend on the internal structure and organization of the cells. The contributions of this work are:

- a complete implementation for modeling approximate SRAM (EOR, EOR_nd, EOW fault models);
- a complete implementation of DRAM models for approximate memories (cell-orientation fault models);
- a complete implementation of bit dropping fault models for both SRAM and DRAM;
- a preliminary implementation of models for ECC protected cells.

The benefits derived by this work are the possibility of exploring the design space regarding approximate memories, showing its effects on output quality and providing a complete characterization of the application, allowing a step toward the determination of the trade-off between saved energy and output quality. The fault models have been introduced in a modular way in the emulator, in order to allow extensions as new techniques and circuits for approximate memories will be proposed.

In particular, as a future work, the introduction of energy consumption models for approximate memories in the emulator will allow to explore and directly determine the energy-quality tradeoff given the combination of an application and an approximate hardware platform. In fact, having a simulation environment which allows the exploration of different solutions in terms of performance and power consumption is one of the basic requirements of embedded system design, since producing many hardware prototypes would require time and a vast economic effort. Moreover, the hardware prototypes would not even have the flexibility of a simulation environment, resulting in limited possibilities for design exploration, in the effort to find the best solution for a particular problem.

7.3 Impact of approximate memory allocation on ETAs

The third contribution has been the study of the impact of approximate memory allocation on ETAs. The introduction of approximate memory support in Linux kernel and the development of a hardware emulator for platforms containing approximate memory allow to write programs using approximate memory and execute them, emulating a system where faults can be injected at run time with predefined statistical properties. In this way, it is possible to evaluate output degradation on real input data and study the relationships between output quality and memory cell error probability/configuration. In fact, knowing the tolerable level of hardware errors is an important step to validate and tune approximate circuits and architectures, eventually quantifying energy savings.

In particular two Error Tolerant classes of applications have been developed and studied, and their effects on output quality have been analyzed:

1. the H.264 video encoder;

2. signal processing applications (digital filters), working on audio signals.

Concerning the first one, the contribution of this work has been to propose a strategy for selecting error tolerant data structures and to study the impact of faults on the quality of the decoded output H.264 video stream. While data fault resilience of H.264 has already been studied considering unwanted and random faults due to unreliable hardware platforms, an analysis considering controlled hardware faults and the corresponding energy quality tradeoff has never been proposed in literature. The strategy for selecting error tolerant data buffers has been found by profiling the application memory usage through the memory profiling suite called *Valgrind*. Then the modified application was run on the *AppropinQuo* emulator, for several combination of approximate memory parameters (e.g. fault rate, fault masking at bit level (*looseness level*)).

The obtained results show the importance of exploring the relation between these parameters and output quality, since the grade of approximation (and hence power savings) is dependent on the amount and the kind of errors that the application can tolerate. For example, leaving some of the MSBs exact in memory cells demonstrated to be an effective way of allowing error probabilities up to 100x higher with the same output quality. Since leaving exact a portion of memory cells reduces global energy saving, this knowledge is also fundamental in order to drive research on hardware techniques specifically tailored to the application, revealing the tradeoff between designing more aggressive approximate circuits and the number of bit cells that must be kept exact.

Future works can consider better allocation strategies in order to increase the fraction of data allocated in approximate memory and more advanced DRAM architectures for embedded systems, as DRAMs chips with integrated ECC units. Another important aspect is a more accurate quantification of power savings, which could be obtained, as said before, by integrating a power consumption model in the approximate DRAM memory model.

Regarding other applications, a signal processing benchmark consisting in a digital FIR filter was implemented and fed with audio samples. It was possible to evaluate the impact on SNR when input and output buffers, along with tap registers, are allocated in approximate memory. A variant of this benchmark has been developed for analyzing the impact on output (SNR) when quality aware approximate memory zones are present. In this case, the application allows to explore the allocation of different data structures depending on sensitivity to errors and desired output quality. The obtained results reveal that, even with just two levels of approximation quality and manual allocation strategy, output SNR can be risen by moving a small number of more sensitive data to a portion of memory with lower level of approximation, while leaving the large and less sensitive buffers on memory with higher level of approximation. As future work in this direction, more complex applications could be targeted, where partitioning of data in multiple quality zones is not directly evident. Automatic allocation strategies will also be considered as a way of reaching more significant savings.

7.4 Transprecision FPU implementation

The last contribution has been the design and implementation of a fully reconfigurable FPU, which can be synthesized to work with reduced precision formats tailored to the application, as required by the Transprecision Computing paradigm. The obtained results show that this approach is a promising solution that can be adopted in all low-power and/or low-cost microprocessor systems requiring significant area and energy saving.

The analysis reveals that a FPU working with non-standard reduced precision formats (e.g. formats with lengths between 18 bit and 24 bit) allows to sensibly lower area and gates requirements, while still producing sensible advantages (in terms of precision and overflow robustness) with respect to fixed-point arithmetic.

The savings in terms of resources, for the analyzed FP formats, range from about 38% for *m17_e6* format and reach about 63% for *m11_e6* format with respect to IEEE-754 single precision. Reducing precision considerably decreased propagation delay. In particular, the propagation delay on reduced-precision implementations was about 20 ns, with a gain of about 25% and 45% with respect to single and double precision formats on the same technology.

Since the synthesis target was an FPGA, estimates concerning power consumption were found to be less significant (leakage power consumption could not be predicted). In fact, the power consumption, as reported by *Vivado*, was characterized by a large leakage value due to the whole chip, while differences in dynamic power consumption were evident.

Future works will add the following contributions:

- synthesis of the FPU on ASIC. It will allow more accurate estimate of area occupation and speed gain and will also add an estimate and a comparison on power consumption, which revealed to be not reliable using an FPGA as target;
- development of a wrapper for the block *Floatig_Point_Unit_core* which exposes an interface based on standard IEEE-754 formats, in order to transparently insert the reduced precision FPU is in an existing ISA;

- implementation of multi-cycle FP operations, as for example the division.

Appendix A

Linux kernel files for approximate memory support

A.1 Patched Kernel files

#gfp flag for zone approximate
include/linux/gfp.h

#Approximate Zone creation
include/mm/mmzone.h

#Approximate zone fallback + Alloc fair policy
mm/page_alloc.c

#Enable zone approximate messages
mm/vmstat.c
include/linux/vm_event_item.h

#Approximate memory system call
include/uapi/asm-generic/unistd_32.h
generated/uapi/asm/unistd_32.h
arch/x86/entry/syscall/syscall_32.tbl
include/linux/syscall.h
kernel/sys.c
kernel/sys_ni.c

#Prototype and call to function for bootmem allocator (Exclude zone approximate from bootmem allocator)
include/linux/memblock.h
mm/memblock.c

#i386 approximate memory support
arch/x86/Kconfig
arch/x86/kernel/setup.c
arch/x86/mm/init.c
mm/nobootmem.c
include/linux/bootmem.h

#ARM approximate memory support
arch/arm/Kconfig
#Added device tree with approximate memory (makefile)

arch/arm/boot/dts/Makefile
#Setup memory zones (Approximate zone) and patch for bootmem allocator (Exclude zone approximate from bootmem allocator)
arch/arm/mm/init.c
#Interrupt vectors
arch/arm/mm/mmu.c
#Device tree (Register approximate memory)
arch/arm/boot/dts/vexpress-v2-ca9

#RISCV approximate memory support
arch/arm/riscv

#Setup memory zones (Approximate zone) and patch for bootmem allocator (Exclude zone approximate from bootmem allocator)
arch/riscv/mm/init.c

A.2 New Kernel source files

#*Approx mem device driver*
drivers/char/approxmem.c
drivers/char/approxmem_multizone.c
drivers/char/Makefile
drivers/char/Kconfig

A.3 Approximate Memory Configuration (Make menuconfig)

1. Enable approximate memory;
2. *Kernel features–> High Memory support–> off*;
3. *Device driver–>character devices–>Device approximate memory (Load approximate memory module.Depends on* ZONE_APPROXIMATE, *if* ZONE_APPROXIMATE *is enabled, approximate memory device = M)*

Appendix B

AppropinQuo: list of approximate memory models

B.0.1 QEmu 2.5.1 patched files for approximate memory support

- *Parser of approximate memory configuration*
- qemu-options.hx
- vl.c
- *approximate memory in x86 architectures*
- hw/i386/pc.c
- *approximate memory in ARM architectures*
- hw/arm/boot.c
- hw/arm/arm-vexpress.c
- *approximate memory in RISCV architectures*
- hw/riscv/virtio.c
- hw/riscv/sifive.c
- *approximate memory log utilities*
- qemu/util/log.c
- qemu/include/log.h
- Makefile.obj

B.0.2 New QEmu 2.5.1 source files

- *approximate memory model*
- approxmem.c
- approxmem.h
- approxmem_multiple.c
- approxmem_multiple.h
- *approximate memory scripts and configuration file*

- qemu_run.sh
- qemu_run_arm.sh
- qemu_run_riscv.sh
- qemu_run_riscv64.sh
- approx_config.cfg

Appendix C

Transprecision FPU: list of vhd files

- FPU_CORE_comb.vhd
- shift_dx_core_comb.vhd
- moltiplicatore_core_comb.vhd
- normalizzatore_core_comb.vhd
- sommatore_core_comb.vhd

Appendix D

Publications and Presentations

- Stazi, G., Menichelli, F., Mastrandrea, A., Olivieri, M. (2017, June). *"Introducing approximate memory support in Linux Kernel"* In Ph. D. Research in Microelectronics and Electronics (PRIME), 2017 13th Conference on (pp. 97-100). IEEE.
- Menichelli, F., Stazi, G., Mastrandrea, A., Olivieri, M. (2016, September). *"An Emulator for Approximate Memory Platforms Based on QEmu."* In International Conference on Applications in Electronics Pervading Industry, Environment and Society (pp. 153-159). Springer, Cham.
- Stazi, G., Menichelli, Adani, L. , F., Mastrandrea, A., Olivieri, M. *"Impact of Approximate Memory Data Allocation on a H.264 Software Video Encoder"* (Frankfurt 2018, June, In ATCET2018: Approximate and Transprecision Computing on Emerging Technologies, ISC HIGH PERFORMANCE).
- Stazi, G., Menichelli, F., Mastrandrea, A., Olivieri, M. *"Approximate Memory support for Linux Early Allocators in ARM architectures"* (Pisa 2018, September, In International Conference on Applications in Electronics Pervading Industry, Environment and Society).
- Stazi, G.,Silvestri, F. , Menichelli, F., Mastrandrea, A., Olivieri, M. *"Synthesis Time Recongurable Floating Point Unit for Transprecision Computing"*(Pisa 2018,September, In International Conference on Applications in Electronics Pervading Industry, Environment and Society).
- Stazi, Giulia, et al. *'AppropinQuo: A Platform Emulator for Exploring the Approximate Memory Design Space.'* 2018 New Generation of CAS (NGCAS). IEEE, 2018.
- Stazi, Giulia, et al. *'Full System Emulation of Approximate Memory Platforms with AppropinQuo.'* Journal of Low Power Electronics 15.1 (2019): 30-39.
- Stazi, Giulia, et al. *'Quality Aware Approximate Memory in RISC-V Linux Kernel.'* 2019 15th Conference on Ph. D Research in Microelectronics and Electronics (PRIME). IEEE, 2019.
- Stazi, G., Menichelli, F., Mastrandrea, A., Olivieri, M. *"Quality aware selective ECC for approximate DRAM"*(Pisa 2019,September, In International Conference on Applications in Electronics Pervading Industry, Environment and Society).
- Stazi, G., Menichelli, F., Olivieri, M. *"The first porting of Linux OS support for Approximate Memory management on RISC-V"*(The RISC-V Workshop, ETH Zurich, 11-13 June 2019).

Bibliography

Amdahl, Gene M (1967). "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, pp. 483–485.

Asma, Ben Hamida, Nedra Jarray, and Zitouni Abdelkrim (2017). "Low-Power Hardware Design of Binary Arithmetic Encoder in H. 264". In: *International Journal of Advanced Computer Science and Applications* 8.7, pp. 412–416.

Bailey, David H, Hida Yozo, Xiaoye S Li, and Brandon Thompson (2002). *ARPREC: An arbitrary precision computation package*. Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

Bellard, Fabrice (2005). "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46.

Bellotti, F., R. Berta, A. De Gloria, and L. Primavera (June 2009). "Enhancing the Educational Value of Video Games". In: *Comput. Entertain.* 7.2, 23:1–23:18. ISSN: 1544-3574. URL: http://doi.acm.org/10.1145/1541895.1541903.

Benini, Luca, Francesco Menichelli, and Mauro Olivieri (2004). "A class of code compression schemes for reducing power consumption in embedded microprocessor systems". In: *IEEE Transactions on Computers* 53.4, pp. 467–482.

Binkert, Nathan, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. (2011). "The gem5 simulator". In: *ACM SIGARCH Computer Architecture News* 39.2, pp. 1–7.

Bovet, Cesati (2005). *Understanding the Linux Kernel 3 edition*. O'Reilly Media. ISBN: 0596005652.

Breuer, M (2004). "Error-tolerance and related test issues". In: *Asian Test Symp.*

Burger, Doug and Todd M Austin (1997). "The SimpleScalar tool set, version 2.0". In: *ACM SIGARCH computer architecture news* 25.3, pp. 13–25.

Burns, Ellie (2016). *An Introduction To Reducing Dynamic Power*. URL: https://semiengineering.com/an-introduction-to-reducing-dynamic-power/.

Chandrasekar, Karthik, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens (2012). "DRAMPower: Open-source DRAM power & energy estimation tool". In: 22. URL: URL: http://www.drampower.info.

Chang, Ik Joon, Debabrata Mohapatra, and Kaushik Roy (2011). "A priority-based 6T/8T hybrid SRAM architecture for aggressive voltage scaling in video applications". In: *IEEE transactions on circuits and systems for video technology* 21.2, pp. 101–112.

Chen, Linbin, Jie Han, Weiqiang Liu, and Fabrizio Lombardi (2016). "On the design of approximate restoring dividers for error-tolerant applications". In: *IEEE Transactions on Computers* 65.8, pp. 2522–2533.

Chen, Yuanchang, Yizhe Zhu, Fei Qiao, Jie Han, Yuansheng Liu, and Huazhong Yang (2017). "Evaluating data resilience in cnns from an approximate memory perspective". In: *Proceedings of the on Great Lakes Symposium on VLSI 2017*. ACM, pp. 89–94.

Chippa, Vinay K, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan (2013). "Analysis and characterization of inherent application resilience for approximate computing". In: *Proceedings of the 50th Annual Design Automation Conference*. ACM, p. 113.

Cho, Minki, Jason Schlessman, Wayne Wolf, and Saibal Mukhopadhyay (2011). "Reconfigurable SRAM architecture with spatial voltage scaling for low power mobile multimedia applications". In: *IEEE transactions on very large scale integration (VLSI) systems* 19.1, pp. 161–165.

De Cock, Jan, Aditya Mavlankar, Anush Moorthy, and Anne Aaron (2016). "A large-scale video codec comparison of x264, x265 and libvpx for practical VOD applications". In: *Applications of Digital Image Processing XXXIX*. Vol. 9971. International Society for Optics and Photonics, p. 997116.

Esmaeilzadeh, Hadi, Adrian Sampson, Luis Ceze, and Doug Burger (2012). "Architecture support for disciplined approximate programming". In: *ACM SIGPLAN Notices*. Vol. 47. 4. ACM, pp. 301–312.

Esmaeilzadeh, Hadi, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger (Feb. 2013). "Power Challenges May End the Multicore Era". In: *Commun. ACM* 56.2, pp. 93–102. ISSN: 0001-0782. URL: http://doi.acm.org/10.1145/2408776.2408797.

Fousse, Laurent, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann (2007). "MPFR: A multiple-precision binary floating-point library with correct rounding". In: *ACM Transactions on Mathematical Software (TOMS)* 33.2, p. 13.

Frustaci, Fabio, Mahmood Khayatzadeh, David Blaauw, Dennis Sylvester, and Massimo Alioto (2014). "13.8 A 32kb SRAM for error-free and error-tolerant applications with dynamic energy-quality management in 28nm CMOS". In: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*. IEEE, pp. 244–245.

Frustaci, Fabio, David Blaauw, Dennis Sylvester, and Massimo Alioto (2015a). "Better-than-voltage scaling energy reduction in approximate SRAMs via bit dropping and bit reuse". In: *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2015 25th International Workshop on*. IEEE, pp. 132–139.

Frustaci, Fabio, Mahmood Khayatzadeh, David Blaauw, Dennis Sylvester, and Massimo Alioto (2015b). "SRAM for error-tolerant applications with dynamic energy-quality management in 28 nm CMOS". In: *IEEE Journal of Solid-State Circuits* 50.5, pp. 1310–1323.

Frustaci, Fabio, David Blaauw, Dennis Sylvester, and Massimo Alioto (2016). "Approximate SRAMs With Dynamic Energy-Quality Management". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.6, pp. 2128–2141.

Gong, Na, Shixiong Jiang, Anoosha Challapalli, Sherwin Fernandes, and Ramalingam Sridhar (2012). "Ultra-low voltage split-data-aware embedded SRAM for mobile video applications". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 59.12, pp. 883–887.

Gorman, Mel (2004). *Understanding the Linux Virtual Memory Manager*. Prentice Hall. ISBN: 0131453483.

Gupta, Sumitha and Sukanya Padave (2016). "Power Optimization for Low Power VLSI Circuits". In: *International Journal of Advanced Research in Computer Science and Software Engineering* 6.3.

Gupta, Vaibhav, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy (2011). "IMPACT: imprecise adders for low-power approximate computing". In: *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*. IEEE Press, pp. 409–414.

Gupta, Vaibhav, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy (2013). "Low-power digital signal processing using approximate adders". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.1, pp. 124–137.

Han, Jie and Michael Orshansky (2013). "Approximate computing: An emerging paradigm for energy-efficient design". In: *2013 18th IEEE European Test Symposium (ETS)*. IEEE, pp. 1–6.

Hardavellas, N., M. Ferdman, B. Falsafi, and A. Ailamaki (2011). "Toward Dark Silicon in Servers". In: *IEEE Micro* 31.4, pp. 6–15.

Hauser, John R (1996). "Handling floating-point exceptions in numeric programs". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18.2, pp. 139–174.

Hegde, Rajamohana and Naresh R Shanbhag (1999). "Energy-efficient signal processing via algorithmic noise-tolerance". In: *Proceedings. 1999 International Symposium on Low Power Electronics and Design (Cat. No. 99TH8477)*. IEEE, pp. 30–35.

Hennessy, John L. (2018). *Future of Computing*. URL: https://web.stanford.edu/~hennessy/.

Holık, Lukáš, Ondrej Lengál, Adam Rogalewicz, Lukáš Sekanina, Zdenek Vašıcek, and Tomáš Vojnar (2016). "Towards formal relaxed equivalence checking in approximate computing methodology". In: *2nd Workshop on Approximate Computing (WAPCO'16)*, p. 48.

Höller, Andrea, Armin Krieg, Tobias Rauter, Johannes Iber, and Christian Kreiner (2015). "QEMU-based fault injection for a system-level analysis of software countermeasures against fault attacks". In: *Digital System Design (DSD), 2015 Euromicro Conference on*. IEEE, pp. 530–533.

Huang, Jiawei, John Lach, and Gabriel Robins (2012). "A methodology for energy-quality tradeoff using imprecise hardware". In: *Proceedings of the 49th Annual Design Automation Conference*. ACM, pp. 504–509.

Itoh, Kiyoo and Masashi Horiguchi (2009). "Low-voltage scaling limitations for nano-scale CMOS LSIs". In: *Solid-State Electronics* 53.4, pp. 402–410.

Jiang, Honglan, Cong Liu, Naman Maheshwari, Fabrizio Lombardi, and Jie Han (2016). "A comparative evaluation of approximate multipliers". In: *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*. IEEE, pp. 191–196.

Jung, Matthias, Christian Weis, and Norbert Wehn (2015). "DRAMSys: a flexible DRAM subsystem design space exploration framework". In: *IPSJ Transactions on System LSI Design Methodology* 8, pp. 63–74.

Jung, Matthias, Éder Zulian, Deepak M Mathew, Matthias Herrmann, Christian Brugger, Christian Weis, and Norbert Wehn (2015). "Omitting refresh: A case study for commodity and wide i/o drams". In: *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, pp. 85–91.

Jung, Matthias, Deepak M Mathew, Christian Weis, and Norbert Wehn (2016). "Efficient reliability management in SoCs-an approximate DRAM perspective". In: *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*. IEEE, pp. 390–394.

Kahan, William (1996). "IEEE standard 754 for binary floating-point arithmetic". In: *Lecture Notes on the Status of IEEE* 754.94720-1776, p. 11.

Kernel, The Linux. *Energy Aware Scheduling*. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-energy.html.

Kwon, Jinmo, Ik Joon Chang, Insoo Lee, Heemin Park, and Jongsun Park (2012). "Heterogeneous SRAM cell sizing for low-power H. 264 applications". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 59.10, pp. 2275–2284.

Lee, Insoo, Jinmo Kwon, Jangwon Park, and Jongsun Park (2013). "Priority based error correction code (ECC) for the embedded SRAM memories in H. 264 system". In: *Journal of Signal Processing Systems* 73.2, pp. 123–136.

Liu, Hao-Ran (2010). *Physical Memory Management in Linux*. URL: https://hzliu123.github.io/linux-kernel/Physical%20Memory%20Management%20in%20Linux.pdf.

Liu, Jamie, Ben Jaiyen, Richard Veras, and Onur Mutlu (2012a). "RAIDR: Retention-aware intelligent DRAM refresh". In: *ACM SIGARCH Computer Architecture News*. Vol. 40. 3. IEEE Computer Society, pp. 1–12.

Liu, Jamie, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu (2013). "An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms". In: *ACM SIGARCH Computer Architecture News*. Vol. 41. 3. ACM, pp. 60–71.

Liu, Song, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G Zorn (2012b). "Flikker: saving DRAM refresh-power through critical data partitioning". In: *ACM SIGPLAN Notices* 47.4, pp. 213–224.

Lucas, Jan, Mauricio Alvarez-Mesa, Michael Andersch, and Ben Juurlink (2014). "Sparkk: Quality-scalable approximate storage in DRAM". In: *The memory forum*, pp. 1–9.

Malossi, A Cristiano I, Michael Schaffner, Anca Molnos, Luca Gammaitoni, Giuseppe Tagliavini, Andrew Emerson, Andrés Tomás, Dimitrios S Nikolopoulos, Eric Flamand, and Norbert Wehn (2018). "The transprecision computing paradigm: Concept, design, and applications". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 1105–1110.

Masadeh, Mahmoud, Osman Hasan, and Sofiene Tahar (2018). "Comparative study of approximate multipliers". In: *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. ACM, pp. 415–418.

Mastrandrea, Antonio, Francesco Menichelli, and Mauro Olivieri (2011). "A delay model allowing nano-CMOS standard cells statistical simulation at the logic level". In: *Ph. D. Research in Microelectronics and Electronics (PRIME), 2011 7th Conference on*. IEEE, pp. 217–220.

Mathew, Deepak M, Martin Schultheis, Carl C Rheinländer, Chirag Sudarshan, Christian Weis, Norbert Wehn, and Matthias Jung (2018). "An Analysis on Retention Error Behavior and Power Consumption of Recent DDR4 DRAMs". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE.

Merritt, Loren and Rahul Vanam (2006). *x264: A high performance H. 264/AVC encoder*. URL: http://neuron2.net/library/avc/overview_x264_v8_5.pdf.

Montgomery, C et al. (1994). *Xiph. org Video Test Media (derf's collection)*. URL: https://media.xiph.org/video/derf.

Natarajan, Vithyalakshmi, Ashok Kumar Nagarajan, Nagarajan Pandian, and Vinoth Gopi Savithri (2018). "Low Power Design Methodology". In: *Very-Large-Scale Integration*, p. 47.

Nepal, Kumud, Yueting Li, R Iris Bahar, and Sherief Reda (2014). "ABACUS: A technique for automated behavioral synthesis of approximate computing circuits". In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 1–6.

Nethercote, Nicholas and Julian Seward (2007). "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices*. Vol. 42. 6. ACM, pp. 89–100.

Nguyen, Duy Thanh, Hyun Kim, Hyuk-Jae Lee, and Ik-Joon Chang (2018). "An Approximate Memory Architecture for a Reduction of Refresh Power Consumption in Deep Learning Applications". In: *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*. IEEE, pp. 1–5.

Olivieri, M., R. Mancuso, and F. Riedel (May 2007). "A Reconfigurable, Low Power, Temperature Compensated IC for 8-segment Gamma Correction Curve in TFT, OLED and PDP Displays". In: *IEEE Trans. on Consum. Electron.* 53.2, pp. 720–724. ISSN: 0098-3063. URL: http://dx.doi.org/10.1109/TCE.2007.381751.

Organization, VideoLan. *VideoLan*. URL: https://www.videolan.org/index.it.html.

Pagliari, Daniele Jahier, Andrea Calimera, Enrico Macii, and Massimo Poncino (2015). "An automated design flow for approximate circuits based on reduced precision redundancy". In: *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE, pp. 86–93.

Palem, Krishna V (2003). "Energy aware algorithm design via probabilistic computing: from algorithms and models to Moore's law and novel (semiconductor) devices". In: *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, pp. 113–116.

Parasyris, Konstantinos, Georgios Tziantzoulis, Christos D Antonopoulos, and Nikolaos Bellas (2014). "GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates". In: *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, pp. 622–629.

Pennisi, Agatino. *Low Power Principles*. URL: https://www.unirc.it/documentazione/materiale_didattico/599_2007_60_1102.pdf.

Pilo, Harold, Chad A Adams, Igor Arsovski, Robert M Houle, Steven M Lamphier, Michael M Lee, Frank M Pavlik, Sushma N Sambatur, Adnan Seferagic, Richard Wu, et al. (2013). "A 64Mb SRAM in 22nm SOI technology featuring fine-granularity power gating and low-energy power-supply-partition techniques for 37% leakage reduction". In: *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*. IEEE, pp. 322–323.

Platt, Susan (2018). *Metamorphosis of an industry, part two moores law*. URL: https://www.micron.com/about/blog/2018/october/metamorphosis-of-an-industry-part-two-moores-law.

projektovanje. *Low Power Design in VLSI*. URL: http://leda.elfak.ni.ac.rs/education/projektovanjeVLSI/predavanja/10%20Low%20Power%20Design%20in%20VLSI.pdf.

R., Kavya (2016). *Optimization Techniques for Low Power VLSI Design*. URL: https://pdfs.semanticscholar.org/Optimization-Techniques-for-Low-Power-VLSI-Design.

Raha, Arnab, Hrishikesh Jayakumar, Soubhagya Sutar, and Vijay Raghunathan (2015). "Quality-aware data allocation in approximate DRAM". In: *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE Press, pp. 89–98.

Raha, Arnab, Soubhagya Sutar, Hrishikesh Jayakumar, and Vijay Raghunathan (2017). "Quality configurable approximate DRAM". In: *IEEE Transactions on Computers* 66.7, pp. 1172–1187.

Rehman, Semeen, Muhammad Shafique, Florian Kriebel, and Jörg Henkel (2011). "ReVC: Computationally reliable video coding on unreliable hardware platforms:

A case study on error-tolerant H. 264/AVC CAVLC entropy coding". In: *Image Processing (ICIP), 2011 18th IEEE International Conference on*. IEEE, pp. 397–400.

Rethinagiri, Santhosh Kumar, Oscar Palomar, Rabie Ben Atitallah, Smail Niar, Osman Unsal, and Adrian Cristal Kestelman (2014). "System-level power estimation tool for embedded processor based platforms". In: *Proceedings of the 6th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM, p. 5.

Rossi, Davide, Francesco Conti, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, Giuseppe Tagliavini, Alessandro Capotondi, Philippe Flatresse, and Luca Benini (2015). "PULP: A parallel ultra low power platform for next generation IoT applications". In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, pp. 1–39.

Roy, Kaushik and Sharat C Prasad (2009). *Low-power CMOS VLSI circuit design*. John Wiley & Sons.

Roy, Pooja, Rajarshi Ray, Chundong Wang, and Weng Fai Wong (2014). "Asac: Automatic sensitivity analysis for approximate computing". In: *Acm Sigplan Notices*. Vol. 49. 5. ACM, pp. 95–104.

Sampson, Adrian, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman (2011). "EnerJ: Approximate data types for safe and general low-power computation". In: *ACM SIGPLAN Notices*. Vol. 46. ACM, pp. 164–174.

Shafique, Muhammad, Rehan Hafiz, Muhammad Usama Javed, Sarmad Abbas, Lukas Sekanina, Zdenek Vasicek, and Vojtech Mrazek (2017a). "Adaptive and energy-efficient architectures for machine learning: Challenges, opportunities, and research roadmap". In: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, pp. 627–632.

Shafique, Muhammad, Semeen Rehman, Florian Kriebel, Muhammad Usman Karim Khan, Bruno Zatt, Arun Subramaniyan, Bruno Boessio Vizzotto, and Jörg Henkel (2017b). "Application-Guided Power-Efficient Fault Tolerance for H. 264 Context Adaptive Variable Length Coding". In: *IEEE Transactions on Computers* 66.4, pp. 560–574.

Shampine, Lawrence F and Mark W Reichelt (1997). "The matlab ode suite". In: *SIAM journal on scientific computing* 18.1, pp. 1–22.

Shim, Byonghyo and NR Shambhag (2003). "Performance analysis of algorithmic noise-tolerance techniques". In: *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS'03.* Vol. 4. IEEE, pp. IV–IV.

Shim, Byonghyo, Srinivasa R Sridhara, and Naresh R Shanbhag (2004). "Reliable low-power digital signal processing via reduced precision redundancy". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.5, pp. 497–510.

Shoushtari, Majid, Abbas BanaiyanMofrad, and Nikil Dutt (2015). "Exploiting partially-forgetful memories for approximate computing". In: *IEEE Embedded Systems Letters* 7.1, pp. 19–22.

Stazi, Giulia, Francesco Menichelli, Antonio Mastrandrea, and Mauro Olivieri (2017). "Introducing approximate memory support in Linux Kernel". In: *Ph. D. Research in Microelectronics and Electronics (PRIME), 2017 13th Conference on*. IEEE, pp. 97–100.

Stazi, Giulia, Antonio Mastrandrea, Mauro Olivieri, and Francesco Menichelli (2019). "Full System Emulation of Approximate Memory Platforms with AppropinQuo". In: *Journal of Low Power Electronics* 15.1, pp. 30–39.

Tagliavini, Giuseppe, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benin (2018). “A transprecision floating-point platform for ultra-low power computing”. In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 1051–1056.

Teman, Adam, Georgios Karakonstantis, Robert Giterman, Pascal Meinerzhagen, and Andreas Burg (2015). “Energy versus data integrity trade-offs in embedded high-density logic compatible dynamic memories”. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, pp. 489–494.

Tian, Ye, Qian Zhang, Ting Wang, Feng Yuan, and Qiang Xu (2015). “Approxma: Approximate memory access for dynamic precision scaling”. In: *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*. ACM, pp. 337–342.

Varadharajan, Senthil Kumaran and Viswanathan Nallasamy (2017). “Low power VLSI circuits design strategies and methodologies: A literature review”. In: *2017 Conference on Emerging Devices and Smart Systems (ICEDSS)*. IEEE, pp. 245–251.

Vasicek, Zdenek and Lukas Sekanina (2016). “Evolutionary design of complex approximate combinational circuits”. In: *Genetic Programming and Evolvable Machines* 17.2, pp. 169–192.

Venkataramani, Swagath, Kaushik Roy, and Anand Raghunathan (2013). “Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits”. In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 1367–1372.

Venkataramani, Swagath, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan (2012). “SALSA: systematic logic synthesis of approximate circuits”. In: *Proceedings of the 49th Annual Design Automation Conference*. ACM, pp. 796–801.

Weis, Christian, Matthias Jung, Éder F Zulian, Chirag Sudarshan, Deepak M Mathew, and Norbert Wehn (2018). “The Role of Memories in Transprecision Computing”. In: *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*. IEEE, pp. 1–5.

Widmer, Marco, Andrea Bonetti, and Andreas Burg (2019). “FPGA-Based Emulation of Embedded DRAMs for Statistical Error Resilience Evaluation of Approximate Computing Systems”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, p. 36.

Winkler, Stefan and Praveen Mohandas (2008). “The evolution of video quality measurement: From PSNR to hybrid metrics”. In: *IEEE Transactions on Broadcasting* 54.3, pp. 660–668.

Yang, Xinghua, Nanyang Huang, Yuanchang Chen, Fei Qiao, and Huazhong Yang (2016). “A priority-based selective bit dropping strategy to reduce DRAM and SRAM power in image processing”. In: *IEICE Electronics Express* 13.23, pp. 20160990–20160990.

Yang, Zhixi, Ajaypat Jain, Jinghang Liang, Jie Han, and Fabrizio Lombardi (2013). “Approximate XOR/XNOR-based adders for inexact computing”. In: *2013 13th IEEE International Conference on Nanotechnology (IEEE-NANO 2013)*. IEEE, pp. 690–693.