technische universität
dortmund

fakultät für
informatik

Software Exploitation of Traditional Interfaces for Modern Technologies

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Christian Hakert

Dortmund

2024

Tag der mündlichen Prüfung:	18. Juli 2024
Dekan / Dekanin:	Prof. Dr.-Ing. Gernot A. Fink
Gutachter / Gutachterinnen:	Prof. Dr.-Ing. Jian-Jia Chen
	Prof. Dr.-Ing. Jeronimo Castrillon

Acknowledgments

In the first place, I would like to express my deep gratitude to my advisor (Doktorvater) Prof. Jian-Jia Chen. Besides his great supervision during my studies, he was always open for new and unconventional ideas. He always supported me in following my ideas, motivated me to continue specific topics and supported bringing results to a publishable status. Beyond this, I have to express my appreciation for his support in involving me in the writing of new project proposals, which lead to two successfully accepted research projects, which stem from topical areas of this dissertation. Furthermore, I would like to thank my second examiner, Prof. Jeronimo Castrillon, to commit to review my dissertation.

Next, I express my deep appreciation to my colleagues, who collaborated with me for my studies and co-authored my published papers. Most important, I would like to thank my colleague Kuan-Hsun Chen for accompanying me during my entire study and working on many topics with me. I further would like to thank my other colleagues, who collaborated with me throughout my different studies. From the DAES group of the LS12 at TU Dortmund University, these are Nils Hölscher, Mario Günzel, Georg von der Brüggen and Daniel Biebert. From the CES group from KIT, these are Lars Bauer, Paul Genssler and Prof. Jörg Henkel. Beyond my colleagues, my studies have been supported by different research projects. These include the DFG project OneMemory(405422836), the DFG special priority program 2377(460954224) and the DFG project MemoryDiplomat(502384507). I would like to express special appreciation to the collaborators of the OneMemory project in Taiwan. Furthermore, I have to express my deep gratitude to my non-scientific colleagues and to the people who collaborated with me in teaching and administration. Without their help and support, I would not have been able to conduct my studies in the way I did. Special appreciation goes to the secretary of the LS12, namely Claudia Graute.

Finally, but nevertheless important, I would like to express my deepest gratitude to my family and friends for their unbreakable support. Especially, I would like to thank my parents for making my academic career so far possible at all and never stopped in supporting me on my way. I further want to express my appreciation to all of my friends for supporting me all along.

Abstract

Modern computer Technologies are skyrocketing to spheres, which frequently seemed unimaginable years ago. Quantum effect petabyte-sized storage devices or deep cache hierarchies, acting within nanoseconds, make only a few examples. At the same time, interfaces to communicate with such technologies are settled and remain largely unaffected by the technology development. While loading and storing a word to a given memory address was the standard interface to communicate with memory devices in very early stages of computer systems, it still features a similar shape nowadays. Unsurprisingly, modern computing technologies come with increasing demand of management, such as lifetime management for NON-VOLATILE MEMORY (NVM) or prefetching and eviction strategies for cache hierarchies. Leaving this management to the hardware solely provides a limited design space and space for optimization. Consequently, software has to find ways, which allow an either direct or indirect management of the technologies over the traditional interfaces.

This dissertation picks up this need and studies selected modern technologies and their need for management. Methods are presented in this thesis, which systematically exploit existing traditional interfaces in order to provide extended functionalities for the management of modern technologies. The exploitations in this thesis are solely conducted on a software level and do not require any actions in the available hardware. In a first part, memory technologies are picked up as a target technology. In greater detail, NON-VOLATILE MEMORY (NVM) is studied. This thesis discusses the lifetime issue of these technologies and the resulting need for wear-leveling. Various approaches are introduced, which allow different forms of wear-leveling on different levels of the software. This ranges from wear-leveling procedures inside the operating system and the system software towards direct application integration to extend the memory lifetime. Apart from the lifetime issue, the latency and energy property of a specific type of emerging memory, namely RACETRACK MEMORY (RTM), is considered. Dedicated to the application of RANDOM FOREST (RF) models, the access properties are optimized in the application level directly.

In the last part of this thesis, the focus is moved from memories to arithemtic compuation. RANDOM FOREST (RF) models are kept as a target application and their execution on modern computation technologies is considered. The usage of floating-point numbers is put to a major focus and the memory behavior of floating-point numbers is optimized. By proposing alternative computation schemes for floating-point numbers, which are entirely realized in software and leave the hardware untouched, significant performance improvement is gained.

Publications

The topical contributions of this thesis are published for major parts in the form of peer-reviewed articles in journals and conferences. The following publications form the topical contributions of this thesis:

[HCC22a] C. Hakert, K.-H. Chen, and J.-J. Chen. “FLInt: Exploiting Floating Point Enabled Integer Arithmetic for Efficient Random Forest Inference”. In: *arXiv preprint arXiv:2209.04181* (2022) (Cited on pages 16, 148).

[HCC22b] C. Hakert, K.-H. Chen, and J.-J. Chen. “Immediate Split Trees: Immediate Encoding of Floating Point Split Values in RandomForests”. In: *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases* (2022) (Cited on pages 16, 136).

[HCC24] C. Hakert, K.-H. Chen, and J.-J. Chen. “FLInt: Exploiting Floating Point Enabled Integer Arithmetic for Efficient Random Forest Inference”. In: *Design, Automation and Test in Europe Conference* (2024) (Cited on page 148).

[HCK+20] C. Hakert, K.-H. Chen, S. Kuenzer, S. Santhanam, S.-H. Chen, Y.-H. Chang, F. Huici, and J.-J. Chen. “Splitn Trace NVM: Leveraging Library OSes for Semantic Memory Tracing”. In: *9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 2020 (Cited on pages 16, 41, 71).

[HCS+21] C. Hakert, K.-H. Chen, H. Schirmeier, L. Bauer, P. R. Genssler, G. von der Brüggen, H. Amrouch, J. Henkel, and J.-J. Chen. “Software-Managed Read and Write Wear-Leveling for Non-Volatile Main Memory”. In: *ACM Transactions on Embedded Computing Systems Special Issue on Memory and Storage Systems for Embedded and IoT Applications*. 2021 (Cited on pages 16, 58).

[HKC+21a] C. Hakert, A. A. Khan, K.-H. Chen, F. Hameed, J. Castrillon, and J.-J. Chen. “BLOwing Trees to the Ground: Layout Optimization of Decision Trees on Racetrack Memory”. In: *58th ACM/IEEE Design Automation Conference (DAC), accepted*. 2021 (Cited on pages 16, 104).

[HKC+21b] C. Hakert, R. Kühn, K.-H. Chen, J.-J. Chen, and J. Teubner. “OCTO+: Optimized Checkpointing of B+Trees for Non-Volatile Main Memory Wear-Leveling”. In: *The 10th IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2021 (Cited on pages 16, 90).

[HKC+22] C. Hakert, A. A. Khan, K.-H. Chen, F. Hameed, J. Castrillon, and J.-J. Chen. “ROLLED: Racetrack Memory Optimized Linear Layout and Efficient Decomposition of Decision Trees”. In: *IEEE Transactions on Computers* (2022) (Cited on pages 16, 116).

[HYC+19] C. Hakert, M. Yayla, K.-H. Chen, G. v. d. Brüggen, J.-J. Chen, S. Buschjäger, K. Morik, P. R. Genssler, L. Bauer, H. Amrouch, and J. Henkel. “Stack Usage Analysis for Efficient Wear Leveling in Non-Volatile Main Memory Systems”. In: *1st ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*. Alberta, Canada, 2019 (Cited on pages 16, 83).

Contents

1 Introduction **1**
- 1.1 Exploiting Traditional Interfaces . . . 3
  - 1.1.1 Relevance to the Research Community . . . 4
  - 1.1.2 Interface Exploitation for NON-VOLATILE MEMORY (NVM) Lifetime 5
  - 1.1.3 Interface Exploitation for RACETRACK MEMORY (RTM) Properties 6
  - 1.1.4 Interface Exploitation for Immediate Arithmetic . . . 6
- 1.2 Contribution of this Work . . . 7
  - 1.2.1 Application Transparent NVM Wear-Leveling . . . 8
  - 1.2.2 Application Cooperative NVM Wear-Leveling . . . 9
  - 1.2.3 Memory Optimization for Random Forests . . . 11
  - 1.2.4 CPU Optimization for Random Forests . . . 13
- 1.3 Organization of the Thesis . . . 15
- 1.4 Author's Contribution to this Thesis . . . 15

2 Background and Related Work **17**
- 2.1 Technological Background . . . 18
  - 2.1.1 Disruptive Memory Technologies . . . 18
  - 2.1.2 Memory Hierarchies . . . 23
  - 2.1.3 Computer Arithmetic . . . 25
  - 2.1.4 RANDOM FOREST (RF) Ensembles . . . 26
- 2.2 Related Work . . . 28
  - 2.2.1 NVM Wear-Leveling . . . 28
  - 2.2.2 RTM Optimization . . . 32
  - 2.2.3 RF Performance Optimization . . . 33

3 System Model **35**
- 3.1 Non-Volatile Memory Model . . . 36
  - 3.1.1 Technical Overview . . . 36
  - 3.1.2 Wear-Out Model . . . 38
  - 3.1.3 Iterative Memory Writes . . . 39
  - 3.1.4 Hybrid Memories . . . 40
  - 3.1.5 Simulation Setup . . . 41
  - 3.1.6 Latency Model for RACETRACK MEMORY (RTM) . . . 43
- 3.2 Random Forest Execution Model . . . 44
  - 3.2.1 Probabilistic Execution Model . . . 45
  - 3.2.2 Implementation . . . 46
  - 3.2.3 Arithmetic Considerations . . . 48
  - 3.2.4 Performance Consideration . . . 48
  - 3.2.5 Tooling . . . 49

3.3 CPU Model . . . 50
3.3.1 Memory Hierarchy . . . 50
3.3.2 Floating-Point Arithmetic . . . 51

4 Application-Transparent NVM Wear-Leveling 53
4.1 Modern Technologies and Traditional Interfaces . . . 54
4.2 Overview . . . 55
4.2.1 Wear-Leveling Decisions . . . 55
4.2.2 Wear-Leveling Actions . . . 56
4.2.3 Wear-Leveling Flow . . . 57
4.3 Software-Managed Read and Write Wear-Leveling . . . 57
4.3.1 Scope . . . 58
4.3.2 Problem Analysis and Statement . . . 58
4.3.3 Coarse-Grained Wear-Leveling . . . 59
4.3.4 Fine-Grained Wear-Leveling . . . 63
4.3.5 Evaluation . . . 66
4.3.6 Wrap-Up . . . 70
4.4 Semantic Memory Tracing . . . 70
4.4.1 Scope . . . 71
4.4.2 Problem Analysis and Statement . . . 71
4.4.3 Modular Analysis . . . 72
4.4.4 Case Study . . . 74
4.4.5 Wrap-Up . . . 78
4.5 Concluding Interface Exploitation . . . 78

5 Application-Cooperative NVM Wear-Leveling 79
5.1 Modern Technologies and Traditional Interfaces . . . 80
5.2 Overview . . . 80
5.2.1 Application-Cooperative Decisions . . . 81
5.2.2 Application-Cooperative Actions . . . 82
5.3 Stack Usage Analysis and Wear-Leveling Hints . . . 82
5.3.1 Scope . . . 83
5.3.2 Problem Analysis and Statement . . . 83
5.3.3 Stack Usage Analysis . . . 84
5.3.4 Stack Wear-Leveling Overhead Optimization . . . 85
5.3.5 Evaluation . . . 86
5.3.6 Wrap-Up . . . 89
5.4 B^+-Tree Checkpoint Wear-Leveling . . . 90
5.4.1 Scope . . . 90
5.4.2 Problem Analysis and Statement . . . 91
5.4.3 B^+-Tree Organization . . . 92
5.4.4 OCTO$^+$ Algorithm . . . 92
5.4.5 Evaluation . . . 95

5.4.6 Wrap-Up . . . 98
5.5 Concluding Software-Based Wear-Leveling . . . 98

6 Memory Optimization for Random Forests 101
6.1 Modern Technologies and Traditional Interfaces . . . 102
6.2 Overview . . . 103
6.3 Unified Layout Optimization of DECISION TREEs (DTs) on Racetrack Memory . . . 104
6.3.1 Scope . . . 104
6.3.2 Problem Analysis and Statement . . . 105
6.3.3 BLOwing Trees . . . 106
6.3.4 Evaluation . . . 112
6.3.5 Wrap-Up . . . 115
6.4 Decomposed Layout Optimization of DTs on Racetrack Memory . . . 115
6.4.1 Scope . . . 116
6.4.2 Problem Analysis and Statement . . . 116
6.4.3 Decomposed Tree Optimization . . . 118
6.4.4 Evaluation . . . 124
6.4.5 Wrap-Up . . . 129
6.5 Concluding Memory Optimization of Random Forests . . . 130

7 CPU Optimization for Random Forests 131
7.1 Modern Technologies and Traditional Interfaces . . . 132
7.2 Overview . . . 133
7.2.1 Numeric Formats . . . 133
7.2.2 Memory Encoding and Hierarchy . . . 134
7.3 Immediate Encoding of Floating-Point Split Values . . . 136
7.3.1 Scope . . . 136
7.3.2 Problem Analysis and Statement . . . 136
7.3.3 Immediate Encoding . . . 140
7.3.4 Evaluation . . . 142
7.3.5 Wrap-Up . . . 147
7.4 FLInt: Exploiting Floating-Point Enabled Integer Arithmetic for Efficient Random Forest Inference . . . 148
7.4.1 Scope . . . 148
7.4.2 Problem Analysis and Statement . . . 149
7.4.3 Providing Correct Floating-Point Comparisons with Integer and Logic Arithemtic . . . 149
7.4.4 Evaluation . . . 157
7.4.5 Wrap-Up . . . 163
7.5 Concluding CPU Optimization for Random Forests . . . 164

8 Conclusion and Future Work **165**
8.1 Conclusion 166
8.1.1 Summary 166
8.1.2 Outreach 167
8.2 Future Work 168

Acronyms **171**

Bibliography **173**

CHAPTER 1

Introduction

Contents

1.1 Exploiting Traditional Interfaces 3
1.1.1 Relevance to the Research Community 4
1.1.2 Interface Exploitation for NON-VOLATILE MEMORY (NVM) Lifetime 5
1.1.3 Interface Exploitation for RACETRACK MEMORY (RTM) Properties 6
1.1.4 Interface Exploitation for Immediate Arithmetic 6
1.2 Contribution of this Work 7
1.2.1 Application Transparent NVM Wear-Leveling 8
1.2.2 Application Cooperative NVM Wear-Leveling 9
1.2.3 Memory Optimization for Random Forests 11
1.2.4 CPU Optimization for Random Forests 13
1.3 Organization of the Thesis 15
1.4 Author's Contribution to this Thesis 15

> ***“We’re on the cusp of another Golden Age that will significantly improve cost, performance, energy, and security. [...] We’ve identified areas that are critical to this new age: 1) Hardware/Software Co-Design for High-Level and Domain-Specific Languages [...]”***
>
> John Hennessy and David Patterson [HP18]

Computer Science as a domain features multiple properties, which are unique to this discipline. Two of these properties are 1) the rapid and disruptive development of modern technologies [LR03] and 2) the extensive use of traditional abstractions in order to allow managing the complexity of the evolving computer systems [CS07].

While tens of years ago, tape drives, spinning hard disks and megabyte-sized main memory without deep cache hierarchies formed the standard of the memory subsystem of a computer system, as of the time of this thesis, terabyte-sized, complex main memory with hundreds of megabytes sized cache hierarchies, quantum effect storage devices and mass data backup facilities in the form of SOLID STATE DRIVE (SSD) clusters are a standard to computing applications. Active research interest promises disruptive technology changes in a similar scope in the future. Integration densities, which would have been considered impossible a few years ago, in memory and even in memory material computing, even larger memory capacities and amounts of computing cores are promising to show up within the next years. Also shifting the focus away from the development in hardware technologies towards modern software technologies, disruptive changes are visible over the recent years. While algorithmic research, complexity analysis and the design of efficient problem-solving strategies belong together with computers since their beginning of existence, other application types, such as resource-demanding machine-learning estimators, error-tolerant algorithms and even complex management schemes inside operating systems belong to the standard set of applications nowadays.

On the other hand, studying abstractions and the creation of interfaces of computing systems, reveals a contrary development speed, compared to disrupting technologies. Consider, for instance, the memory interface of computers. In the Von-Neumann architecture, memory plays a central role to store code and data memory. Consequently, INSTRUCTION SET ARCHITECTURES (ISAs) formed a simple abstraction of memory accesses years ago. They provide load and store instructions, which specify a memory location to load from or store to. Modern computers offer still the same interface for software to access the memory. On both sides of this interface, modern technologies are applied by introducing further levels of abstractions. For instance, considering memory hierarchies, memory accesses caused by the application are not redirected to the memory device directly, but rather go through the various levels of caches upfront. Also,

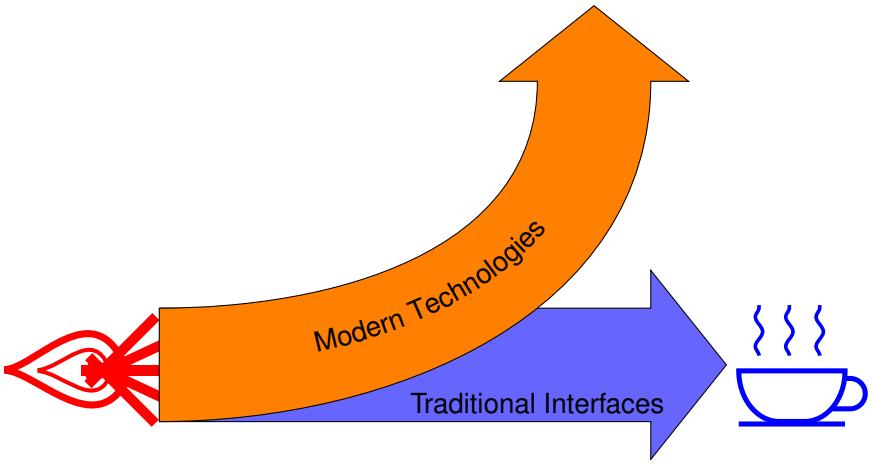


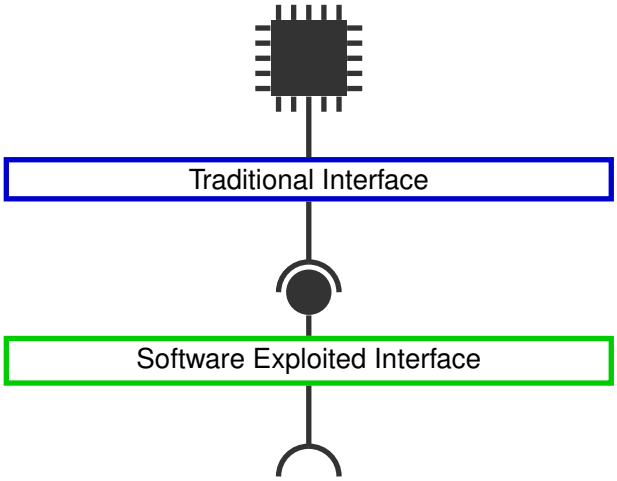
Figure 1.1: Development of Technologies and Interfaces

from the perspective of applications, neural networks are seldom implemented directly by computing memory addresses and doing load and store operations to these addresses. Instead, levels of abstraction are put on top of the basic software interface, which allow implementation in object-oriented or scripting based languages. As highlighted in the opening quote of this section, Hennessy and Patterson identify the co-design between hardware and software innovations as one core challenge of the current “golden age” [HP18].

From the previously mentioned aspects, it can be noted that the development of modern technologies, including hardware and software development, is skyrocketing into new spheres in the past and most probably also in the near future. The basic interfaces, however, which connect modern technologies with each other and probably also with not that modern technologies, keep crawling on the ground and remain largely unaffected by the technological development. Bringing both worlds together remains as an open challenge. It should be clarified, that this thesis does not focus on methods against this uneven development, but rather studies the need to effectively make use of the existing interfaces, in other words to exploit traditional interfaces in order to operate modern technologies in computer systems.

1.1 Exploiting Traditional Interfaces

As the landscape of technology is massively large, this thesis does not aim to provide universal solutions, but rather focuses on specific technologies and specific interfaces and contributes methods for the explicit exploitation of the relevant interfaces. The following illustration highlights the basic principle behind every contribution of this thesis and can be found in a more specific version in the corresponding chapters.



Given that certain technologies are available, indicated by the illustration of a chip with pins, traditional interfaces exist to use these technologies. In order to properly operate the technologies, a broader interface is required. One way to provide such an extended interface is to employ special software solutions, which exploit the existing interface and on their own provide a broadened, software exploited interface. This then can be used by target applications to operate the modern technology.

1.1.1 Relevance to the Research Community

As of the time of this thesis, research efforts are actively conducted explicitly into the direction of software exploitation of interfaces for modern technologies. This thesis puts a major focus on disruptive memory technologies and the required management to operate them efficiently and effectively. In a more specific scope, this is a defined target of the DFG (Deutsche Forschungsgemeinschaft) Project *Design and Optimization of Non-Volatile One-Memory Architecture (NVM-OMA)*[1]. The concept behind this project is to develop solutions for a unified memory landscape, consisting of NVM only, which is one type of disruptive memory. This project aims to provide software based solutions and hence enable the software management of NVMs. Beyond specific software management of NVMs, there is active research interest in disruptive memory technologies itself. In the scope of a special priority program, the DFG conducts a number of research projects to actively pursue research on disruptive memory. This program can be found under the number *SPP 2377*[2]. Specifically, the project *Memory Diplomat*[3] is dedicated to the development of extended technology interfaces. The core idea of this project is to develop a central software instance, the so-called memory diplomat, which has knowledge about applications and the hardware and can negotiate required management strategies. The memory diplomat further contains extension and exploitation means

[1] https://gepris.dfg.de/gepris/projekt/405422836
[2] https://gepris.dfg.de/gepris/projekt/460954224
[3] https://gepris.dfg.de/gepris/projekt/502384507

to manage the hardware. These extensions in this project are entirely achieved by a central software instance. In consequence, the contributions from this thesis align with the declared target of such existing research efforts.

1.1.2 Interface Exploitation for NVM Lifetime

Bringing a clearer focus to the technologies and topics, addressed by this thesis, one relevant technology is NON-VOLATILE MEMORY (NVM). Although many different hardware technologies behind NVM exist and also many special characteristics are caused by these technologies, this thesis picks up a central issue, which is common to many of these technologies, namely the lifetime issue. Due to technical properties of different memory materials used for NVM, they can break after a comparably low amount of memory accesses. Without active lifetime maintenance, the memory would endure for a few months only. This is mainly caused by the fact that executing software tends to use some memory portions more often than other portions. Hence, although the total volume of accesses may not be very high, the intensively used memory portions can break early. On the other hand, this behavior allows a significant extension of NVM lifetime, when the memory accesses are well distributed. The process of achieving such a distribution is called wear-leveling.

From the perspective of interfaces used for emerging NVM technologies, the basic memory interface has to be considered as a baseline. In order to properly operate the process of wear-leveling, two extensions to the interface should be considered: 1) the possibility to determine current memory wear-out and 2) the possibility to redirect memory accesses to other portions. Although it is not crucially necessary to know the current state of the memory and how much lifetime is left per memory portion, it can help to pursue a further optimized utilization of the memory. Extracting information about the current memory aging over the traditional memory interface is not possible due to its focus of loading and storing memory requests. If software aims to exploit this interface and keep track of the memory lifetime internally, a straightforward approach is to hook into memory accesses directly from a software level and maintain an internal model of the aging of memory portions. Hooking into memory accesses by software is challenging, since the default design of computers does not provide standard means to achieve this. Consequently, appropriate solutions are required. Either a combination of other available technologies and interfaces is used in order to achieve the required goal, or the application is modified in order to hook into memory accesses. Both directions are explored in this thesis.

The interface extensions of being able to redirect the target of memory accesses can be approached from similar directions. Either, other existing means and interfaces are combined in a creative way to redirect memory accesses, or the application is again modified in order to achieve this. Both directions are explored, partially in an overlay manner in this thesis. It should be noted that of course a large research community explores possibilities to directly change the traditional memory interface in order to achieve the management of NVM regarding their lifetime. Such approaches can usually

gain a high efficiency, especially when additional hardware controllers are introduced into the system. This thesis, however, aims for another research direction and aims to provide alternatives, namely the software exploitation of interfaces. Following both of these research directions opens a larger design space for creation of future computing systems.

1.1.3 Interface Exploitation for RTM Properties

One special type of NVM, which comes with unique shift properties, is RACETRACK MEMORY (RTM) [BKF+20]. RTM is organized in a way, that memory contents are stored at adjacent physical locations, but only one of the locations can be read or written. When a specific element of the memory should be accessed, the memory positions have to be moved towards the access port first. This operation is called *shift*. Naturally, shifts take time and also cause an impact on the energy consumption of the memory. These effects correlate to the distance of a shift. That is, the more memory positions have to be skipped upfront to an access, the more latency and energy overhead is caused. The previous position of the memory depends on the previous access. Hence, the latency and energy consumption of RTM is access sequence dependent.

Towards interface extensions, the caused overhead for shift operations requires a mechanism to be controlled by the running software. This can be achieved by offering interface extensions to specify the location of memory elements, which essentially allows gaining control over the memory access sequence. Broadening the traditional memory interface towards such a behavior by software can be potentially achieved in applications themselves. For example, applications can utilize data structures, which are constructed by pointers between single elements of the data structure. This allows to change the actual physical location of single data structure elements and keeping the view from the application on the logical correctness of the data structure untouched. In this thesis, tree-based data structures are considered as such a controllable data structure. In addition to the action space of controlling the memory access sequence, a second extension to the interface is required, i.e. an extension allowing to decide for a good reordering of the memory access sequence. One possible approach is to formulate a cost model for the cause latency and energy overhead. This cost model then can be used as an objective function for optimization. Execution-specific behavior, i.e. knowing which memory elements are accessed subsequently, are crucial to create such a cost model. Since this thesis focuses on an application-specific interface extension, an application-specific cost model is derived for DECISION TREE (DT) applications.

1.1.4 Interface Exploitation for Immediate Arithmetic

Beyond concrete NVM technologies, considering a generic concept of memory hierarchies and the efficient utilization of system caches, immediate arithmetic becomes an interesting aspect. In modern memory hierarchies, instruction memory and data memory can be separated in the high cache levels. That means, instruction memory accesses,

i.e. accesses to encoded instructions target a different portion from the system cache than data memory accesses. If applications now explicitly want to optimize their cache behavior, controlling the amount of accessed to data and instruction memory can be a desirable mechanism. Immediate encoding of constants, i.e. storing numeric values directly in the immediate field of instructions offers a mechanism to move stored values from data memory to instruction memory and vice versa. Traditional ISAs, however, only offer such a possibility for integer numeric values.

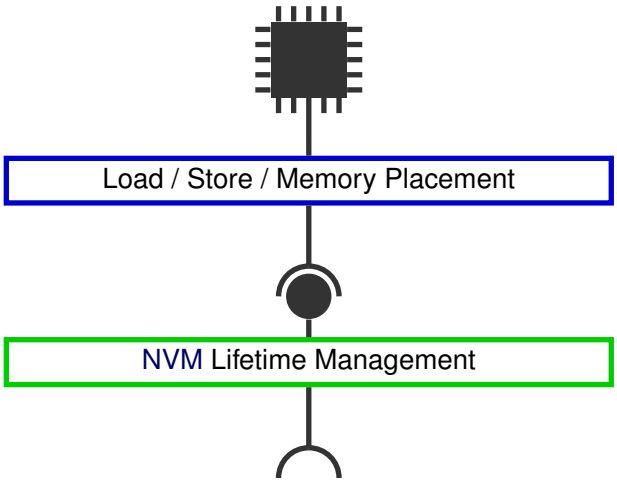
With the rise of MACHINE LEARNING (ML) applications into various classes of computing systems, spanning a range from embedded low-power systems up to higher performance systems, also the need to intensive floating-point arithmetic arises in these systems. If a target data set is provided with floating-point numeric values, the MACHINE LEARNING (ML) model has to include floating-point arithmetic at a certain level to properly reflect this data set. Consequently, ML models can be forced to rely on intensive floating-point arithmetic. It is also a desirable target to optimize the cache utilization of ML models, which can be assisted by the option for immediate encoding. Consequently, in order to broaden the traditional interface for this case, immediate encoding for floating-point arithmetic should be enabled. Achieving this in a software extension of the interface is not entirely intuitive, since floating-point arithmetic is usually hardware based. Hence, this thesis explicitly studies the comparison operation of floating-points (e.g. $\geq$), which is the only operation required for DECISION TREE (DT) models and ensembles and provides software based solutions to offer immediate encoding.

1.2 Contribution of this Work

This thesis addresses the problem of software exploitation of traditional interfaces for specific modern technologies and applications. For concrete cases, methods are proposed, studied, and evaluated, which can offer an extended interface to allow an effective operation of the considered technology. As highlighted before, these methods do not aim to be universally applied or to dominate other solutions, but rather to provide alternative options, which are based on a software exploitation of the interface. The first two topical chapters of this thesis (Chapter 4 and Chapter 5) pick up the lifetime issue of NVM. The basic extensions, which need to be added to the memory interface, follow the purpose to handle lifetime issues. The third topical chapter (Chapter 6) picks up RTM as a specific technology and discusses interface extensions for effective latency and energy management of this memory technology. The last topical chapter (Chapter 7) picks up the possibility for immediate encoding of numeric values and discusses an interface extension, allowing immediate encoding for floating-point values.

1.2.1 Application Transparent NVM Wear-Leveling

The first type of NVM lifetime management, considered in this thesis is application transparent wear-leveling (Chapter 4). In this scope, applications remain untouched and the required steps for wear-leveling are conducted in a software-based manner outside the applications. The suitable target, consequently, is the operating system and the system software in order to realize wear-leveling functionality:



Wear-leveling requires two basic components: 1) extracting information about the remaining lifetime of certain memory portions and 2) redirecting the physical target location of memory accesses in order to level the utilization of lifetime across the entire memory space.

Towards the component of extracting information, this thesis introduces two orthogonal approaches. In the first approach, existing functionality of modern CPUs is creatively reused. Performance counters, which can count certain hardware events, are used to count the number of read and write accesses in the system. An interrupt mechanism is used to trigger a system interrupt after a specified number of memory accesses. With the help of memory access permissions in the MEMORY MANAGEMENT UNIT (MMU), the target of the next memory access is recorded. This is achieved by forbidding access to the entire memory, such that the subsequent access causes a memory access violation, which delivers the violated address to the interrupt handler. Afterwards, the default state is restored. In consequence, this method provides a sampled histogram of memory accesses, which serves as an estimation of the memory wear-out, caused by the running software. Accumulating the histogram over time provides an estimation about the remaining memory lifetime. In addition to this methodology, the thesis introduces semantic enrichment of the collected information. The target application is deployed to a modular unikernel, which allows extraction of the relation of the binary layout to single components of the unikernel. This can then be used to enrich the previously collected memory access information with knowledge about the unikernel component.

Wear-leveling can then be targeted to specific components. The semantic enrichment is achieved with a static tracing methods, which analyzes the binary memory layout of the entire kernel instance and a dynamic tracing method, which checks the current PROGRAM COUNTER (PC) during the execution of a memory access to determine the corresponding component.

Towards the redirection of memory accesses, this thesis also introduces two orthogonal approaches. These two approaches operate on a different target granularity, making them naturally combinable. For coarse granularities, MMU page tables are modified in order to redirect memory accesses to the virtual address space to other physical locations. By copying the data also to the corresponding new physical location, application transparency is achieved. For fine granularities, the text and stack segments of the running program are modified. Both are moved in fine-grained steps through the memory and therefore the accesses to these segments are distributed across the physical memory space. Position-independent code ensures the application transparency for the text segment. Stack pointer relative addressing ensures the application transparency for the stack segment.

The combination of both components is also implemented in the form of an intuitive wear-leveling algorithm in this thesis. It should be, however, noted that the design of a wear-leveling strategy is not within the scope of this thesis. The focus of this work is on providing methods in software to exploit the existing memory interface for wear-leveling strategies. In short, the contributions for application-transparent wear-leveling are summarized in the following:

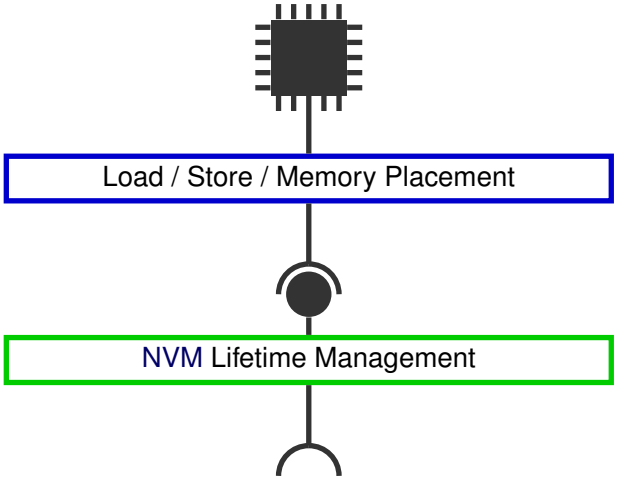
Contributions: Applications Transparent NVM Wear-Leveling

- Online read and write approximation with performance counters and the MMU in order to derive a statistically approximated trace of real read and write accesses of the application during runtime.
- Coarse-grained aging-aware wear-leveling, utilizing the approximate read and write trace to execute replacement operations with the virtual to physical memory mapping of the MMU.
- Fine-grained non aging-aware wear-leveling, operating on the stack memory for read and write accesses and on the text memory for read accesses.
- Static memory tracing, utilizing compiler information from a unikernel to map the binary layout of the compiled unikernel to the single included libraries.
- Dynamic memory tracing to further provide a semantic association of memory regions in the dynamically allocated memory segments to the corresponding library of the unikernel.

1.2.2 Application Cooperative NVM Wear-Leveling

Based on the methods for application-transparent NVM wear-leveling, this thesis contributes with design approaches towards application-cooperative wear-leveling. The

interface extension targets the same functionality, but the methods to achieve this shift from the operating system and the system software into the application directly:



The advantage of achieving wear-leveling mechanisms in the application directly can be efficiency and effectiveness. However, this comes with the cost of tailoring the wear-leveling scheme specifically to a certain target application. This thesis follows two approaches, an application-cooperative wear-leveling scheme, which is not directly tied to any specific application, and the other approach specifically proposes a wear-leveling scheme for B^+ trees.

For the first approach of application-cooperative wear-leveling, without targeting specific applications, the focus is put on the stack memory. Stack memory, due to its usage for intermediate data, can be a significant driver for memory wear-out. Consequently, it is specifically focused. When deciding to move the target location of stack memory in application-transparent wear-leveling schemes, a certain overhead is caused for moving the relevant memory contents to a new position. This overhead can be not only noticed in the form of a time overhead, but also in the form of a lifetime overhead, since additional memory accesses are caused. Consequently, the amount of memory to be moved when the stack position is relocated should be minimized. The proposed method in this thesis employs a GENETIC ALGORITHM (GA) on offline-profiled stack behavior of a target application in order to identify optimized parts in the application code, to trigger a relocation of the stack memory. The determined relocation points are then fed back into the application source code in the form of special annotation, which notify the underlying operating system and the application transparent wear-leveling subsystem to cause a stack relocation.

For the second approach for application-cooperative wear-leveling, this thesis proposes a wear-leveling scheme, dedicated for B^+ trees, which are used as an important component of database management systems. The collection of information about the memory usage and the impact on the lifetime of memory portions is directly performed

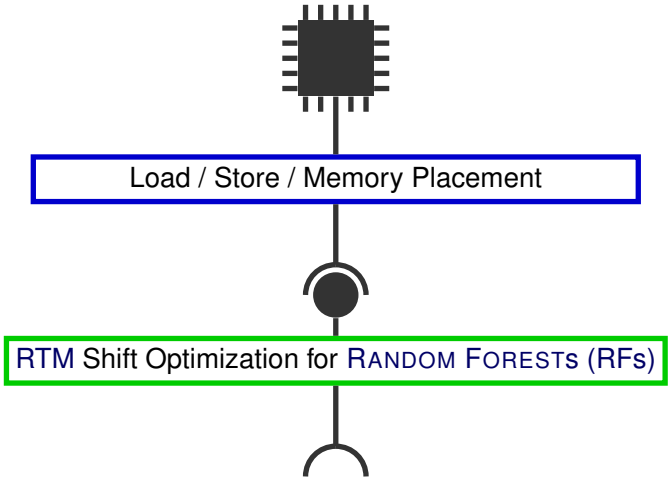
in the tree operations themselves. Whenever the tree implementation touches a node, an access histogram is updated and maintained to keep track of memory modifications. In order to effectively utilize this histogram for wear-leveling, a checkpointing mechanism is explored in this thesis, where an entire checkpoint of the tree is written from volatile memory to NVM at the time of a checkpoint. The histogram is used to decide for a mapping of tree nodes to memory portions in the checkpoint. An efficient search algorithm is directly integrated to provide a fast heuristic solution to optimize the mapping for the checkpoint memory. In short, the contributions for application cooperative wear-leveling can be summarized as follows:

Contributions: Application Cooperative NVM Wear-Leveling

- A ptrace-based stack analysis-framework for stack-size recording of single instructions.
- A GENETIC ALGORITHM (GA) for finding instructions with low stack sizes.
- An annotation mechanism to trigger stack wear-leveling at specific instructions.
- A modified B^+-tree implementation, which maintains modification information within tree nodes, which are updated in a lightweight manner during updates and inserts.
- A wear-leveling implementation, utilizing the collected modification information during checkpoints to apply a wear optimized mapping.

1.2.3 Memory Optimization for Random Forests

The third topical chapter of this thesis (Chapter 6) shifts the focus from the common property reduced lifetime for NVM memories towards a special type of NVM, namely RACETRACK MEMORY (RTM). RTMs feature a unique property of an access-dependent latency and energy consumption. The reason is that memory contents are stored in adjacent positions, but read and write operations can only be performed at the position of the access head. Contents, consequently need to be shifted to this access head upon an access first. The introduced overhead for latency and energy should be minimized in order to properly operate RTM memory. In the context of the exploitation of the memory interface, this requires gaining explicit control over the shift behavior in RTM:



This interface exploitation requires two components: 1) the possibility to change locations of memory elements in order to influence the shift cost for access to this memory element and 2) the formulation of a cost model in order to apply an optimization algorithm. In this thesis, these two exploitation are realized as an application-specific solution for RANDOM FOREST (RF) ensembles.

In the application-specific context of RANDOM FOREST (RF) ensembles, mainly focusing on the internal DTs, modifications of the memory positions of single elements can be achieved straightforward. Towards this, the implementation as native trees is considered, where DT nodes are realized as array elements and the child nodes are indicated by an index to the node array. When a node is mapped to an arbitrary position in the array, only the child pointers need to be adjusted accordingly. In this thesis, the RTM optimized layouting is done during the implementation time of the ensemble; hence the corresponding code can be generated directly.

The contribution of this chapter then focuses on deriving the cost model and the optimization of the layout with respect to this cost model. Towards the cost model, empirical branching probabilities are collected during the training time of the single DTs. Since ML applications in general assume the test data to stem from a similar distribution as the training data, the probabilities are considered to be accurate also during productive use. Starting from the relative branching probabilities, an absolute probability model is created, which indicates the probability to visit a node from its parent node. Together with the shifting distance from the parent to the node, this forms a cost model of the expected amount of shifts during inference of a DT. Leaving the position of each node as the solution space, the cost model can be interpreted as an optimization problem. This optimization problem can be formulated an instance of the quadratic assignment problem or an instance of the optimal linear order problem, which is NP hard and infeasible to be solved for larger DTs in tolerable time.

Due to the specific focus on DT models, a reduction of the problem complexity is possible. The optimal linear ordering problem can be solved in $\mathcal{O}(n \cdot \log n)$ for rooted tree models. The cost model for DTs on RTM is not exactly transferable to

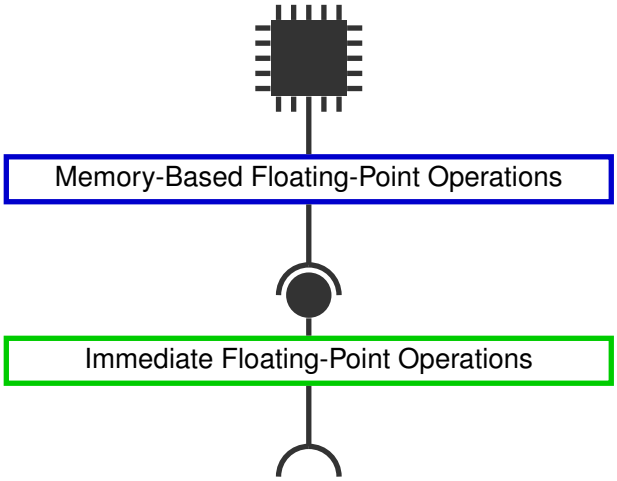
the rooted tree problem, but comes somewhat close. This thesis contributes a slightly modified version of the fast optimization algorithm and contributes a formal proof about the relation to the optimal solution. Towards this, two organization approaches are proposed: 1) tree nodes are entirely stored at one RTM position and 2) tree nodes are decomposed in three subcomponents and stored in individual clusters of the RTM. The formal proof guarantees an upper bound of $4\times$ for the unified organization and $12\times$ for the decomposed organization. Experimental evaluation even indicates that these upper bounds are pessimistic in most cases. In short, the contributions can be summarized as follows:

Contributions: Memory Optimization for Random Forests

- A formal cost model for shift overhead of racetrack memory in a unified and decomposed organization with respect to a probabilistic execution model is specified. This model is expressed as an INTEGER LINEAR PROGRAM (ILP), which allows for deriving the optimal solution, if the required time is spent.
- A fast optimization algorithm, which reduces the problem of DT layouting in RTM to an efficient solvable optimization problem. The algorithm is further modified for the specific characteristics of the studied problem.
- A formal proof of the upper bound of $4\times$ for the unified organization and $12\times$ for the decomposed organization for the fast optimization algorithm compared to the optimal solution.

1.2.4 CPU Optimization for Random Forests

The last topical chapter of this thesis (Chapter 7) leaves the area of disruptive memory technologies and puts a focus on the design of the memory subsystem itself. The organization into memory hierarchies allows applications to optimize their memory behavior for performance optimization in a given memory hierarchy. As described above, distinguishing data and instruction memory can be an important knob to control the behavior in caches, since they are distributed into instruction and data caches usually in the highest cache levels. One way of gaining active control over this is to encode numeric values either in data memory or in the immediate field of instructions directly. For floating-point numerical values, this is not possible, consequently the software exploitation to enable this is considered here:



In this thesis, again specific solutions for RF ensembles are considered since cache optimization is a known desirable target for them. In addition, the arithmetic structure of DTs is very simple, allowing for creative solutions. This thesis follows two approaches in order to exploit existing immediate encoding to achieve immediate arithmetic for floating-point values. Firstly, a generic method is discussed, which is not directly bound to DTs. Secondly, a specific method is presented, which only enables the comparison operation for immediate floating-point numbers in a more effective manner.

For the generic solution, the basic concept is to consider the binary representation of a floating-point number as a bit vector, which can also be interpreted as an integer number. This bit vector is then immediately encoded into a set of instructions. For instance, in a 32-bit architecture, instructions naturally cannot have 32-bit wide immediate fields. Therefore, a set of instructions is synthesized to load the 32-bit floating-point bit vector into a register from immediate encoded portions of the bit vector. Afterward, the integer register is copied bit by bit to a floating-point register without conversion, such that the number is available for arbitrary floating-point arithmetic. In consequence, only the load operation to a floating-point register, which would normally have to be done from data memory, is replaced by corresponding immediate encoded operations. This method has the advantage that afterward the number is present in a floating-point register and can serve for any type of floating-point computation.

To provide a further optimized solution, a specific solution for the comparison operation between floating-point values is contributed in this thesis. In order to omit the conversion steps, required in the previous method, the relation between floating-point numbers and the interpretation of their bit vector as two's complement integer numbers is investigated. A formal proof is contributed, which shows that, except a few special cases, the floating-point and two's complement interpretation of an arbitrary bit vector is monotonic, hence comparison operations are equivalent in both interpretations. Consequently, the bit vector representation can be treated as an integer value directly and the comparison can be evaluated. The handling of the special cases, which break the equivalence, can be simplified to simple logic operations, when one of the numbers

for comparison is known upfront. In DTs this is exactly the case, since the split value, which is used for comparison of incoming data tuples, are fixed during training time. Consequently, optimized implementations of the special case handling can be directly synthesized during implementation time. In short, the contributions to enable immediate based floating-point arithmetic can be summarized in the following overview:

Contributions: CPU Optimization for Random Forests

- The implementation of immediate encoded floating-point split values in the text memory, shifting the pressure between data and text memory.
- FLInt: A two's complement and logic operation based comparison operator for floating-point numbers, where the correctness is formally proven.
- An efficient implementation of FLInt in RFs with if-else tree implementations, where the special case handling is resolved offline during the implementation time.

1.3 Organization of the Thesis

The rest of this thesis is structured as follows:

- Chapter 2 provides technological background about the relevant technologies to this thesis. In addition, this chapter contains an overview about the relevant related work to the contributions of this thesis.
- Chapter 3 provides further details about used technologies and methods. Assumptions, metrics and tools are clarified, which are relevant to the contributions.
- Chapter 4 contains the first chapter of topical contributions towards application transparent NVM wear-leveling.
- Chapter 5 contains the second chapter of topical contributions towards application cooperative NVM wear-leveling.
- Chapter 6 contains the third chapter of topical contributions towards memory optimization of RFs.
- Chapter 7 contains the fourth chapter of topical contributions towards CPU optimization of RFs.
- Chapter 8 concludes this thesis by giving an overview of a broader scope and of future work.

This thesis has gray bars on the side of the text to indicate that the presented contribution is a published work. The reference appears on the first page of the corresponding block as a footnote.

1.4 Author's Contribution to this Thesis

In accordance to §10 (2) of the *Promotionsordnung der Fakultät für Informatik der Technischen Universität Dortmund* from 29.08.2011, this section provides an explanation

about the collaboration with other people on the contributions and the own contribution of the author of this dissertation. The author of this dissertation is referred to by the term “me” in the following.

Before giving a detailed explanation of the single contributions, all publications, which are the basis for the presented contributions in this thesis, are conducted by me as the first author. Implementations are done mainly by myself. The writing of the publications is further done in major parts by me. The co-authors of the single publications mainly assisted in discussions, providing alternative ideas and in formulation of single text elements in the publications. Especially the review of the related work was done in tight collaboration with the co-authors.

- The work about Software Managed Read and Write Wear-Leveling (published in [HCS+21]) in Chapter 4 is implemented entirely by myself. Major parts of the implementation stem from my master thesis. This includes the basic form of the memory access estimation and the wear-leveling algorithm for write accesses. The estimation of read accesses, the read wear-leveling and the special handling of the text segment are contributed by me beyond the scope of my master thesis. Evaluation and refinements were also added beyond the scope of the master thesis. My co-author Kuan-Hsun Chen provided significant assistance for the review of the related work. All co-authors provided significant assistance for the writing of the publication and defining the presentation flow.
- The work about Semantic Memory Tracing (published in [HCK+20]) in Chapter 4 is conducted by myself. For realizing the presentational flow, my co-authors, especially Felipe Huici provided major assistance.
- The work about Stack Usage Analysis and Wear-Leveling Hints (published in [HYC+19]) in Chapter 5 is also conducted by myself. The co-authors assisted in topical discussions and writing of the publication.
- The work about B^+ Tree checkpointing (published in [HKC+21b]) in Chapter 4 is realized by me together with the co-author Roland Kühn. We implemented the method together, where I contributed the checkpointing implementation, the simulation and the wear-out evaluation. Roland Kühn provided the implementations specific to the B^+ tree, especially the collection of modification information. The writing of the publication is mainly shared between me and Roland Kühn. The other co-authors assisted in writing and topical discussions.
- The work about Layout Optimization of DTs on RTM (published in [HKC+21a; HKC+22]) in Chapter 6 was mainly conducted by me together with Asif Ali Khan. While I contributed the implementation and the DT related work, he contributed with the modeling of the RTM specific behavior. Especially the evaluation of the energy consumption is contributed by him. Jian-Jia Chen further provided major assistance in formalizing the proof.
- The work about immediate encoding for floating-point (published in [HCC22b; HCC22a]) in Chapter 7 was mainly conducted by myself. The co-authors assisted in topical discussion and in refining the presentation flow of the publication.

CHAPTER 2

Background and Related Work

Contents

2.1 Technological Background 18
2.1.1 Disruptive Memory Technologies 18
2.1.2 Memory Hierarchies 23
2.1.3 Computer Arithmetic 25
2.1.4 RF Ensembles 26
2.2 Related Work 28
2.2.1 NVM Wear-Leveling 28
2.2.2 RTM Optimization 32
2.2.3 RF Performance Optimization 33

	SRAM	DRAM	HDD	NAND Flash	STTM	ReRAM	PCM	FeRAM
Cell size (F^2)	$120-200$	$60-100$	N/A	$4-6$	$6-50$	$4-10$	$4-12$	$6-40$
Write Endurance	10^{16}	$>10^{15}$	$>10^{15}$	10^4-10^5	$10^{12}-10^{15}$	10^8-10^{11}	10^8-10^9	$10^{14}-10^{15}$
Read Latency	$\approx 0.2-2$ns	≈ 10ns	$3-5$ms	$10-35\mu$s	$2-35$ns	≈ 10ns	$20-60$ns	$20-80$ns
Write Latency	$\approx 0.2-2$ns	≈ 10ns	$3-5$ms	$200-500\mu$s	$3-50$ns	≈ 50ns	$20-150$ns	$50-75$ns
Leakage Power	High	Medium	mechanical	Low	Low	Low	Low	Low
Dynamic Energy	Low	Medium	mechanical	Low	Low/High	Low/High	Medium/High	Low/High
Maturity	Mature	Mature	Mature	Mature	Test chips	Test chips	Test chips	Manufactured

Table 2.1: NVM Characteristics [BRC+17]

2.1 Technological Background

This section aims to provide a background of various technologies, which are important throughout this thesis. It should be noted, that this section does not target to pick up the technological aspects, which are crucially relevant to the discussed methods in detail, but rather provides a broad overview about the used technologies. The specific details, which are relevant to the methods, are detailed in Chapter 3. Referring to the title of this thesis, this chapter discusses the *modern technologies*. The specific relation to the traditional interface is highlighted with the methods to exploit them by software.

There are basically two modern technology fields relevant to this thesis: One is disruptive memory technologies, the other is memory and computation organization. Especially the organization in memory hierarchies is relevant to this thesis. The third and the fourth technical chapter (Chapter 6 and Chapter 7) intensively consider RANDOM FOREST (RF) ensembles as a dedicated use case, consequently a broad overview of random forests is provided in this section as well.

2.1.1 Disruptive Memory Technologies

The first three technical chapters of this thesis (Chapter 4, Chapter 5 and Chapter 6) discuss problems and develop solutions for a software based management of specific types of non-volatile memory through traditional memory interfaces. The actual name giving property of the non-volatility thereby is not of central interest to this thesis. The term NVM itself describes memory technologies, which do not lose their stored values when being powered off. Indeed, hard disk drives, tape drives or flash storage block devices belong to this category of memory. However, these storage technologies are not the focus of this thesis. Instead, a focus is put to non-volatile storage class memory or byte-addressable non-volatile main memory, i.e. NVM technologies which can be used as a one by one replacement for main memory. A variety of such technologies is disrupting the memory landscape, all facing slightly different technological aspects. However, most NVM technologies utilize physical properties of materials to encode stored values, which can induce a significant wear-out. Consequently, lifetime is an issue, which is common to many disrupting NVM technologies. Table 2.1 summarizes

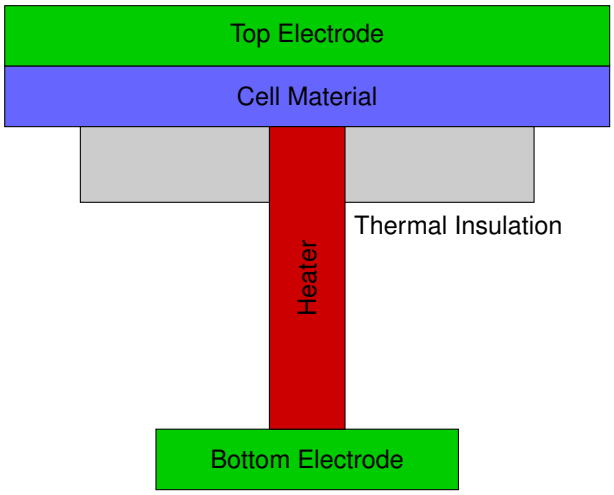


Figure 2.1: PCM Cell Design Example

technical properties of a few NVM technologies in comparison to classic DYNAMIC RANDOM ACCESS MEMORY (DRAM) and STATIC RANDOM ACCESS MEMORY (SRAM), but also HARD DISK DRIVES (HDDs) and NAND flash memories. Although the detailed information in this table can be considered a little outdated, the trend clearly shows a significantly lower endurance for emerging NVM technologies (SPIN-TORQUE TRANSFER MAGNETORESISTIVE RAM (STTM), RESISTIVE RAM (ReRAM), PHASE CHANGE MEMORY (PCM) and FERROELECTRIC RAM (FeRAM)). The latency and energy characteristics, reveal good chances to be an adequate candidate to replace DRAM as main memory. In order to give a better intuition, how storing memory values works in these technologies, a few examples are provided in the following.

PHASE CHANGE MEMORY (PCM)

The explanations and illustrations about PCM are taken from [LW08; BRC+17]. The basic concept of PCM is to use a dedicated material (e.g. chalcogenide alloy), which can be transferred to a crystalline or amorphous state by controlled heating. Heating the material up quickly to a high temperature and letting it cool down quickly again achieves an amorphous state. Heating the material up slowly to a lower temperature and keeping the temperature for a certain duration before cooling down leads to a crystalline organization. From an electrical perspective, the crystalline state has a low electric resistance, while the amorphous state has a high resistance. Taking this together, a memory cell can be assembled: The material is deployed with a controllable heater and proper thermal insulation. Utilizing a conductive heater, electrodes can be connected to two sides of the material, which allow measuring the resistance. Programming the cell is achieved by heating the material either to the crystalline or amorphous state. Reading the cell is achieved by measuring the cell resistance. An exemplary cell design is depicted in Figure 2.1. It should be noted that the state of the cell material is not

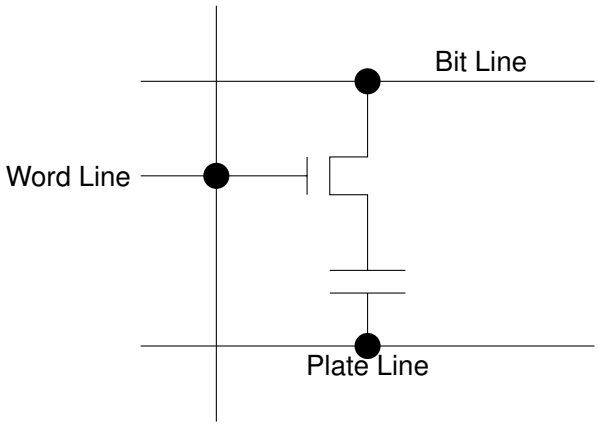


Figure 2.2: DRAM Cell Design Example [GMS+21]

binary, but continuous between crystalline and amorphous. This allows, on the one hand, the design of multi-level cell memory and on the other hand efficient programming of the cells with iterative writes, where the cell state is changed in incremental steps until the target state is achieved. This omits the need for a full programming procedure partially. However, the continuous cell state also comes with a downside. Resulting from a natural operation temperature of the memory device, the amorphous state is meta stable and may crystallize over time. This results in a limited retention time for the amorphous state. The less crystallized the material is, the longer the encoded value can be stored. Consequently, the time and effort invested in the programming of the amorphous state results in a longer retention time. This opens a possible trade-off between programming effort and retention.

FERROELECTRIC RAM (FeRAM)

In comparison to the previously presented PCM, FERROELECTRIC RAM (FeRAM) features a design, which is very close to classic DRAM. One possible design of a DRAM device is the 1T1C architecture, where one transistor and one capacitor is used to form a single memory cell [GMS+21]. Such a dram cell could be realized as illustrated in Figure 2.2. The basic concept behind this architecture is to store the memory value as a charge in the capacitor and activate the specific cell with the transistor. For DRAM, the charge in the capacitor is volatile due to leakage and hence the architecture forms a volatile memory.

The basic concept of FeRAM is highly similar to a DRAM cell, just that the capacitor is replaced by a ferroelectric capacitor (Figure 2.3) [ETA14]. The ferroelectric material in the capacitor does not store a charge, but can be permanently polarized by an electric field. Hence, applying an electric field in the capacitor can store memory information in the polarization. Applying a sense pulse to the capacitor results in a different voltage depending on weather the polarization is linear or reverse. This can be measured and the stored value can be read out. In comparison to PCM, this leads to a specific behavior

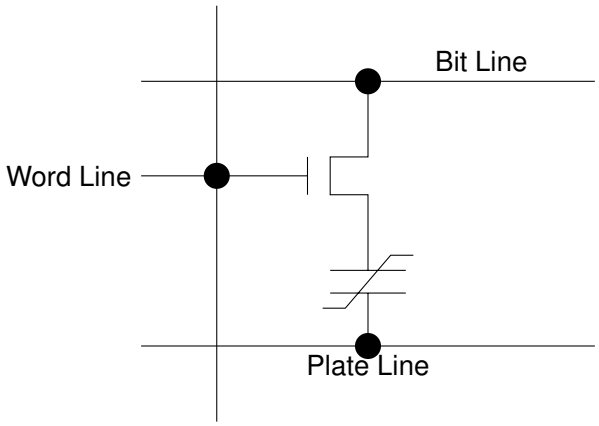


Figure 2.3: FeRAM Cell Design Example [ETA14]

of FeRAM, namely the *read destructiveness*. The sensing process of FeRAM forces a charge flow through the capacitor. If the polarization is reversal, a higher voltage appears at the bit line. This process, however, reverses the polarization of the capacitor, destroying the stored value. If the stored value should be kept, this makes a rewriting of the stored value after reading the memory cell unavoidable. Consequently, every read operation in FeRAM requires a subsequent write operation. Despite possible effects on the latency and energy characteristics, this affects the memory lifetime, when the maximal amount of possible write operations to the memory is limited. Classically, memory wear-out would be only expected for write operations, because only this operation impacts the cell state. FeRAM, however, also features read-destructiveness as well.

SPIN-TORQUE TRANSFER MAGNETORESISTIVE RAM (STTM)

Beyond material crystallization and ferroelectric charging, there exist further material properties, which can be used to permanently encode a stored memory value in a non-volatile fashion. One example for this is SPIN-TORQUE TRANSFER MAGNETORESISTIVE RAM (STTM) [BSH+17]. In this technology, a material is used, which can store a spin polarization. In addition to the storage layer, which can be programmed, a cell contains a fixed layer with constant spin polarization as a reference layer. A current flow through the spin polarized fixed layer creates a spin polarized current flow. If the free storage layer is polarized parallel, another electrical resistance is caused as if the free layer is polarized antiparallel. This enables reading out the stored value. Programming the cell is consequently done with an according programming current. If a parallel spin polarization in the free layer should be achieved, a current flow through the fixed layer into the free layer is realized. The current flow is spin polarized in the fixed layer and achieves a parallel spin polarization in the free layer. If an antiparallel spin polarization should be achieved, an inverse programming current is applied, which goes through the free layer first and through the fixed layer afterwards. The current consists of a mix of

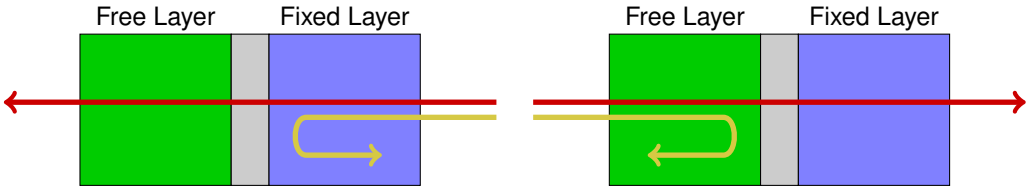


Figure 2.4: Schematic STTM Cell Overview

parallel and antiparallel spin polarized electrons to the fixed layer. The parallel polarized electrons can easily pass the fixed layer, while the antiparallel polarized electrons gather in the free layer. This achieves an antiparallel polarization of the free layer. A schematic overview of the STTM cell and the two write operations is illustrated in Figure 2.4. The current flow is indicated by two arrows, where each of them illustrates one spin polarization. It can be seen, that the fixed layer filters for one polarization and rejects the other one, which can lead to an according polarization of the free layer.

Beyond PCM, FeRAM and STTM, there are several other non-volatile memory technologies. Usually, the memory value is encoded in certain physical properties, which induces a possibly limited lifetime. However, except measurable factors, such as lifetime, energy consumption, density, unit cost and the latency, integrateability is also discussed as an argument. Producing the memory devices imposes its own challenges, especially integrating the process into COMPLEMENTARY METAL-OXIDE-SEMICONDUCTOR (CMOS) production. Consequently, compatibility with CMOS production can be another argument.

RACETRACK MEMORY (RTM)

This thesis, despite the aforementioned memory technologies, picks up another specific NVM technology, namely RACETRACK MEMORY (RTM). Basically, two types of cells could be distinguished for RTMs: Domain walls or skyrmions [BKF+20]. The special property of these cells is that charges can be transferred to adjacent cells, which allows the assembly of a nanowire of cells, where contents can be *shifted* across the positions in the wire. This further allows to place an access port only at one position of the wire and therefore reduce the complexity of the memory device. Access to the memory then requires a shift of the memory content towards the access port first. This results in a shift dependent access latency and energy consumption. The detailed implications on this model are discussed in Chapter 3. Although the question of domain walls vs. skyrmions opens interesting design choices, it is out of the scope of this thesis. The main focus of this thesis lies on the common shift property. Concrete experimental evaluation is conducted for domain wall RTM.

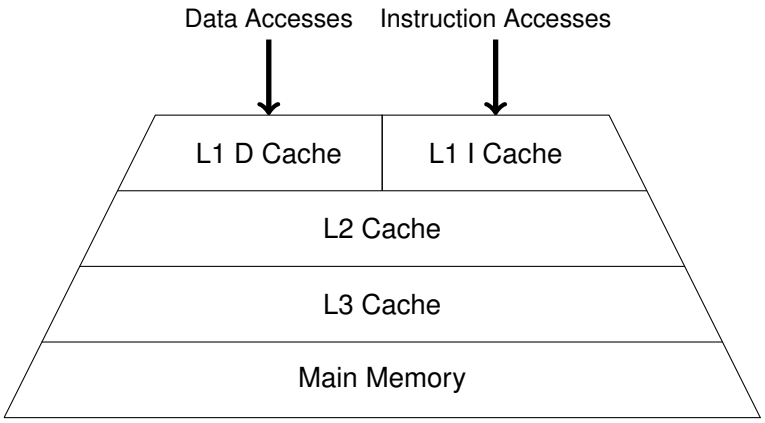


Figure 2.5: Memory Hierarchy Illustration

2.1.2 Memory Hierarchies

The basic explanations about the hierarchical organization of memory are taken from the text book [PH17]. This basic book also serves as an excellent source for further details. In the development of computer history, at a certain point memory could not simply be further scaled up to adequate sizes, while maintaining low memory access latencies. Larger memories are way more complex in their device design, which makes it infeasible to realize them with low access latencies. Also, the energy consumption for a large and fast memory would be infeasible. However, large and fast memories are well feasible with adequate energy consumption. Consequently, the straight-forward idea is to combine large, slow, but energy efficient memories with small and fast memories. Ideally, this combination is done in a way, that the application experiences a virtually large and fast memory. The key aspect, which is crucial to provide this illusion, is the local memory access behavior of applications. Applications tend to access a small working set of memory frequently. If memory management can achieve to store this working set in the small and fast memory, and exchange this working set on a behavior change of the application, the illusion of a large and fast memory works out.

The illusion of a large and fast memory is created by a hierarchical organization of different types of memory, as illustrated in Figure 2.5. Small and fast memories go to the top, large and slow memories to the bottom. Usually, the lowest layer is the main memory, storing all contents. All levels above are caches. A memory access always targets the uppermost level of the hierarchy. Only if the requested content is not found there, the next lower level is requested. This implies, that each level stores a subset of memory contents from the level underneath. Such a memory hierarchy could, in theory, be built with arbitrary amounts of levels, in current systems, four layers can be commonly found for high performance systems, whilst two or three layers for embedded processors. The management of which content to store in which level is another aspect, which requires intelligent design. For most systems, the upper levels of the memory hierarchy are organized as write back caches. That is, they keep a copy the value,

stored underneath, and do not propagate modifications to this value until the stored value needs to be evicted to gain some free space. This is advantageous, since, as long as the content is stored in a certain level, only the latencies of this level apply to accesses to that specific value. The eviction strategy, i.e. deciding which contents should not be stored any longer in a cache, when no space is left, essentially determines the policy of which contents are stored in the cache. Even though the details of eviction strategies in modern systems are usually not published by the manufacturers, some form of age-based eviction is usually applied. This means, that contents, which have not been accessed for a certain time, are more likely to be candidates for eviction.

This structure forms the most basic principle of a memory hierarchy. There are many more aspects to the management of the caches in the memory hierarchy, such as speculative prefetching or cache coherence across different CPU cores, which are largely out of the scope of this thesis. One interesting aspect, which is of certain importance to this thesis, is that many systems employ separate instruction and data caches at the upper levels of the memory hierarchy. This, up to a certain degree, contradicts the von Neumann concept, because data and instruction memory is separated. Especially when optimizing applications to efficiently perform with cache capacities and cache eviction, instruction and data memory should be distinguished for the higher levels of cache.

Memory Wall(s)

The basic problem, mitigated by the memory hierarchy, as mentioned before, is the so-called memory wall. Historically, the speed of the CPU increased faster than the memory speed, which left the memory behind and essentially left the memory as a bottleneck. One could call this also a latency wall. Although cache hierarchies seem to solve this problem by employing small and fast memories, the latency wall is only traded in exchange to a capacity wall. As long as the local working set of an application is small enough, memory hierarchies can be fast. However, if the working set of the application grows too large, memory hierarchies also reach their limits. The later in this thesis studied RANDOM FOREST (RF) application is a typical application, which can exceed the cache capacity in the local working set.

There exist further technologies, which aim to mitigate the capacity wall, namely HIGH BANDWIDTH MEMORY (HBM). However, this technology again basically exchanges the capacity wall to an energy wall, since energy consumption and heat production of HIGH BANDWIDTH MEMORYS (HBMs) can exceed feasible application in many systems. NVMs may promise to further mitigate the energy wall while maintaining extreme fast access latencies. Then, however, the energy wall is exchanged to a lifetime wall, which can limit the applicability of the NVM technology. Concluding from this, the problem of the memory speed cannot be solved unconditionally. Instead, properties can be exchanged to other disadvantages, which may be better suited for certain applications. This, however, requires explicit support and management of the different memory technologies, which is realized in a software exploited manner in this thesis.

Memory Technologies Across the Hierarchy

The aforementioned concept of a hierarchical organization of different memory types opens a design space to place various memory technologies, including NVM technologies, at different levels of the hierarchy in a native way. Some technologies may be better or worse suited for different levels. In a traditional system with volatile memory, DRAM is usually used for the main memory and the lower cache layers. The upper cache layers are realized in SRAM, which can achieve way faster access latencies. Existing systems mostly employ NVM as the main memory. One example for this is the Intel Optane Storage technology, which can be used as a replacement for the DRAM main memory [PGG19]. Another example are the FeRAM based integrated microcontrollers , e.g. the MSP430FR5994 from Texas Instruments. This system integrates FeRAM as well as a main memory. Furthermore, a combination of NVM and volatile memories could be considered, where for instance the main memory is a NVM and the upper cache levels are realized as volatile memories.

2.1.3 Computer Arithmetic

Chapter 7 focuses on arithmetical computation in the CPU and partially the impact on the memory subsystem. In order to provide an abstract background explanation, the arithmetical computations in RISC-V CPUs are picked up here as a simple example [WA17]. Although Chapter 7 focuses on X86 and ARMv8 CPUs, the principles behind the design of the arithmetic instructions are transferable. It should be noted that CPU instructions do not always follow the same scheme, but rather different format templates for different purposes exist. For instance, in RISC-V, base instructions can be either encoded in the R, I, S or U format. These formats encode different requirements for different types of instructions. For instance, the R format can encode two source registers and no immediate field, while the I format can encode only one source register and one immediate field. In RISC-V, the I format can encode a 12 bit immediate field, which allows appropriate instructions to directly store a constant value of up to 12 bits in the instruction itself.

RISC-V instructions which exploit this format are usually indicated with a trailing I in the instruction name, e.g. the ADDI, ANDI, ORI or XORI instruction. Comparing, for instance the ADDI to the ADD instruction, the ADD instruction takes two source registers and stores the computed value in the destination register. The ADDI instruction takes one source register, adds the immediate value and stores the result in the destination register.

Unlike RISC-V, ARMv8 and X86 also apply immediate encoding to control flow instructions. That means, conditional branches, which are realized as a pair of a compare instruction and a branch instruction, can compare one register to an immediate field and do the branch accordingly. This again allows a trade-off between loading a comparison value from memory or directly store it inside the instruction in the instruction memory. It should be noted that the limitation, that the immediate field has only 12

bits, only allows immediate encoding of numbers, not exceeding this length. If a larger number should be encoded as an immediate constant, it is unavoidable to load the number in extra steps.

While the processing of integer numbers (including comparisons for greater or less relations and conditional branches) allows the aforementioned trade-off of immediate encoding or memory encoding, floating-point numbers do not allow this trade-off. While integer numbers can be easily cropped down to 12 bits, since the uppermost bits are only sign extended, floating-point numbers employ discrete fields of a sign bit, an exponent number and a mantissa, which cannot be simply cropped to an arbitrary amount of bits. Consequently, neither RISC-V, nor ARMv8 or X86 provide any means to encode floating-point numbers in immediate fields. Consequently, the trade-off between memory and immediate encoding of constants is not available for floating-point numbers.

2.1.4 RF Ensembles

Chapter 6 and Chapter 7 study RF ensembles as a premier specific application. Execution and tooling of RFs and a brief overview about the general working principle is given here. The basic explanations are adopted from [Bre01] and [BFO+]. RFs are machine-learning models, which are either trained for a classification or regression purpose on a given dataset. In order to provide intuition, consider a very simple data-set. The data-set has two features and one prediction column. The data-set simulates a collection of recordings, which restaurant on the campus a virtual work group visited. The features are the current weekday and the existence of fries in the student lunch plan. The prediction is the visited restaurant:

Weekday	**Fries?**	**Restaurant**	...	...	...
Monday	yes	Mensa	Monday	yes	Mensa
Tuesday	yes	Mensa	Tuesday	yes	Mensa
Wednesday	no	Mensa	Wednesday	no	Mensa
Thursday	no	Galerie	Thursday	yes	Mensa
Friday	yes	Food Fakultät	Friday	yes	Food Fakultät
Monday	yes	Mensa	Monday	yes	Mensa
Tuesday	yes	Mensa	Tuesday	no	Galerie
Wednesday	no	Mensa	Wednesday	yes	Mensa
Thursday	yes	Mensa	Thursday	no	Galerie
Friday	yes	Food Fakultät	Friday	no	Food Fakultät

Table 2.2: Demonstration Training Datset

If now a random forest should be trained to predict the decision of the virtual work group, which restaurant to visit, the training process has to be reduced to training of single DTs first. This could be done, for instance, by random sampling the data-set and train each one DT on a subset. For simplicity, only a single DT should be trained on the

entire given data set. If multiple DTs exist in an ensemble, the prediction of the single trees is usually combined by, e.g. a majority vote for classification problems. If only a single tree exists, the prediction of this tree is directly applied. The training of a single tree is achieved by recursively splitting the data set into partitions, which can be well distinguished by a threshold value of a feature. To determine such a well-suited split criterion, impurity metrics can be used. The impurity is basically an indicator for the mix of prediction classes in a data partition. If the impurity is high, many classes are mixed in a partition. If the impurity is low, the partition is very pure and only contains a small amount of prediction classes. One example for an impurity metric is the gini impurity, which is computed according to the following equation [BFO+]:

$$\sum_{j=1}^{J} p(j) \cdot (1 - p(j)) \tag{2.1}$$

In this equation, J is the number of classes in the analyzed data partition and $p(j)$ is the relative frequency of the appearance of this class in the data partition. The CLASSIFICATION AND REGRESSION TREES (CART) training algorithm [BFO+] basically tries different split criterions by either doing full explorative search or random sampling and determines the impurity score of the resulting partitions. The split with the lowest impurity in the resulting partitions is then taken as the applied split. In this example, a split with a high reduction in impurity would be to filter the weekday by Friday and not Friday. For the outcome of true, the resulting partition has only one class and therefore minimal impurity.

Consequently, the root node of the decision is formed by the criterion `Weekday == Friday`. The training further continues the recursively on two data sets, one where the weekday is Friday and one where the weekday is not Friday. The data set with the weekday set to Friday only contains one prediction class, namely Food Fakultät. Therefore, no further splitting is needed, because the impurity cannot be further lowered. This is a stop criterion, which leads to the creation of a leaf node. This leaf node predicts the outcome Food Fakultät. The other data partition, is further analyzed for possible good splits. Splitting the remaining data set by the weekday to be Wednesday or not Wednesday also results in one entirely pure partition. The remaining data partition than can be split on the feature of the existence of fries and results in two entirely pure data partitions. This then forms the final DT (Figure 2.6). In the illustration, split nodes are green, and leaf nodes are blue. If the ensemble had been created with multiple decision trees, the process would be similar for the single trees. The predictions of the single trees would be combined by, for instance, a majority vote. It should be noted that in this example, for the purpose of intuitive illustration, equality split criteria are applied to the features. If a general data set is given, however, features are numerically encoded and the split criterion normally is done by a $\leq$ operation with the threshold value.

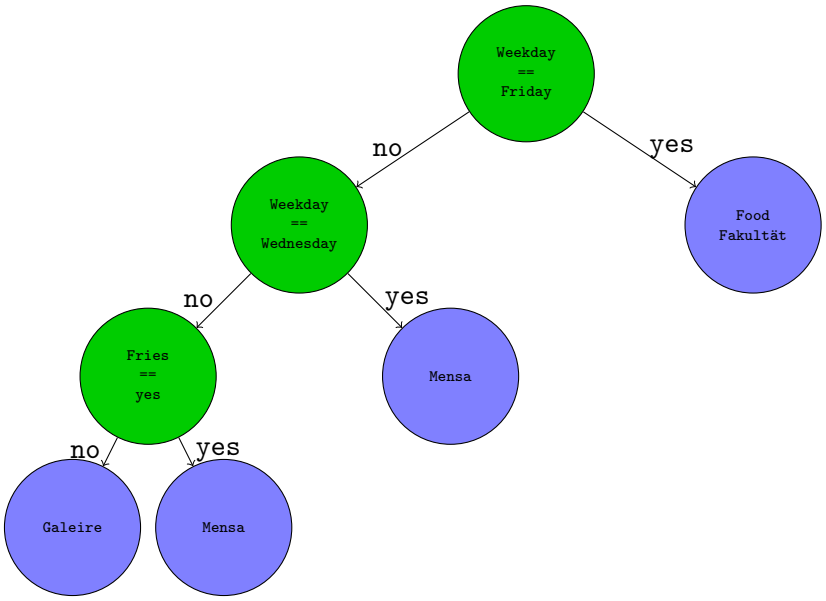


Figure 2.6: Demo DT

2.2 Related Work

On an abstract level, the methods detailed in this thesis can be roughly categorized in two categories: 1) NVM management and 2) RF performance optimization. The management of NVM needs to be further divided into common wear-leveling and special optimization of the properties of RTM. All of these fields experience active research interest, which makes capturing the state-of-the art difficult. This section aims to provide a comprehensive overview about research efforts in the related work.

2.2.1 NVM Wear-Leveling

Over the last decades, several approaches for in-memory wear-leveling for NVM are proposed. Independent of the exact design principle, the target of memory accesses is changed during the process of wear-leveling, in order to achieve a more lifetime-friendly memory access scheme in consequence. The decision, which memory access is redirected to which memory location, is usually based on certain strategies. These strategies can be roughly categorized in aging aware strategies, i.e. management strategies which take the current memory aging, or an estimation of the current aging into account and non aging-aware strategies. [LSX+19; DZH+11; LWW+13; HDW+16; AXY+14; SNM+15; CHK+12; FZB+10; ZZY+09; GWD+19] can be considered to be aging aware. The detailed design principle behind these works spans a wide space. Approaches like [LSX+19; SNM+15] propose wear resistant memory allocation schemes, where the allocated portions of memory are placed to different memory locations, depending on the current wear-out. The information about the memory usage can be

partially extracted from the allocation requests itself. This forms a less invasive method, since the majority of wear management happens inside the memory allocation. Other approaches, which target a management of memory wear more close to the hardware [AXY+14; CHK+12; FZB+10; ZZY+09] employ various schemes of hardware controllers to redirect memory accesses on the basis of certain sized memory blocks. Also, the management, which block to choose as a target for a relocation spans a wide space. Most of the methods, however, have in common that frequent memory accesses target less worn-out physical memory locations in order to distribute the wear, caused by these accesses. Many approaches consider the number of memory accesses as a central indicator for the memory aging. However, [DZH+11; HDW+16] consider the wear-rate, i.e. the frequency of memory accesses as an indicator as well. Memory is assumed to wear-out more from accesses with a short time in between than from accesses with a large time distance.

A counterpart to the previously mentioned aging-aware approaches is formed by non aging-aware approaches, where neither the precise memory age, nor an estimated memory age is taken into account for the making of wear-leveling decisions. Towards this, randomized approaches [QKF+09; FZB+10; ZZY+09] apply wear-leveling in a circular or randomized manner. Memory accesses are redirected to adjacent memory locations in a circular fashion over time, essentially spreading the memory wear-out across the memory space. Non aging-aware and aging-aware wear-leveling can be combined to achieve a randomized wear-leveling on fine granularities inside of memory blocks, while an aging-aware approach is used to target these coarse-grained memory blocks. Towards, this different approaches in the literature target different granularities for the memory management. It varies from single bits [CL09; ZSY+14] over cache-lines [QKF+09; ZZY+09] for fine-grained approaches to memory pages [FZB+10; AXY+14; SNM+15; CHK+12; GWD+19] or even bigger memory segments [ZZY+09; ZL09] for coarse-grained approaches.

Memory Tracing

One of the core functionalities for aging-aware wear-leveling algorithms is the tracing of memory accesses in order to model and estimate the current memory wear-out. Work in the literature proposes many ways to analyze the memory access behavior of computer applications. Capturing of memory accesses is proposed to be done on various levels. Bao et al. propose HMTT [BCR+08], which is a hardware based memory simulator that uses an FPGA between the processor and the memory DIMM to snoop all memory accesses. A less invasive method is provided by the combination of the gem5 full-system simulator [BBB+11] and the non-volatile memory simulator NVMain2.0 [PZX15]; this setup simulates an entire system, including CPU, memory and peripherals. The simulator then allows users to trace all memory accesses. Nethercote et al. propose Valgrind [NS07], which is a method that hooks in at the application level and requires code instrumentation such that every memory access can be traced out.

The aforementioned mechanisms are subsequently used to analyze applications memory behavior. Jiang et al. focus on a memory analysis of computing frameworks (e.g. Hadoop) by investigating hardware characteristics (e.g. cache behavior) [JZH+14]. They employ Intel's VTune Amplifier [Amp19] to collect architectural metrics, but also conduct their analysis on the HMTT platform to acquire the full memory trace. Nalli et al. present a benchmark suite with various applications [NHH+17], using the gem5 simulator to analyze memory traces and to investigate the memory behavior of these applications. Byma et al. perform an analysis of the usage of heap allocated memory [BL18] by instrumenting the code, similarly as Valgrind [NS07] does. Consequently, they analyze the memory behavior of their application based on these data.

Software Based Wear-Leveling

Studying the problem of wear-leveling towards the scope of this thesis, software exploitation of existing hardware for wear-leveling is of a special interest. Gogte et al. propose a software-only coarse-grained wear-leveling approach by using a sampled approximation of the write distribution [GWD+19]. They make use of advanced debugging capabilities, i.e. Intel Processor Event Based Sampling (PEBS), which allows them to sample the write requests from the CPU. These debugging capabilities, however, can rarely be found in embedded systems and resource-constrained hardware.

All other mentioned aging-aware approaches rely on the current write-count information of the memory. Most approaches introduce specialized hardware into the memory controller to collect the write-count information, which is not available in commonly available systems and might be hard to realize. Dong et al. [DZH+11] use an offline recorded memory trace to estimate the write distribution, which limits the approach to a subset of well-known applications only.

Stack Memory Wear-Leveling

Specialization in the process of wear-leveling can help to derive efficient and tailored methods. Stack memory is a desirable candidate for specifically optimized solutions, since a high volume of memory accesses targets the stack memory. In addition, the patterns of accesses to the stack memory are indirectly controlled by the management of stack memory from the compiler. This offers opportunities for software-based wear-leveling to specifically end efficiently target stack memory. Specific software-based wear-leveling approaches consider specialized solutions for the stack memory in their work, i.e., a memory allocation is performed for the stack of each function call [LSX+19; LHC+14]. They also mitigate the caused overhead for stack wear-leveling by only hooking in the wear-leveling algorithm for new memory requests. Thus, copying of old content is omitted. However, these approaches rely on the application's corporation to, for instance, perform a sufficient amount of function calls.

Read Wear-Leveling

As most memory technologies only wear out from write accesses, many proposed methods in the related work also only target write accesses to the memory as a premier target for the wear-out analysis. However, FeRAM as a produced and market-ready technology features also read wear-out. There are no dedicated algorithms or methods for read wear-leveling in read-destructive NVMs. However, hardware-based approaches that are not aging-aware or that directly decide based on the wear of each cell are compatible with read-destructive memories by default. If the wear is estimated from the write count, it could be also estimated from the read and write count together. This implies that hardware-software interplay algorithms can obtain the accurate wear estimation by extending the hardware to count read accesses as well. As long as generic mechanisms (e.g. virtual-memory page remapping) are used [AXY+14; CHK+12], the modifications to the algorithms are minimal. When in contrast specific mechanisms (e.g. heap allocation or stack allocation) are used for wear-leveling [LHC+14; SNM+15; LSX+19], then read wear-leveling cannot be integrated easily. Thus, another special mechanism for read wear-leveling is required in those cases. Also, for algorithms that ship with their own write approximation [HMH20; GWD+19], a specialized read approximation has to be added.

Database-Specific NVM Management

Application specific NVM management can offer a huge potential for effective and efficient management. Especially when the application can be highly customized, NVM management can be directly integrated into the application implementation. Parts of database management systems, in this context, offer such a highly customizable application. The use of NVM for indexes in databases is extensively studied in the last years. However, most work is dedicated to exploit features such as byte addressability. The problem of higher latency compared to DRAM, especially during writes, is thereby addressed by many researchers in this effort, leading to new index structures that mainly try to avoid unnecessary writes to NVM. Many of these approaches attempt to reduce data movement within leaf nodes. Some of them allow unsorted leaf nodes and maintain additional helper structures to improve the performance of search operations [CJ15; YWW+15; OLN+16]. Other approaches enhance search operations by dividing leaf nodes in cacheline-sized chunks that are sorted internally, even if the node itself is not sorted [WC20]. Still other approaches allow unsorted inserts into leaf nodes, whilst the nodes are sorted occasionally [ALM+18].

While writing to NVM is identified as a critical problem in almost all the approaches mentioned above, the focus is primarily on reduced write performance rather than wear-leveling. Chi et al. [CLX15] propose a cost model for B^+ trees in their work that estimates wear out of nodes, but they do not focus on wear-leveling.

2.2.2 RTM Optimization

One special type of NVM is RTM, which is picked up as a premier target in the second half of this thesis. For RTM, the issue of lifetime management is less interesting as for other types of memory, RTM, however, comes with a unique property, requiring dedicated management. RTM features access-dependent latency and energy consumption, which opens a wide range for optimization. A rich body of research explores the efficient employment of RTM at various levels in the memory hierarchy for numerous application domains and system setups. In this context, optimization techniques for RTM are proposed to facilitate their adoption in the register file [Ato15; MXM+16; MWZ+14], scratchpads [KHB+19; KRH+19; MZS+15], caches [XAM+16; VKS+16; XLM+15; ZSZ+15a; SBW+16; MIG14; SBJ+14], network-on-chip [KXM+15], off-chip memory [HSS+16], and solid state drives [PYL+14]. Therefore, RTM can be fitted in all levels of the memory hierarchy, making it a promising candidate for universal memory.

To provide performance, area, and energy benefits, various optimizations are proposed in the literature at cell-level [SBW+16], circuit-level [MIG14], layout-level [SBJ+14; ZSZ+15a; MIG15], and cross-level [SZL+15]. RTM's leakage power and capacity advantages give it a competitive edge over existing memory technologies, but the expensive shift operations present a daunting challenge. In this context, various techniques for RTM shift cost reduction are proposed, such as runtime data swapping [SWL13; VKS+16; SBW+16], data compression [RRV+15; XLM+15], preshifting [CPA+19; Ato15], access port management [VKS+16; SBW+16], intelligent instruction [MJK+19], and data placement [KHB+19; CSZ+16]. For data placement, Chen et al. in [CSZ+16] present a heuristic appending data objects according to the adjacency information sequentially. Khan et al. in [KHB+19] formulate the data placement problem with an integer linear programming and further propose the ShiftsReduce heuristic to enhance the previous heuristic by introducing a tie-breaking scheme and a two-directional objects grouping mechanism assuming a single access port RTMs.

It is shown that domain-specific approaches not only guarantee better performance and energy consumption but also enable better predictability of the runtime [KRH+19]. In fact, the studied data placement problem can be treated as an instance of the quadratic assignment problem (QAP), which was introduced in 1957 [KB57], considering the problem of allocating a set of facilities to a set of locations. When the facilities are all in a line (like the locations within in a DBC), such a special case is named the *linear ordering/arrangement* problem [BÇP+98]. Suppose that the number of vertices is m and the length of an edge is defined as the linear distance between the vertices involved. For undirected trees, Shiloach proposes an $\mathcal{O}(m^{2.2})$ algorithm [Shi79]. For directed trees, Adolphson and Hu in [AH73] present an algorithm to derive an optimal placement in $\mathcal{O}(m \log m)$.

The imperfection in the fabrication technologies and fluctuation in the current density required for the shift operation may cause pinning faults and position errors in RTMs. Many position error detection and correction schemes are proposed to guard RTMs against such errors and improve their reliability [ZSZ+15b; OJR+19; VMS+17].

2.2.3 RF Performance Optimization

The second part of this thesis discusses efficient execution of RFs on modern technologies. Despite RTM, performance considerations for other system properties are being made. Towards modern hardware technologies, several techniques are proposed in the literature to speed up the execution of inference for tree-based ensembles. For binary search trees, Kim et al. in [KCS+10] present an optimized realization by using vectorization units on X86, considering the register sizes, cache sizes, and page sizes specifically. However, such a technique requires a specific support from the underlying architectures. The concept of vectoring the tree structures is also applied to the context of ranking models in [LPN+16], which enhances the QuickScorer algorithm for gradient boosted trees [LNO+15; DLN+16]. Ye et al. in [YZZ+18] further improve the scalability of such vectorization methods by encoding the node representation to compact the memory footprint. These techniques decompose the tree-ensembles into different data structures based on the feature values, which is especially effective for large ensembles of smaller trees. Without executing trees one by one, however, the target applications are mainly limited to batch-processing. Hummingbird is proposed to compile decision trees into a small set of tensor operations and batch tensors for each tree together for tree inference [NSY+20].

Architecture-aware implementations for decision trees start from [VMG+12], which optimizes the implementations of decision trees on different architectures, i.e., CPUs, FPGAs, and GPUs. By fixing the tree-depth, Prenger et al. in [PCM+13] further show an effective pipelining approach over these computing units, based on the CATE algorithm during training. However, the impact of cache misses is not taken into account. The two common implementations for decision trees, i.e., native trees and if-else trees, are first distinguished in [ALD13], which provides the first attempt to increase data locality for native trees. By leveraging the probability model of accessing nodes during tree inference in [BM18], Buschjäger et al. in [BCC+18] propose several optimizations for memory layout over different tree implementations to improve the memory locality and show the potential speed-ups can be up to 2-4x over different architectures. Chen et al. enhance this method further by compiler-based binary size estimation [CSH+22].

RF Floating-Point Arithmetic

Considering floating-point arithmetic and the relation to two's complement integer arithmetic, several starting points can be found in the literature. It is reported that some CPUs internally use the same hardware unit for floating-point and integer comparisons with a few additions for the floating-point computation [Bra]. Furthermore, there are explicit considerations about the binary floating-point format regarding accuracy and efficient programming [DDR95; Bli97].

CHAPTER 3

System Model

Contents

3.1 Non-Volatile Memory Model 36
3.1.1 Technical Overview 36
3.1.2 Wear-Out Model 38
3.1.3 Iterative Memory Writes 39
3.1.4 Hybrid Memories 40
3.1.5 Simulation Setup 41
3.1.6 Latency Model for RACETRACK MEMORY (RTM) 43
3.2 Random Forest Execution Model 44
3.2.1 Probabilistic Execution Model 45
3.2.2 Implementation 46
3.2.3 Arithmetic Considerations 48
3.2.4 Performance Consideration 48
3.2.5 Tooling 49
3.3 CPU Model 50
3.3.1 Memory Hierarchy 50
3.3.2 Floating-Point Arithmetic 51

3.1 Non-Volatile Memory Model

The term NON-VOLATILE MEMORY (NVM) covers a large variety of memory technologies and memory architectures. This section gives a general overview about the technologies, which belong to this term and are relevant to this thesis, and introduces the relevant key aspects.

3.1.1 Technical Overview

The term NVM itself refers to memories, that persist their value over a longer time, when the memory is powered off. This also includes, for instance, hard disks or tape drives. However, this thesis studies a more specialized subclass, which is *non-volatile main memories* or *storage class memories*. This, in fact, refers to memory technologies, which are non-volatile, but also byte-addressable and usable as a main memory technology. In this technology class, hard disks are for instance not included since they are block based and not byte addressable.

In the current state of the art, several technologies are proposed and discussed for the use as non-volatile main memory. Examples for such technologies are PHASE CHANGE MEMORY (PCM) [LW08; BRC+17], RESISTIVE RAM (ReRAM) [BRC+17], FERROELECTRIC RAM (FeRAM) [GMS+21], SPIN-TORQUE TRANSFER MAGNETORESISTIVE RAM (STTM) [BSH+17] or RACETRACK MEMORY (RTM) [BKF+20]. In comparison to classic DRAM, all these technologies make use of physical properties of a certain material in order to store data in a persistent manner. While DRAM stores the data in an electric field in a capacitor, which is volatile due to electric leakage, PCM for instance encodes information by bringing a material into a crystalline or amorphous state, which impacts the electric resistivity of the material. Another example is STTM, where information is encoded by spin polarizing a free layer. This is achieved by applying a spin polarized current to this layer. This spin polarization again causes a different electric resistivity when a read current is applied. It has to be noted that NVM technologies similarly face leakage or drift effects, which change the encoded value of memory cells when they are not written for a certain time. However, while a value in DRAM can lose the encoded information within a time range of seconds, for most NVM technologies information persists for the time range of years when contents are not overwritten. It should be further noted that the aforementioned technologies usually only refer to a conceptual realization of the memory cell technology itself. The realization of the memory device, i.e. how to organize single cells to a full memory device, is another question which can take another huge impact of the performance characteristics of the memories.

When discussing NVM technologies the following characteristics are important to consider and are explained in the following:

- **Endurance**: Since the information in NVM is encoded by almost permanent physical changes of a material, way higher stress on the material is caused in comparison to DRAM. This results in a limited lifetime of memory cells for many

technologies. The lifetime of DRAM cells is limited but usually in such a large scale that the lifetime of DRAM is practically considered as infinite. For many NVM technologies, the lifetime of cells is limited in such a scale, that realistic workloads can destroy memory cells within a few years or even shorter periods [BRC+17]. In this thesis, the term of endurance is used to refer to the lifetime of the memory cells in a certain technology. Under a certain assumption that every access causes the same physical stress on a cell, the endurance can be expressed as a number of accesses the cell can endure before being destroyed. Although endurance may be measurable in experiments for certain memory technologies, the endurance should not be considered as a hard boundary. Effects, such as processing variation, can impact the endurance and even cause slight differences in the endurance within one memory chip [ZJZ+14]. Hence, when the endurance of memory cells in general is limited, the system should manage to cause in total a low number of access to each memory cell in order to prevent failing of single cells.

- **Lifetime**: Directly related to the term of endurance, the lifetime of a memory chip can be discussed. The endurance is a central driver for the lifetime, since this effect causes single memory cells to become unusable. However, it strongly depends on the chip architecture and management strategies how the total lifetime of a chip can be modeled. If, for instance, a chip, a driver, an operating system or an application implements tolerance mechanisms to detect failing cells and exclude these from usage, the lifetime of a chip can go beyond the failing of the first cell.
- **Retention**: As previously explained, also NVM is affected by leakage or drift effects, which cause the values stored in a cell to change over time when no further write operation is applied to the cell. The time, a value can be expected to be available in a memory cell is referred to as the retention time. As already mentioned, the retention time for NVM is usually considered ranging in the scope of years while for DRAM it is at most a few seconds. For some NVM technologies, retention time is not necessarily fixed or determined by the technology. In PCM, for instance, the length of the write pulse determines also the retention time of the written value. This basically can offer a trade-off between retention time and time/energy/endurance consumption for write operations. It is also worth mentioning that the drift effects are asymmetric for most technologies. In detail, a written value of 1 may drift towards 0 over time, but a 0 is always persistent[1].
- **Density**: Density refers to the potential realization of memory devices, i.e. the integration density. Many NVM technologies promise to achieve a high integration density, allowing low unit cost with large capacities. Although market ready devices with NVM are not very popular for large capacities at the moment, the possible density opens discussions for the omission of larger capacity storage beyond the NVM main memory at all.

[1] The state of 1 and 0 is only a logic interpretation of the cell state, it can also be flipped.

- **Latency**: Although the materials used for NVM technologies usually do not feature special properties impacting the required time to read them out and write them, several concepts for NVM have a certain effect on the access latency. As mentioned before, retention may be controlled by application of different length write pulses. This essentially also impacts the write latency of the corresponding operation. Furthermore, RTM as a special technology features varying latencies based on the state of the memory, since memory contents are organized in a nanowire with a central access port. Furthermore, the chip design could include various buffers which also can cause a non-uniform access latency.

3.1.2 Wear-Out Model

All methods presented in this thesis in regard to the lifetime of NVM study basic approaches to use traditional memory interfaces to execute required wear-leveling operations in a software-driven manner. Consequently, a generic model for the memory lifetime and the wear-out is chosen. Naturally, other methods can be deployed which operate on other assumptions as the ones made in this thesis and further improve the lifetime. This thesis, however, aims to make as generic assumptions as possible and provide a generic basic solution to the lifetime issue in a software-based manner by exploiting traditional interfaces.

The amount of write operations a cell can endure is limited in a scale of 10^8 operations for some technologies [BRC+17]. Assuming that software writes a certain memory portion only once every second, this memory portion would be worn out after ≈ 3 years. If the software applies more frequent writes, the cells are destroyed even faster. In this thesis, no additional error detection mechanisms are considered, thus the entire memory becomes unusable once the first cell wears-out. This potentially limits the lifetime of NVM equipped systems to a few years or even less. This points out the urgent need for wear-leveling in such systems, i.e. the process of distributing write operations to all memory cells approximately equally in order to reach a drastically improved system lifetime.

Before precisely defining wear-out measures, NVM technologies further have to be distinguished in write-destructive and read-destructive memories. For some technologies, the read operation does not impact the cell value and therefore causes no or a neglectable impact to the cell endurance. For other technologies, such as FeRAM, the read operation entirely drains the memory cell, which makes a subsequent write operation mandatory in order to keep the stored value [Phi96]. This essentially makes read operations destructive as well, since each read operation results in a write operation. For wear-leveling in a hardware-aware fashion this may require no special care, because the corresponding write operation becomes visible in the hardware level, for wear-leveling on the software level, in contrast, it requires employing wear-leveling means for logical read and write accesses.

Laying out the basics about the wear-out assumptions allows to subsequently formalize the memory wear-out. This thesis distinguishes between read-destructive and

write-destructive NVM. For write-destructive memories, write accesses are assumed to wear-out their targeted memory cells equally. For read-destructive memories, read and write accesses are assumed to wear-out their targeted memory cells equally. This thesis does not make assumptions about absolute lifetimes of NVM technologies and hence evaluates wear-out in a relative manner. Under the assumption that the memory becomes unusable once the first cell wears out, the idealized way of achieving maximal memory lifetime is to shuffle all memory accesses of the program such that all memory cells face the same amount of accesses, i.e. the mean amount of accesses. Comparing this value to the most heavily-stressed cell (practically determining the lifetime of the device) results in a relative measure of how *good* in terms of lifetime achievement a certain memory usage is. This value is further referred to as *achieved endurance (AE*):

$$AE = \frac{\text{mean\_access\_count}}{\text{max\_access\_count}} \tag{3.1}$$

Low values of the achieved endurance motivate the use of wear-leveling methods, which modify software in a way such that the distribution of memory accesses is modified towards better endurance achievement. Given the memory access distribution without wear-leveling as a baseline, the relative effect of wear-leveling methods can be determined by comparing both values of achieved endurance, which is referred to as *endurance improvement (EI)* in the following:

$$EI = \frac{AE_{\text{analyzed}}}{AE_{\text{baseline}}} \tag{3.2}$$

Although the EI provides a relative measure of how well memory wear-out is distributed, it does not compare the absolute stress in the memory. For instance, after wear-leveling the AE could be $4\times$ better distributed, but also cause $4\times$ the absolute stress on the memory, effectively gaining no benefit. In order to include this consideration, the absolute overhead (percentage of the additional total number of destructive accesses), denoted as OV, is included, resulting in the measure of *lifetime improvement (LI)*:

$$LI = \frac{EI}{OV + 1} \tag{3.3}$$

It should be noted that this overhead measure does not directly relate to additional execution time overheads. Such overheads, however, can be measured separately and do not impact the considerations about relative memory lifetime.

3.1.3 Iterative Memory Writes

Section 3.1.2 defines wear-out measures based on the term of write accesses, or in addition read accesses for read-destructive memories. In the simplest form of a realization, write accesses to the memory are triggered directly by the CPU in the form of a store operation to the memory. The possibility of layers of caches between the CPU and the NVM is not further discussed in this thesis, since this is not often

present in the considered target class of embedded systems. However, whenever a store request reaches the NVM, there are two possible ways to realize the execution: 1) blind application of a write pulse to the memory and 2) iterative sense and update procedures. The first version requires less overhead and can be realized in a simple manner. Independent of the previously stored memory content, memory cells are overwritten. For the wear-out model, CPU store requests can be considered as NVM writes.

For iterative sense and update procedures [QFJ+; ZZY+09], in contrast, the stored memory cell value is iteratively sensed, and smaller update pulses are applied to the NVM cell. This implies, that whenever the CPU requests a store operation with the same content as already stored, no write operation to the memory cell occurs and no wear-out is caused. In this thesis, SINGLE LEVEL CELL (SLC) memories are considered, i.e., every bit in the memory space is stored in a separate memory cell. Under this assumption, NVM write accesses can be extracted from the CPU store requests by knowing the previously stored content: A cell requires a write operation, whenever the stored bit in the cell has to be flipped. It is furthermore assumed that memory cells face the same wear-out from set and reset operations, i.e., bit flips from 1 to 0 and 0 to 1. In consequence, the memory wear-out then is determined by the total sum of flips per memory bit. The metrics in Section 3.1.2 can be applied to CPU write requests for non-iterative memories, as well as to the sum of bit flips for iterative memories. In this thesis, these metrics are denoted as $LI_{iterative}, EI_{iterative}$ and so on.

3.1.4 Hybrid Memories

When it comes to the realization of systems with NVM, one possibility in addition to realizing the entire main memory with NVM is to equip the system with multiple types of memory. For instance, NVM and volatile memory can be realized side by side. It could, however, also be considered to equip a system with different NVM types, e.g. different technologies or different realizations with different latencies. Such hybrid memory systems can offer different properties that can potentially be exploited by the software towards the ideal combination of memories.

From a technical perspective, the simplest realization of hybrid memories is to map the different memory types to different addresses in the physical address space. Examples for such systems are the FeRAM based microcontrollers from Texas Instruments, which are equipped with FeRAM and SRAM, and the Optane DC Persistent Memory technology from Intel [IYZ+19], which can be applied together with DRAM in a system. If software should not gain control over the hybrid memory architecture, concepts like using one type of memory as a cache for another type of memory can be realized as well. When systems with NVM and volatile memory are available, and can be exclusively accessed by the software, one strategy that can be employed is checkpointing. With this strategy, a working copy in the volatile memory would be modified and regularly written to the NVM to ensure persistence.

3.1.5 Simulation Setup

In order to analyze the aforementioned information about memory wear-out, the knowledge about the amount of accesses to single memory locations (i.e. memory cells) is crucial. As this information is not available in a normal computer during usual execution, several techniques exist to derive such information. Such techniques range from code instrumentation, over special tracing hardware up to full system simulations (Section 2.2.1). In this thesis, full system simulation is used as the main source for collecting memory access information. For all presented methods in this thesis, a modified framework is used for the full system simulations, which is detailed as follows:

As a basis for the simulation framework, the full system simulator gem5 [BBB+11] is used in combination with the NVM simulator NVMain2.0 [PZX15]. Gem5 is configured to simulate an ARMv8 64-bit CPU with according peripheral units. The simulated CPU implementation of gem5 is chosen as the DerivO3 CPU with pipelining and out-of-order execution. The machine is chosen as the VExpress_GEM5_V2 machine. NVMain2.0 is integrated into the implementation of gem5, such that memory requests to the main memory are forwarded to NVMain2.0 as a memory device. NVMain2.0 then simulates NVM-specific properties and produces memory access traces of the performed memory requests. These traces can be used to extract the sum of write requests, as well as to extract the sum of flips per bit. Hence, the trace can be used in a lightweight post-processing manner in order to determine the memory wear-out for non-iterative memories and for iterative memories.

Gem5 is configured to run in the full system simulation mode, i.e. it simulates a bare metal system without any operating system or library support. Hence, the simulation setup delivers this component as well. Basically, there are two exchangeable components, which can be used as part of the operating system layer, which is booted bare metal in gem5:

1. A bare metal custom **runtime system**, which is an extreme lightweight software layer, only including required boot code and simple device driver implementations. This runtime system is used in [HCY+20].
2. The library based unikernel unikraft [KSV+19b], which is a configurable bare metal operating system, shipping with many standard components, such as standard libraries or file system implementations. The advantage of the unikernel is that these components can be selected or deselected during compilation and hence, a custom binary image can be created. Unikraft is ported to support gem5 and NVMain2.0 and provided as a modified version with the simulation setup in [HCK+20].

Figure 3.1 gives an overview of the simulation setup, including the various components and the interplay between the layers. In the place of the runtime system, either the bare metal lightweight runtime system can be used or the modified instance of unikraft can be used.

published work: [HCY+20]

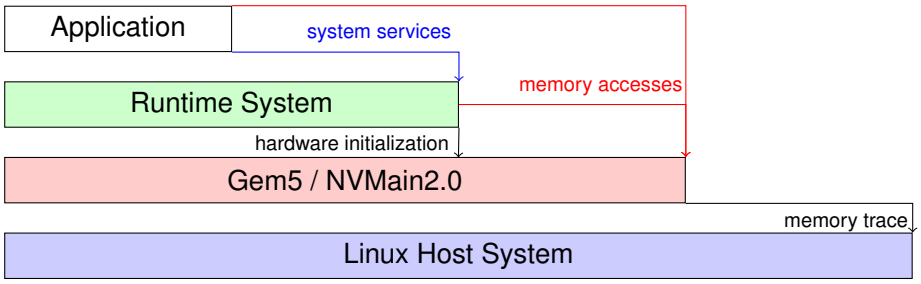


Figure 3.1: Overview of the Simulation Setup

Isolation

Using the simulation setup, described before, could execute an application and derive a memory trace from the application together with the runtime system, which can be directly used to compute wear-out measures. In many scenarios, however, it is of particular interest to investigate the behavior of a single application, e.g. a specific benchmark application only, without the interaction from the runtime system. In order to accommodate for this, two isolation mechanisms are integrated into the simulation setup, which separate the memory space of the application from the runtime system, such that the memory behavior of the application can be analyzed in separation. The first isolation mechanism is spatial isolation during linking. In greater detail, the build process is modified in such a way, that the text, data and bss segment of the application relevant code are placed to separate memory locations. Providing the boundaries of these allocated segments, they can be clearly identified in the memory trace[2]. This spatial isolation does not provide isolation for the stack segment, since the stack is used by function calls., which can be deeply nested and interleaved, when the application makes calls to components of the runtime system. To accommodate for this, interrupt isolation is provided in the simulation setup by executing the runtime system on a higher privileged execution level of the CPU than the application. The ARMv8 CPU supports an automatic change of when jumping privilege levels. Calls to the runtime system consequently have to be capsuled in hardware assisted system calls. In consequence, functions from the runtime system use another stack region than functions from the application. If the boundaries for both stack regions are known again at analysis time, the memory trace can be filtered accordingly to achieve fully isolated analysis of the application. Figure 3.2 illustrates a memory layout, which can be derived after applying both isolation mechanisms. Since also the available memory space for the stack is allocated by the linker, all segment boundaries are known at compile time. After the simulations are executed, a memory access trace can be derived by NVMain2.0 and filtered by the corresponding segment boundaries. This then allows for separate and

[2] It should be noted that the memory trace from NVMain2.0 denotes physical memory locations, while the linked addresses are virtual addresses, which can be translated by the MMU. In order for the isolation to work, the MMU needs to apply an identity mapping or the mapping needs to be known at analysis time.

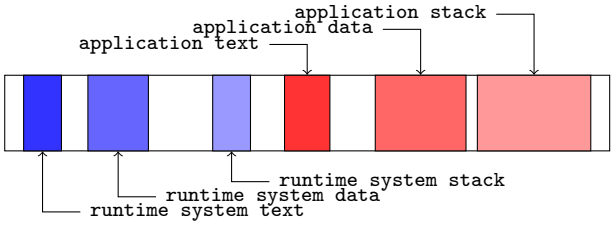


Figure 3.2: Exemplary Memory Layout for Isolation

isolated analysis of a specific application or even of single memory segments of a single application.

Based on the derived memory access information from this setup, post-processing can be used to simulate various aspects of NVM. For instance, in order to study the impact on the lifetime of iterative write scheme memories, bit flips between the old and new content in the memory can be determined. For the analysis of hybrid memory systems, the traces can be split at specific address boundaries and used in different post-processing steps.

3.1.6 Latency Model for RACETRACK MEMORY (RTM)

As highlighted in Section 3.1.1, in addition to the lifetime properties, also the access latency of NVMs can have a significantly different shape than classical DRAM. Especially RTM features a specific access-dependent latency model, which is briefly introduced in this subsection. RTM is a magnetic tunnel junction memory type, where single bits are physically stored in the form of magnetic orientation in small regions [BKF+20]. Multiple of these regions are organized in a track, which forms a magnetic nanowire. Each track is equipped with an access port, which can read or write an aligned magnetic region. In order to access an arbitrary region within the track, the magnetic charges need to be shifted first, such that the region to be accessed is aligned with the port. This shifting is achieved by applying a shift pulse at one end of the track. Since the track remains at the position, the distance to shift always depends on the previous access to this track. The induced latency for shifting is proportional to the shift distance.

In order to ease the organization of the memory device, multiple tracks are grouped together, such that the same shift pulse is applied to the entire group. The regions at a certain position in this group then form a memory word, which is shifted to and accessed at once. Such a group of tracks is referred to as a DOMAIN BLOCK CLUSTER (DBC). The DOMAIN BLOCK CLUSTERS (DBCs) are organized similarly as in classical memory devices to enable random access. Figure 3.3 Illustrates the organization of a single DBC with an illustrative word size of 4 bits. It can be seen that the access port and the shift pulse are unified per DBC such that the single tracks always form a memory word at the aligned regions. Consequently, the latency model should be considered for single DBC. A consecutive number of memory words (depending on the length of the tracks) is considered to be stored in a single DBC, Accesses within this consecutive region cause

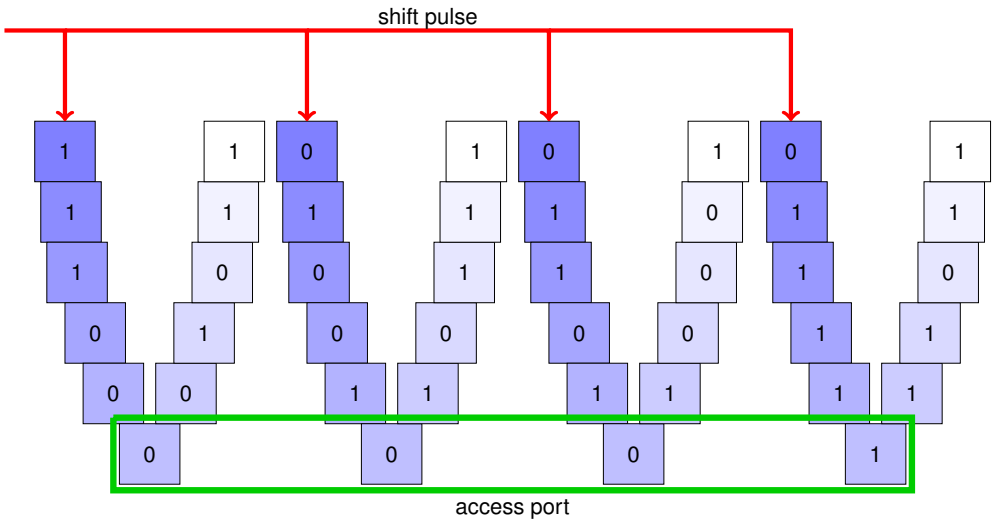


Figure 3.3: RTM DBC Organization

a latency, which is proportional to the distance to the previous accessed word within this DBC.

3.2 Random Forest Execution Model

The second part of this thesis focuses specifically on RANDOM FOREST (RF) execution and the impact to the memory subsystem and to the CPU. Consequently, low level realizations are considered, where the impact on the hardware is controllable. A basic introduction to RFs and their hardware-aware implementation is given.

RFs are machine-learning models, i.e. estimator models, which can be used for classification and regression problems. RFs are a collection of DECISION TREES (DTs), where every single tree derives a prediction value for a given input feature vector X. Depending on the exact realization of the RF, the results from the single DTs are combined to form global prediction. In the case of classification problems, a majority vote of the predicted classes from the single trees is a basic choice, in the case of regression problems the arithmetic mean can be a basic choice [Bre01]. The exact way of construction of the RF and the inner DTs is a matter of training. The basic training algorithm goes back to the CART algorithm (Classification and Regression Trees) from Breimann et al. [BFO+]. In this algorithm, the training data set is recursively split into two halves, by defining a threshold value on one element of the feature vector. This split then forms a DT node, which checks that specific feature index against the defined

threshold. Recursive continuation of this process on the resulting halves of the data set until a stop criterion forms the final tree. The prediction value is delivered by leaf nodes, which are determined as, for instance, the majority vote or arithmetic mean, of the remaining data set elements for the leaf node. Within a RF, all trees may be trained with the same training data set or with a certain strategy to only train on subsets. However, details of training are not relevant to neither this section, nor this thesis and no deeper explanation is provided.

After a DT is trained, it results in a logical model, which describes the inference rule from the given input feature vector to a prediction. This model can be described in a simple model, where X is the input feature vector and Y is the corresponding prediction value. A DT then consists of a set of m nodes $\{n_0, n_1, ..., n_{m-1}\}$, where n_0 is the root node of a tree. Each node n is associated with a feature index $FI(n)$, a split value $S(n)$, a left child index $LC(n)$ and a right child index $RC(n)$. Leaf nodes are further associated with a prediction value $P(n)$. The inference then follows a sequence of nodes $n_{i_0}, n_{i_1}, ..., n_{i_p}$, where $i_0 = 0$ is the root node and i_p is the only leaf node in the sequence. The inference follows the following rule:

$$i_{x+1} = \begin{cases} LC(n_{i_x}) \\ RC(n_{i_x}) \end{cases} \begin{array}{|l} X[FI(n_{x_i})] \leq S(n_{i_x}) \\ \text{else} \end{array} \tag{3.4}$$

Basically, every node checks a certain item of the feature vector against a threshold and either follows the path to the left or right subtree. When a leaf node is reached, $Y = P(n_{i_p})$ is the prediction value of the DT. This logical tree structure can be derived by training frameworks, such as scikit-learn [KK16]. Such tools deliver the tree structure either in an internal representation or allow an export to, for instance, a JSON object. Although means exist to execute a tree in such a format, efficient execution requires an implementation of the derived logical model as a follow-up step.

3.2.1 Probabilistic Execution Model

In order to optimize implementations of DTs for efficient execution on a target hardware, a deeper investigation of the execution structure is beneficial. If, for instance, the utilization of limited sized caches is to be optimized, it is beneficial to know which parts of the tree are accessed more often. This information can be derived by considering a probabilistic model on the tree execution. This probabilistic model is constructed entirely on the training data set at training time, such that it is available at the implementation time to serve as a basis for hardware aware optimization.

During training of a DT, the data set is recursively split in two halves by defining a threshold (split value) on a feature index and assign the data set elements to one set with a feature value at the corresponding index less or equal to the split value, and another set with a feature value larger than the split value. The choice of the split criterion does not intend to derive two equal-sized subsets, but rather minimize an impurity metric, such that prediction classes of the remaining subsets become purer and deliver a more precise prediction. As a consequence, the remaining subsets for the

left and right subtree can have different sizes. Assuming that the training data set is a representative sample of the data distribution, the sizes of these two subsets reflect the probability of data elements to belong to the right or left subtree. Normalizing these sizes, consequently leads to a relative node access probability $prob(n_x)$, which describes the relative probability of a node n_x to be visited from its parent node. The relative probability of the root node $prob(n_0) = 1$ is always 100%. The relative probabilities of all leaf nodes summed up must also result in a sum of 100%.

3.2.2 Implementation

For a subsequent implementation of DTs after training, many versions can be considered. As this thesis discusses hardware optimization in the following, the focus is drawn to implementations which allow to gain control over certain hardware aspects. Consequently, this thesis focuses on realizations of RFs and DTs in C/C++, which allows explicit control over certain hardware aspects. While the combination of single DTs into a RF can be done pretty straightforward (implementing a majority vote or an arithmetic mean results in a simple loop in C/C++), the implementation of a single DT can be done in two basic versions: **native trees** and **if-else** trees [ALD13]. These versions are referred in the literature sometimes with different terms, e.g. struct and codegen.

Native Trees

For native trees, the basic concept is to encode tree nodes as memory objects and link the tree nodes by pointers or indices to their child nodes. A tree node can become a C struct in this version, where all instances of nodes are stored in a large array. The left and right child nodes then can be referred to by an index within the array. The inference of the tree can be done in a simple loop, updating a running index through the tree.

```
1 struct Tree_Node {
2         bool isLeaf;
3         unsigned int prediction;
4         unsigned char feature;
5         int split;
6         unsigned char leftChild;
7         unsigned char rightChild;
8 };
```

Listing 3.1: Native Tree Node Struct Example

An example for the definition of a node struct is given in Listing 3.1. It should be noted that the data types have to be chosen carefully. For instance, the definition of the left and right child pointers as unsigned chars is only possible when the entire tree has not more than 256 nodes.

```
while(!tree[i].isLeaf) {
        if (pX[tree[i].feature] <= tree[i].split){
                i = tree[i].leftChild;
        } else {
                i = tree[i].rightChild;
        }
}
return tree[i].prediction;
```

Listing 3.2: Native Tree Node Loop Example

The loop for executing the tree consequently can be realized similarly to the example illustrated in Listing 3.2. For the realization of native trees, it should be noted that the tree is mainly encoded in data memory. The instruction memory is constant and not related to the tree shape. In addition, the realization of the tree structure by left and right child indices allows an arbitrary reordering of nodes within the array. Basically, any mapping of nodes to memory locations is realizable, as long as the child indices are set correctly.

If-Else Trees

The realization of if-else trees is the counterpart to the realization of native trees. In contrast to native trees, if-else trees encode the entire tree in instruction memory and ideally make no use of data memory at all. This is achieved by unrolling the shape of the tree into nested if-else statements. Every node checks a feature index against a threshold, which can be realized in a single if statement. Depending on the outcome, either the left or the right subtree is further executed. Hence, the corresponding code of the left and right subtrees can be placed in the if and in the else blocks, respectively. Whenever a leaf node is reached, a simple return statement with the corresponding prediction value stops the tree inference and delivers the prediction value.

```
if(pX[49] <= 0){
        if(pX[27] <= 0){
                if(pX[62] <= 2365){
                        if(pX[39] <= 0){
                                if(pX[38] <= 0){
                                        return 0;
                                } else {
                                        return 0;
                                }
                                ...
```

Listing 3.3: If-Else Tree Example

Listing 3.3 provides an excerpt from an example if-else tree.

Comparing this implementation to native trees, it can be noted that the freedom of placement of nodes is strongly limited. While native trees allow an arbitrary placement of

nodes in memory due to the usage of pointers, child nodes in if-else trees must always be placed in the if block or else block of their corresponding parent node. Although an arbitrary placement within the instruction memory could be achieved by introducing additional branch instructions (e.g. goto instructions in C/C++), this also introduces an additional overhead. Without introducing additional overheads, the only freedom of placement can be achieved by swapping the if and the else block of a node. Consider the statement `if(c) A else B`, where c is the condition to test, A is the left subtree implementation and B the right subtree implementation, an equivalent realization is achieved by `if(!c) B else A`. Although this does not allow arbitrary placement of nodes in the instruction memory, it offers a certain degree of freedom of placement during implementation.

3.2.3 Arithmetic Considerations

Since this thesis discusses optimization for the arithmetic computations in RFs, the basic operations for native and if-else trees are summarized shortly. The arithmetic operations to compute the final outcome of the ensemble after deriving the results of the single trees is neglected in this thesis, since this is a comparably short part of the execution of the ensemble and furthermore depends on the values used in the data set. Both realizations, i.e. native trees and if-else trees, require basic address arithmetic in order to perform branches, load array elements or maintain the running index. This is usually designed as standard integer arithmetic, more specifically as additions, multiplications and bit shifts of integer numbers. It should be noted that these operations are entirely independent of the data types used for the feature vector or the output, since they are only required for the tree structure. Specific for the used data in the tree is only the comparison operation, which compares the feature element against the split value. Consequently, only the comparison operation needs to be realized specific for the data types of the data set during inference. Comparison is a rather simple operation and consequently does not impose strong constraints on the underlying computation platform.

3.2.4 Performance Consideration

Comparing the performance of if-else trees and native trees is a non-obvious task. It has to be noted, that due to the different usage of instruction and data memory, both implementations can perform significantly different on certain target systems. For classic desktop computers, if-else trees are reported to perform significantly faster than native trees [CSH+22]. There are two factors, which can have a major impact on the execution time and may serve as an explanation towards this observation: 1) if-else trees require less assembly instructions per node and 2) if-else trees benefit from specific optimizations for code execution. An if statement, like in Listing 3.3, translates basically into four assembly instructions. First, the feature value at the specific index is loaded from memory. Second, the split value is loaded to a register. Third, a comparison instruction compares both loaded values. Fourth, a branch instruction finalizes the if

statement. It should be noted that certain ISAs may allow combining certain steps in one special instruction. Comparing this to the if statement in Listing 3.2, at least the same amount of instructions is needed for the if statement. In addition, the maintenance of the running index i is required. On top of that, overheads for the loop execution are introduced. In consequence, visiting a single node in a native tree requires significantly more assembly instructions than visiting a single node in an if-else tree. As an additional factor, if-else trees may benefit better from code performance optimization techniques, such as branch prediction, speculative execution, out-of-order execution and speculative prefetching than native trees.

Even though if-else trees are reported to perform faster on desktop class systems and there exist certain factors to this performance gap, if-else trees should not be considered as a superior implementation in general. The larger freedom of node placement in native trees allows accounting for special hardware characteristics. In addition, both implementations utilize instruction memory and data memory to a different degree. Depending on concurrently running software in the system, a better balance between the usage of instruction and data memory may be achieved by using either on or the other implementation.

3.2.5 Tooling

As described before, generating, implementing and execution hardware-aware realizations of random forests requires a couple of steps. The summary of this process is illustrated in Figure 3.4. The utilized tools for the generation process of random forests are summarized in a tool frame, called arch-forest [CSH+22]. The UCI machine-learning repository serves as a source of comparable benchmark data sets [AN07]. To achieve representative evaluation, the hyperparameters, such as the maximal depth for trees and the number of trees are usually varied in order to cover a wide range of interesting sizes. The maximal depth is almost only varied between 1 and 50, because most data sets anyway do not result in a depth of more than 50. The number of trees is also varied in a range between 1 and 50. For certain experiments, only single trees are studied, since there may be no impact on forest implementations.

The code generation, after deriving the logical tree model, is the module which allows the implementation of strategies to be analyzed. In this module, the entire forest model and the training data set is known and can serve for optimization. The code generator can transform the model to an arbitrary implementation. This includes standard implementations like native or if-else trees in C or C++, but also direct assembly implementations. After the code generation, compilation and execution achieves real execution of the random forest model. Utilizing tools like perf allows gathering runtime statistics of the execution, which forms a basis for evaluation results.

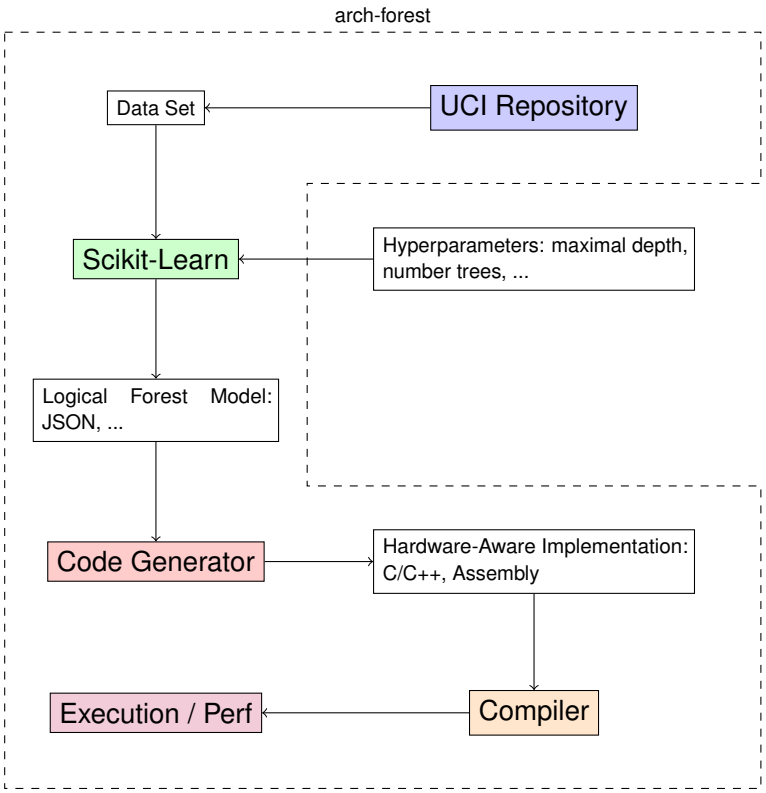


Figure 3.4: Tooling Overview for Random Forest Generation

3.3 CPU Model

Modern CPUs feature many properties due to their integration of complex technologies. Technical details about these technologies are for large parts not published by the manufacturers. Prefetching and eviction strategies of the caches, for instance, are rarely available in detail for off the shelf CPUs. Hence, this thesis makes some basic assumptions about the execution and the path of memory accesses inside of CPUs, which is valid for many existing CPUs.

3.3.1 Memory Hierarchy

In this thesis, CPUs are assumed to be equipped with memory hierarchies, i.e. with different levels of caches. It is assumed, that memory accesses target the cache levels in ascending order, until the requested value is found in a cache. For instance, a memory request targets the Level-1 cache first and continues with the Level 2 cache, if the value is not stored in the Level-1 cache. It is assumed that cache levels increase in capacity and decrease in speed. When a requested memory content is not found at the target cache level, it assumed to be loaded into the corresponding cache and persist there, until eviction. For the eviction strategy, it is assumed that a temporal component is

considered, i.e. that values which have not been accessed for a longer time are more likely to be evicted than values which have been accessed a short time ago. In total, this forms a cache behavior, that benefits temporal local memory accesses.

For the prefetching strategy of the CPU caches, it is assumed that a certain form of spatial locality is favored. This means, that memory accesses to locations close by the previous accesses are more likely to be hit by prefetching than memory accesses more far away. Furthermore, caches are assumed to be split into instruction and data caches at least on the highest cache level. Accesses to the instruction memory target a different set of cache memory than accesses to data memory.

3.3.2 Floating-Point Arithmetic

Although the hardware internal handling of floating-point numbers may be different for different CPUs, the programming model has certain properties in common, which are assumed in this thesis. It is assumed that CPUs provide a dedicated set of registers, which is used for floating-point computation. These registers may be also used for vector instructions or other specialized arithmetic, however, the important property is that general purpose registers are not commonly used for floating-point computations. In order to use the floating-point registers efficiently, it is assumed that a set of floating-point instructions exist, which perform operations directly on these registers. It is also assumed that there are instructions to transfer from general purpose registers to floating-point registers and vice versa.

CHAPTER 4

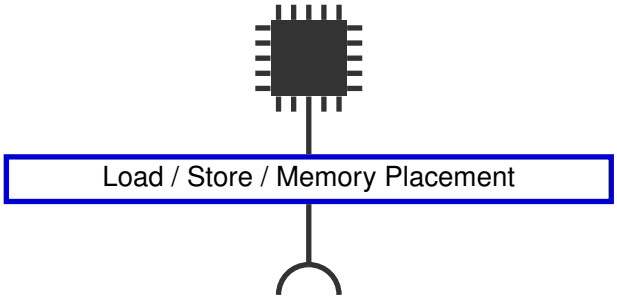
Application-Transparent NVM Wear-Leveling

Contents

4.1 Modern Technologies and Traditional Interfaces 54
4.2 Overview 55
4.2.1 Wear-Leveling Decisions 55
4.2.2 Wear-Leveling Actions 56
4.2.3 Wear-Leveling Flow 57
4.3 Software-Managed Read and Write Wear-Leveling 57
4.3.1 Scope 58
4.3.2 Problem Analysis and Statement 58
4.3.3 Coarse-Grained Wear-Leveling 59
4.3.4 Fine-Grained Wear-Leveling 63
4.3.5 Evaluation 66
4.3.6 Wrap-Up 70
4.4 Semantic Memory Tracing 70
4.4.1 Scope 71
4.4.2 Problem Analysis and Statement 71
4.4.3 Modular Analysis 72
4.4.4 Case Study 74
4.4.5 Wrap-Up 78
4.5 Concluding Interface Exploitation 78

4.1 Modern Technologies and Traditional Interfaces

Studying memory technologies leads to a very limited picture of the traditional memory interface. The only well established interface to communicate with main memory is the existence of load and store instructions. Naturally following from this, software can arbitrarily control the placement of memory contents across the address space. The traditional memory interface is illustrated in the following.

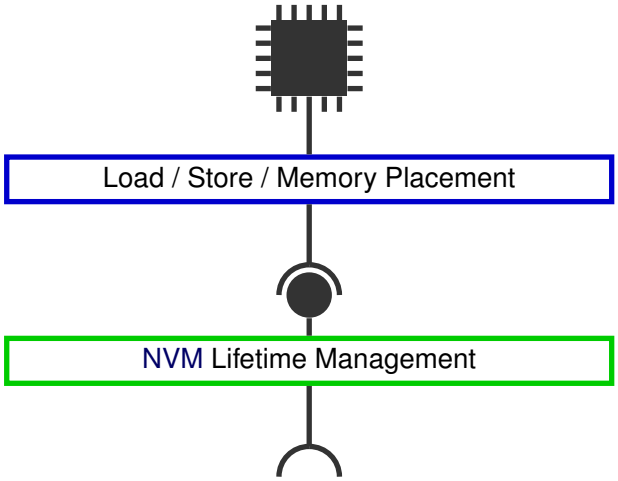


There exist many attempts to extend this traditional interface, for instance by adding software prefetching instructions to gain control over cache hierarchies, interfaces to report NUMA architectures, or even multiple versions of load and store instructions, which should result in different hardware operations [Kle05]. However, such proposals are either experimental or only available in a few systems and cannot be considered as a well established traditional interface.

With the limitation to only load and store instructions and arbitrary placements in the traditional memory interface, non-functional properties of the memory, such as latencies, lifetimes or other technological impacts cannot be transferred or accounted for. Hence, it is a major goal to find methods and means to exploit the existing interface to account for such technologies. Emerging memory technologies, especially in the field of NVM, bring in a couple of non-functional properties. One of the most interesting properties is the limited memory lifetime. While DRAM or SRAM can be considered to have a practically infinite memory lifetime [BRC+17], some other technologies have a highly limited lifetime, which can make the memory unusable within very short time periods, if no counter means are employed. It should be expected, that such technologies, however, will be integrated into systems with a traditional memory interface of only allowing store and load accesses.

Consequently, the question to be studied in the following chapter is how a memory lifetime management can be realized with the limited means of the traditional memory interface. In other words, the question is studied, which software components have to be added as an exploitation of the traditional memory interface to account for limited lifetime memory. This chapter studies a centralized approach, where a central software instance is employed in the operating system, which internally manages the states of

the NVM. This broadens the interface towards lifetime management:



The central software instance is realized by maintaining an internal lifetime surrogate model. Appropriate actions are submitted to the memory by changing memory locations of accesses through the traditional memory interface. The interface exploitation by a central instance makes the approach application transparent.

4.2 Overview

The task of wear-leveling for NVM is crucial to the ultimate lifetime of a system, since a system can become unusable once the first memory location is worn out. The process of wear-leveling can be further divided into the process of making a *wear-leveling decision*, including considerations about current aging and memory accesses, and performing a *wear-leveling action*. This chapter investigates approaches to perform both of these actions software-based, but transparent to the application, i.e. not requiring any corporation from the application. Hence, generally the binary image of an application is sufficient to work on top of these approaches. No source code from the application is required for wear-leveling. This section gives a summarized overview about possible approaches to software based application transparent wear-leveling decision-making and wear-leveling actions.

4.2.1 Wear-Leveling Decisions

For the process of making wear-leveling decisions, two general concepts could be considered in an application transparent manner: 1) aging-aware and 2) non aging-aware. While the former would try to estimate the current age of certain memory regions and make decisions for necessary wear-leveling actions accordingly, the latter would perform more general wear-leveling decisions, which independent of the current

aging of memory would lead to a more wear-leveled result. While for the latter version schemes like rotational replacement of memory locations or random memory location replacement can be promising, the estimation of memory aging on a software basis is a complex task. Realization as a software based solution, while being application transparent, requires the memory age estimation to take place within the operating system / runtime environment. In this chapter, a method for memory age estimation with the use of commonly available hardware components is presented. In addition, a method is presented, which interacts with the operating systems libraries, in order to achieve a more precise memory age estimation, separated by subcomponents of the application and the operating system.

Another alternative to memory age estimation in the running software is offline profiling of the target application. Tools like Valgrind [NS07] can be used to derive an instrumented binary, generating an approximate trace of memory accesses. Although such methods are fast and require low effort, the memory trace is only approximate and may only cover a subset of the possible memory access patterns. Nevertheless, memory age estimation in the running software also can only derive an approximate memory trace, the current memory access pattern of the application is always approximated. This ensures that the approximation is not drifting apart from the real memory trace over time.

Regardless of aging-aware or non aging-aware wear-leveling, decisions for wear-leveling have to be made at a certain point. A wear-leveling decision includes a logic memory region, which has to be physically relocated to another location. In the case of aging-ware wear-leveling, the target of a decision could be, for instance, chosen because the current physical location a some memory region is already more heavily worn out than other locations. For non aging-aware wear-leveling, locations could be, for instance, targeted randomly or in a repetitive circular manner.

4.2.2 Wear-Leveling Actions

Once a wear-leveling decision is made, i.e. a logic memory region is triggered to be relocated to another physical position, a proper action has to be performed. Making wear-leveling actions includes two crucially important aspects: 1) the choice of the target memory location and 2) the technical realization of the relocation in an application transparent manner. The choice of the target memory location can be ideally combined with the principle of the wear-leveling decision. Aging-aware wear-leveling, for instance, can choose the least worn out memory location as a target, random based non aging-aware wear-leveling could choose a random location as a target or rotational wear-leveling could choose also the target in a rotational manner. When it comes to the technical realization of the relocation of a logic memory region to another physical location, software based solutions quickly reach a limit when trying to main application transparency. Thus, this chapter presents a method, where commonly available hardware support is used for the relocation. This is realized by the virtual to physical memory mapping of the MMU, which is based on the granularity of memory pages, usually set to 4kb.

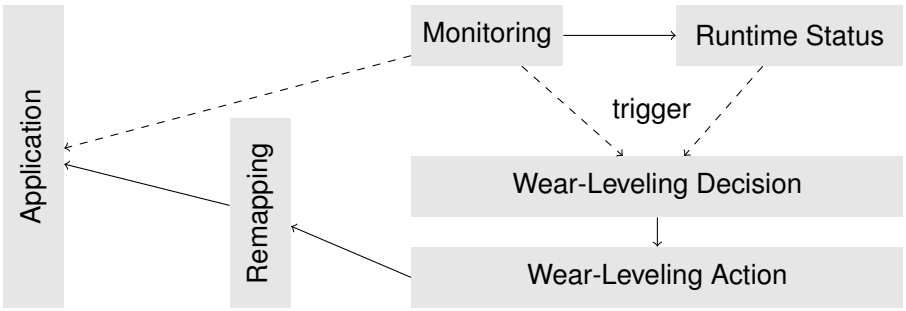


Figure 4.1: Schematic Overview of Application Transparent Wear-Leveling

Since this rather coarse granularity limits the effectiveness of wear-leveling for particular memory regions, an additional entirely software based scheme is developed, which only operates on the stack memory. Since the stack memory, by convention, is always used relative to the location of the stack pointer, the physical location of contents on the stack can be modified in an application transparent manner by setting the stack pointer to a different location.

4.2.3 Wear-Leveling Flow

In order to round up the overview of required basic steps for application transparent wear-leveling, this subsection summarizes the general flow of wear-leveling, including the process of making wear-leveling decisions and executing wear-leveling actions. Figure 4.1 gives an overview about the general flow of application transparent wear-leveling. The initiating component for every wear-leveling process is a combination of a monitoring module and a capture of some runtime status. In the case of aging-aware wear-leveleing, the monitoring has to track the age, i.e. the number of accesses, for certain memory regions and maintain an age distribution in the runtime status. In the case of non aging-aware wear-leveling, the monitoring could be, for instance, a monitoring module for system time in order to maintain frequent wear-leveling. The runtime status in that case could be offsets for rotational wear-leveling, random seeds, passed system time or other. Upon a certain condition, e.g. if a certain memory region becomes too old or if a timer expires, the monitoring triggers a wear-leveling decision to be made. This decision can include the current runtime status. The decision results in a wear-leveling action, which again can include the runtime status The wear-leveling action then is executed with the help of a remapping technique, which directly influences the execution of the application in a transparent manner.

4.3 Software-Managed Read and Write Wear-Leveling

This section presents methods for entirely software based read and write wear-leveling in detail. Hence, the methods can be used to increase the lifetime of write destructive, read

destructive or write and read destructive NVM, according to which parts of the method are applied. Furthermore, the methods include all required modules for wear-leveling, as described before: monitoring, keeping of a runtime status, wear-leveling decisions, wear-leveling actions and remapping techniques. Parts of the method are aging-aware, parts are non aging-aware. All methods are implemented as a library component in the library based unikernel unikraft [KSV+19a]. This allows execution and evaluation with the aforementioned simulation setup (Section 3.1.5).

4.3.1 Scope

The work presented in this section covers an entire implementation of software based wear-leveling for a commonly available system. This includes methods and implementations for the following points:

- Online read and write approximation with performance counters and the MMU in order to derive a statistically approximated trace of real read and write accesses of the application during runtime.
- Coarse-grained aging-aware wear-leveling, utilizing the approximate read and write trace to execute replacement operations with the virtual to physical memory mapping of the MMU.
- Fine-grained non aging-aware wear-leveling, operating on the stack memory for read and write accesses and on the text memory for read accesses by relying on conventions for stack and text in order to perform remapping of memory locations in an application transparent manner.

4.3.2 Problem Analysis and Statement

In order to analyse the requirement for wear-leveling in general, specifically within the different memory segments of a program (i.e. stack, text, data and bss), a set of benchmark applications is chosen first and analyzed for the memory access patterns without any modification of the application. The exact memory traces of these applications are derived by a previously developed simulation framework [HCY+20], including the full system simulator gem5 [BBB+11] and the NVM simulator NVMain 2.0 [PZX15]. It should be noted that these exact traces are only used to analyze the memory accesses of the application and the impact on the memory lifetime, but cannot be used as an input for any wear-leveling method, since a specific simulation environment is required to gather them. As a subset of illustrative applications, this work considers six small benchmark programs, which are taken from the MiBench suite [GRE+01] and from the nvm simulation setup [HCY+20]:

- **dijkstra** from MiBench applies the dijkstra algorithm to a graph, given as input. This benchmark uses an internal queue in the data segment to manage the steps of the algorithm.

published work: [HCS+21]

- **lesolve** from the nvm simulation setup applies the Gaussian elimination algorithm on a set of linear equations. The input data is directly modified in place to derive the solution.
- **sha** from MiBench features the SHA-1 has calculation of input data.
- **qsort** from the nvm simulation setup implements quicksort in a recursive manner. This causes intensive use of the stack segment.
- **qijndael** from MiBench computes encryption of input data based on the rijndael algorithm.
- **crc32** from MiBench computes crc checksums of input data.

Due to the high time consumption of full system simulations, the benchmark applications are chosen to be simple test applications, such that simulations are feasible within several hours. However, the benchmarks are chosen to represent applications with different memory usage patterns to analyze the behavior and the effect on memory lifetime of such different patterns.

As mentioned before, the impact of applications on the lifetime of write destructive and read destructive memories is studied. Since FeRAM, as a target read destructive memory, causes the wear out by succeeding read operations with a mandatory write operation, write and read accesses are considered as destructive for read destructive memories. Hence, the impact on the memory lifetime is analyzed for 1) write accesses only, considering the case of write destructive memories and 2) write and read accesses together, considering the case of read destructive memories. The results of the simulated memory trace of the 6 benchmark applications for write accesses and read and write accesses are accordingly illustrated in Figure 4.2 and Figure 4.3.

The results, presented in these figures allow two problem statements: 1) Different regions of the memory face largely different access frequencies and thus wear-out. Some regions, for instance, are not accessed at all, other regions are accessed intensively. 2) Some portions of the memory face a comparably high peak in the access frequency. For the case of write accesses only, these peaks are often found in the stack segment, for read and write accesses additionally in the text segment. In order to apply wear-leveling, both problems should be tackled.

In order to handle both problems in an application transparent manner, two methods are developed: 1) a generic coarse-grained wear-leveling, handling the regions with different access frequencies on a coarse granularity and achieve global wear-leveling. 2) specific fine-grained wear-leveling, targeting the text and stack segment to reduce the comparably high peaks. Since this specialized solutions only operate locally on the text and stack, both methods need to be applied in combination.

4.3.3 Coarse-Grained Wear-Leveling

The problem statement (Section 4.3.2) points out the need for generic global coarse-grained wear-leveling and specialized local fine-grained wear-leveling. This section provides a detailed explanation of a realization of application transparent software based

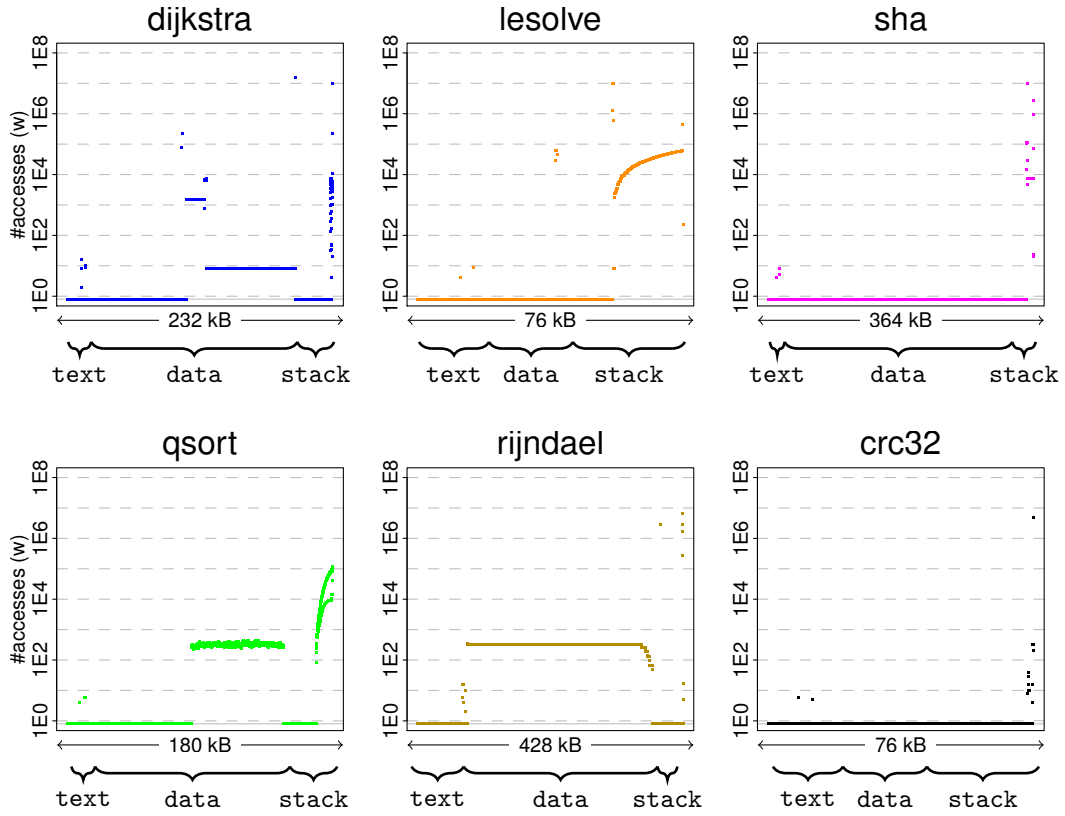


Figure 4.2: Benchmark baseline memory traces (write accesses)

aging-aware coarse-grained wear-leveling. Within this method, the current memory age is estimated with the help of configurable memory access permissions and performance counters for memory accesses. The wear-leveling actions are realized by a custom pagetable in the MMU, allowing to exchange the physical memory location of virtual memory addresses, which are used by the application.

Memory Age Estimation

Estimating the current aging of memory regions would intuitively require some hardware support to track memory ages or at least count the frequency of accesses. Within the hereby proposed method, however, no dedicated hardware support is required and memory age estimation is still realized in an application transparent manner. This is realized by two components: Performance counters for memory accesses and configurable memory access permissions.

First, a sampling of memory accesses is realized by configuring a performance counter to count write accesses and another performance counter to count read accesses. Both performance counters are configured to cause an interrupt (trap) on an overflow. By setting the performance counter value manually to the maximal value ($2^{32} - 1$ for 32 bit performance counters) minus a configurable number after each over-

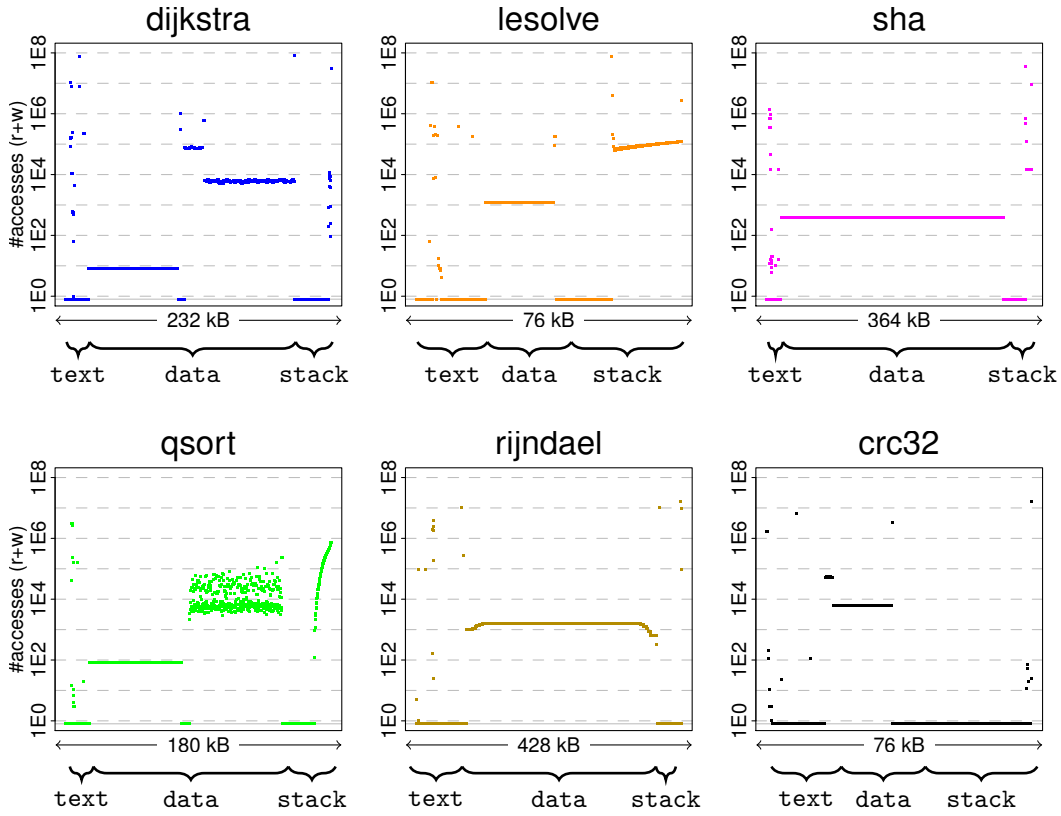


Figure 4.3: Benchmark baseline memory traces (write and read accesses)

flow, interrupts are caused whenever exactly the configured amount of write respectively read accesses happened. This provides the chance to realize statistical sampling after a configurable amount of read and write accesses each. The rate of the sampling further can be configured for write and read accesses separately, since two distinct performance counters are used. In the following, the sampling rate for write accesses is denoted as C_{sample}^{write} and C_{sample}^{read} for read accesses, respectively.

Second, whenever an interrupt of the performance counter overflow happens, the target address of the subsequent write or read access (depending on which counter overflowed) is captured, providing one sample of the real memory access distribution. Sine the performance does not necessarily cause synchronous interrupts, it cannot be assumed that the PROGRAM COUNTER (PC) points to the instruction, which caused the overflow. Hence, an additional trapping mechanism is implemented. Whenever the performance counter overflow interrupt is caused, the memory permissions of the traced memory are set to read only or not accessible, depending on weather the write or read counter overflowed. Consequently, the subsequent write, respectively read instruction causes a memory access violation trap, which provides the causing address to the trap handler. The trap handler resets the memory permissions to the original state and continues execution. In consequence, the target address of the sampled access is

recorded and a full sample of the write or read distribution is recorded. This process builds an approximate memory trace over time.

As mentioned above, the mechanism to capture the target of the subsequent read access has to set the memory permissions to restrict all accesses. In consequence, also write accesses cause a violation and call the corresponding trap handler. In this case, memory permissions have to be restored to the original state as well, since the write operation has to complete. In order to still capture the next read access, the instruction after the write access is replaced with a breakpoint instruction, where the corresponding breakpoint handler sets the memory access permissions again to not accessible. This process is repeated until a read access is reached. Afterwards normal execution continues. It should be further noted that this mechanism cannot be used to capture an approximate memory trace of the text segment, since every instruction fetch causes a read access to this segment.

The described realization of estimating memory accesses results in an approximate memory access trace for read and write accesses. The samples from this trace are used to build a histogram on a configurable granularity, revealing the approximate wear-out of the observed memory regions. This histogram builds a basis for aging-aware wear-leveling decisions. The rate of the read access sampling C_{sample}^{read} may be configured to another value as the rate for the write access sampling C_{sample}^{write}. In order to accommodate for this, the histogram weights the access with the fraction of the sampling rates in order to approximate the real memory wear-out.

Wear-Leveling Decisions

Based on the approximated memory ages, wear-leveling decisions are deferred. Within this method, a straightforward scheme is realized, where *hot* logical memory regions are identified and physically relocated to *cold* memory regions. Repeating this process frequently realizes incremental wear-leveling, where all physical memory regions are similarly worn out over time. This concept further does not require persisting any aging state of the memory, since the wear-leveling is incremental at any time. The identification of hot logical memory regions and cold physical memory regions is based on the approximated memory ages. Whenever a configurable amount of n_{reloc} samples is added to one logical memory region, this region is considered as hot. The physical region with the lowest value in the approximated access histogram is considered as the most cold region.

Upon ever sampled added to the histogram, the condition of a hot logical memory region is checked. If this condition is true, a wear-leveling action is triggered. The wear-leveling decision includes the exchange of the physical locations of the identified hot logical memory region and the coldest physical region. In order to quickly identify the coldest memory region in the histogram, physical regions are stored together with their estimated age in a sorted red black tree, which allows fast search for the smallest element and easy insertion of an updated region.

Once the pair of physical memory regions (and hence also their mapped logical memory regions) is decided to be exchanged, a two-step process is applied to perform the exchange in an application transparent manner. With the help of the MMU, the mapping of virtual memory pages for the two physical memory pages is exchanged. In addition, the content is copied from one page to the other and vice versa. In consequence, the application sees the same content at a virtual page, but the physical location is exchanged. Using this process, the granularity of memory regions is set to the size of memory pages, managed by the MMU, in the implemented case 4 kB. It should be noted that the histogram of estimated memory ages is not updated every time a sample of the access distribution is recorded. Instead, a temporary histogram of virtual memory regions and their number of recorded samples is maintained, which is used to easily decide for hot memory regions. Whenever a wear-leveling action is triggered, the value of this virtual memory region is reset to 0 in the temporary histogram. The stored value in the red-black tree (for the physical regions) is updated by 1 upon a wear-leveling action. Hence, the red-black tree is only updated upon a wear-leveling action and only stores a scaled histogram of the approximated age distribution for physical memory regions.

4.3.4 Fine-Grained Wear-Leveling

Section 4.3.2 points out the need for global coarse-grained and local fine-grained wear-leveling. While Section 4.3.3 provides an approach for application transparent coarse-grained global wear-leveling, this section focuses on local fine-grained wear-leveling for the stack and the text segment, as the most dense access hotspots are experimentally found in these regions. Stack and text memory are suitable candidates for application transparent wear-leveling, since their usage is not strictly determined and rather follows certain conventions during runtime. Stack memory is managed with the contents of the STACK POINTER, text memory is managed with the content of the PROGRAM COUNTER. Accordingly, memory contents can be physically relocated in an application transparent manner, when the STACK POINTER (SP) or PC are adjusted as well.

Although the movement of the stack and text segment in general can be realized in an application transparent manner, arbitrary relocation is not possible. The segments are used by the application relative to the value of the PC and SP. This implies that the entire text and stack segment has to be accessible in the virtual memory space at positions relative to these pointers. In consequence, arbitrary relocation on fine granularities of memory regions is not possible. The wear-leveling method presented here therefore applies a circular non-aging aware relocation of the text and stack segment. In greater detail, the entire stack and text segment is moved by a small offset to another position regularly. The according adjustment of the PC and SP allows the application to still access the correct memory contents. In order to limit the relocation of the stack and heap in a bounded region, circular movement of these regions is realized. Memory contents are regularly relocated towards higher memory addresses. When the end of

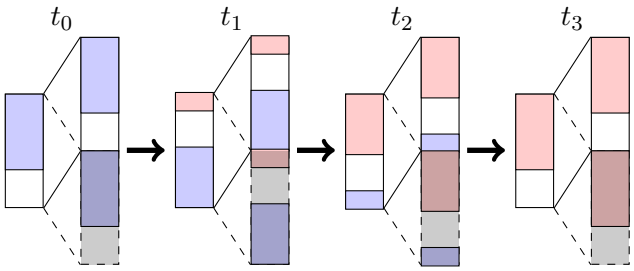


Figure 4.4: The physical memory pages (each on the left) and the main and shadow virtual memory map (each on the right) during the movement steps. The colored blocks contain the allocated and used memory; the red color indicates that this block already performed the wraparound.

the bounded region is reached, the relocation starts from the beginning of that region again. In order to realize this in an application transparent manner, a method, called *shadow memory* in the following, is implemented.

To realize shadow memory, a region of physical memory is mapped consecutively two times to the virtual address space with the help of the MMU. Accordingly, when accesses overshoot the upper bound of the first virtual map, they automatically target the bottom of the physical memory region again. Thus, the text and stack region can be relocated towards higher memory addresses and any content that leaves the upper boundary is automatically wrapped around to the beginning of the region again. Accesses from the application, which add a certain offset to the PC or SP also automatically target the wrapped around physical memory. Once the relocated memory region entirely leaves the first virtual map, the PC or respectively the SP is reset to the beginning of the first virtual map, which already contains the correct contents. This process is illustrated in Figure 4.4. It can be seen that although physically the memory is wrapped around to the beginning of the region again, the virtual shadow mapping always ensure consecutive accessibility of the memory. It is worth noticing that this method relies on the virtual to physical memory mapping of the MMU and the text and stack regions have to be rounded up to the size of multiple memory pages.

The previous description may suppose that application transparent relocation of stack and text memory can be straightforward achieved by setting the PC or SP to the corresponding address. However, the process requires special care of further circumstances, which stem from the targeted hardware. Hence, the following explanation relates to the specific implementation on the ARMv8 test system.

In order to relocate the stack, the relocation of the SP and the memory contents of the stack by the same offset is straight forward realized. For all kinds of accesses, relative to the SP this is also sufficient. Memory addresses relative to the stack, however, may be computed by the software and stored in memory, for instance to be passed to a function as an argument. For these *materialized* pointers to the stack memory, the adjustment of the SP has no effect and the movement of the stack memory causes errors

accordingly. To accommodate for this effect, several mechanisms could be considered. An intuitive solution would be to scan the stack, data and bss segment for values, which could be a pointer to the stack memory and adjust these values accordingly. This, however, would cause errors when memory content by coincidence has a value, which could be a pointer to the stack. Therefore, the implementation ships with a MMU based pointer consistency mechanism, which eliminates the risk of accidental value changes on the cost of requiring a MMU and massive availability of virtual memory space. The basic concept is to not only adjust the SP by a small offset but by the entire size of the stack region in addition (assuming the stack region is the size of multiple memory pages) and adjust the virtual memory map in such a way that the newly targeted virtual memory pages point to the same physical memory as the previously used memory pages. Subsequently, the memory map for the previously used virtual memory pages is invalidated such that any access to a beforehand materialized address results in a memory trap. The trap handler consequently adjusts the register, which holds the materialized address to the new virtual memory region and fixes the relocation offset accordingly. It should be noted that this does not take an effect on materialized stack addresses in the memory and cause a trap again every time such an address is loaded to a register and used. It should be further noticed that to eliminate the risk of any overlap in the virtual memory space, unused virtual memory pages have to be used every time. This practically limits the lifetime of this method. However, assuming a 48 bit address space, $2.8 \cdot 10^{11}$ memory pages are available. Configuring the fine-grained wear-leveling to e.g. one relocation per second with a stack region of 8 virtual memory pages, the virtual memory space would suffice for 136 years.

Relocating the text segment is realized in a very similar manner. For the implemented case of ARMv8, the text is used for parts only relative to the current PC, for other parts like function calls, addresses are also materialized to memory. The provided implementation compiles the target code as a POSITION INDEPENDENT CODE (PIC) in order to make sure that addressing is performed in a PC relative mode. This introduces two data structures to the compiled code: The GLOBAL OFFSET TABLE (GOT) for data objects and the PROCEDURE LINKAGE TABLE (PLT) for function addresses, which both materialize absolute addresses. Both these data structures are adjusted accordingly when the text segment is relocated. The data structures itself are addressed PC relative. The PC relative addressing for that purpose in ARMv8 is in greater detail only PC relative for a coarse granularity. The compiler introduces *adrp* instructions to compute addresses relative to the current 4 KB page of the PC. Since the offset within pages is not PC relative, the developed implementation keeps the GOT and PLT at static addresses and excludes them from relocation. The only problem remaining is that when an instruction is relocated across the boundary of a 4KB pages, the *adrp* mechanism delivers a wrongly computed address. This is fixed by rewriting *adrp* instructions during relocation and adjusting the immediate offset in the instruction. In addition, the same address consistency mechanism as explained before is used in order to protect against materialized instruction addresses for instance in the form of function pointers.

Whenever the PC is set to an invalidated virtual memory region a trap is caused and the PC is adjusted to the correct address, including the adjusted offset.

4.3.5 Evaluation

Designing software based wear-leveling for NVM is described as the realization of coarse-grained and fine-grained wear-leveling beforehand. In this section, both approaches are experimentally evaluated by executing the real implementation in the simulation setup, recording memory traces, illustrating these and computing the measurement means according to the wear-out model. As test applications, the initially used benchmark applications are used again. Configuring the parameters for the described wear-leveling methods is done by experimental tests and deciding for a trade-off between caused overhead and gained memory lifetime. This results in setting the sampling rate for the coarse-grained estimation of write accesses to $C^{write}_{sample} = 2000$, the sampling rate for the estimation of read accesses to $C^{read}_{sample} = 12000$ and the trigger for wear-leveling decisions $n_{reloc} = 64$. For the fine-grained wear-leveling, the distance of a relocation in every step is 64 bytes, since the memory subsystem is assumed to always write and read a full 64 byte line and cause equal wear-out for all bytes within. The action for fine-grained wear-leveling is triggered together with every action of the coarse-grained wear-leveling, i.e. on the rate of $n_{reloc} = 64$.

The illustrative results of the memory traces after applying the corresponding coarse-grained wear-leveling for write accesses only (write destructive case) and write and read accesses (read-destructive case) can be found in Figure 4.5 and Figure 4.6. The corresponding illustrations for the application of coarse-grained and fine-grained wear-leveling can be found in Figure 4.7 and Figure 4.8. In addition to the illustration of the resulting memory access distribution, the measures according to the wear-out model are computed and summarized for all investigated cases in Table 4.1. Investigating the provided evaluation results, several observations can be made. Generally, the observations can be distinguished between read-destructive and write destructive memories and coarse-grained and coarse-grained plus fine-grained wear-leveling.

Coarse-Grained Wear-Leveling

From the graphical illustration of the memory traces, it can be observed that the aging-aware design principle of the coarse-grained wear-leveling generally works out and distributes memory page sized blocks in such a way that all physical blocks face similar access distribution. This works out for the write-destructive and read-destructive case. Although the initial situation of a few dense peaks in the memory access distribution is relaxed a bit, dense peaks still can be found within every memory page (forming a repeating pattern across the memory space). This observation supports the initial design principle that global coarse-grained wear-leveling is required to distribute stress to memory regions and local fine-grained wear-leveling is required to flatten peaks within small memory regions. Also investigating the analytical LI results for the coarse-grained

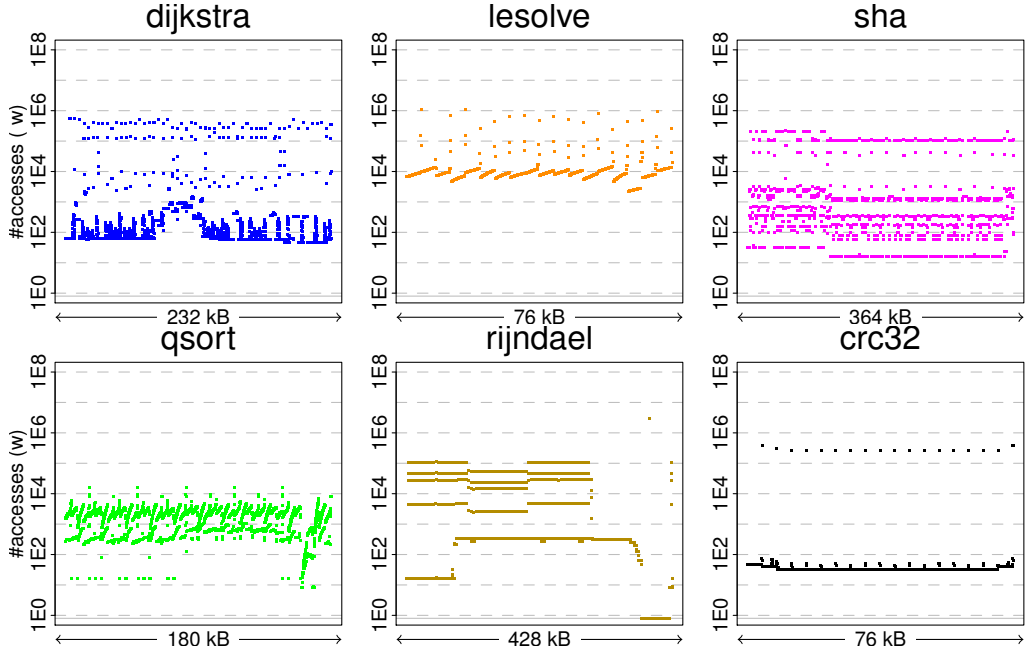


Figure 4.5: Coarse-Grained Wear-Leveling (write accesses)

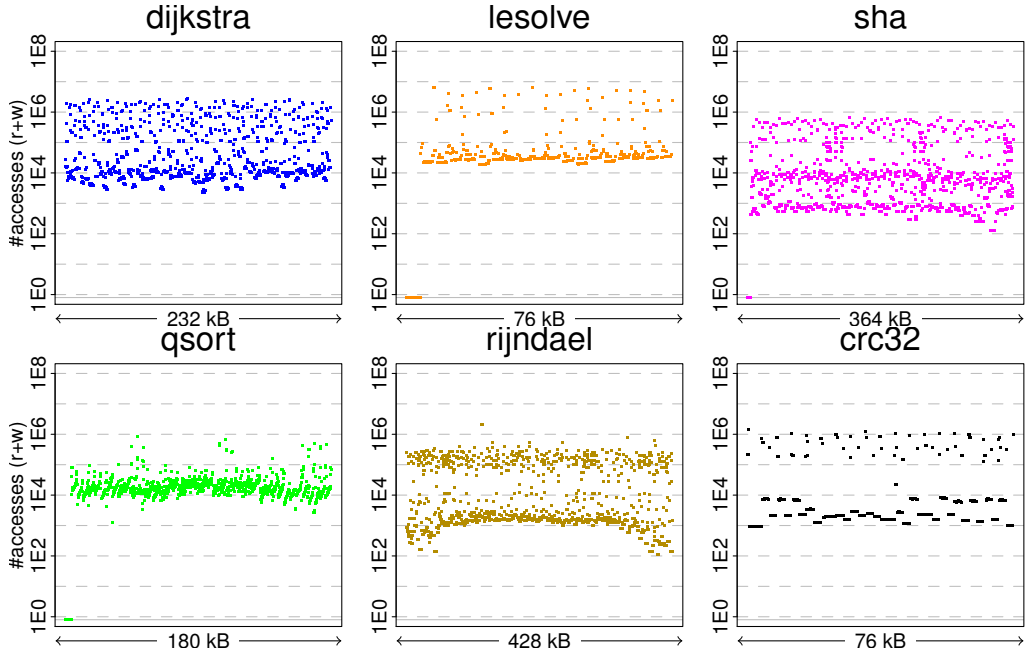


Figure 4.6: Coarse-Grained Wear-Leveling (write and read accesses)

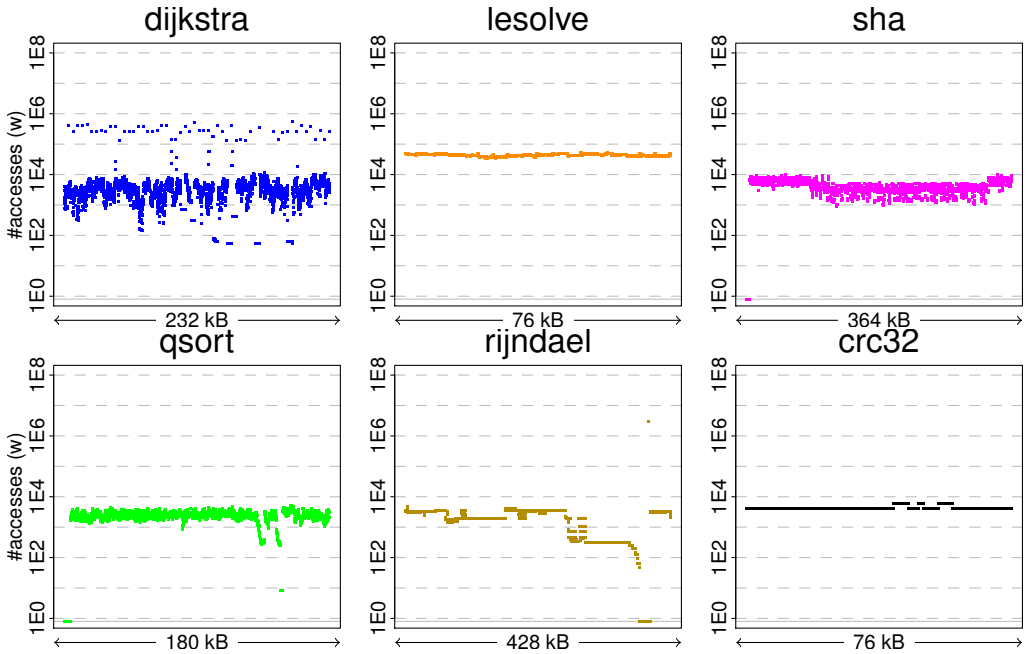


Figure 4.7: Fine-Grained Wear-Leveling (write accesses)

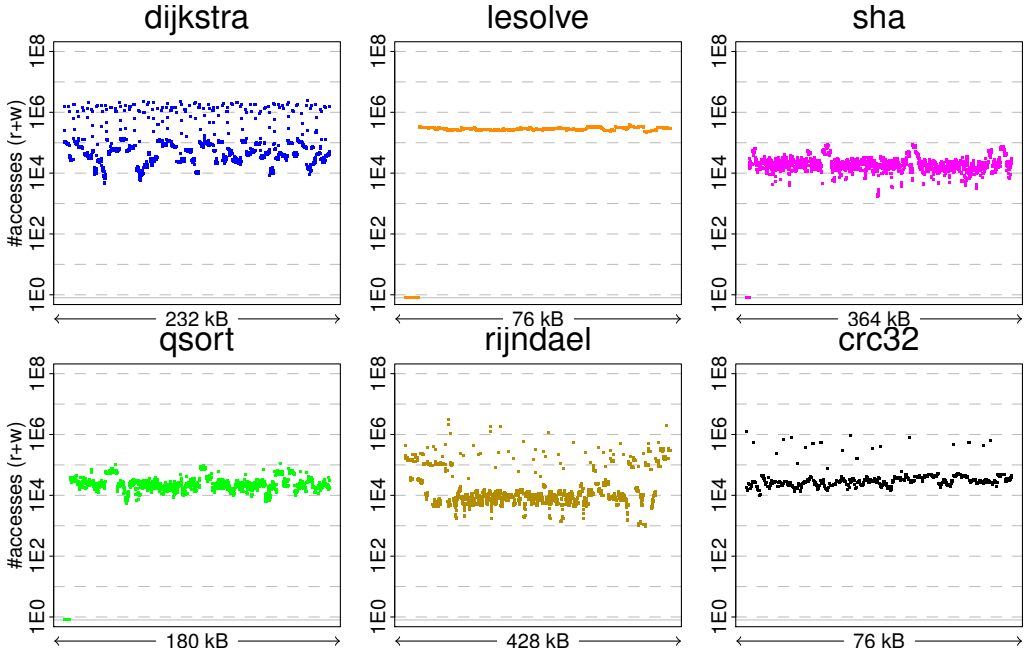


Figure 4.8: Fine-Grained Wear-Leveling (write and read accesses)

wear-leveling conveys a similar conclusion. The lifetime, considering caused overheads, is extended by reasonable factors of $\approx 10\times$ for some test applications and even up to $\approx 40\times$ for the sha benchmark. Although such an improvement can already help to extend

Application / Configuration	AE	EI	LI	Application / Configuration	AE	EI	LI
dijkstra				**qsort**			
-baseline[w]	0.00048			*-baseline[w]*	0.01609		
-baseline[r+w]	0.00110			*-baseline[r+w]*	0.00599		
-coarse-grained [w]	0.01349	28.1042	27.8912	*-coarse-grained [w]*	0.12174	7.5662	7.5097
-coarse-grained [r+w]	0.03283	29.8455	29.5888	*-coarse-grained [r+w]*	0.02292	3.8264	3.7901
-fine-grained [w]	0.01385	28.8542	28.2420	*-fine-grained [w]*	0.44067	27.3878	23.0112
-fine-grained [r+w]	0.04527	41.1545	35.6377	*-fine-grained [r+w]*	0.19900	33.2220	27.9929
lesolve				**rijndael**			
-baseline[w]	0.00189			*-baseline[w]*	0.00035		
-baseline[r+w]	0.00183			*-baseline[r+w]*	0.00082		
-coarse-grained [w]	0.01712	9.0582	8.9899	*-coarse-grained [w]*	0.00123	3.5143	3.4978
-coarse-grained [r+w]	0.02132	11.6503	11.5302	*-coarse-grained [r+w]*	0.00610	7.4390	7.3514
-fine-grained [w]	0.82097	434.3757	189.2855	*-fine-grained [w]*	0.00088	2.5025	2.3027
-fine-grained [r+w]	0.67184	367.1257	194.7539	*-fine-grained [r+w]*	0.01431	17.4564	5.1970
sha				**crc32**			
-baseline[w]	0.00028			*-baseline[w]*	0.00087		
-baseline[r+w]	.00033			*-baseline[r+w]*	0.00200		
-coarse-grained [w]	0.01182	42.2143	41.8878	*-coarse-grained [w]*	0.01117	12.8396	12.7390
-coarse-grained [r+w]	0.01796	54.4242	53.8633	*-coarse-grained [r+w]*	0.02316	11.5798	11.4619
-fine-grained [w]	0.42223	1507.9643	955.6642	*-fine-grained [w]*	0.70932	815.3117	798.0718
-fine-grained [r+w]	0.22706	688.0606	418.8788	*-fine-grained [r+w]*	0.02764	13.8225	13.0103

Table 4.1: Memory Lifetime Indicators

the system lifetime from a few weeks to several years and may make the application of NVM feasible in certain cases, the graphical illustration reveals that still improvement potential is existent when relaxing the local peaks.

Fine-Grained Wear-Leveling

Figure 4.7 and Figure 4.8 provide a graphical illustration for the fine-grained wear-leveling results for specific memory traces. Especially with a focus on the observable shortcomings of the coarse-grained wear-leveling, it can be observed, that local peaks are further relaxed by fine-grained wear-leveling. This supports the initial design principle of executing fine-grained wear-leveling as an addition to aging-aware coarse-grained wear-leveling for intra page wear-leveling. For both cases of write destructive and read destructive memories, it can be observed that most benchmark applications do not feature significant uneven peaks after the employment of the fine-grained wear-leveling. An exception to this is formed by the dijkstra benchmark. This benchmark makes heavy use of the data segment for internal management, resulting in uneven peaks

inside this segment. Since the fine-grained wear-leveling only covers the stack segment for the case of write destructive memories and the stack and text segment for read destructive memories, peaks in the data segment cannot be reduced. As a general trend, the resulting memory trace for read-destructive memories is less even than for write destructive memories. This stems from the fact that while the write destructive fine-grained wear-leveling can move the entire stack in memory, the read destructive fine-grained wear-leveling can only relocate the code parts of the text segment as has to maintain the absolute position of the GOT and PLT. As the benchmarks naturally make use of these data structure during execution, dense peaks are still caused and not wear-leveled to these data structures.

Investigating the analytical LI results for fine-grained wear-leveling (Table 4.1), reasonable improvements in comparison to the coarse-grained wear-leveling can be observed for most benchmarks. For the sha benchmark, an absolute lifetime improvement of almost $1000\times$ can be observed. It has to be noted that the fine-grained wear-leveling causes generally a larger overhead than the coarse-grained wear-leveling, which is why the EI and LI indicators have a larger difference for the case of fine-grained wear-leveling. Considering the gained improvement in lifetime, however, it can make a reasonable trade-off to invest such larger overheads for the gain in memory lifetime.

4.3.6 Wrap-Up

This section details methods for entirely software based read and write wear-leveling in an application transparent manner. This includes an aging-aware coarse-grained memory page based wear-leveling, which largely works similar for the case of read and write destructive memories. In addition and to compensate the shortcomings of such coarse-grained wear-leveling, specific extensions for fine-grained stack and text wear-leveling are integrated. The experimental results show largely improved memory lifetimes when coarse-grained and fine-grained wear-leveling is employed. However, a few memory regions cannot be targeted by the presented methods and case suboptimal memory wear-leveling.

4.4 Semantic Memory Tracing

The previous part of this chapter discusses application transparent wear-leveling on an entirely software based implementation. Orthogonal to this, this sections discusses the aspect of memory tracing in greater detail. For the overview of the wear-leveling process, memory tracing can potentially impact the aspect of monitoring and of making wear-leveling decisions. As a special focus of this section, semantic memory tracing is performed. This means that the application is logically decomposed into certain units, which are then logically isolated in memory, such that the process of wear-leveling could be executed separately, for instance with different configurations, for the application units. A crucial component to this method is formed by the library based unikernel

unikraft [KSV+19a], hence all the following methods are imlemented and tested in the environment of unikraft.

4.4.1 Scope

The work in this section covers two central methods for semantic memory tracing. Both methods are built on top of the full system simulation setup and the unikernel unikraft. The results of this work are methodological, i.e. they are not directly integrated into the software based wear-leveling framework. The methods in this section are intended to aid development of specific applications and to find and configure according wear-leveling and memory management schemes for these applications. To provide an intuition for the usability, a case study on an exemplary application is appended. The two central methods in this work cover:

- Static memory tracing: In this method, compiler information from unikraft is used to map the binary layout of the compiled unikernel to the single libraries, which are used for the compilation of the unikraft instance. This method is limited to the statically allocated memory segments (dara, bss and text). Advantageous to this approach is that the analysis can be performed offline and results on a simple map of memory addresses to components of the unikernel.
- Dynamic memory tracing: In order to further provide a semantic association of memory regions in the dynamically allocated memory segments (heap and stack) to the corresponding library, dynamic analysis is implemented based on the program counter information. As a part of this, the simulation framework is extended to output the current program counter to the memory trace. This information in combination with the static memory trace of the text segment then allows to form a clear association of every memory access (including accesses to the stack and to the heap) to the executing library of the access.

4.4.2 Problem Analysis and Statement

Performing memory tracing with full system simulations results in a memory trace, which delivers detailed information about single memory accesses during the execution. Although this information can be arbitrarily detailed, they are collected on a system level, i.e. they usually have no strong relation to the executed software architecture. When wear-leveling should be conducted system-wide for the entire software, this can be considered as sufficient. When, however, wear-leveling should not be conducted uniformly, further semantic memory tracing is needed. This can become necessary when, for instance, wear-leveling is anyway configured on a fine granularity for single libraries of a unikernel and the libraries have a largely different demand for wear-leveling. In such a scenario, the semantic memory analysis can be used to come up with a suitable wear-leveling configuration for all libraries. Another scenario, which could be

published work: [HCK+20]

considered, is the presence of a heterogeneous memory architecture, where memory portions with different lifetime characteristics and hence different demand for wear-leveling are present. In such a scenario, a possible strategy would be to map the memory of single libraries to the corresponding memory type, to match the memory access characteristics from the library with the demands of the memory. Semantic memory tracing in this case can deliver the important information about the memory behavior of the single libraries.

Motivated by the examples above, the problem of semantic memory tracing is stated to provide a semantic association of memory accesses in the full system trace to the causing software component. Since the semantic memory tracing is implemented in a unikernel, the granularity of software components is chosen to be single libraries of the unikernel. This can be considered as an application transparent method, since no modifications are made to the libraries or the applications within the unikernel. The semantic memory tracing is entirely realized in the unikernel internals and the memory simulation.

4.4.3 Modular Analysis

The semantic memory tracing itself is realized by hooking into the compilation process of unikernel. The compilation process is separated by libraries, where all source files of a library are first compiled to separate object files and partially linked afterwards to a single binary file of the library. All the pre linked library files are linked to a single binary file of the unikernel in a final step. This file is further post processed in order to derive an executable memory image. It has to be noticed, that the final linking step to a single binary file merges the memory segments of the libraries together. For instance, all compiled code from all libraries is merged to a single text segment. In consequence, investigating the memory segment boundaries of the compiled unikernel does not reveal any further semantic information.

Static Memory Tracing

Executing the compiled unikernel on the full system simulation delivers a memory access trace from the address space of the unikernel. It may be possible to identify if a memory access is within a specific memory segment, but not from which library it was caused, since all libraries are merged within the memory segments. To overcome this, debug symbols are utilized for static memory tracing. During compilation and linking, the compiler enriches the output files with debug symbols, which identify single variables, functions or even basic blocks of functions in the compiled output. These symbols are unique within the unikernel, since they are used also for the final linking. The working principle of the static memory tracing consequently is to scan the final unikernel image for all existent debug symbols and their corresponding memory addresses in the linked binary and match each debug symbol to the pre linked binary files of the single libraries. For each symbol, one originating library is identified by this method. This results in a

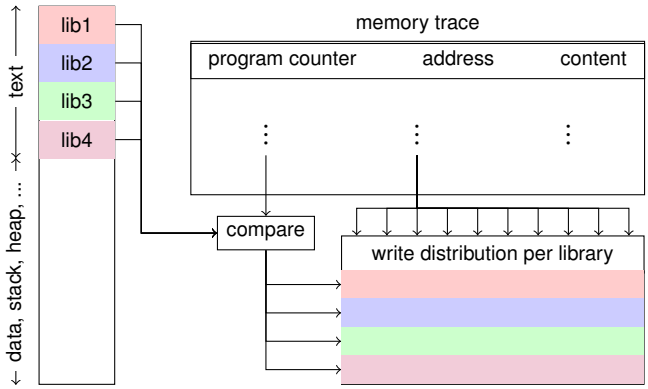


Figure 4.9: Dynamic Memory Trace Analysis

debug symbol to library map. Together with the memory locations of the debug symbols in the final memory image, a memory library map is derived. This map can be then associated with the simulator output and provides the semantic information about the memory ownership of a library for every single memory access. It has to be noted, that memory ownership does not necessarily mean that also the implementation of this library is causing the memory access. It may happen that a library passes a pointer to some portion of owned memory to another library, which then performs accesses to this portion of memory.

Dynamic Memory Tracing

As mentioned above, the static memory tracing can provide semantic information about memory ownership, but not about the executing library of a memory access. Furthermore, the static memory tracing can only operate on memory segments, which are allocated during compile team. Hence, the stack and heap segment cannot be targeted by static memory tracing. To account for these two missing features, dynamic memory tracing is implemented as an additional module. The working principle of dynamic memory tracing is illustrated in Figure 4.9. The executing library of memory access can be identified by the memory ownership of the currently executed instructions, i.e. of the portion of the text segment. This information can be derived from static memory analysis, since the text segment is statically allocated and debug symbols are added for basic blocks and at least for functions. The currently executed instruction can be derived by investigating the PC. Consequently, the full system simulator is modified in order to append an additional entry of the current PC for each single memory access. When deriving the semantic memory trace, the PC is compared with the memory to library map of the text segment and the causing library of the access is identified. The resulting memory trace from the simulation then can be further divided into single library memory traces and used for potential separated wear-out analysis.

When comparing static and dynamic memory analysis, it should be noted that static memory analysis identifies memory ownership, while dynamic memory analysis

identifies the runtime executor of memory accesses. While only dynamic memory tracing can target the analysis of dynamically allocated memory segments, static and dynamic analysis can be executed together for statically allocated memory segments. When, for instance, a memory mapping of libraries to a heterogeneous memory landscape should be decided, the results from static memory tracing can serve as an important input. When the configuration of software based wear-leveling schemes is to be decided for different libraries, the behavioral information of the dynamic memory tracing can deliver better insights.

4.4.4 Case Study

In order to provide an illustrative example of how the decomposition into single libraries of an application can be performed, a typical database use case is considered in this case study as the application. In greater detail, a sqlite3 database [New04] implementation is compiled into unikraft together with required dependencies (libc and pthreads). In addition, the storage layer is realized with a virtual main memory file system. It is worth noticing that all these implementations are provided in the unikraft source repositories as electable components. As the workload for the database implementation, the TPC-H queries Q1 and Q6 [Cou08] are executed on a scaled lineitem table of 500 rows. The limitation to the scaled data set is due to the massive time consumption of the full system simulation. The mentioned application and workload is executed on the full system simulation setup and the memory traces are processed by static and dynamic memory tracing. This results in per library memory traces, which are graphically illustrated in the following. In addition, analytic measure are conducted on the semantically separated memory traces. Determining the achieved endurance (AE) of single libraries provides an intuition of how effective wear-leveling could be performed for different libraries and for which library the effort for wear-leveling may pay out best.

Static Memory Tracing

Figure 4.10 provides a graphical illustration of the semantically enriched results for the static memory tracing. The graphs basically show the full system simulation output, i.e. increasing memory addresses on the x-axis and the amount of write accesses[1] to the specific memory locations on the y-axis. The black line in the plots illustrates the simulation result, i.e. the amount of write accesses per memory byte. The colorful background illustrates the result from static memory tracing, i.e. which portion of the presented data and bss segment belongs to which library.

Investigating the displayed results, it can be observed that the application (app) itself has no present memory in the data and bss segment. This stems from the fact that the app is only coordinating calls to the sqlite implementation in this scenario

[1] The evaluation for this method is entirely conducted for the case of write destructive memories. It should be noted that the methods are fully transferable to read destructive memories without further limitations.

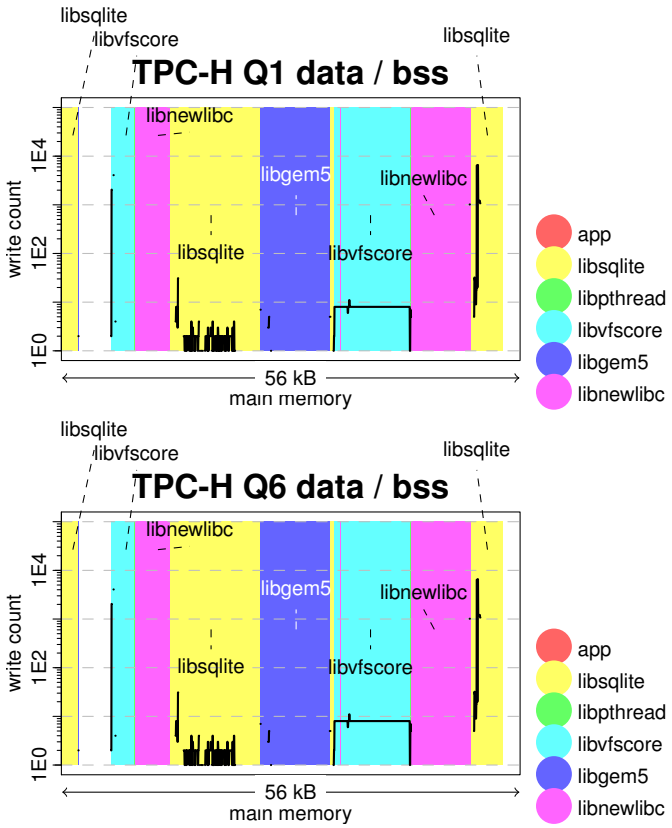


Figure 4.10: Data / BSS Section for TPC-H Q1 and Q6

and has no large memory demand. It can be further observed, that the library for the sqlite implementation (libsqlite) and the library for the virtual file system (libvfscore) feature the majority of memory accesses. While the sqlite implementation faces more uneven and densely peaked memory accesses and hence may require a certain fine-grained wear-leveling, the file system implementation faces evenly distributed memory accesses across the allocated memory space and hence may only require a lightweight coarse-grained wear-leveling.

Dynamic Memory Tracing

As previously discussed, the results for static memory tracing only reveal the memory ownership, but do not guarantee that the owning library is also responsible for the accesses. In addition, dynamically allocated segments (stack and heap) cannot be semantically enriched. Figure 4.11 illustrates the derived results for dynamic memory tracing for the stack segment[2]. Again, the x-axis denotes memory space, while the

[2]The analysis can work without any modifications for a heap segment as well. However, the used configuration of unikraft and of the libraries do not include any dynamic memory management and thus no heap segment

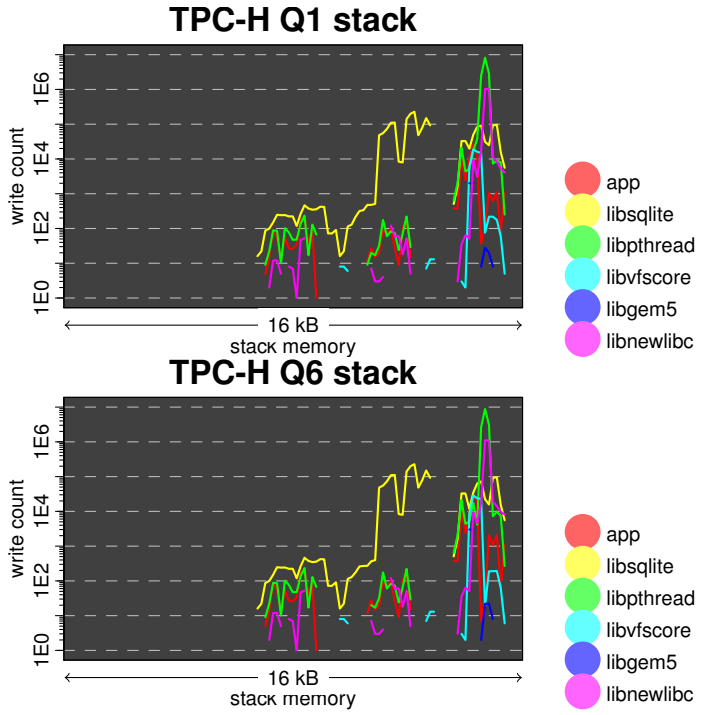


Figure 4.11: Stack Section for TPC-H Q1 and Q6

y-axis indicates the amount of write accesses per memory location (byte). In contrast to static memory tracing, where the memory space can be segmented by the owning libraries, each library has another trace across the memory space, which is indicated by different color lines in the plots.

Investigating the presented results, it can be observed that most of the libraries operate on overlapping segments of the stack memory. This stems from the basic usage principle of stack memory: Function calls allocate reverse order growing parts of the stack. When the call is done and another function is called, the same parts of the stack can be allocated by another function from another library. Bridging the gap to analyzing the possible demand of wear-leveling, it can be again observed that some libraries (i.e. libsqlite) face a closely uniform memory wear-out and hence could be equipped with lightweight coarse-grained stack wear-leveling, while other libraries (i.e. libpthread) induce dense peak usage of small portions of the stack segment and cause the demand for fine-grained stack wear-leveling. It is worth noticing, and somewhat counterintuitive, that the analysis reveals two different wear-leveling demands for the data / bss and stack segment of the sqlite implementation.

As mentioned before, the dynamic analysis is not limited to be applied on dynamically allocated memory segments and can reveal a relation between memory ownership and memory usage when being applied to static allocated memory segments. Figure 4.12 illustrates the corresponding results for the dynamic analysis of the data and bss segment. It can be observed, that most of the libraries have a strong similarity between

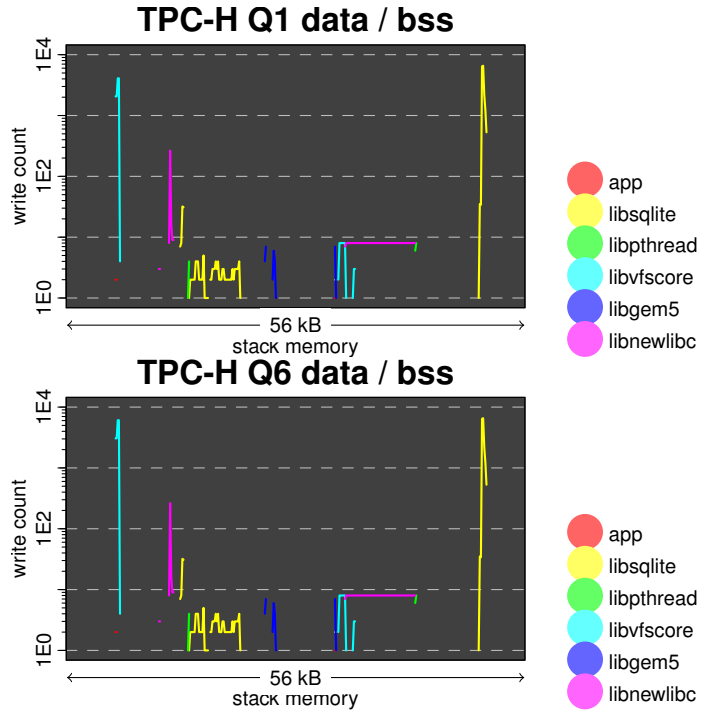


Figure 4.12: Dynamic Data / BSS Section for TPC-H Q1 and Q6

	app	libsqlite	libpthread	libvfscore	libgem5	libnewlibc
Q1						
stack	2.44%	7.80%	1.53%	2.74%	1.87%	1.89%
data	1.17%	0.69%	0.62%	0.79%	1.27%	0.47%
Q6						
stack	2.53%	7.47%	1.52%	2.73%	1.85%	1.90%
data	1.17%	0.69%	0.62%	0.79%	1.27%	0.40%

Table 4.2: achieved endurance after dynamic analysis

memory ownership and memory usage, few memory portions, however, are intensively used by other libraries. The largest difference can be observed for the virtual file system, where a majority of the owned memory is accessed by the c library implementation (libnewlibc).

In addition to the graphical illustration, as mentioned above, analytic measures can be conducted on the semantically separated memory traces. For the case of dynamic memory analysis and the measure of the achieved endurance (AE), this is performed, and the results are shown in Table 4.2. The table shows the computed AE values for each library each for the stack and data segment for the two cases of the TPC-H Q1 and Q6 workload. The calculation of the AE is done across the entire data / stack segment and not limited to the owned memory regions. As the achieved endurance provides an

intuition of how far lifetime of NVM can be extended by proper wear-leveling, it can be observed that libpthread reveals the highest optimization potential for the stack memory, while libnewlibc reveals the highest optimization potential for the data memory. The data also support the finding mentioned before, that libsqlite uses the stack memory way more evenly than the data memory. The collection of these findings can assist in the proper configuration and application of wear-leveling methods in order to apply an efficient, yet effective wear-leveling scheme.

4.4.5 Wrap-Up

Semantic memory tracing, as presented in this section, is an orthogonal extension for wear-leveling in NVM systems. It does not provide any wear-leveling methodology, but rather a semantically enriched view on the memory behavior of the application together with the required OPERATING SYSTEM (OS) components. To achieve this, debugging information of the compiler in the build process of the unikernel unikraft are used to map memory locations to owning libraries in static memory tracing. In order to also account for dynamically allocated memory segments and to not only focus on memory ownership, but also on the causing software components of memory accesses, dynamic memory tracing is implemented by considering the PC for every single memory access additionally. An illustrative use case highlights how the semantic memory tracing could be used on a typical database workload in order to derive an efficient, yet effective wear-leveling scheme, which is dedicated to the individual demands of the single libraries.

4.5 Concluding Interface Exploitation

Towards exploitation of the traditional memory interface, this chapter introduces the concept of a central software instance in the operating system, which provides additional interface functionality to enable lifetime maintenance of NVM. The central software instance keeps an internal incremental representation of memory age, which allows the making of wear-leveling decisions. The appropriate decisions are executed by modifying the locations of future memory accesses in an aging aware manner. By only exploiting this traditional interface by this software instance, the functionality of lifetime management for NVM can be provided. On the cost of requiring a set of available hardware features (performance monitoring and MMU), this approach can realize lifetime management without any corporation from the application. Even the source code of the application is not necessarily required.

CHAPTER 5

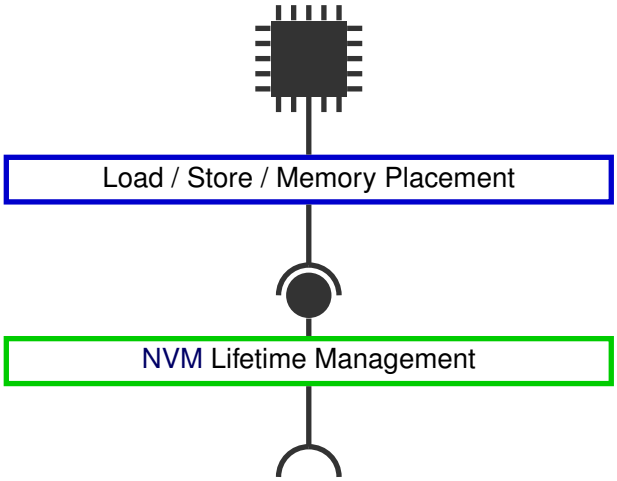
Application-Cooperative NVM Wear-Leveling

Contents

5.1 Modern Technologies and Traditional Interfaces 80
5.2 Overview 80
5.2.1 Application-Cooperative Decisions 81
5.2.2 Application-Cooperative Actions 82
5.3 Stack Usage Analysis and Wear-Leveling Hints 82
5.3.1 Scope 83
5.3.2 Problem Analysis and Statement 83
5.3.3 Stack Usage Analysis 84
5.3.4 Stack Wear-Leveling Overhead Optimization 85
5.3.5 Evaluation 86
5.3.6 Wrap-Up 89
5.4 B^+-Tree Checkpoint Wear-Leveling 90
5.4.1 Scope 90
5.4.2 Problem Analysis and Statement 91
5.4.3 B^+-Tree Organization 92
5.4.4 OCTO$^+$ Algorithm 92
5.4.5 Evaluation 95
5.4.6 Wrap-Up 98
5.5 Concluding Software-Based Wear-Leveling 98

5.1 Modern Technologies and Traditional Interfaces

While the previous chapter introduces methodology to enable memory lifetime management through the traditional memory interface in an application transparent manner by employing a centralized software instance, this chapter aims for conceptually the same interface exploitation, but with an entirely different approach:



Instead of employing a centralized software instance, applications are directly modified in the following and the management of the memory lifetime is performed directly within application operations. Partially, this can omit the maintenance of an internal aging model, since the application operations directly contribute to partially incremental wear-leveling. Although this approach differs largely from the previously presented centrally-managed technique, both offer a possibility to software exploitation of the traditional memory interface for NVM lifetime maintenance. In addition, both methods can be applied jointly in an adequate situation.

5.2 Overview

The general process of wear-leveling, including the required division into corresponding decisions and actions is introduced in Chapter 4. In addition to the general concept, the chapter introduces application transparent wear-leveling mechanisms in a software-based manner, i.e. which require no special cooperation from the executed application. This makes the implementation of complex wear-leveling mechanisms unavoidable, which employ means for the making of wear-leveling decision and actions, without any knowledge of the application. To overcome the need for such complex, resource consuming mechanisms, in case the application is well known, the source code is available, and it can be modified, application cooperative wear-leveling schemes can be considered, which include the application into the wear-leveling process. Generally,

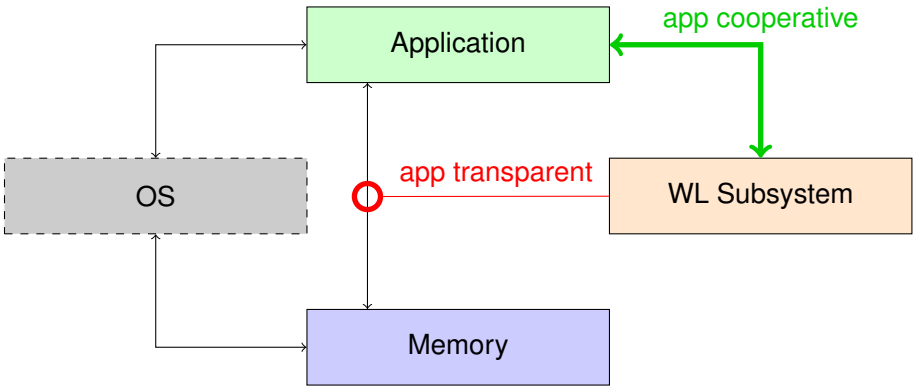


Figure 5.1: Application Cooperative Wear-Leveling Principle

the rough categorization of wear-leveling decisions and wear-leveling actions can be kept for this. This can be imagined as two directions of cooperation of the application: When the application *provides* information about memory usage, this realizes application cooperative wear-leveling decisions. When the application *receives* instructions to change memory locations or change the memory usage of certain locations, the application realizes wear-leveling actions. The concept of application cooperative versus application transparent wear-leveling is illustrated in Figure 5.1. While for application transparent wear-leveling, the wear-leveling system has to estimate the memory usage of the application with the help of the OS, application cooperative wear-leveling allows the application to directly interact with the wear-leveling system in order to perform decisions and actions.

5.2.1 Application-Cooperative Decisions

The crucial aspect of application cooperative wear-leveling decisions is the exploitation of domain specific knowledge about the application. Wear-leveling decisions should be made in a way, such that the allover wear-out of memory is reduced, and the total lifetime is extended. It is further of strong interest to keep overheads low due to efficient making of wear-leveling decisions. The latter goal makes the usage of aging-aware wear-leveling highly interesting, which requires knowledge about the memory usage of the application to a certain degree. In case of application transparent wear-leveling, this knowledge has to be gathered in a very general way, e.g. as presented in Section 4.3. This however, requires unavoidable large overheads. Similar information can be gathered in a highly efficient way directly from the application. Consider the example of a tree based data structure, which forms the major part of an application and where tree nodes are allocated in main memory. While application transparent wear-leveling only sees bare memory and can build an access histogram on the global memory space, all tree inference implementations could be easily modified in such a way that for every access a histogram for the specific node is updated. This then can form a global access histogram for the tree. Not only can such a way of gathering memory access information

be more precise, but more efficient, since accesses can be directly captured and do not need to be sampled, e.g. with the help of interrupts.

In addition to bare efficiency of information collection due to application cooperative wear-leveling decisions, domain specific information can further allow making wear-leveling decisions based on knowledge, which is not available in an application transparent manner. Considering the tree based data structure example again, the application knows the semantics of the tree structure well. E.g. when a single node is accessed, the application can know that a future memory access will target one of the child nodes. The application could further provide a probabilistic model of the access frequency distribution of child nodes. This can be used by the wear-leveling subsystem to form a global probabilistic model and anticipate memory wear-out directly with the application of wear-leveling actions, following the probabilistic mode.

5.2.2 Application-Cooperative Actions

Similarly to application cooperative wear-leveling decisions, also the performed actions of the war-leveling subsystem can be assisted in an application cooperative manner. The process of wear-leveling actions usually aims to alter the target of memory access operations, such that the usage and hence the wear-out can be evenly distributed across the memory space. These actions are triggered based on the corresponding wear-leveling decisions. In an application transparent manner, wear-leveling actions have to use hardware assisted features, like MMU page mapping, or have to hook into compiler managed structures, like the text or stack management. This comes with natural limitations, e.g. the page granularity of the MMU and with possibly induced high overheads. For instance, relocation of the stack or text memory require a large overhead and sophisticated management to maintain the semantic correctness.

In an application cooperative manner, memory is managed often directly by the application. Considering again the example of a tree based data structure, the nodes are allocated in memory and linked with pointers. Changing the location of a node in memory only requires a lightweight operation of a pointer update. Furthermore, the granularity of such operations can be significantly finer compared to e.g. MMU operations. Even though the application may not manage all the allocated memory in a way that allows such lightweight application cooperative memory relocation, the application can further assist the application transparent principles from the wear-leveling subsystem. Providing information about unused or invalid memory can reduce the overhead for MMU relocation. Providing hints before calling of long-lasting functions can reduce overheads for stack based relocation.

5.3 Stack Usage Analysis and Wear-Leveling Hints

This section introduces an extension to application transparent wear-leveling (Section 4.3). In greater detail, the approach for fine-grained stack wear-leveling is refined.

The method in this section aims to reduce the spent overhead by triggering copies of the stack to other memory locations whenever the stack is small. The retrieval of this information makes the method application cooperative. The source code is modified in order to notify the wear-leveling system when the stack is small, such that synchronous wear-leveling actions can be triggered. However, in order to provide a method, which does not depend on heavy domain specific knowledge, i.e. knowing the exact stack sizes, machine-learning models are also used to predict the stack usage of an application in a not invasive way.

5.3.1 Scope

This section covers implemented methods for systematic analysis of the stack size of an application. The stack sizes are determined by employing a special analysis framework, based on self-hosted debugging capabilities of the hardware and the OS. This framework determines the current size of the stack for a given instruction in the application over repetitive executions. This framework forms part of the fitness function of a GA, which determines instructions for the low overhead triggering of wear-leveling actions for the stack. These instructions are then enriched with annotations and inserted into the application source code and compiled again. In short, the method covers following elements:

- A ptrace based stack analysis framework for stack size recording of single instructions
- A GA for finding instructions with low stack sizes
- An annotation mechanism to trigger stack wear-leveling at specific instructions

5.3.2 Problem Analysis and Statement

Software based wear-leveling for stack memory can become a resource intensive task, since due to the usage semantics of stack memory, the entire actively used stack region has to be relocated to a new consistent memory locations. Especially due to the need of fine-grained wear-leveling, copying of the active stack region becomes necessary. In consequence, the overhead for such wear-leveling operations is strongly coupled with the size of the active stack region. Depending on the structure of the application, the number and depth of function calls and the use of local stack memory, applications can feature highly varying sizes of active stack memory during their execution. For the objective of minimizing overheads for stack wear-leveling, it is favorable to perform wear-leveling actions on the stack at points during the execution, when the stack size is small. In order to surpass overheads for runtime monitoring of the stack, such points during execution should be determined in advance. In addition, when the analysis is performed offline, complex analysis methods can be used to configure the wear-leveling

published work: [HYC+19]

mechanism to the determined stack sizes in order to provide a trade-off between wear-leveling quality and caused overheads.

Consequently, the problem is stated twofold: 1) the problem is to analyze the stack usage of an application in an offline manner, such that the stack size at specific points during the execution can be predicted. As points during execution, instructions of the application are used. 2) the problem is to determine instructions in the application, where the stack size is predictably small, such that wear-leveling actions should be ideally triggered synchronous to the execution of these instructions.

5.3.3 Stack Usage Analysis

As motivated above, determining the stack size at a certain point (i.e. instruction of a program) is one core problem to be solved. Hence, the input to this problem is a target program and one instruction of the program. The output of the desired analysis component is the stack size (cumulative or average) at the different executions of the target instruction. This section details the implementation of such an analysis component, which is based on the linux API ptrace [Man19]. By utilizing this library, target programs can be executed in a native linux environment, which allows the use of powerful and highly parallelized hardware. If, for instance, the target system is an embedded system with a single ARMv8 core, the ptrace based analysis can be run on a many core ARMv8 server CPU and deliver fast results for the subsequent optimization.

Ptrace Based Analysis

The linux API ptrace [Man19] provides means to debug user level applications. As one part of the API, registers and memory can be read and modified at any point during the execution of the target program. In addition, the program can be interrupted at specific events, in order to perform memory or register reads and modifications at these events. As common debuggers, ptrace supports a single step mode, where the target application is interrupted after every single instruction and the analysis tool could read registers. In addition, the use of debug instructions is supported. In order to use this mechanism, a debug instruction must be present in the binary code segment of the target application and be executed. The execution of this breakpoint causes an interrupt of the target application and the invocation of the analysis tool. Since ptrace allows modifications of the memory of the target application, breakpoints can be inserted by replacing an arbitrary instruction of the target application. Whenever the breakpoint is executed, it has to be replaced with the original instruction and the PC has to be reverted to the previous instruction.

In the context of the stack size analysis, it is desired to analyze the stack size at the moment of a specific instruction being executed. Consequently, overheads can be saved by using breakpoint instructions instead of single step debugging. Additionally, stack modifications are deterministic with the instruction stream and not affected by pipelining or out of order execution effects. Consequently, the interrupts due to breakpoint

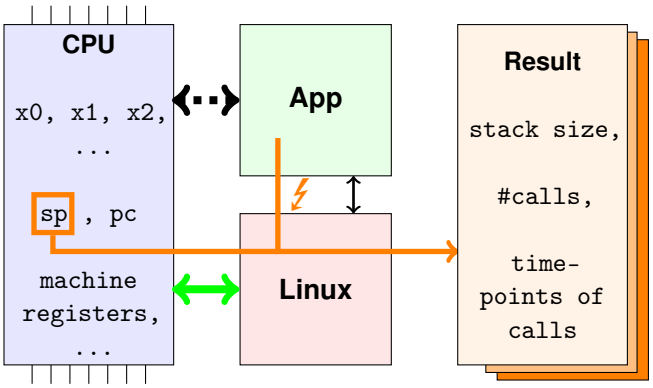


Figure 5.2: Ptrace based analysis framework for instruction evaluation

instructions and the modifications of the code segment in order to introduce breakpoint instructions do not affect the stack usage, since the execution of breakpoint instructions does not modify the stack.

Analysis Module

The definition of the studied problem states a program and an instruction of interest as the input to the analysis module. As the output, the stack size during the execution of the specific instruction is desired. The stack size can be determined by reading the SP register at the execution of the instruction and subtract the base address of the stack from the value. The value of the SP is read by employing the breakpoint mechanism of ptrace, as described before. The analyzed instruction is replaced by a breakpoint instruction. When this breakpoint is executed, the SP register is read and the stack size is calculated and stored. Subsequently, the breakpoint instruction is replaced by the original instruction and the PC is reverted. The execution continues in single step mode for exactly one instruction. Afterwards the target instruction is again replaced by a breakpoint instruction and the execution continues normally. This process continues until the target program terminates or a specified timeout is reached. The analysis module the returns the cumulative stack size, the number of calls of the instruction and the time-points of the calls of the instruction as a result. This analysis workflow is illustrated in Figure 5.2.

5.3.4 Stack Wear-Leveling Overhead Optimization

The second part of the problem statement is to find instructions in the target application, which promise to reduce the overhead for stack wear-leveling, when wear-leveling actions are triggered synchronous to these instructions. Consequently, instructions with a predictably small stack size should be used as a premier target. However, in order to maintain the wear-leveling quality, it has to be ensured that wear-leveling actions are executed with a certain target frequency. This frequency is denoted as f_{wl} in the

following, where a value of $f_{wl} = 1000$, for instance, means that in average after the execution of 1000 instructions a wear-leveling action should be performed. Consequently, the objective is to find a set of instructions, which feature a minimal stack size and are called in average with a frequency of f_{wl}. This is pursued by employing a GA with domain specific post-processing steps. The population is formed by set of assembly instruction from the target program $P = \{a_0, a_1, ..., a_n\}$, which are initially random sampled and limited to the population size n. The fitness value for each population element is computed as the average stack size, derived from the stack size analysis module (Section 5.3.3). In a first step after evaluating the fitness, an elitism strategy is applied and k the best instructions are kept in the population. Subsequently, the population is further pruned by eliminating instructions, which are called too frequent within f_{wl}. The analysis module delivers the time points of the execution of the analyzed instructions. Whenever one instruction is called within f_{wl} instructions, other instructions, which are called within the same f_{wl} are removed, unless they are the only instruction being called within another f_{wl}. Mutations are performed by adding random bounded offsets to population items, recombination are achieved by selecting the arithmetic mean instruction address between two random population items. For recombined and mutated population elements, a sanity check ensures that the address is a valid instruction address of the target program. Mutation and recombination is omitted for the final iteration of the GA.

5.3.5 Evaluation

In order to empirically evaluate the effectiveness of the presented stack wear-leveling optimization, two aspects have to be considered. First, an empiric case study on benchmark applications is conducted, where the stack usage is analyzed over time. This reveals the potential for overhead optimization. Second, wear-leveling together with the overhead optimization is evaluated end to end, and the caused overhead is recorded. This studies the overall effectiveness of the presented approach.

Stack Usage Analysis

A slightly modified version of the stack analysis module (Section 5.3.3) is used in the following to obtain the stack size during the execution of benchmark applications. Rather than obtaining the cumulative stack size across various calls of a single instruction, the stack size at every executed instruction is recorded. Even if the same instruction is executed multiple times, this results in two data points in the result. Hence, the result forms a mapping $f : \mathbb{N}_0 \to \mathbb{N}_0$, which maps points in time during execution (i.e. the instruction cycle) to the current stack size. This analysis is conducted for a set of benchmark applications, which are taken for parts from the MiBench suite [GRE+01] and for other parts are self written implementations of various algorithms. The resulting stack size distribution for a subset of the benchmark applications is illustrated in Figure 5.3. Generally, the observed stack usage distribution can be described as three sub patterns:

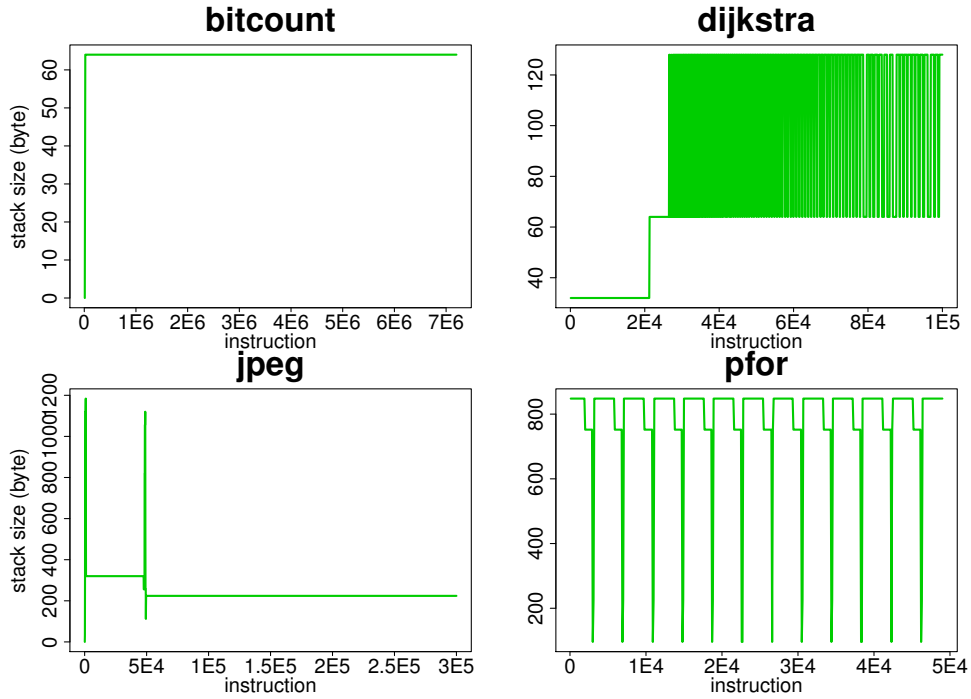


Figure 5.3: Exemplary stack usage analysis

1. **Constant**: As can be exemplary seen in the bitcount benchmark in Figure 5.3, the size of the used stack does not change significantly over the execution of the majority of the application and thus can be described as constant.
2. **Periodic**: For some applications, as for instance the pfor benchmark in Figure 5.3, the changes of the used stack size are periodic throughout the execution of the application. This can be caused by periodic invocations of the same or similar parts of the code.
3. **Irregular**: When the stack usage over time is neither constant, nor periodic, it can be described as irregular. In Figure 5.3, the dijkstra benchmark features such a pattern, since the stack usage changes with a varying frequency.

Since some benchmark applications do not feature one of the aforementioned patterns during the entire execution, categorizing benchmark applications to these classes is not obvious. The dijkstra benchmark, for instance, features constant stack usage as first, then changes to irregular usage. Considering, however, the major present class in each benchmark, the categorization in Table 5.1 can be made. In this table, the number in the braces behind the benchmark denotes the maximum change in the used stack during the benchmark execution. Generally, it can be said that benchmark applications with constant stack usage patterns are not likely to be a good candidate for overhead optimization of stack wear-leveling. Although the recorded data for the classification is just an artificial run and the stack usage may differ in reality, no meaningful points during the execution can be determined, which are promising to reduce the overhead.

Constant:	Bitcount (0), Susan (10224), Jpeg (1072), Patricia (240), CRC32 (1072), Blowfish (304)
Periodic:	Basicmath (688), Stringsearch (64), Adapcm (112) Pfor (752), Sha (176), FFT (176)
Irregular:	Dijkstra (688)

Table 5.1: Categorization of 13 embedded applications

In contrast, benchmark applications with periodic or irregular usage patterns offer the potential for overhead optimization by triggering stack wear-leveling actions exactly when the stack is small. For applications with periodic usage patterns, this can be likely achieved by triggering stack wear-leveling actions synchronous to a few instructions, which are executed periodically. This helps to keep the overhead for triggering the wear-leveling actions small. For applications with irregular patterns, wear-leveling actions may need to be triggered synchronous to multiple instructions in order to ensure wear-leveling at low stack sizes on the desired target frequency. This potentially causes a larger overhead for ensuring the synchronous wear-leveling actions. Regardless of periodic or irregular stack usage patterns, the maximal difference in the stack usage can further help to assess the optimization potential. Considering two examples of benchmarks with periodic stack usage, pfor promises to achieve a higher overhead reduction than stringsearch.

Stack Wear-Leveling Overhead Optimization

As highlighted before, several benchmark applications promise to allow a reduction of the overhead for stack wear-leveling due to not constant usage of stack memory during execution. The method described in Section 5.3.4 allows to determine instructions, which should be used to synchronously trigger wear-leveling actions to target this overhead reduction. In order to provide an end to end evaluation, the results from the GA are fed beck to the benchmark application in the form of a relocation hint. A relocation hint is a small set of instructions, which is inserted into the source code of the benchmark application at the specific instruction, which is determined by the GA for low overhead stack wear-leveling. The relocation hint triggers the wear-leveling subsystem and allows it to trigger a stack wear-leveling action, if this fits to the desired target frequency f_{wl}. The instructions of the relocation hint maintain the stack size after completion, hence, introducing relocation hints does not impact the stack usage of the benchmark application. It should be noted, that for analysis and determination of the relocation hint, the linux ptrace based framework (Section 5.3.3) is used. For evaluation, assessment of the end to end overheads and wear-leveling quality, the relocation hints are deployed to the wear-leveling subsystem from Section 4.3.4 and simulated in the setup from Section 3.1.5.

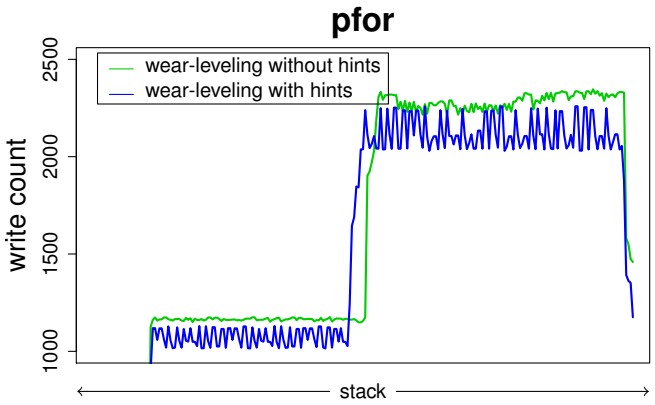


Figure 5.4: Memory write access distribution

As the target application for evaluation, the pfor benchmark is chosen. This benchmark implements the PFOR compression algorithm [ZHN+06]. The benchmark runs decompression in a stream-like manner and applied subsequent aggregation of the decompressed results. In order to analyze optimal points for stack wear-leveling, the GA from Section 5.3.4 is configured with a population size of $n = 200$, an elitism strategy of $k = 100$ and a target frequency of $f_{wl} = 10000$. The boundary for random mutations is set to 50 instructions in either positive or negative direction. Deploying stack wear-leveling without any overhead optimization purely timer driven, an overhead in terms of write accesses of 9.96% is caused. Introducing the predetermined relocation hints reduces the cause overhead to 1.41%, which is a reduction of 85.83%. In addition to the reduction of overhead, the wear-leveling quality should also be checked. Figure 5.4 shows the amount of write accesses across the stack without any optimized overhead (green) and with optimized overhead due to relocation hints(blue). It can be observed that the write distribution does not cause higher peaks after the optimization of overhead, which indicates that the wear-leveling quality is not negatively impacted by the overhead optimization. Contrarily, the amount of write accesses are reduced for most parts of the stack due to the overhead optimization, which further extends the memory lifetime.

5.3.6 Wrap-Up

Wear-Leveling on the stack memory can achieve significant lifetime extensions of NVM due to the reduction of heavy memory access peaks. However, the caused overhead can be immense, since in order to maintain the access semantics, the entire used stack memory has to be relocated to a new position in memory. Investigating the stack usage over time of certain benchmark applications, it can be observed that the stack memory does not allocate the same amount of memory during the execution and hence, several moments during execution are more optimal for stack wear-leveling actions, than others. The work presented in this section details a framework to conduct such a stack size analysis on an offline fashion. The analysis can be massively parallelized

and be executed in linux host systems. Based on this analysis, an optimization process, including a specialized GA is invoked, which determines instructions in a benchmark application, which should be used as a synchronous trigger for stack wear-leveling actions. Including the resulting wear-leveling hints to an end to end evaluation show to reduce the overhead for stack wear-leveling by 85% and not negatively impact the wear-leveling result.

5.4 B$^+$-Tree Checkpoint Wear-Leveling

In this section, an example of a deeply integrated application cooperative wear-leveling scheme is discussed. While Section 5.3 introduces a scheme, which is application cooperative but also generally applicable to arbitrary applications, this section focuses on a specific application and provides wear-leveling means, which are dedicated to this specific application. By exploiting the specific structure of the application, an extremely low overhead wear-leveling scheme can be realized. In this approach, B$^+$-trees [Bay72] are considered on a hybrid memory system, which regularly performs checkpoints between the volatile and non-volatile memory. The wear-leveling scheme operates integrated into this checkpointing mechanism.

5.4.1 Scope

This section details a wear-leveling method for B$^+$-trees, running on a hybrid memory system. In detail, the database system, including instances of B$^+$-trees is assumed to operate on volatile memory, as for instance DRAM without specific endurance considerations. In order to ensure persistency and aid robustness, a write back like scheme is employed, where the latest modified version of the B$^+$-tree is copied to the NVM in the form of a checkpoint. Although the database system can operate such a strategy on arbitrary data structures, B$^+$-trees as a central indexing structure in many database systems are studied as a highly important case here. The checkpointing of a B$^+$-tree basically copies tree nodes to NVM blocks. Since the tree nodes may not be stored continuously in the volatile memory, a mapping of volatile memory locations to non-volatile memory locations is applied. This mapping can be realized as a simple mapping table between B$^+$-tree blocks and NVM locations. Wear-leveling can be realized with extremely low overhead by altering this mapping table between two checkpoints in an aging-aware manner. In order to make realize such aging-aware modifications, a modified B$^+$-tree implementation is realized, which tracks modified regions of tree nodes on insert and update operations. Considering the accumulated modification information on the NVM side allows assessing the current aging of NVM regions. Taking the local modification information of a tree node into account allows to find an optimized mapping of tree nodes to memory locations, which reduces the total memory wear-out. This approach

published work: [HKC+21b]

is implemented and evaluated in this work. In short, the implementation covers the following aspects:

- A modified B+-tree implementation, which maintains a modification information within tree nodes, which is updated in a lightweight manner during updates and inserts.
- A wear-leveling implementation, utilizing the collected modification information during checkpoints to apply a wear optimized mapping.

5.4.2 Problem Analysis and Statement

Considering a hybrid memory system, where a database system executes a B+-tree in volatile memory and regularly checkpoints the tree to NVM, the problem of wear-leveling NVM in order to extend the system lifetime focuses on the checkpoint operation. In this context, a pair of coupled states needs to be distinguished: 1) memory contents in the volatile memory can be either modified or unmodified, 2) physical locations on the NVM can be older / more worn out or younger / less worn out. When writing a checkpoint from volatile memory to NVM, unmodified memory contents likely will not impact the age of a NVM location under the assumption of an iterative memory update scheme. Writing modified memory contents, in contrast, likely ages the NVM due to the required changes of stored memory contents. The information about modified contents of the volatile memory is collected by tracking tree update and insert operations in the following. The aging of the NVM is tracked by accumulating the amount of updated memory contents written at a checkpoint.

The realization of wear-leveling considered in the following operates by employing a mapping between tree nodes in volatile memory and in NVM. Without any weaer-leveling actions, this mapping is constant during the entire execution and tree nodes are always checkpointed to the same memory location. When a wear-leveling action of a change to the mapping is applied, the corresponding mapping is applied within the next checkpoint. The mapping itself can be stored as a simple table, in the form of a metadata block of the B+-tree. This block can be considered as every other data block of the tree for checkpointing and wear-leveling. Only when restoring a checkpoint of the tree from NVM to volatile memory, the offset of the mapping table needs to be stored at a central location.

Although modifications of the mapping table open a space for wear-leveling operations to improve the memory wear-out, a modification of the mapping table potentially causes a high overhead in terms of memory lifetime. With a modification of the mapping table, tree nodes will be stored at NVM locations in the next checkpoint, which previously contained different memory contents. Thus, potentially a high amount of changed information needs to be written, which wears out the memory. Hence, the problem of wear-leveling in this context is stated to modify the mapping of checkpoints carefully in an age extending manner, without causes excessive age overheads.

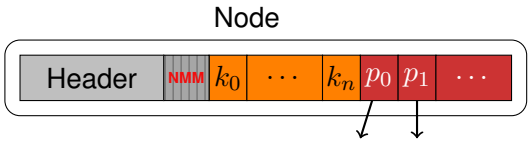


Figure 5.5: Layout of a B$^+$-tree node

5.4.3 B$^+$-Tree Organization

Since the collection of information about modifications in B$^+$-tree nodes is a crucial component for the checkpoint wear-leveling, the layout and organization of the B$^+$-tree nodes is discussed here. From a logic perspective, B$^+$-trees are tree data structures, where every node can have multiple child nodes. Although this number must not be constant within the tree, an implementation with fixed sizes is often easier and more efficient in terms of memory overhead. Hence, for this method, a B$^+$-tree with fixed node sizes is considered. Each node then stores two central components: 1) an array of keys and 2) an array of pointers. In these arrays, the pointer at index i belongs to the key at index i. While executing the tree, the lookup or insert key is compared to the key array and the child node is visited, which is pointed to by the corresponding pointer. In order to provide a fully working implementation, some additional information needs to be stored in each node, including the fill level and the information weather this node is a leaf or inner node. This additional information is stored in a header part of each tree node.

In order to gain control over the memory layout, the tree is implemented in C. The nodes are stored as packed memory arrays, such that the memory layout of tree nodes results in a scheme, illustrated in Figure 5.5. The header, the key array and the pointer array are stored in contiguous memory locations. When a modification operation happens to the tree, as for instance an insert operation, the key array, the pointer array and parts of the header may be modified. Since the key array is stored in a sorted manner and a newly inserted key must be inserted in a sorted manner as well, larger parts of these memory portions may be modified. Hence, the collection of the information about node modifications is directly integrated into the tree implementation. On this level, the information about modified parts of the key and pointer array and of the header can be collected with a low overhead. The tree implementation then stores a *node modification map*, indicated by nmm in Figure 5.5. This map is a binary vector where a 1 encodes that a portion of the node is modified and a 0 indicates unmodified, respectively. Since the node modification map is only relevant to the volatile version of the tree node and is further evaluated for the wear-leveling algorithm, it is omitted while storing checkpoints of the tree nodes.

5.4.4 OCTO$^+$ Algorithm

As mentioned above, the process of wear-leveling in the context of the introduced setup is to map B$^+$-tree nodes to NVM locations for their checkpoint in an aging aware manner.

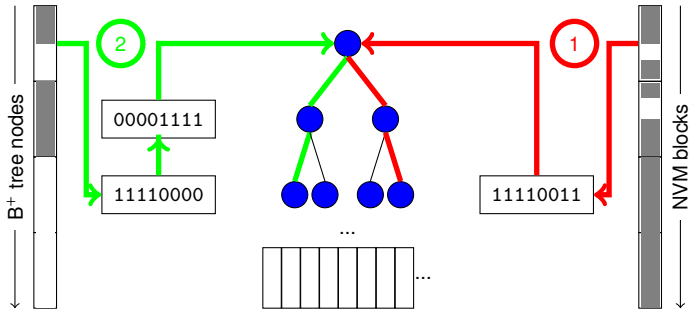


Figure 5.6: OCTO⁺ tree based algorithm

Towards this, the OCTO⁺ algorithm (Optimized Checkpointing for B⁺-Trees) aims to achieve two objectives: 1) intra block wear-leveling, where the distribution of modified and unmodified memory portions within every single node is matched to a corresponding NVM block, such that the block is uniformly aged and 2) inter block wear-leveling where the total age of all NVM blocks is uniformly leveled.

Write Information Collection

The basic principle of the collection of modification information is already explained in Section 5.4.3. After every checkpoint, the node modification map or node modification map (nmm) is reset to all 0. Read operations result in no modification of this map. The nmm is a fixed length bit vector, where every bit indicates modifications to the corresponding equi-sized memory region of the tree node. Insert and update operations may flip the corresponding modification bit of the header to 1, if the header is modified. Further, the corresponding bits of the key and pointer arrays are set to 1 for the modified parts. If modifications occur to a certain part of the tree node, where the corresponding modification bit is already set to 1 no further action is taken. Double modifications are not explicitly reflected, since in between two checkpoints the number of modifications to the volatile memory version of the tree is irrelevant.

The size of the node modification map, and thus the granularity, can be chosen arbitrarily between the full node size (every bit in the nmm would represent one bit of the node then) down to a single bit, representing the entire node. Choosing the right granularity opens a trade-off between preciseness of the collected information and overheads for storing and maintaining the nmm. For this work, the size of the nmm is chosen as 8 bit for two reasons. First, the mask forms a single byte, which can be efficiently packed into the header data structure of the node implementation. Second, the aging of the NVM is estimated by accumulating the nmms of the checkpointed nodes to a certain NVM location. By using 8 bit counters for this accumulation, the age estimation every tree node sized NVM block can be stored within 64 bit, which is the usual word width of a CPU. Once a single of these counters overflows, all counters are divided by 2, which maintains the relative order of the aging.

Remapping Decisions

With the aforementioned procedure of maintaining the nmm on modifications and maintaining the accumulated value on checkpoints as the NVM age, each checkpoint allows identifying the need for wear-leveling actions and to perform these. The general process of making these decisions is illustrated in Figure 5.6. This process is executed in advance to every checkpoint to potentially update the mapping table. In a first step, blocks in the NVM are identified, which require wear-leveling actions. This is done by computing the average of the accumulated counters within each block and check if the maximum counter value exceed the average by a configurable threshold. Next, the accumulated counters are reduced to an 8 bit bit mask, where a value of 1 indicates that the counter exceeds the average value of the block. The blocks are then further inserted into a binary tree, evaluating the determined bit mask bit by bit in each layer. In consequence, the released NVM blocks are inserted into one of 256 lists, where the determined bit mask is the index of the list.

In the next step, the B$^+$-tree nodes, which were mapped to the released NVM blocks are shuffled to new blocks by inverting the current nmm and looking the mapping tree up for a suitable NVM block. In detail, of the nmm has a 1 at index i, the node has modifications at this piece of memory and potentially causes wear-out at this location. Due to the inversion, the tree node is ideally mapped to an NVM block, which has a 0 in the bit mask at index i, meaning that the counter for this memory location was less than the average, i.e. the block is relatively young at this position. Ideally, this process maps B$^+$-tree nodes in an aging-aware manner to NVM locations. It also has to be considered that a perfectly matching NVM block does not exist for a tree node. To accommodate for this, the mapping tree maintains counters in every node of the available unmapped NVM blocks under this node. If then the ideal mapping would choose to visit the right child node, for instance, but no blocks are left under the right child, the left child is taken instead. It should also be noted that this process performs intra block wear-leveling only, i.e. the relative age between NVM blocks is not considered at all.

In order to extend the algorithm for inter block wear-leveling, the absolute values of the accumulated age counters for all NVM blocks are further compared at every checkpoint. For every node, the maximum value is determined. The youngest and the oldest block, according to this maximal value, are then further swapped in their mapped B$^+$-tree node when the maximal and minimal age differ more than a configurable threshold. It should be noted that all the computations for the intra and inter block wear-leveling can be done in a single scan of the NVM blocks. Inserting NVM blocks and tree nodes into the mapping tree has a constant time complexity of 8 steps for each block / node.

Although the algorithm, presented in this section, aims to perform wear-leveling on a rather fine granularity within B$^+$-tree nodes, the practical granularity is limited by a meaningful chosen size for the nmm. Although such as granularity suffices for wear-leveling, a special case of non-uniform memory usage happens within data words. Consider, for instance, an integer numeric value stored in a 64 bit variable, but only

storing value between 0 and 255. One byte of the word would face heavy wear-out, while all others do not face any wear-out at all. To overcome this, the OCTO$^+$ algorithm implements a byte offset strategy for the checkpoint of tree nodes. Each node is stored with a constant byte offset between 0 and 7, which is statically determined by the node ID. This achieves a more uniform wear-out of the memory within words.

5.4.5 Evaluation

In order to evaluate the effectiveness of the OCTO$^+$ algorithm, an exemplary B$^+$-tree is implemented together with the wear-leveling algorithm in the full system wear-out simulation framework (Section 3.1.5).The configurable node size of the B$^+$-tree is chosen to 1024 bytes to offer a compromise between a reasonable fan out of the tree and precise granularity of write information collection. The tree is constructed by inserting three different data sets into the tree to simulate different usage scenarios:

1. **Linear**: This data set contains data in monotonic order, which simulates the scenario of presorted data. This results in a tree with many half filled nodes, since old nodes are likely not modified again during insertion.
2. **Random**: This data set contains random insertion keys, which likely modifies nodes in the tree in a uniform manner during insertion.
3. **YCSB**: This data set contains data from the Yahoo cloud serving benchmark [CST+10].

For all the data sets, keys and values are chosen as 8 byte values. To further simulate different usage characteristics for the various datasets, the dataset elements are split into three different setups each: 100% inserts / 0% lookups, 75% inserts / 25% lookups and 50% inserts / 50% lookups. For each experiment, the operations are mixed, such that modifications of the tree happen during the entire execution of the experiment. The datasets are not used entirely, but rather the total number of operations is fixed to make the experiments more comparable. The total number of operations is set to 20000 operations to simulate small trees and to 50000 operations to simulate big trees. The checkpoints are then performed after each 50 operations for the small trees (400 checkpoints in total) and after each 100 operations for the big trees (500 checkpoints in total).

To apply a mapping from tree nodes to NVM blocks, 5 different strategies are applied for comparison:

- **Static**: This strategy always employs the same mapping (e.g. an identity mapping) and leaves it unchanged. This strategy is used as a baseline to compare other strategies to.
- **OCTO$^+$**: This is the implementation of the algorithm discussed before, including intra and inter block wear-leveling.
- **AA**: This is only the inter block wear-leveling (aging aware) part from the OCTO$^+$ algorithm.

- **RANDOM**: This strategy decides an entirely randomized new mapping on every checkpoint.
- **RING**: This strategy moves every blocked to the next NVM block at every checkpoint with a wrap around semantics, such that the tree nodes move around the NVM in a ring like fashion during the checkpoints.

With the help of initial experiments, the threshold for aging divergence within blocks, which triggers a tree node to be mapped to a new NVM block in the OCTO$^+$ algorithm, is set to 15. The threshold for the age difference to swap the oldest and the youngest block is set to 5.

Metrics

In order to evaluate the resulting improvement in terms of memory wear-out, the lifetime improvement metric $LI_{iterative}$ is utilized (Section 3.1.2) in adoption to iterative write scheme memories. In this metric the resulting memory lifetime, under the assumption that the memory becomes unusable once the first cell wears out, is compared to a baseline execution and related to the caused overheads. This leads to a factor of how much the memory lifetime is improvement with the studied wear-leveling scheme. Although this metric suffices to analyze the lifetime improvement in isolation, it does not reveal any insights on how well the B$^+$-tree with the wear-leveling scheme is integratable into a larger system. In a larger system, multiple applications may be executed and checkpointed. A global wear-leveling scheme may consider all NVM blocks from different applications together and achieve wear-leveling by shuffling these blocks within checkpoints. Hence, it should be also studied how well such global wear-leveling schemes can profit from the analyzed B$^+$-tree implementation. Towards this, the definition of wear-leveling potential WLP is introduced. This metric determines the maximal memory wear-out in fixed size NVM blocks and computes the mean of these ages:

$$WLP(g) = \textbf{mean}\left(\max_{x\in[0,g]}(\textbf{age}(x)), ..., \max_{x\in[(m-1)\cdot g,n]}(\textbf{age}(x))\right) \tag{5.1}$$

In this equation, n denotes the total number of bits in the NVM, g denotes the granularity of NVM blocks to be investigates and $m = \frac{n}{g}$ the number of blocks. The age of a bit refers to the total amount of bit flips. For this evaluation, the granularity is considered to be $g = 4096 \cdot 8$, i.e. the usual size of virtual memory pages. This is motivated by the fact that software based global wear-leveling schemes may utilize the MMU to perform global wear-leveling. Comparing the WLP from an analyzed configuration with a baseline configuration leads to an improvement factor of the WLP.

Results

For the previously discussed evaluation settings and metrics, the results are summarized in Figure 5.7. The upper two graphs illustrate the lifetime improvement (left) and the wear-leveling potential (right) for the small B$^+$-trees, while the lower two graphs illustrate

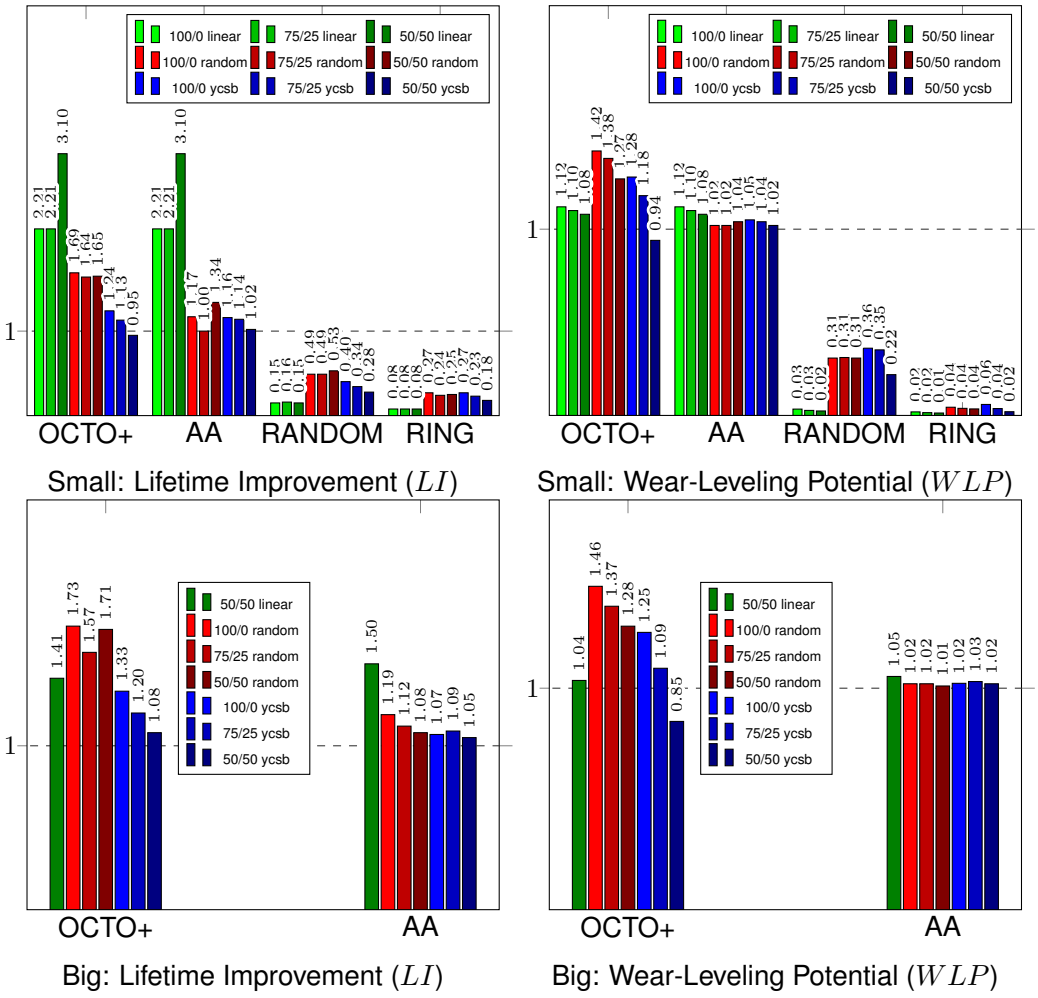


Figure 5.7: Wear-Leveling Improvement

the results for the big trees, respectively. The data sets, which are used to create the tree are indicated by color (green denotes the linear data set, red the random data set and blue the ycsb data set). The insert / lookup configurations are indicated by shading of the respective color. The analyzed wear-leveling configurations are indicated in blocks on the x-axis. For the big trees, the Random, and Ring strategies are omitted, since they already highlight a bad performance for the small trees and are hence not feasible to be considered as a viable strategy. Further, due to massive memory and time consumption for the simulation, only the 50% insert scenario is evaluated for the linear data set for big trees.

Investigating the shown results, it can be seen that the Random and Ring strategy lead to a consequent reduction in lifetime and wear-leveling potential in the setting of the small trees for all data sets. This suggests the conclusion that the initial problem statement, that wear-leveling actions have to be performed carefully, has high significance. Random and Ring are both blindly applied strategies, without any careful considerations

about caused overheads and gained improvements. This leads to a decrease in the memory lifetime by at least 50% in most cases, even up to massive decrease in lifetime and wear-leveling potential of more than 90%. On the other side, the carefully designed wear-leveling algorithms (OCTO$^+$ and AA) can improve the memory lifetime and the wear-leveling potential in all cases except for the ycsb benchmark with 50% inserts. This further suggests the conclusion that a careful design of wear-leveling can help to gain improvements in terms of memory lifetime. It can be further observed that for the random data set the OCTO$^+$ algorithm can achieve better results than the AA algorithm, i.e. additional intra block wear-leveling pays out in this scenario. While many results range around an improvement of $\approx 50\%$ in terms of memory lifetime, an improvement of up to 200%, i.e. a factor of $3\times$ can be achieved for the linear data set with 50% inserts and lookups.

5.4.6 Wrap-Up

This section illustrates a wear-leveling scheme for B$^+$-trees on a hybrid NVM system, where the wear-leveling is performed in the form of a remapping of memory locations for each checkpoint. The initial problem statement suggests that such remapping decisions have to be performed carefully, because the inherent overheads can easily degrade the memory lifetime. An algorithm, called OCTO$^+$ is introduced, which aims to perform such careful wear-leveling operations. This algorithm aims for intra blocks and inter block wear-leveling. Considering only the part of the inter block wear-leveling, the strategy is called AA (Aging Aware) in this section. An experimental comparison of blind wear-leveling strategies (Random and Ring) reveals that the memory lifetime indeed is degraded by up to 98% with such a not careful strategy. The OCTO$^+$ algorithm, in contrast can achieve a significant lifetime improvement of up to $3\times$, depending on the studied data set. In addition to this pure lifetime improvement, also the potential for possible cross application system-wide wear-leveling schemes is improved by the OCTO$^+$ algorithm. It has to be noted, however, that the presented algorithm here is an application specific scheme for B$^+$-trees, which heavily bases on the fact that the memory access information collection can be precisely and easily integrated into the tree implementation. When such methods are applied to other applications, a similar scheme for information collection has to be designed.

5.5 Concluding Software-Based Wear-Leveling

The previous two chapters introduce concepts for software-based NVM wear-leveling. It is the major goal of these concepts to broaden the traditional memory interface of only load ans store accesses in a way that extends the lifetime of memories with limited lifetime. This is mainly achieved by following a model of iterative wear-leveling, i.e. altering the software in such a way that an even usage of all memory cells is targeted

at any time. This approach has the key advantage that the internal state does not need to be persisted or complex model of the NVM needs to be maintained.

Even memory usage is achieved in an application transparent manner as a component of the executing operating system (Chapter 4). The MMU and the usage principles of the stack memory are used to change the target of memory accesses in a transparent way to the application. The even memory usage is achieved by tracking memory accesses with the help of performance monitoring hardware and applying simple wear-leveling strategies. Summing this method up, the traditional memory interface of loads and stores is broadened by information collection with the help of performance monitoring and by memory behavior changes with the help of the MMU. These two technologies are well established and traditioned and can be found in a variety of computer systems.

Chapter 5 further introduces methods to achieve wear-leveling in an application cooperative manner. For this, the need for performance monitoring and also the need for MMU based remapping is removed. This is achieved by investigating the internals of the running software and altering as one approach the stack memory usage and hook into a checkpointing algorithm as a second approach. The usage of stack memory by a C/C++ compiler is as well a well established and traditioned technology, which is used to broaden the memory interface for the sake of wear-leveling. Checkpointing between volatile memory and NVM may not be as traditioned and well established as the usage of stack memory, but the concept of checkpointing is very generic and widely used in the scope of certain applications, e.g. databases. Hence, for this specific case, it is also used to broaden the memory interface.

CHAPTER 6

Memory Optimization for Random Forests

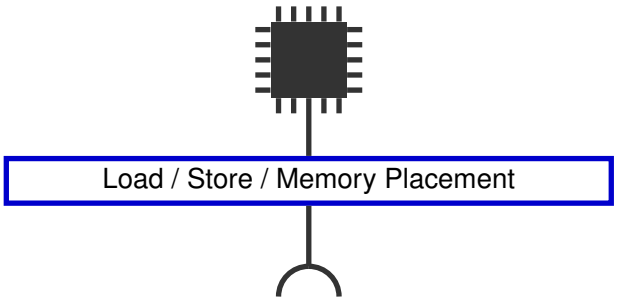
Contents

6.1 Modern Technologies and Traditional Interfaces 102
6.2 Overview 103
6.3 Unified Layout Optimization of DTs on Racetrack Memory 104
6.3.1 Scope 104
6.3.2 Problem Analysis and Statement 105
6.3.3 BLOwing Trees 106
6.3.4 Evaluation 112
6.3.5 Wrap-Up 115
6.4 Decomposed Layout Optimization of DTs on Racetrack Memory . 115
6.4.1 Scope 116
6.4.2 Problem Analysis and Statement 116
6.4.3 Decomposed Tree Optimization 118
6.4.4 Evaluation 124
6.4.5 Wrap-Up 129
6.5 Concluding Memory Optimization of Random Forests 130

6.1 Modern Technologies and Traditional Interfaces

Although the previously studied traditional memory interface of load and store accesses and controllable memory placement cannot achieve direct specific hardware management, it opens a wide design space for software exploitation for many use cases. Towards the application to other technologies, two main observations can be summarized from the previous chapters: First, the introduced work towards application transparent wear-leveling shows that a central management agent can overcome the shortcoming of the traditioned interface and provide a broadened interface towards wear-leveling, where the guessing about application behavior can be omitted. Second, the work on application cooperative wear-leveling shows that certain applications are well suited to efficiently integrate effective wear-leveling schemes directly into the application. Summarizing the interface extensions, required for wear-leveling, a memory access distribution is the most crucial part, which has to be captured and utilized. Providing means for capturing and utilizing memory access distributions, as presented before, broadens the traditional memory interface to a degree that lifetime management for modern memory technologies can be achieved by exploiting the traditional memory interface.

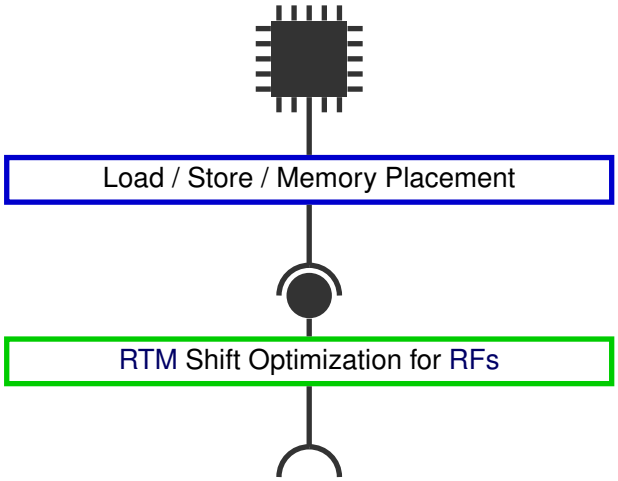
This chapter of this thesis picks up the same traditional hardware interface for memory accesses:



In the following, technologies are considered, where capturing a simple memory access distribution is not sufficient for a proper operation of the technology. Consequently, interfaces have to be further broadened. The major focus in this chapter is put to RACETRACK MEMORY (RTM), which is an emerging memory technology. Despite other technical characteristics, which are not further studied in this thesis, RTM features an access dependent shift latency, i.e. the latency and hence the energy consumption of a memory access depends on the previous memory access. In order to efficiently operate RTM, software should lay out memory contents in such a way that memory access, which follow each other, target RTM friendly memory locations. Towards this, not only a memory access distribution, but a conditional memory access distribution is required, i.e. the probability of a memory location to be accessed, given that another memory location was accessed beforehand. Intuitively, such a conditional distribution can be

captured in a straight forwards extension of the previously introduced methods as an empirical model and can serve for optimization. Considering existing methods towards this, such an approach also exists with compiler support [KHB+19]. The problem with this intuitive approach, however, is the large complexity. In an empirical model, every memory location could be accessed after every other location with a certain probability, which makes optimization of memory locations extremely challenging.

Consequently, a conditional empirical memory access distribution with a drastically reduced complexity is required. This chapter aims to provide such a model by exploiting application specific knowledge about the used data structures, i.e. tree data structures. In tree based structures, the parent relation clearly limits the possible combinations of subsequent memory accesses. The information about this reduced model is subsequently fed into a dedicated optimization interface for RTM. Ultimately, this allows a latency optimized operation of RTM by a software based extension of the traditional memory interface of placing memory contents at dedicated addresses and issuing load and store accesses:



6.2 Overview

The previous chapters of this thesis mainly focus on the lifetime issue of NVM and means to extend the lifetime by broadening the traditional memory interface. Lifetime, however, is not the only property, that comes with NVM in a significant different manner than in traditional memories. Latencies and access time behavior can also feature significant different structures than for classic random access memories. This chapter enlightens this aspect with a dedicated focus on the latency structures of RACETRACK MEMORY (RTM). Section 3.1.6 gives an overview of the latency structure of RTMs. These memories induce an access dependent latency model due to their inner realization.

In greater detail, the latency of a single memory access depends on the previous accesses and is not constant. Although generic optimization frameworks for RTM exist [KHB+19], the complexity of the latency optimization problem is extremely high, making optimal solutions infeasible in many situations. One chance to overcome this, is to develop specialized solutions for dedicated applications, such that the complexity of the optimization problem can be drastically reduced. In this section, the target application of RANDOM FOREST (RF) is studied. RFs, when deployed after training, allow building a probabilistic model of the distribution of visited memory locations during inference. It should be noted that such a model always is constructed on the training data and therefore must not entirely describe the test data. However, a basic principle of ML is that training and test data come from a similar distribution, such that probabilistic models should ideally hold for training and test data similarly. The probabilistic model of the memory accesses of RFs can be used to model the corresponding memory latencies in RTM, as explained in detail in Section 3.2. The degree of freedom when deploying RFs to hardware then allow for optimization of the allover latency.

6.3 Unified Layout Optimization of DTs on Racetrack Memory

As discussed above, random forest and decision tree implementations allow for a variety of modifications, which also allow hardware aware implementations. Especially the step of transforming a given model to a hardware-aware implementation allows considering hardware specific characteristics and account for these in a hardware optimized layout. In this section, such a hardware optimized implementation is discussed for the case of RTM (Section 3.1.6). Due to their inner structure, RTMs have a memory access latency, which is dependent on the previous access. The previous access within decision tree inference is deterministically defined by the tree structure itself. Following a probabilistic model of the tree, most probable accesses are optimized in such a way, that the allover memory access latency is minimized.

6.3.1 Scope

This work presents, as mentioned before, memory access latency optimization for decision trees on RTM. The specific property of memory access latencies depending on the previous accesses stems from the organization of RTM into DBCs. These DBCs organize memory cells in a nanowire with the need to shift them towards an access head before accessing, requiring a shift length dependent latency. The usual size of these DBCs allows to place one DT at most. Consequently, RFs are not specifically considered in this work, since the latency optimization is applied to every single DT within the ensemble. As the optimization provide optimized memory access latency for RTM, native trees are considered in this work. Native trees allow an arbitrary placement of tree nodes in the data memory, hence allow for arbitrary encountering for RTM related

published work: [HKC+21a]

access latencies. Basically, the studied problem in this work boils down to a formalizable optimization problem, i.e. finding the layout in data memory for a native tree, which minimizes the RTM induced latency with respect to an empirical probabilistic model. Since solving this problem optimally is highly time-consuming, this work introduces a fast optimization strategy. It is proven, that the fast optimization strategy in this section has an upper bound of $4\times$ compared to the optimal solution, i.e. achieves a result at most with $4\times$ the latency as the optimal solution. Experimental evaluation shows that the fast optimization performs even better, even close to the optimal solution in most cases. In short, the following points are covered in this work:

- A formal cost model for shift overhead of racetrack memory with respect to a probabilistic execution model is specified. This model is expressed as an ILP, which allows for deriving the optimal solution, if the required time is spent.
- A fast optimization algorithm, which reduces the problem of DT layouting in RTM to an efficient solvable optimization problem. The algorithm is further modified for the specific characteristics of the studied problem.
- A formal proof of the upper bound of $4\times$ for the fast optimization algorithm compared to the optimal solution.

6.3.2 Problem Analysis and Statement

Before elaborating the actual problem formulation, the scope of the target system has to be clarified first. As motivated, this work aims to optimize access latencies in RTM for DTs. Hence, a system equipped with RTM is considered as the target system. Due to the specific access properties of RTM, not the entire system memory may be mapped to the RTM, hence a hybrid memory system is considered, where RTM is available as a form of scratchpad memory, which can be mapped to the target memory, which should be stored in RTM. The other type of memory in the system can be e.g. realized with SRAM. If such a system, for instance, serves as an on the edge inference device with a limited energy budget, storing the DT in RTM is desired due to the non-volatility and the corresponding energy saving. Improving the access latency of the model inference then further reduces the energy consumption and can help to reduce maintenance cycles when such a device, for instance, serves as a battery powered embedded sensor node. In order to enable such a system architecture, no further caches are assumed to be applied with the RTM, i.e. all memory accesses from the CPU to that specific region directly lead to access within the RTM.

The deployed RF including the inner DTs follows exactly the same structure and logical model as described before. Furthermore, the realization of native trees is considered for this specific case. In native trees, each node can be stored at an arbitrary index within the node array, since the logical tree structure is realized by left and right child pointers, which only need to be set correctly. In consequence, for a DT with $m-1$ nodes $N = \{n_0, n_1, ..., n_{m-1}\}$, an arbitrary bijective mapping $I : N \to \{0, 1, ..., m-1\}$ of tree nodes to array positions can be realized. It is assumed that a single node of the DT can be stored in one memory word. Referring to the system model (Section 6.3),

the cost for access latency and the relative energy consumption for shifting racetrack memory is linearly related to the length of the shift. Hence, shifting a single RTM DBC from position i to position j is assumed to cause an abstract cost of $|i-j|$ in the following. When one DT is mapped to a single DBC, accessing node n_b directly after n_a with an applied mapping I causes the cost $|I(n_b) - I(n_a)|$. Minimizing this abstract cost minimizes the access latency and the corresponding energy overhead.

The organization of RTM in DBCs leads to the aforementioned abstract cost model within a single DBC. Accesses to different DBCs can be considered individually, since they are realized in an architecture, which enables random access to DBCs without varying costs. A typical length of a DBC is up to 64 positions. If only a single DT is stored in such a DBC, at least a maximal depth of 5 can be realized. Larger trees could be split into sub trees of depth of 5 and handled as individual trees. Hence, the methods presented in the following assume one DT to be mapped within a single DBC exclusively.

Effectively, the studied problem in this section can be formulated as finding an optimal mapping I for a single DT, exclusively mapped to a single RTM DBC, minimizing the abstract cost model. Solving this problem directly derives a strategy for larger trees and RFs by splitting into subtrees and employing multiple instances as a RF. Consequently, the following only discusses the case of single DTs with a maximal depth of 5. In order to tackle the studied problem, a precise cost model is formulated first. Due to the high time consumption of minizing the cost model straight forward, a fast optimization algorithm is derived by reducing the cost model to an instance of the Optimal Linear Ordering (O.L.O) problem, which can be efficiently solved the special case of rooted trees. Since the cost model does not exactly describe a rooted tree, the derived solution is not necessarily optimal. Therefore, an upper bound to the optimal solution is derived with a formal proof.

6.3.3 BLOwing Trees

As introduced, the first step towards latency and energy optimization of DTs on RTM is to derive a precise cost model, which serves as a target for optimization. This cost model includes the abstract cost model on RTM as explained before, as well as the probabilistic model of DT inference. From the training data, the probability of each node n_x to be visited from its parent node $prob(n_x)$ is determined by counting the samples to be assigned to that specific node during training. Although this distribution must not exactly be valid for the test data set, it is assumed that the distributions are similar. Hence, the probabilistic model serves as an optimization target here. The relative probability of a node to be visited is between 0 and 1, hence an absolute probability of a node n_x to be visited within the tree across a sufficient amount of inferences can be stated as $absprob(n_x) = \Pi_{n_z \in \{path(n_x)\}} prob(n_x)$, where $path(n_x)$ describes a set of nodes, which contains all nodes on the path from the root node n_0 towards the specific node n_x in the order of their appearance in the path. Every node within the tree n_x forms the root of a subtree, which contains a set of leaf nodes $leafs(n_x) \subseteq N_l$, where

N_l is the set of all leaf nodes of the tree. Then $\forall n_y \in leafs(n_x) : n_x \in path(n_y)$. This further helps to define a relation of the absolute probabilities:

Definition 1. For a given node $n_x \in N$, the sum of probabilities of its direct children must always be 1. The absolute probability of n_x then by definition can be expressed as

$$absprob(n_x) = \sum_{n_y \in leafs(n_x)} absprob(n_y) \tag{6.1}$$

Considering the execution of the DT during inference, nodes are allover visited on their absolute probability. Hence, across the entire tree, the abstract cost for shifting the RTM DBC for inference under a node mapping I can be stated as the cost for shifting from a parent node to the node, weighted with the absolute probability (Equation ((6.2))):

$$C_{down} = \sum_{n_x \in N \setminus \{n_0\}} absprob(n_x) \cdot |I(n_x) - I(P(n_x))| \tag{6.2}$$

Since not only a single inference is executed, the cost for shifting the RTM DBC back to the root node has to be accounted for as well. This only happens when a leaf nodes was accessed before, hence this cost can be described in Equation ((6.3)).

$$C_{up} = \sum_{n_x \in N_l} absprob(n_x) \cdot |I(n_x) - I(n_0)| \tag{6.3}$$

This consequently leads to the overall cost model for DT inference on RTM in Equation ((6.4)).

$$C_{total} = C_{down} + C_{up} \tag{6.4}$$

Finding a mapping of nodes to memory positions, wich minimizes C_{total} is considered to be a minimal mapping I^*. As mentioned before, this problem is an instance of the *Optimal Linear Ordering* (O.L.O.) problem [AH73; BÇP+98; DPS02]. The O.L.O. problem in general is to map the nodes of a graph G to slots, where all slots are in a row and adjacent slots are one unit apart, such that the total sum of arc weights multiplied with the distance between the nodes, connected by the arc, is minimal. The O.L.O. (or also called *Optimal Linear Arrangement*) problem is an instance of the Quadratic Assignment problem and is NP-complete [GJ79]. As a special case, the O.L.O. problem for rooted trees with the root node on the leftmost position (i.e. only optimizing C_{down}) can be optimally solved in time complexity $O(m \log m)$ [AH73]. However, reducing the problem formulation in Equation ((6.4)) does not form a rooted tree, but rather a cyclic graph due to arcs from leaf nodes to the root node. Hence, applying the algorithm from Adolphson and Hu [AH73], leads to a mapping, which must not be I^*, but rather a suboptimal mapping, only minimizing Equation ((6.2)) as part of the total cost model. Furthermore, the algorithm limits the solution to trees with the root node mapped to the left most position, which can be considered as a further limitation compared to the optimal mapping. The mapping, derived by the fast algorithm from Adolphson and Hu, is denoted as $\overleftarrow{I^*}$ in the following.

The following discusses the relation between the optimal mapping for the studied problem, which requires a long time to be found and the suboptimal mapping, derived by the algorithm from Adolphson and Hu. The two limitations, namely the limitation to rooted trees and the condition that the root node is mapped to the left most position, are analyzed and their effect on the total cost is formally upper bounded in the following. This requires the introduction of certain terminology for different mappings and their resulting cost in Table 6.1.

Placement	Explanation
I	arbitrary mapping
I^*	optimal mapping which optimizes C_{total}, resulting in the total cost C^*_{opt}
$I^{*\downarrow}$	optimal mapping which optimizes C_{down} only, resulting in the optimal down cost $C^{*\downarrow}_{down}$
$\overleftarrow{I}$	arbitrary mapping with the root on the left
$\overleftarrow{I^*}$	optimal mapping with the root on the left and with expected down cost $\overleftarrow{C^*}_{down}$

Table 6.1: Placement Notation

Suppose that C^*_{opt} is the minimally expected cost C_{total} of the optimal placement I^* of the DT. The following shows how to derive a suboptimal mapping, which causes at most 4 times the cost of C^*_{opt}. A path, defined as $path(n_\ell)$, from the root node n_0 to a leaf node $n_\ell \in N_l$ in a placement I is *monotonically increasing* if $I(n_x) > I(P(n_x))$ for every node n_x in $path(n_\ell) \setminus \{n_0\}$. Contrarily, such a path is *monotonically decreasing* if $I(n_x) < I(P(n_x))$ for every node n_x in $path(n_\ell) \setminus \{n_0\}$.

Definition 2. A placement I is defined as **unidirectional** if all paths in the given DT are monotonically increasing in this placement. □

Definition 3. A placement I is defined as **bidirectional** if every path in the DT is either monotonically increasing or monotonically decreasing. □

Lemma 1. Let $I^{*\downarrow}$ be a mapping which only minimizes C_{down}, resulting in $C^{*\downarrow}_{down}$, and ignores C_{up}. Then,

$$C^{*\downarrow}_{down} \leq C^*_{opt} \tag{6.5}$$

Proof. This comes from the definition as certain terms in the objective function are removed, and all terms are positive. □

Next, a property is restated, which comes from Adolphson and Hu [AH73]. It regards the optimization of $I^{*\downarrow}$ when the root *has to be put* on the leftmost position.

Lemma 2 (Page 410 in [AH73])**.** (restated) There exists an optimal unidirectional placement $\overleftarrow{I^*}$ for the O.L.O. problem when the input is a rooted tree, i.e., $\overleftarrow{C^*}_{down} = C^{*\downarrow}_{down}$, under the constraint that the root is on the leftmost position.

Deriving a unidirectional or bidirectional placement induces the special property that optimizing C_{down} implicitly optimizes C_{up}, which is shown by the following lemma.

Lemma 3. If a placement I is unidirectional or bidirectional, $C_{down} = C_{up}$.

Proof. $C_{down} = C_{up}$ has to be shown. Since I is known to be unidirectional or bidirectional, it is also known that a leaf node $n_x \in N_l$ is always the rightmost node or the leftmost node within its path $path(n_x)$ if the path is monotonically increasing or decreasing, respectively. It is further known that following the path from parents to their children must always be a movement monotonically to the right or monotonically to the left. Therefore, it can be followed that the distance from the root to a leaf node is equal to the sum of all distances on the path:

$$\forall n_y \in N_l : |I(n_y) - I(n_0)| = \sum_{n_z \in path(n_y) \setminus n_0} |I(n_z) - I(P(n_z))| \tag{6.6}$$

This leads to:

$$C_{up} = \sum_{n_y \in N_l} \left(absprob(n_y) \cdot \sum_{n_z \in path(n_y) \setminus \{n_0\}} |I(n_z) - I(P(n_z))| \right) \tag{6.7}$$

The summation is reorganized with respect to each node $n_x \in N$ by using the following observation: if n_z is in $path(n_y)$, then n_y is in $leaves(n_z)$. That is, a node $n_x \in N$ contributes to Equation ((6.7)) exactly $|I(n_x) - I(P(n_x))| \cdot \sum_{n_y \in leaves(n_x)} absprob(n_y)$. Therefore,

$$C_{up} = \sum_{n_x \in N \setminus \{n_0\}} \left(|I(n_x) - I(P(n_x))| \cdot \sum_{n_y \in leaves(n_x)} absprob(n_y) \right) \tag{6.8}$$

Applying Definition 1 leads to Equation ((6.9)):

$$C_{up} = \sum_{n_x \in N \setminus \{n_0\}} (|I(n_x) - I(P(n_x))| \cdot absprob(n_x) = C_{down} \tag{6.9}$$

□

In the following, the relation between a mapping I and a mapping $\overleftarrow{I}$ which puts the root on the leftmost position is highlighted.

Lemma 4. Any placement I can be converted into a placement $\overleftarrow{I}$ which places the root on the leftmost position by increasing the expected cost of $\overleftarrow{C}_{down}$ with at most a factor of 2:

$$\overleftarrow{C}_{down} \leq 2 \cdot C_{down} \tag{6.10}$$

Proof. Suppose that the root of the DT is assigned at position r in the placement I. Due space limitation, only the proof of the case that $m - r \geq r$ is presented, as the other case is symmetric. The placement is replaced as follows:

- reassign every node in position $r+i$ in I to $r+2\cdot i$ for $i=1,2,\ldots,r$.
- reassign every node in position $r+i$ in I to $2\cdot r+i$ for $i=r+1,r+2,\ldots,m$.
- reassign every node in position $r-i$ in I to $r+2\cdot i-1$ for $i=1,2,\ldots,r$.

After that, every node is then shifted by r positions towards the left and the root of the decision tree is on the leftmost position, i.e., 0.

For notation brevity, $P(n_x)$ is denoted as n_z for the rest of this proof. According to the above reassignment,

$$\overleftarrow{I}(n_x)=\begin{cases}2\cdot(r-I(n_x))-1 & I(n_x)<r\\ 2\cdot(I(n_x)-r) & r\le I(n_x)\le 2r\\ I(n_x) & 2r<I(n_x),\end{cases} \tag{6.11}$$

which also holds in the same manner for $\overleftarrow{I}(n_z)$. Four cases for different conditions of $I(n_z)$ and $I(n_x)$ based on Equation ((6.11)) are analyzed to prove

$$|\overleftarrow{I}(n_x)-\overleftarrow{I}(n_z)|\le 2|I(n_x)-I(n_z)|. \tag{6.12}$$

Case 1: $I(n_z)\le 2r$ and $I(n_x)\le 2r$: The following scenarios are considered:

- **Case 1a**: $I(n_x)$ and $I(n_z)$ are both $\ge r$: Then, $|\overleftarrow{I}(n_x)-\overleftarrow{I}(n_z)|=2|I(n_x)-I(n_z)|$, i.e., Equation ((6.12)) holds.
- **Case 1b**: $I(n_x)$ and $I(n_z)$ are both $<r$: Then, $|\overleftarrow{I}(n_x)-\overleftarrow{I}(n_z)|=2|I(n_x)-I(n_z)|$, i.e., Equation ((6.12)) holds.
- **Case 1c**: one of $I(n_x)$ and $I(n_z)$ is $<r$ and the other is $\ge r$: Suppose for the first subcase that $I(n_x)>I(n_z)$. Then, $|\overleftarrow{I}(n_x)-\overleftarrow{I}(n_z)|=2\cdot(I(n_x)-r)-2(r-I(n_z))+1<2\cdot(I(n_x)-r)-2(r-I(n_z))+4(r-I(n_z))=2\cdot(I(n_x)-I(n_z))=2|I(n_x)-I(n_z)|$, where $<$ is due to the assumption that $I(n_z)<r$ and $I(n_z)$ is an integer, i.e., $1\le r-I(n_z)$. The other case that $I(n_z)>I(n_x)$ is symmetric. Therefore, the condition in Equation ((6.12)) remains to hold.

Case 2: $I(n_z)>2r$ and $I(n_x)>2r$: In this case, the reassignment does not change their positions, i.e., $\overleftarrow{I}(n_z)=2r+(I(n_z)-2r)=I(n_z)$ and $\overleftarrow{I}(n_x)=2r+(I(n_x)-2r)=I(n_x)$. As a result, $|\overleftarrow{I}(n_x)-\overleftarrow{I}(n_z)|=|I(n_x)-I(n_z)|$, and Equation ((6.12)) holds.

Case 3: $I(n_z)>2r$ and $I(n_x)\le 2r$: When $I(n_x)\ge r$, $|\overleftarrow{I}(n_x)-\overleftarrow{I}(n_z)|=I(n_z)-2|I(n_x)-r|=I(n_z)-2I(n_x)+2r\le 2\cdot|I(n_z)-I(n_x)|$ holds. When $I(n_x)<r$, $|\overleftarrow{I}(n_x)-\overleftarrow{I}(n_z)|=I(n_z)-2r+2I(n_x)+1<I(n_z)-2r+2I(n_x)+4r-4I(n_x)=I(n_z)+2r-2I(n_x)\le 2\cdot|I(n_z)-I(n_x)|$ holds, where $<$ above is due to the assumption that $I(n_z)<r$ and hence $r-I(n_z)\ge 1$. Therefore, Equation ((6.12)) holds.

Case 4: $I(n_z)\le 2r$ and $I(n_x)>2r$: This is the symmetric case of Case 3.

As a result, Equation ((6.12)) holds for all cases and the lemma is proved. □

Suppose that $\overleftarrow{I^*}$ is an optimal unidirectional mapping of the rooted tree (with the root on the leftmost position) and optimizes the cost C_{down}, resulting in the minimal cost $\overleftarrow{C^*}_{down}$. Further suppose that $I^{*\downarrow}$ is an optimal mapping which optimizes C_{down}, resulting in $C^{*\downarrow}_{down}$. The following corollary can be concluded:

Corollary 1.

$$\overleftarrow{C^*}_{down} \leq 2 \cdot C^{*\downarrow}_{down} \tag{6.13}$$

Proof. $I^{*\downarrow}$ is an unconstrained placement that achieves the optimal $C^{*\downarrow}_{down}$. By Lemma 2, it is known that $\overleftarrow{I^*}$ is an optimal placement for the cost $\overleftarrow{C^*}_{down}$ under the condition that the root is on the leftmost position. Therefore, $C^{*\downarrow}_{down}$ is a lower bound of any solution when the root is on the leftmost position. By Lemma 4, $I^{*\downarrow}$ can be converted into a placement $\overleftarrow{I}$, in which the root is put to the leftmost position, with a cost up to $\overleftarrow{C}_{down} \leq 2 \cdot C^{*\downarrow}_{down}$. Therefore, $\overleftarrow{I}^*$, as the optimal placement under the root constraint, must not cause a higher cost $\overleftarrow{C^*}_{down}$ than $\overleftarrow{C}_{down}$. □

Combining the previous lemmas and the corollary leads to an assessment of the relation of the total cost between an optimal mapping and an optimal unidirectional mapping with the root on the left most position:

Theorem 1. *An optimal unidirectional placement has an approximation factor of* 4 *of the studied problem.*

Proof. Based on Lemma 3, we know that the expected cost, denoted as $\overleftarrow{C^*}_{total}$, of the optimal unidirectional placement for the decision tree (including the down- and up-parts) is exactly $2 \cdot \overleftarrow{C^*}_{down}$. Therefore, together with Corollary 1 and Lemma (6.5), the conclusion can be reached.

$$\overleftarrow{C^*}_{total} = 2 \cdot \overleftarrow{C^*}_{down} \leq 4 \cdot C^{*\downarrow}_{down} \leq 4 \cdot C^*_{opt}.$$

□

In order to set the previous proof into a context of application, it is shown how to derive an optimal unidirectional solution that minimizes $\overleftarrow{C^*}_{down}$ efficiently. Adolphson and Hu [AH73] proposed an algorithm to optimally solve this case. Specifically, according to [AH73], the O.L.O. problem for rooted trees with the root mapped to the leftmost slot is to find an optimal allowable linear ordering of tree nodes. An allowable linear ordering in their terminology means that if node $n_p = P(n_x)$ is the parent of node n_x, it has to be left of n_x in the ordering. The algorithm from Adolphson and Hu always derives an optimal allowable linear ordering to minimize the O.L.O. problem in $O(m \log m)$ time complexity.

Deriving a mapping by the algorithm from Adolphson and Hu at most causes $4\times$ the cost compared to the optimal solution for our placement problem. The algorithm from Adolphson and Hu has the main drawback that it places the root node to the leftmost

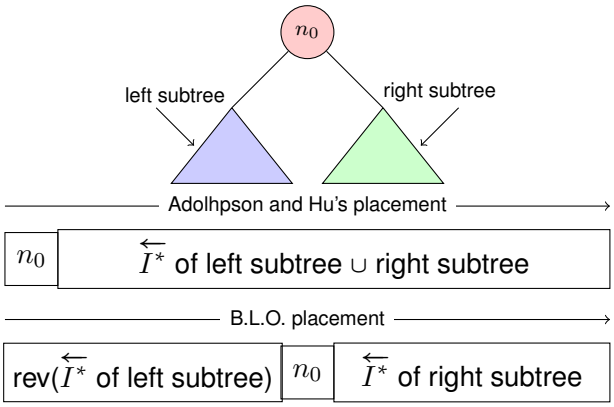


Figure 6.1: Suboptimal Placement Correction

slot in any solution, which is not optimal when the cost for going back from leaves to the root between inferences is considered. Consequently, this section introduces a final algorithm, which computes a *Bidirectional Linear Ordering* (B.L.O.). This algorithm maps the two subtrees underneath the root by the algorithm from Adolphson and Hu, which derives a mapping I_L for the left subtree and a mapping I_R for the right subtree. Both mappings cause an expected cost which is at least 2 shifts less than the total expected cost of the entire tree since one node, and therefore a shift at least by one slot, is missing on every path to a leaf and back to the root. Then, the final B.L.O. mapping is formed by placing $I^\diamond = \{reverse(I_L), 0, I_R\}$. In this mapping two shifts are then added again to every path into and out of the right and left subtree, thus $C^\diamond_{total} \leq C_{total}$.

Considering the exemplary decision tree in Figure 6.1, each access would start at the leftmost position in the first placement, target a leaf within the rest of the mapping and shift back to the leftmost position. In the second mapping, as long as leaves from the left and right subtree are accessed on a similar ratio, the expected shifting distance is divided by a factor of 2. The reverse ordering can be done in $O(m)$, the placement of the root is performed with constant time overhead. Therefore, the time complexity of B.L.O. is $O(m \log m)$.

6.3.4 Evaluation

In order to compare Bidirectional Linear Ordering (B.L.O.) to the state-of-the-art generic RTM frameworks (i.e., ShiftsReduce [KHB+19] and Chen et al. [CSZ+16]), the previously introduced summarized tool frame for C/C++ realizations of random forests is used [CSH+22]. 8 typical machine-learning classification datasets are selected from the UCI Machine-Learning Repository [AN07] and [LeC98]: adult, bank, magic, mnist, satlog, sensorless-drive, spambase and wine-quality. For each data set, 75% of the data is used for training and 25% is used for testing.

To derive different sized trees, the maximum depth of the trees is limited, e.g., DT1 means that the tree has 2 levels and DT3 means that the tree has 4 levels. After the

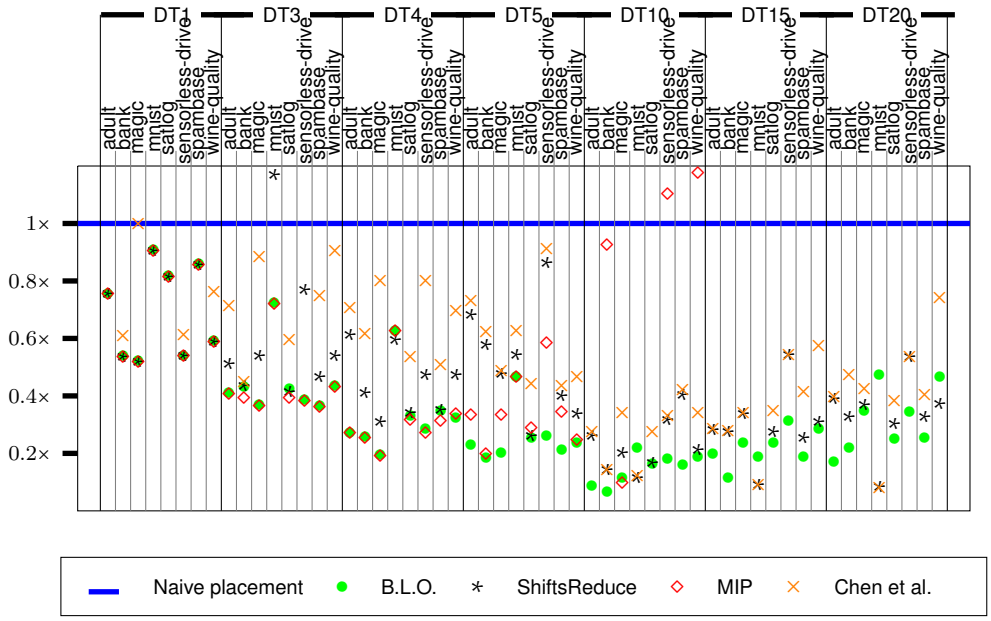


Figure 6.2: Comparison of Total Shifts During Inference

trees are generated, the node probabilities are profiled on the training data, delivering the probabilistic model. Furthermore, the data points from the test data are inferred on the trees and generate a node access trace, which provides the node access paths on a logic level. Subsequently, the trees are mapped to a memory layout with the compared approaches and the node access traces are replayed to derive the total amount of required racetrack shifts under the assumption that the entire tree is placed in a single DBC. Although this already allows a quantitative comparison of the placement approaches, the energy consumption and total runtime on a realistic model is further computed, derived from the various memory mappings. For the runtime, the per-access and per-shift latencies in Table 6.2 are used and lead to the overall runtime. Given the amount of RTM accesses $n_{accesses}$ and the total amount of shifts in between n_{shifts}, the total runtime is $runtime = \ell_R \cdot n_{accesses} + \ell_S \cdot n_{shifts}$. The total energy consumption is derived from read and shift dependent dynamic energy consumption and from the runtime dependent static energy consumption (leakage): $energy = e_R \cdot n_{accesses} + e_S \cdot n_{shifts} + p \cdot runtime$, where the parameters can be found in Table 6.2.

Ports/track, tracks/DBC, domains/track		1, 80, 64
Leakage power [mW]	p	36.2
Write / Read / Shift energy [pJ]	$e_W/e_R/e_S$	106.8 / 62.8 / 51.8
Write / Read / Shift latency [ns]	$\ell_W/\ell_R/\ell_S$	1.79 / 1.35 / 1.42

Table 6.2: RTM Parameters Values for a 128 KiB SPM

As previously mentioned, only the racetrack shifts are investigated, which are caused when inferring data points on the decision trees. Since it is assumed that for the target system, the decision trees are mapped to isolated scratchpad memory, the memory accesses to the decision trees are not disrupted by any operating system interaction. The overall energy consumption and latency, however, still strongly depend on the parallel running applications and the underlying system software. This could be investigated by further full system simulation, which is out of the scope of this evaluation.

Result Discussion

Figure 6.2 depicts the experimental results for the reduction of the total amount of shifts by the different placement approaches. All results indicate the relative amount of racetrack shifts compared to a *naive* placement, which is derived by traversing the tree in breath-first order while placing the nodes consecutive in memory as they are traversed. Despite applying the proposed B.L.O. algorithm, ShiftsReduce [KHB+19] and Chen et al. [CSZ+16], the mapping problem is also formulated as a mixed integer program (MIP), which optimizes Equation ((6.4)). This MIP is implemented in the Gurobi optimizer [Bix07] and a time limitation of three hours per dataset and tree configuration is set. For all datasets, the MIP converges to the optimal solution only for DT1 and DT3. For all other cases, the result is based on the Gurobi heuristic. Results which are worse than $1.2\times$ of the naive placement are not included.

Investigating the illustrated results, it can be observed that for the cases where the MIP finds an optimal mapping (for DT1 and DT3), B.L.O. achieves the same or only marginally worse results than the optimum. This supports the heuristic design principle of B.L.O. Furthermore it can be observed that B.L.O. achieves the best reduction in shifts for most of the investigated cases. Considering the mean improvement over all evaluated datasets and trees, B.L.O. reduces the amount of required shifts by 65.9% compared to the naive placement. ShiftsReduce reduces the required amount of shifts by 55.6%. This implies that B.L.O. further improves the amount of required shifts by 18.7% upon ShiftsReduce.

It should be noted that deciding the placement based on the profiled probabilities from the training dataset does not necessarily result in the expected cost for the test dataset, when both datasets are too different. Hence, the required amount of shifts when the training dataset is inferred on the decision tree is determined, after the mapping is decided on the profiled probabilities of the same dataset. The results report minimal difference: B.L.O. on average reduces the required amount of shifts on the train dataset by 66.1%, and ShiftsReduce reduces the required amount of shifts by 55.7%.

The reduction of the total amount of shifts is an indicator, which does not immediately reflect a realistic improvement in runtime or energy consumption. Therefore, the improvement of the total runtime and energy consumption for the placement approaches is computed. It is pointed our earlier, that in a realistic setup, larger decision trees are split into smaller trees first and the placement heuristic is then executed on multiple DT5 sized trees. Therefore. the average runtime and energy consumption improvements

are presented for all DT5 experiments: B.L.O. improves the overall runtime by 71.9% compared the naive placement, the total energy consumption by 71.3% respectively. ShiftsReduce, in comparison, improves the overall runtime by 60.3% and the total energy consumption by 59.8%. Thus, B.L.O. improves runtime and energy consumption by 19.2% compared to ShiftsReduce. Comparing this to the reduction of shifts for DT5 sized trees only, B.L.O. reduces the required shifts by 74.7%, ShiftsReduce by 48.3%, thus B.L.O. improves ShiftsReduce by 54.7%. This draws the conclusion that despite static energy consumption and read latency having a non-negligible influence, the reduction of the amount of racetrack shifts results in a significant improvement of the runtime and energy consumption.

6.3.5 Wrap-Up

While the previous chapters of this thesis focus on the aspect of lifetime of NVM, this chapter introduces the consideration of special latency and energy properties, especially for RACETRACK MEMORY (RTM). In RTM, memory cells are organized in nanowires, which share a single access head. Consequently, the wire needs to be shifted to the corresponding position in order to read or write a memory cell. As this shifting consumes time, linearly dependent on the shift distance, the memory access latency depends on the previous access and hence the required shift distance. While there exist a few generic frameworks to minimize the shift induced latency and also the inherent energy consumption overhead, finding an optimal solution under general assumptions that every memory location can be accessed at any time is not feasible within a considerable amount of optimization time. In this chapter, a specific placement method for DECISION TREES (DTs) is presented, where the known order of memory accesses within the binary tree (i.e. to always follow paths from the root to the leaf) is exploited to significantly reduce the complexity of the optimization problem. Considering only parts of the required memory accesses, an optimal solution can be found with a time complexity of $O(m \log m)$. It is proved that finding this optimal solution for parts of the required memory accesses derives a solution for all memory accesses, which has an upper bound of $4\times$ to the optimal solution. Experimental evaluation even highlights that the proposed algorithm for finding a fast solution performs very close to the allover optimal solution in most cases. The generic placement solutions for RTM are outperformed in most cases.

6.4 Decomposed Layout Optimization of DTs on Racetrack Memory

Section 6.3 discusses a layout method for DTs on RTM, where the access dependent memory latency is minimized with respect to a probabilistic execution model of the tree. In this method, two key assumptions define the scope: 1) it is assumed that trees can be entirely placed in a single DBC. If a tree exceed the size of the DBC, the method

can be transferred to subtrees with the DBC size. 2) it is assumed that a node of a tree is stored entirely in one memory word within a DBC. Although this is feasible with, for instance, 64-bit memory words, not every component of the node is required during every execution. Consequently, splitting the tree nodes into components, storing them in different DBCs and only shifting the required DBC can help to further reduce the allover latency. Such an approach of decomposing tree nodes into various DBCs is discussed in this section.

6.4.1 Scope

The method presented in this section builds a direct extension to the method presented in Section 6.3. Hence, also the placement of DTs on RTM is discussed and the optimization and relation to optimal solutions is discussed. In contrast to the previous method, tree nodes are not assumed to be stored in one word within a DBC position, but rather are considered to be split into three different DBCs. Since the discussed trees here are binary trees, nodes can be split into their data elements, including feature indices and split values, the left child pointer and the right child pointer. In this method, each of these three is stored in one DBC. While the data value DBC still requires the same amounts of RTM shifts as in the previous method, since nodes are accessed in exactly the same sequence, only one of the left or the right child pointer is accessed at the execution of a node. Hence, only one of the corresponding DBCs needs to be shifted. This leads to a redefinition of the cost model, where the leftmost and rightmost parent of a node (i.e. the previous node in the path, which accessed either the left or right child pointer) is considered. This cost model is used to assess the efficient derived placement solution from the O.L.O algorithm against the optimal solution. A formal proof of an upper bound of $12\times$ is provided. In addition to that, empirical experimental evaluation provides intuition for the realistic performance of the proposed algorithm. In short, these points are covered in this section:

- A redefined cost model for a decomposed organization of DTs into three RTM DBCs. The cost model considers the leftmost and rightmost parent on a path to assess the shift cost for the left and right child pointers independent of the data elements of each node.
- A formal proof of an upper bound of $12\times$ of the efficiently derived O.L.O solution against the optimal solution.
- Experimental evaluation of the proposed B.L.O Decomposed algorithm.

6.4.2 Problem Analysis and Statement

The analyzed and studied problem in this section is basically described in Section 6.3.2. The major difference is the considered cost model, which encounters the decomposed organization of DTs into three DBCs, one for the data elements, such as split values and

published work: [HKC+22]

feature indices, one for the left child pointers and one for the right child pointers. This model requires only a shift of either the right or left child pointer DBC during execution, but not both. This forms another optimization objective, which is studied against the O.L.O. algorithm in this section.

The decomposed approach in general is motivated by two major challenges in the previously presented unified organization approach: (1) it requires very wide DBCs and is less scalable (ii) leaf nodes that make $\approx 50\%$ of the total number of tree nodes do not need to store pointers for left and right child nodes. However, since the node information in the unified approach is tightly coupled, storage can not be optimized. This leads to storage wastage and yields suboptimal latency and energy consumption.

The DBC size is generally defined by two parameters, i.e., the number of (useful) domains per track and the number of tracks per DBC. Increasing the number of domains per track increases the capacity but at the cost of increased latency and increased position-error rate [OJR+19]. Similarly, the number of tracks per DBC affects the number of address bits, decoder's size, and ultimately performance and energy consumption. For a fixed size RTM, increasing the number of tracks per DBC reduces the number of DBCs and requires fewer address bits. However, this comes at the cost of storage wastage and increased energy consumption. Smaller width DBCs allow for storing different memory objects in different parts of the RTM that can be accessed and controlled independently. This also avoids wasting the RTM storage space.

This section proposes a decomposed approach to find a better solution to store DTs in optimized width DBCs. Every tree node is split into three components: (1) the split value/feature index, which is used to decide on an incoming data tuple to follow the tree further to the left or right; (2) the left child pointer, and (3) the right child pointer. All these three components are placed in separate DBCs at synchronized indices, leading to one DBC for right child pointers, one for left child pointers, and one for split values and feature indices. It should be noted here that all DBCs are assumed to have the same width, such that they can be arbitrarily allocated to the split values or pointer values. As the indices need to be synchronized (i.e. the right pointer of node n_x has the same index in the right pointer DBC as the left pointer in the left pointer DBC), the placement I is modeled in the same manner as before. The central advantage of the decomposed DTs is that the width of the DBCs is reduced, and the right pointer and left pointer DBCs do not need to store leaf nodes which can result in a considerable reduction in the memory footprint of the DTs (of $\approx 33\%$). From the programming perspective, only few changes are required to access the decomposed organization during inference. In the unified organization, every tree node is stored as one object in an array, thus access to the three node elements require access at the corresponding array index and the according offset within the object. For the decomposed organization, the three node components are stored as three different objects in three arrays. Thus, the array index for the current node stays the same, but instead of accessing different offsets within one object, accesses for the same index in different arrays need to be performed. This induces minor changes of the decision tree code.

6.4.3 Decomposed Tree Optimization

Although the decomposition can be realized straightforwardly, it yields a different optimization objective. The DT inference causes a different cost in the decomposed structure. Eventually, an optimal placement for a unified DT may not be optimal for its corresponding decomposed tree. Therefore, the upper bound of the previously introduced BLO algorithm (Section 6.3) needs to be revisited, respecting the modified structure of an optimal placement. In order to formalize the decomposition, the following notation is used:

$C_{down}^{decomp}/C_{up}^{decomp}/$ C_{total}^{decomp}	denotes the cost for an unconstrained arbitrary placement I to execute the tree in decomposed DBCs.
$C_{down}^{*decomp}/C_{up}^{*decomp}/$ C_{*total}^{decomp}	denotes the cost in decomposed DBCs for an optimal placement $I^{*decomp}$, which optimizes $C_{total}^{*decomp}$.
$\overleftarrow{C}_{down}^{*decomp}/\overleftarrow{C}_{up}^{*decomp}/$ $\overleftarrow{C}_{total}^{*decomp}$	denotes the cost for an optimal placement $\overleftarrow{I}^{*}$ with the root on the left most position, which is caused on decomposed DBCs and optimizes $\overleftarrow{C}_{down}^{*}$.

Table 6.3: Decomposition Notation

It should be noted here that the cost is considered as a number of shifts within the DBCs. A DBC shift in RTM is different from the bit shifts, which are dependent on the DBC width. Hereby, shifts are counted for the unified organization scenario with the same weight as shifts for the decomposed organization scenario to make the cost definitions comparable and relate them. However, when it comes to the realization of the decomposed DBCs, every shift contributes $\frac{1}{3}$ to the bit shifts and energy consumption compared to a single shift in the unified DBC. Hence, if a placement results in $3\times$ the cost on decomposed DBCs as on unified DBCs, ultimately, the energy consumption penalty is roughly the same in both cases.

For the rest of this section, the cost model for the decomposed approach is first revisited and the objective is defined. Subsequently, the upper bound on the BLO placement is analyzed.

During inference of the decomposed tree, the split value always has to be checked first. Thus, the split value DBC has to be shifted to every node during inference and therefore features the same cost for following the tree down ($C_{split,down}^{decomp}$) and back to the root ($C_{split,up}^{decomp}$) as for the unified organization approach:

$$C_{split,down}^{decomp} = \sum_{n_x \in N \setminus \{n_0\}} absprob(n_x) \cdot |I(n_x) - I(P(n_x))| \tag{6.14}$$

$$C_{split,up}^{decomp} = \sum_{n_x \in N_l} absprob(n_x) \cdot |I(n_x) - I(n_0)| \tag{6.15}$$

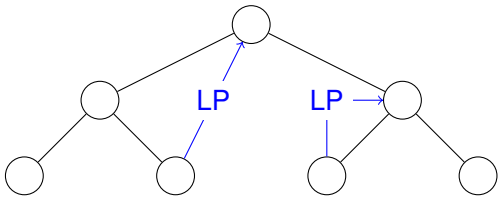


Figure 6.3: Illustration of the left most parent of a node (2 exmaples)

For the right pointer and left pointer DBC, the decision to shift to a certain index depends on the previous decision on the split value. Indeed, only the right pointer DBC or the left pointer DBC needs to be shifted for any node, but never both. Constructing the cost for this requires additional definitions. In the following, the left child of node n_x is denoted by $LC(n_x)$ and the right child by $RC(n_x)$, respectively:

Definition 4. $path(n_x, n_y) = \{n_{i_1}, n_{i_2}, ..., n_{i_m}\}$ is defined as a part of a root leaf path where $n_{i_1} = n_x$ and $n_{i_m} = n_y$ and $P(n_{i_x}) = n_{i_{x-1}}$ or as the empty set if n_x is neither a direct, nor an indirect parent of n_y.

Definition 5. $isleft(n_x)$ is defined for all nodes $n_x \in N \setminus \{n_0\}$ as 1 if $n_x = LC(P(n_x))$ and as 0 for all other cases. $isright(n_x)$ is symmetrically defined for all nodes $n_x \in N \setminus \{n_0\}$ as 1 if $n_x = RC(P(n_x))$ and as 0 for all other cases.

Definition 6. $LP(n_x)$ is defined as the leftmost parent of node n_x for all nodes $n_x \in N \setminus \{n_0\}$:
$\forall n_y \in path(LP(n_x), n_x) \setminus \{LP(n_x)\} : LC(n_y) \notin path(LP(n_x), n_x) \wedge LC(LP(n_x)) \in path(LP(n_x), n_x)$
If such a node does not exist, $LP(n_x) = \varepsilon$. In other words, the leftmost parent is the closest node to n_x on its path from the root, where the left child is taken (illustrated in Figure 6.3).
$RP(n_x)$ is symmetrically defined as the rightmost parent of node n_x for all nodes $n_x \in N \setminus \{n_0\}$:
$\forall n_y \in path(RP(n_x), n_x) \setminus \{RP(n_x)\} : RC(n_y) \notin path(RP(n_x), n_x) \wedge RC(RP(n_x)) \in path(RP(n_x), n_x)$
If such a node does not exist, $RP(n_x) = \varepsilon$.

These definitions imply that for all nodes $n_y \in path(LP(n_x), n_x) \setminus \{LP(n_x)\}$ in between a node n_x and $LP(n_x)$, $isleft(n_y) = 0$. This also holds symmetrically for the RP definition. With the help of Definition 5 and Definition 6 every node within the tree can be investigated and the shifting distance in the left pointer and right pointer DBC can be computed if that specific node requires an inference of the right or left pointer DBC. This leads to the cost for following the right and left pointer DBC down:

$$C^{decomp}_{lptr,down} = \sum_{n_x \in N \setminus \{n_0\}} absprob(n_x) \cdot isleft(n_x) \cdot |I(P(n_x) - I(LP(P(n_x))))| \tag{6.16}$$

$$C^{decomp}_{rptr,down} = \sum_{n_x \in N \setminus \{n_0\}} absprob(n_x) \cdot isright(n_x) \cdot |I(P(n_x) - I(RP(P(n_x))))| \tag{6.17}$$

For simplicity, $|x, \varepsilon| = 0$ for an arbitrary number x. The cost for going up the tree between two inferences is not necessarily the cost for shifting back to the root in the left pointer and right pointer DBC. Instead, there is a set of nodes, which are candidates to be accessed first in the right and left pointer DBCs, i.e. the nodes n_x where $LP(n_x) = \varepsilon$ or $RP(n_x) = \varepsilon$, respectively. Thus, for computing the estimated cost, all these candidates need to be considered with their respective absolute probabilities:

$$C^{decomp}_{lptr,up} = \sum_{n_x \in N_l} absprob(n_x) \cdot \sum_{n_r : LP(n_r) = \varepsilon} absprob(n_r) \cdot prob(LC(n_r)) \cdot |I(n_r) - I(LP(n_x))| \tag{6.18}$$

$$C^{decomp}_{rptr,up} = \sum_{n_x \in N_l} absprob(n_x) \cdot \sum_{n_r : RP(n_r) = \varepsilon} absprob(n_r) \cdot prob(RC(n_r)) \cdot |I(n_r) - I(RP(n_x))| \tag{6.19}$$

Combining these partial costs, the total cost can be deduced by adding all components:

$$C^{decomp}_{down} = C^{decomp}_{split,down} + C^{decomp}_{lptr,down} + C^{decomp}_{rptr,down} \tag{6.20}$$

$$C^{decomp}_{up} = C^{decomp}_{split,up} + C^{decomp}_{lptr,up} + C^{decomp}_{rptr,up} \tag{6.21}$$

$$C^{decomp}_{total} = C^{decomp}_{down} + C^{decomp}_{up} \tag{6.22}$$

Due to the revisited cost model, the considerations about an optimal decision tree placement to the decomposed DBCs also need to be revisited. The following conducts a proof about the relation of the placement solution produced by the O.L.O. algorithm to the optimal solution. Throughout this, the relation between placements for the unified organization approach, the cost they cause on the decomposed organization, and how a placement for unified DBCs can be constructed from a placement for decomposed DBCs is clarified. First, the relation between the cost C_{total} for an arbitrary placement I on a unified DBC and the cost C^{decomp}_{total} the exact placement causes on decomposed DBCs has to be clarified. Intuitively, the cost for the unified DBC can be seen as the cost for the DBC containing the split and feature values since this DBC has to access every node. In the following, a restructuring of the cost model is considered:

Lemma 5.

$$C^{decomp}_{split,down} = \sum_{n_l \in N_l} absprob(n_l) \cdot \sum_{n_x \in rlpath(n_l) \setminus \{n_0\}} |I(n_x) - I(P(n_x))| \tag{6.23}$$

$$C^{decomp}_{lptr,down} = \sum_{n_l \in N_l} absprob(n_l) \cdot \sum_{n_x \in rlpath(n_l) \setminus \{n_0\}} isleft(n_x) \cdot |I(P(n_x)) - I(LP(P(n_x)))| \tag{6.24}$$

$$C^{decomp}_{rptr,down} = \sum_{n_l \in N_l} absprob(n_l) \cdot \sum_{n_x \in rlpath(n_l) \setminus \{n_0\}} isright(n_x) \cdot |I(P(n_x)) - I(RP(P(n_x)))| \tag{6.25}$$

The cost for following the tree down in decomposed DBCs can be restructured as a per path cost, which is weighted with the absolute probability of the leaf node on this root leaf path.

Proof. From the definition of the tree structure, it is known that probabilities are entirely inherited. Thus, summing up the absolute probabilities of all leaf nodes underneath a certain node n_x must result in the absolute probability of this node: $absprob(n_x) = \sum_{n_l \in leafs(n_x)} absprob(n_l)$. In Equation ((6.23)), each distance between each node and the parent is weighted with exactly this sum of absolute probabilities of underlying leafs, since for every leaf the entire root leaf path is considered. Consequently, Equation ((6.23)) can be rewritten to Equation ((6.14)). The same principle can be applied to Equation ((6.24)) (transforms to Equation ((6.16))) and to Equation ((6.25)) (transforms to Equation ((6.17))). □

Lemma 6.

$$C^{decomp}_{lptr,down} \leq C^{decomp}_{split,down} = C_{down} \tag{6.26}$$

$$C^{decomp}_{rptr,down} \leq C^{decomp}_{split,down} = C_{down} \tag{6.27}$$

The summed cost for shifting down in decomposed DBCs in the left and right pointer tree is smaller than the cost for shifting down in the split value DBC, which is equal to the cost for shifting down in the unified DBC case.

Proof. Entire root leaf paths from the root to a leaf node are considered. According to Lemma 5, each path contributes to the total cost with the shifts along the path and the absolute leaf probability. From the definition of the cost function it is known that $C_{down} = C^{decomp}_{split,down}$. Investigating the cost in the left and right pointer DBCs (Equation ((6.24)) and Equation ((6.25))) two cases need to be distinguished. As this consideration is symmetric for $C^{decomp}_{lptr,down}$ and $C^{decomp}_{rptr,down}$, only the left pointer case is discussed here. Considering an arbitrary root leaf path from the root node to a leaf node $rlpath(n_l) \setminus \{n_0\} = \{np_0, np_1, ..., np_m\}$, it is know from the definition of $isleft$ and LP that for all positions $i_0, i_1, ...$ on the path where $isleft(np_{i_x}) = 1$, $LP(P(np_{i_x})) = P(np_{i_{x-1}})$, i.e. the leftmost parent LP of the parent is always the immediate previous parent node which contributes to Equation ((6.24)). Further, $|I(P(np_x)) - I(LP(P(np_x)))| \leq \sum_{np_y \in path(np_x, LP(np_x)) \setminus \{np_x, LP(pn_x)\}} |I(np_y) - I(P(np_y))|$ since an arbitrary path between two indices cannot be shorter than the direct path. If for a certain node n_x on a path $P(n_x) = LP(n_x)$, $isleft(n_x)$ must be 1 by definition. This node then contributes the same cost to $C^{decomp}_{lptr,down}$ as $P(n_x)$ contributes to $C^{decomp}_{split,down}$ on the specific path. If this was the case for all nodes on a path, $C^{decomp}_{lptr,down} < C^{decomp}_{split,down}$ because the shift to the leaf node is not considered in the left pointer DBC. If, however, $LP(n_x) \neq P(n_x)$ on

any path for any node, it is known from the definition of $isleft$ already that all nodes on the path in between n_x and $LP(n_x)$ do not contribute to $C^{decomp}_{lptr,down}$ since $isleft$ must be 0. Thus, the contributed cost to $C^{decomp}_{lptr,down}$ for this specific node is at most the contributed cost of this node and the omitted nodes ($isleft = 0$) to $C^{decomp}_{split,down}$. In total, $C^{decomp}_{lptr,down} \leq C^{decomp}_{split,down}$ and $C^{decomp}_{rptr,down} \leq C^{decomp}_{split,down}$. □

Lemma 7.

$$C^{decmp}_{down} \leq C^{decomp}_{total} \tag{6.28}$$

The cost for following the tree down in a decomposed placement is a part of the total shifting cost (compare to Lemma 1).

Proof. C^{decomp}_{total} is the sum of C^{decomp}_{down} and C^{decomp}_{up}, where C^{decomp}_{up} itself is a sum of non-negative terms. □

Lemma 8.

$$C_{down} \leq C^{decomp}_{down} \tag{6.29}$$

The summed cost for shifting through the decomposed DBCs while following the tree downwards is at least the cost of shifting through a tree on a unified DBC downwards with the same placement.

Proof. From the definition of the cost function, it is known that $C_{down} = C^{decomp}_{split,down}$. It is further known that $C^{decomp}_{rptr,down}$ and $C^{decomp}_{lptr,down}$ only consists of a sum of terms which are either 0 or positive. According to Equation ((6.20)), C^{decomp}_{down} is the sum of only these three components. Thus, $C_{down} = C^{decomp}_{split,down} \leq C^{decomp}_{down}$. □

Next, the cost relation of a linear allowable placement produced by OLO needs to be considered. As reported by Adolphson and Hu, there is always a linear allowable placement, which features the optimal cost C_{down} under the constraint that the root is placed to the leftmost position [AH73]. Thus, the cost of such an optimal linear allowable placement is denoted in the following by $\overleftarrow{C}^{*\cdots}_{\cdots}$.

Lemma 9.

$$\overleftarrow{C}^{*decomp}_{lptr,up} \leq \overleftarrow{C}^{*decomp}_{split,up} = \overleftarrow{C}^{*}_{down} \tag{6.30}$$

$$\overleftarrow{C}^{*decomp}_{rptr,up} \leq \overleftarrow{C}^{*decomp}_{split,up} = \overleftarrow{C}^{*}_{down} \tag{6.31}$$

The cost for shifting up in the left and right pointer DBCs in a linear allowable placement can be upper bounded by the cost for shifting up in the split value DBC, which is the same cost as shifting down in the unified DBC case.

Proof. $\overleftarrow{C}^{*decomp}_{split,up} = \overleftarrow{C}^{*}_{up}$ directly follows from the definition of the cost functions (Equation ((6.15)) and Equation ((6.2))). $\overleftarrow{C}^{*}_{up} = \overleftarrow{C}^{*}_{down}$ follows from Lemma 3. By investigating Equation ((6.15)) and Equation ((6.18)) the outer sum is over the same (leaf)

nodes. In a linear allowable placement, n_0 must have the left most position, further $I(LP(n_x)) < I(n_x)$ since LP is an indirect parent relation. Thus, all terms $|I(n_r) - I(LP(n_x))| \leq |I((n_x) - I(n_0)|$. The nodes considered in the inner sum of Equation ((6.18)), namely $n_r : LP(n_r) = \varepsilon$, must form a single consecutive path of nodes where always the right child is taken by definition. Each node on the path contributes a certain portion of their absolute probability ($absprob(n_r) \cdot prob(LC(n_r))$), the remaining part is inherited to the right child by definition, which then itself contributes a part of the inherited probability. Thus, $\sum_{n_r : LP(n_r) = \varepsilon} absprob(n_r) \cdot prob(LC(n_r)) \leq absprob(n_0) = 1$. Consequently, the inner sum is a weighted average of upper bounded terms, thus the entire sum can be upper bounded by $|I((n_x) - I(n_0)|$. The case for the right pointer DBC is symmetric. □

Corollary 2.

$$\overleftarrow{C}^{*decomp}_{total} \leq 6 \cdot \overleftarrow{C}^{*}_{down} = 3 \cdot \overleftarrow{C}^{*}_{total} \tag{6.32}$$

If a linear allowable placement is deployed to decomposed DBCs, the total cost for shifting through the decomposed DBCs is at most $6\times$ the cost of shifting the unified DBC downwards.

Proof. Equation ((6.32)) follows from the definition of the cost model (Equation ((6.22))) and Lemma 9, Lemma 6 and Lemma 3: $\overleftarrow{C}^{*decomp}_{lptr,down} \leq \overleftarrow{C}^{*}_{down}$, $\overleftarrow{C}^{*decomp}_{rptr,up} \leq \overleftarrow{C}^{*}_{up}$, $\overleftarrow{C}^{*decomp}_{split,down} = \overleftarrow{C}^{*}_{down}$, $\overleftarrow{C}^{*decomp}_{lptr,up} \leq \overleftarrow{C}^{*}_{up} = \overleftarrow{C}^{*}_{down}$, $\overleftarrow{C}^{*decomp}_{rptr,down} \leq \overleftarrow{C}^{*}_{up} = \overleftarrow{C}^{*}_{down}$, $\overleftarrow{C}^{*decomp}_{split,up} = \overleftarrow{C}^{*}_{up} = \overleftarrow{C}^{*}_{down}$. In total, $\overleftarrow{C}^{*decomp}_{total}$ consists of 6 terms, which are all upper bounded by $\overleftarrow{C}^{*}_{down}$. Lemma 3 further leads to $\overleftarrow{C}^{*}_{total} = 2 \cdot \overleftarrow{C}^{*}_{down}$. □

Combining the above considerations, the according upper bound can be constructed.

Theorem 2.

$$\overleftarrow{C}_{down} \leq 2 \cdot C^{decomp}_{total} \tag{6.33}$$

Any placement for decomposed trees can be transformed into a placement with the root on the left most position, where the cost for following the tree downwards in a unified DBC is at most $2\times$ the cost for executing the entire tree on decomposed DBCs.

$$\overleftarrow{C}^{*decomp}_{total} \leq 12 \cdot C^{*decomp}_{total} \tag{6.34}$$

An optimal linear allowable placement for shifting downwards in a unified DBC, as obtained by OLO, is an upper bound of 12 of the optimal placement for decomposed DBCs.

Proof. Equation ((6.33)) directly follows from Lemma 7,Lemma 8 and Lemma 4.

Equation ((6.34)) can be proven by contradiction. Suppose that the optimal linear allowable placement for a unified DBC $\overleftarrow{C}^{*}_{down}$ would cause a cost $\overleftarrow{C}^{*decomp}_{total}$ larger than $12\times$ of the optimal placement for decomposed DBCs $C^{*decomp}_{total}$. According to Corollary 2, it is known that the optimal placement must have at least a cost of $\frac{1}{6}$ on the unified

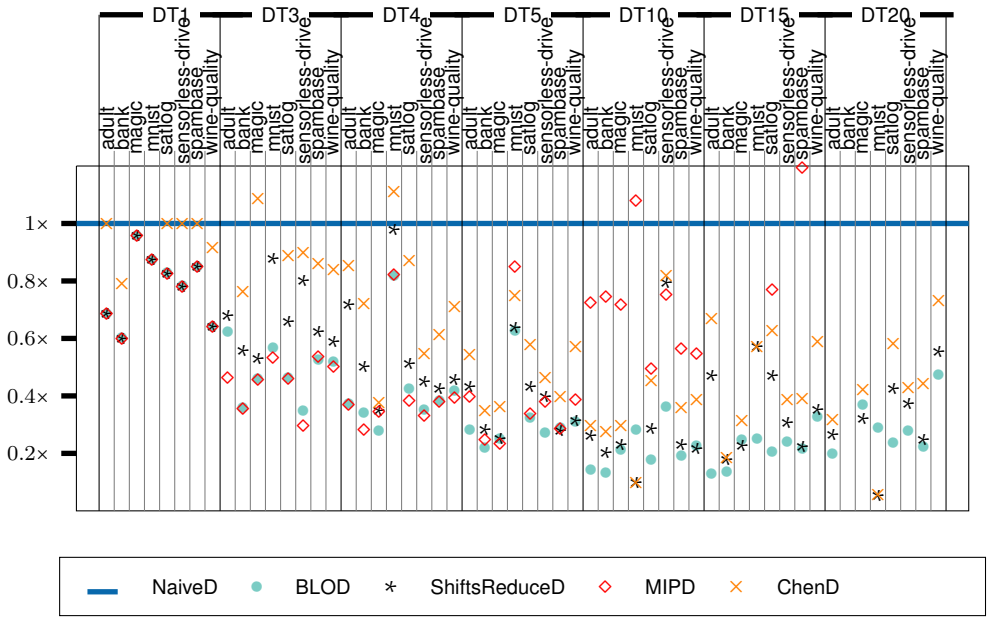


Figure 6.4: Comparison of Total Shifts During Inference on Decomposed Trees

DBC then, thus $\overleftarrow{C}^*_{down} > 12 \cdot \frac{1}{6} \cdot C^{*decomp}_{total} \Leftrightarrow \overleftarrow{C}^*_{down} > 2 \cdot C^{*decomp}_{total}$. It is further known that according to Equation ((6.33)) a solution for the unified DBC with a cost less than $2 \cdot C^{*decomp}_{total}$ can be built, which contradicts the optimality of $\overleftarrow{C}^*_{down}$. □

The BLO heuristic (Section 6.3) can be applied to the decomposed organization scenario without any limitation. The consideration that the BLO extension does not introduce additional shifting cost, however, it does not remain valid for this scenario. Potentially, the left or right pointer DBC can be shifted from a certain node within the right subtree to another node within the left subtree, without loading the root and vice versa. Thus, both nodes may be placed closer in the OLO placement as in the BLO placement. However, the proof upper bounds the cost for going up and down in the left and right pointer DBCs with the cost for the split value DBC, i.e. with the cost of starting at the root and ending at a leaf in Lemma 9. Theorem 2 consequently takes this bound in to determine the ultimate upper bound. Hence, under this worst-case scenario, e upper bound of $12\times$ is valid for BLO and OLO.

6.4.4 Evaluation

In addition to the proven upper bound of the BLO algorithm on the decomposed organization, this section presents experimental evaluation of the BLO algorithm and provides a comparison to the state-of-the-art. The proven upper bound for BLO consequently holds for the state-of-the-art methods, since these cannot achieve better performance than the optimum. The relation between these approach in realistic scenarios, however, is empirically studied in this section. First, the shifts' reduction of different solutions is discussed and then the impact of shifts reduction on the runtime and energy consump-

tion is shown. As a setup for the evaluation, the setup from Section 6.3.4 is used. To provide a comparison between the unified and decomposed organization, the setup is modified in order to account for the different organization approaches. This leads to the comparison of the following versions:

- *Naive / NaiveD:* A baseline breadth-first order placement in which indices are assigned to tree nodes layer-wise in increasing order. The placement is used for the unified (Naive) and decomposed organization (NaiveD).
- *ShiftsReduce / ShiftsReduceD:* The state-of-the-art data placement algorithm from [KHB+19]. We evaluate the heuristic on the unified organization (ShiftsReduce) and the decomposed organization (ShiftsReduceD).
- *Chen / ChenD.:* The data placement algorithm from [CSZ+16], evaluated on the unified organization (Chen) and the decomposed organization (ChenD).
- *BLO / BLOD:* Bidirectional linear ordering for unified trees. It is evaluated on the unified and decomposed organization.
- *MIP / MIPD:* The mixed integer programming formulation of the cost model (Equation ((6.4)) for unified organization and Equation ((6.22)) for decomposed organization). The solver, in case it converges, returns the optimal tree placement.

The node access trace for all configurations are replayed to derive the total amount of required racetrack shifts. For the decomposed trees, the performance and energy numbers reported in this section consider all, i.e., the split value and pointers DBCs. In addition to the pure amount of shifts, the runtime and energy consumption is modeled for the unified and decomposed organization. The runtime per-access and per-shift latencies and the energy consumption is provided in Table 6.4.

Ports per track, domains per track		1, 64
Tracks per DBC: unified, decomposed		96, 32
Leakage power [mW]: unified, decomposed	p	36.2, 36.9
Write energy [pJ]: unified, decomposed	e_W	106.8, 40.7
Read energy [pJ]: unified, decomposed	e_R	62.8, 23.4
Shift energy [pJ]: unified, decomposed	e_S	51.8, 17.3
Write latency [ns]: unified, decomposed	l_W	1.79, 1.75
Read latency [ns]: unified, decomposed	l_R	1.35, 1.32
Shift latency [ns]: unified, decomposed	l_S	1.42, 1.39

Table 6.4: RTM parameters values for a $128\,\mathrm{KiB}$ SPM considering a decomposed organization

Figure 6.4 evaluates the reduction in terms of RTM shifts for the decomposed organization approach. The MIP formulation is implemented in the Gurobi optimizer [Bix07] and is given a time limit of 8 hours per dataset and per tree configuration. For the DT1 and DT3 instances in all datasets, the MIP converges to the optimal solution. In all other cases, the results are based on the Gurobi heuristic. Results which are worse than $1.2\times$ of the baseline are not illustrated in the figures.

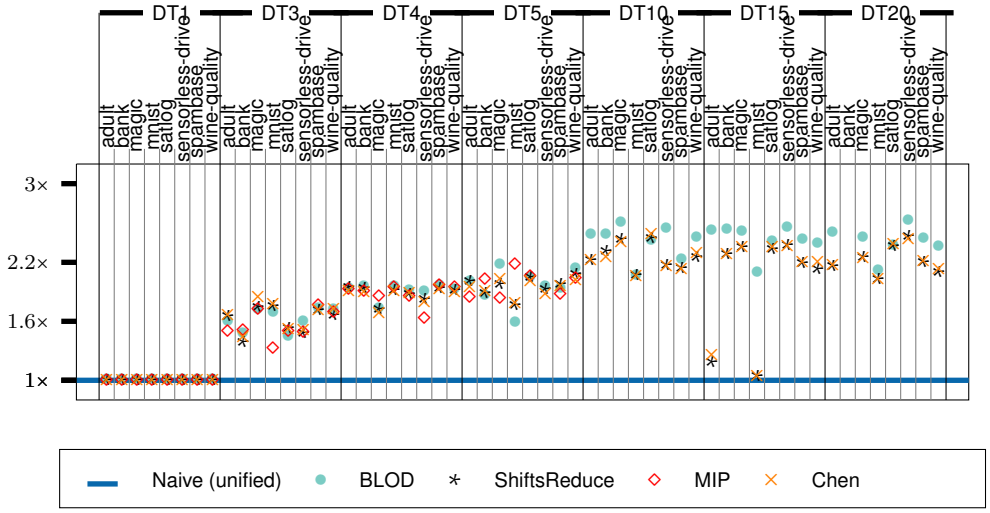


Figure 6.5: Increase of Total Shifts During Inference between Unified and Decomposed Trees

A detailed analysis of the results shows that for the cases where the MIPD finds an optimal placement (for DT1 and DT3), BLOD achieves the same or only marginally worse results than the optimum. Compared to state-of-the-art solutions, it can be observed that BLOD achieves the best reduction in shifts for most of the investigated cases. The reduction of the total shifts does not directly imply a similar improvement in runtime and energy consumption. To estimate the shifts reduction impact on the runtime and energy consumption, a realistic setup as explained in Section 6.3.4 is considered. Larger decision trees are first split into smaller trees, and the placement heuristic is then executed on multiple trees of maximal depth of 5. Note that the assignment of these smaller trees to different DBCs may affect the cost of the overall shift.

Unified vs. Decomposed DTs

Although the previous results report the performance of the BLOD algorithm on the decomposed trees, the relation between the unified and decomposed layout remains an open question, especially which of both realizations should be used for a concrete system remains open. Equation ((6.32)) implies that any linear allowable placement cannot cause more than $3\times$ shifts on the decomposed DBCs as on the unified DBCs. Under the ideal assumption that each single DBC in the decomposed setup only needs $\frac{1}{3}$ of bit-lines and therefore also only yields $\frac{1}{3}$ of the energy consumption, the decomposed setup cannot be worse than the unified setup in no scenario. In reality, however, constructing the decomposed setup may create additional static overheads or consume additional resources (such as chip space or leakage power), which is only desirable if the decomposed setup can significantly reduce the resource consumption.

In order to assess the resource savings when considering the decomposed setup, the placement of all configurations is taken and the node access traces are replayed

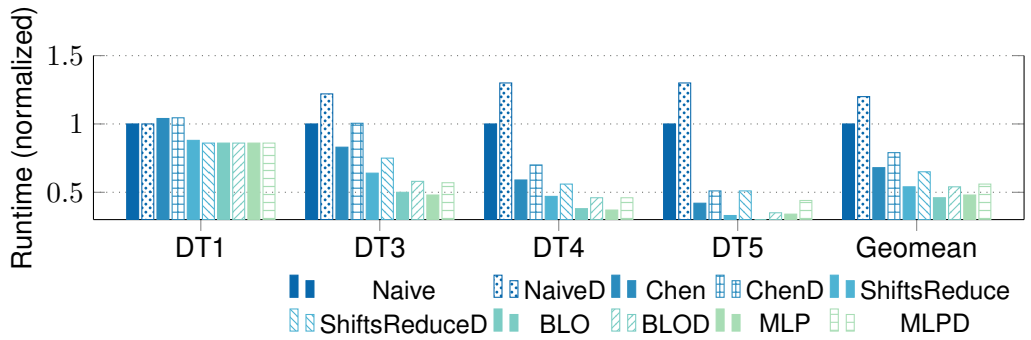


Figure 6.6: Runtime of different configurations for different tree size. The results are average across all benchmarks.

on the unified and decomposed organizations. The relation of the total amount of shifts for all configurations in the unified and decomposed approaches are computed. Theoretically, the ratio between the unified shifts and the decomposed shifts must range between $1\times$ and $3\times$. This is evaluated, and the ratios are shown based on experimental results in Figure 6.5. For trees with a maximum depth of 1 i.e., DT1, the decomposed and unified approaches result in exactly the same amount of shifts in all placements. This is because a DT1 has two levels, thus only a single node with pointers which is mapped to the first location in a single DBC (unified) or multiple DBCs (decomposed). Therefore, no shifts in the right and left pointer DBC are required. Note that it is assumed that the access ports in all DBCs are initially aligned to the first position. For deeper trees, the increase in shifts ratio shows similar trend for all placement approaches. For the deepest trees considered in this evaluation, the number of shifts in the decomposed trees can be as high as 2.59 for the BLO algorithm.

In the decomposed organization, the highest shift reduction is expected from scenarios where the pointer DBCs are rarely shifted. For DT1, the best case is achieved because the left and right pointer DBCs do not need to be shifted at all. As the trees get deeper, the probability of frequently accessing left and right pointers also increases. Thus, for deeper trees the shifts reduction in the decomposed setup is reduced, which can be seen in the reported results as well.

However, focusing on the realistic tree sizes of at most three or four layers, which can be placed into a single DBC, the experimental data suggests that the amount of shifts is increased by at most a factor of $2\times$ when switching to the decomposed setup. This is a considerable margin to leverage static overheads from the decomposition and provide a reduction in the total resource consumption.

Runtime and Energy

BLO reduces the total runtime by 53.8% compared to the naive placement, as shown in Figure 6.6. In comparison, for the same baseline, ShiftsReduce and BLOD reduce the total runtime by 45.7% and 46.3%, which are 13.3% and 13.9% longer compared to the

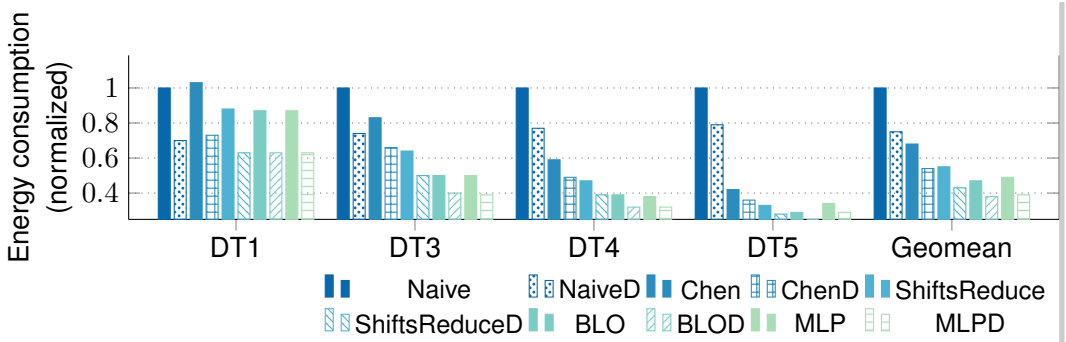


Figure 6.7: Energy consumption of different configurations for different tree size. The results are average across all benchmarks.

BLO, respectively. Comparing this to the reduction of shifts for trees with maximum depth of 5 only, BLOD reduces the required shifts by 85.1%, BLO by 77.5%, and ShiftsReduce by 72.4%. Thus BLOD, compared to BLO and ShiftsReduce, further reduces the amount by shifts by 9.8% and 17.5% respectively. This suggests that a reduction in shifts may not necessarily result in the runtime reduction, or at least not with the same proportion. When comparing Figure 6.2 and Figure 6.4 to Figure 6.6 and Figure 6.7, the different scaling on the y-axis and that results are averaged across datasets for the latter figures should be noted.

In the decomposed placement approach, the total runtime increases due to the alignment time in the pointers DBCs. The split value DBC is checked first to determine whether a pointer DBC needs to be accessed or not. Subsequently, depending on the node access probabilities, a shift request may be sent to the left or the right pointer DBC. The lazy shift approach in pointers DBCs improves the overall shift energy due to the reduced amount of shifts. However, this negatively impacts the runtime due to the shift penalty required to align the access port to the desired location if it is not aligned with the split value DBC. To quantify the impact of the decomposed approach on the runtime, it is compared with other methods, as presented in Figure 6.6. For the same baseline (naiveD), BLOD has an average runtime overhead of 7.5% compared to BLO. Consequently, BLOD also increases the leakage energy compared to BLO. However, this deterioration in the leakage energy is offset by the reduction both in the shift and access component of the energy (cf. Figure 6.8). Similarly, other decomposed approaches (e.g., naiveD, MLPD) induces a runtime penalty compared to their unified counterparts (e.g., naive, MLP).

BLOD achieves the most reduction in energy consumption compared to all other approaches. This is because the total energy consumption of RTM is largely dependent upon the number of bit shifts, which affect the shift energy and the runtime, which determine the leakage energy. Figure 6.7 and Figure 6.8 show the overall energy consumption and the energy breakdown of different placement approaches for the unified and the decomposed DBCs normalized to the naive placement. Compared to

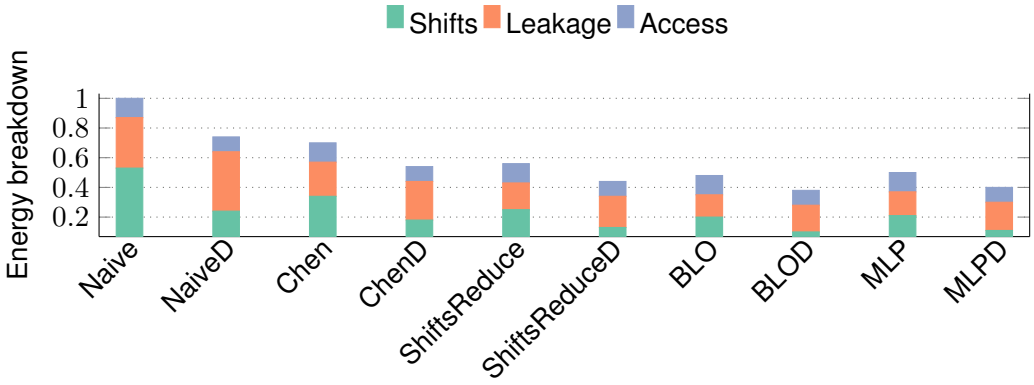


Figure 6.8: Energy consumption breakdown into shifts energy, leakage energy and access energy for various configurations. BLOD records the lowest shift and total energy consumption compared to all other configurations.

the naive solution, BLOD delivers a 61.7% reduction in the RTM energy consumption, compared to 52.6% in BLO and 45.8% in ShiftsReduce for the same baseline.

Figure 6.8 highlights that the energy efficiency of BLOD compared to existing unified approaches is achieved via a significant reduction in the energy consumed by the shift operation and a slight reduction in the access energy. The leakage energy, compared to the naive solution (NaiveD), is also reduced by 44.7%. The improvement in the shift energy is due to reduced shift cost, while the reason for the leakage energy saving is the reduced runtime (cf. Figure 6.6). Compared to the unified BLO solution, despite an increase in the leakage energy by 16.2%, the decomposed approach consumes 17.3% less energy. Overall, for the naive baseline (Naive), BLOD on average achieves (95.3%, 35%, 21.5%, 17.3%, 150%, 1.7%) more energy reduction compared to (Chen, ShiftsReduce, MLP, BLO, naiveD, MLPD).

6.4.5 Wrap-Up

While the previous section (Section 6.3) discusses the optimization of the shifting cost of a DECISION TREE (DT) in RACETRACK MEMORY (RTM) for a unified layout, this section refines the basic organization assumption to allow a decomposed layout and revisits the resulting optimization problem of the shifting cost. The decomposed layout refers to the organization approach to store DT nodes in the RTM DBCs. While in the unified layout an entire node is stored in one domain, requiring a particular shift of a single DBC, the decomposed layout splits tree nodes into three components and stores them in three different DBCs. This allows each of the DBCs to be shifted individually. The nodes are split into a left child pointer component, a right child pointer component and a remaining component, including the feature index and split value.

This section revisits the analysis of the previously presented BLO algorithm and proofs the upper bound of $12\times$ to the optimal solution in the decomposed organization

approach. Assuming a generally reduced cost of $\frac{1}{3}$ for shifts in the decomposed DBCs due to the reduced required size of $\frac{1}{3}$, the decomposed layout cannot perform worse than the unified layout, at least in the upper bound. Experimental evaluation shows that the required overhead for shifts is realistically reduced for the decomposed layout when compared to the unified layout.

6.5 Concluding Memory Optimization of Random Forests

In order to exploit the traditional memory interface for latency optimization for RTM in a software centric manner, two components are of crucial importance: 1) a reduced conditional memory access distribution, which can serve as a basis for computation and 2) a memory location optimization strategy, taking the aforementioned model as an input and derive an optimized memory mapping. Then, the memory mapping can be employed through the traditional memory interlace. Achieving a significant reduction of the conditional memory access distribution is achieved by the methods in this chapter due to an application specific consideration of tree based data structures, i.e. DTs. The focus on DTs further allows a direct collection of local empirical probabilities, due to the inherent relation to a data set distribution. As an optimization strategy, the methods in this chapter use an efficient placement algorithm, which is not optimally suited to the case of RTM, but can be executed very fast. A formal proof guarantees an upper bound on the outcome of this strategy, compared to an optimal mapping. The decided optimized mapping is applied though an assignment of memory locations to data objects, which is compatible to the traditional memory interface.

This chapter shows a possible way to exploit the possibility for data placement in the traditional memory interface to achieve a latency optimization for a modern memory technology, with an uncommon access dependent latency property. This exploitation is achieved by only introducing adequate software components directly in the application and into an optimizer.

CHAPTER 7

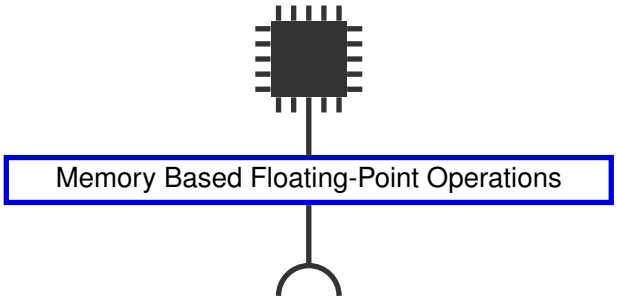
CPU Optimization for Random Forests

Contents

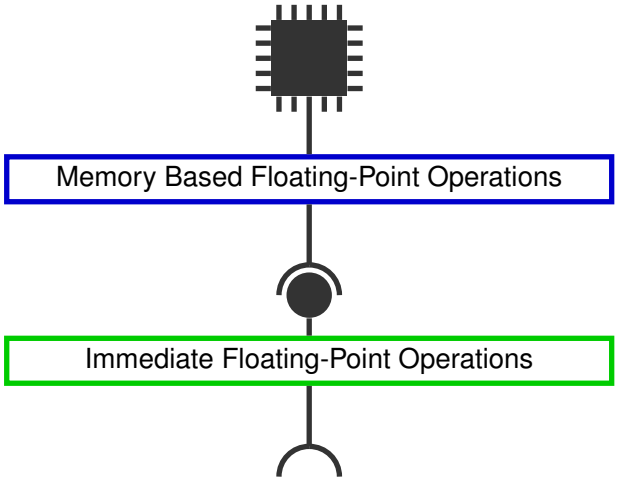
7.1 Modern Technologies and Traditional Interfaces 132
7.2 Overview 133
7.2.1 Numeric Formats 133
7.2.2 Memory Encoding and Hierarchy 134
7.3 Immediate Encoding of Floating-Point Split Values 136
7.3.1 Scope 136
7.3.2 Problem Analysis and Statement 136
7.3.3 Immediate Encoding 140
7.3.4 Evaluation 142
7.3.5 Wrap-Up 147
7.4 FLInt: Exploiting Floating-Point Enabled Integer Arithmetic for Efficient Random Forest Inference 148
7.4.1 Scope 148
7.4.2 Problem Analysis and Statement 149
7.4.3 Providing Correct Floating-Point Comparisons with Integer and Logic Arithemtic 149
7.4.4 Evaluation 157
7.4.5 Wrap-Up 163
7.5 Concluding CPU Optimization for Random Forests 164

7.1 Modern Technologies and Traditional Interfaces

While the previous chapters of this thesis studies interface exploitation around the traditional memory interface, enabling management for modern memory technologies, this chapter focuses on another traditional memory interface, namely the execution of floating-point operations in CPUs. The traditional interface for floating-point operations is formed by CPU instructions, which process floating-point values in registers, which may be general purpose or dedicated floating-point registers. The traditional way to fill such a register with a value, is to perform a memory load:



Comparing this to integer operations, it can be observed that the functionality of immediate encoding for floating-point values is not provided. For immediately encoded integer operations, the value is directly encoded in the CPU instruction itself. Consider, for instance, an imaginary ISA, where registers are named `r0, r1, ...` and an addition instruction with an immediate constant exists: `add r0, r1, #42`. The constant value `42` would be stored as a part of the encoded instruction in the instruction memory directly, and hence allows immediate encoding. For floating-point values, such an option is traditionally not available. Instead, a floating-point value of `42.0` would have to be placed at a dedicated memory position, e.g. `0xAB00`, and loaded for a floating-point operation: `load r2, 0xAB00; fadd r0, r1, r2`. Although this does not cause any functional disadvantage, the possible design of arithmetic operations is consequently limited for floating-points in comparison to integers. Especially for low level optimization for RFs, exploring large design spaces is of crucial interest. Consequently, this chapter aims to exploit the traditional interface by software in order to provide immediate operation of floating-point values. To gain efficiency, the application domain is specifically studied for RFs and specific solutions are derived:



7.2 Overview

The previous chapter (Chapter 6) discusses the impact of RANDOM FOREST (RF)) inference on the memory subsystem, especially the optimization of the shifting cost in RTM. The memory subsystem, however, is not the only subsystem specifically impacted by RF execution and consequently offering potential for optimization. This chapter focuses on the impacts of the CPU subsystem, specifically the arithmetic computation to RF inference. Since the inference of RFs consists of repetitive execution of tree nodes, optimizing the arithmetic operations of a single node can have a large impact on the entire tree. While training of DTs and RFs requires a rather complex set of arithmetic operations, inference is a very simple algorithm. The basic operation is the comparison between the incoming feature value against a certain threshold and following the decision whether the value is larger or smaller. Comparison, i.e. computing the $\leq$ relation, for instance, is a rather simple operation, since the result is a single comparison bit and not a numeric result, as for other arithmetic operations. Consequently, comparison operations open a certain space for optimization. The techniques, which are presented in this chapter, impact the computation of the comparison operation partially, as well as the memory subsystem. Hence, the basic impact schemes are explained in the following.

7.2.1 Numeric Formats

Numeric computation in CPUs can be roughly divided into integer and floating-point computation. Even though integer computation itself covers number formats with various bit widths, signed and unsignedness and probably further specialities, all these formats can be efficiently combined into a single arithmetic unit by implementing a proper

processing and interpretation of computation results, respecting the considered integer format. Floating-point computations, in contrast require a largely different arithmetic unit, since the algorithms to compute arithmetic operations are different from integer operations. Consequently, CPUs implement floating-point units in a different hardware unit than integer arithmetic. In addition, floating-point values itself cannot be directly reinterpreted as integer values or vice versa. This leads to the realization of dedicated floating-point registers in many CPUs, such that integer values and floating-point values are stored and processed in different register sets.

This realization scheme can have at least two potential impacts on the execution performance. First, CPUs may be more optimized for integer computations, such that, for instance, a simple addition can take more time in average on a floating-point value than on an integer value. Second, the handling of floating-point values in dedicated floating-point registers can involve extra steps, as for instance copying values between integer and floating-point registers when memory operations cannot directly load values to floating-point registers. This can lead to negative performance impacts for floating-point values as well.

When discussing RFs, the comparison operation should be considered as a premier target. It is reported, that some CPUs, although implementing dedicated floating-point units, use the integer unit to perform the comparison of floating-point numbers with a dedicated post-processing [Bra]. This surpasses possible performance impacts due to slow floating-point units for RFs largely. However, the dedicated handling of floating-point registers remains as a possible cause for performance impacts. Hence, optimization potential when surpassing the need for the use of floating-point units is existent.

It should be noted at this point that the decision whether a RF operates on integer or floating-point data stems from the data source, which is used for inference. If the input data is provided as floating-points, the RF is naturally trained with floating-point comparison values. The RF can be transformed to use integers values instead, however, a potential accuracy loss can be induced. In addition, this requires an additional conversion step for the incoming data. Hence, this version is not considered in the following. This chapter discusses methods for performance optimization when RFs have to operate on floating-point data.

7.2.2 Memory Encoding and Hierarchy

Despite the direct effect from requiring different hardware units for floating-point computation, whether floating-point or integer values are used in a RF can have, among other causes, indirect effects on the performance. In order to elaborate these effects, this subsection gives a short overview on possible effects on the memory hierarchy. Due to the wide presence of caches in memory subsystems of modern CPUs, the memory access latency, i.e. the memory performance, cannot be assumed to be uniform for the entire memory. Instead, memory accesses hitting the cache can be processed significantly faster than other memory accesses. Knowing if a certain memory element will be stored in a cache at a certain level is a complex question. For most CPUs, this

question even cannot be answered because details of the cache management are not known usually. However, if the pressure on the cache is higher, i.e. the application makes memory accesses to lots of different memory elements in a short time, chances are lower that memory accesses will be cache hits. If, on the other side, the cache pressure is low, i.e. the application only makes use of a few memory elements, chances are higher that memory accesses will be cache hits. Despite the pure pressure on the cache, the frequency of accesses to memory elements plays a major role for most CPUs. In detail, the more often a memory element is accesses within a program, the higher chances are that an access will be cache hit. Although this effect is not further studied in this thesis, it should be noted that RFs offer a strong cache optimization potential in this regard due to the probabilistic model, which can be exploited. [CSH+22] is a contribution in this scope, which, however, is not further detailed in this thesis.

The organization of CPU caches is not only hierarchical, i.e. different levels of caches, but also splits data and instruction memory for many CPUs at a certain level. In detail, a separate data and instruction cache is present for many CPUs, usually in the first cache level. In consequence, at least for this level, pressure on the instruction and data memory can be considered separately. For instance, even though a program may use many different data objects, instruction accesses can still be very likely cache hits when only a few instructions are executed. Although management strategies of instruction and data caches may be highly complex, leveling the pressure on the instruction and data cache can be a good approach in order to optimize the performance of an application. In the context of DTs in RFs, the pressure on instruction and data memory can be directly analyzed. While for native trees (Listing 3.1 and Listing 3.2), a low pressure on the instruction memory and a high pressure on the data memory is mandatory, if-else trees (Listing 3.3) offer a certainly controllable trade-off. For if-else trees, the entire tree is encoded in instruction, which suggests the intuition that only a high pressure on instruction memory is caused. Comparison values, however, may be stored in data memory and loaded during the execution, which creates pressure on the data memory.

In order to elaborate the option to store comparison values in the code itself or in the data memory, immediate encoding of constants needs to be considered. The split values are known during implementation time and hence known to the compiler as constants. In order to compile the comparison, largely independent of the used ISA, two options can be considered. First, comparison values can be stored in a memory array, a load instruction is placed loading the value to a register and a comparison instruction performs the actual comparison between two registers, when the incoming feature value was loaded to another register. Second, if the ISA offers that option, the incoming feature value can be loaded to a register, and an immediate comparison instruction is used, which takes one register as an operand and an immediate bit field as a second operand. It should be noted, that such immediate encoding of comparison values is limited to a certain bit width less than the instruction width naturally. If larger data types are used in the RF, multiple instructions may be needed for the immediate loading or the values have to be stored in data memory. Due to the limited bit width, usually only integer numbers are supported.

7.3 Immediate Encoding of Floating-Point Split Values

As explained before, this thesis focuses on the specific impact of RF execution on the number computation and memory subsystem. As a main focus, execution of floating-point numbers in comparison to integer numbers is considered. This section assumes RFs with floating-point split values to be given. The considered CPU architectures (i.e. X86 and ARMv8) both cannot encode floating-point constants in immediate fields. Consequently, the compiler places the floating-point constants in data memory and loads them during execution as a normal memory load. This creates additional pressure to the instruction caches, which potentially causes not optimal cache utilization. Hence, in this section a method is presented, which forces the floating-point constants to be stored in text memory by applying an immediate encoding scheme. Due to the more balanced pressure on the caches, performance improvement is gained.

7.3.1 Scope

As mentioned before, this section introduces a method, which enables the encoding of floating-point constants as immediate constants during the code generation if RFs. In detail, neither the training, nor the logical output of ensemble, nor the logical structure is modified. Only the way, single nodes of the inner DTs are translated into machine code is modified, such that the constants for the split values are not stored in data memory, but in text memory instead. In order to achieve this, the binary encoding of floating-points, i.e the IEEE754-1985 [ECS85] is investigated. This standard determines how floating-point numbers are encoded in, for instance, a 32 or 64 bit register. This encoding, can be also stored as an integer variable. Loading this integer variable to a register and performing a direct copy between a integer register and a floating-point register without reinterpretation can then enable the usage of the floating-point value. This method offers the possibility to encode the corresponding integer in an immediate field of an instruction, i.e. in the text memory. In short, the following contributions are presented in this section:

- An analysis of state-of-the-art RF optimization techniques with their impact on the pressure on data and instruction caches.
- The implementation of immediate encoded floating-point split values in the text memory, shifting the pressure between data and text memory.
- Experimental evaluation with a detailed comparison to the data and instruction cache impact of state-of-the-art RF optimization techniques.

7.3.2 Problem Analysis and Statement

After training of a RF model (e.g. with scikit-learn [PVG+11]), the model is derived in a logic representation (e.g. encoded in JSON). Executing this model without special

published work: [HCC22b]

operating system or library support requires a realization in a programming language and compilation to machine code. The realization of DTs and RFs as if-else trees, as introduced by [ALD13] intensively utilizes instruction caches during the execution, as the entire tree structure is encoded in instructions itself. Only loading of the data point for inference is mandatory from data memory and therefore uses data caches. Listing 7.1 depicts an example of the implementation of a single tree node as an if-else tree in C++.

```
1 if(pX[3] <= (float) 1.500000){
2         return 1;
3 } else { ...
```

Listing 7.1: C++ node example

It can be seen that the loading of the data point (stored in pX) is an array access and therefore a data memory access. The split value, which is used to decide in combination with the data point if the left or right subtree should be further followed, is immediately encoded in the source code, also the prediction value is immediately encoded with the return statement. To illustrate the conversion to assembly code, the assembly code for an X86 machine, produced by the gcc compiler in version 11.1.0 is investigated in the following. Later, both, X86 and ARMv8 architectures are considered.

```
1 movss      0xe50(%rip),%xmm9
2 comiss     0xc(%rdi),%xmm9
3 jmp        2fa0
```

Listing 7.2: Assembly node example

Listing 7.2 illustrates the relevant assembly code for the implementation of the node from Listing 7.1. Line 1 is responsible for loading the split value, Line 2 loads the feature value from the data point and performs the comparison to the split value. Line 3 then performs the according jump. Counter intuitively to the C++ implementation, the split value is not encoded as an immediate value, but rather leads to a data memory load within the `comiss` (compare scalar ordered single-precision floating-point) instruction. Since the X86 instruction set does not offer immediate values for floating-point instructions, the compiler decides to place the split values at a central position in data memory and translate the accesses to regular data loads[1]. It should be noted that the `movss` instruction (which loads a floating-point number to a register) uses the immediate encoding for the offset within main memory, but not for the floating-point constant itself. In consequence, two out of the three relevant instructions for an if-else tree perform data accesses and utilize the data cache. The motivational concept of intensively utilizing instruction memory and caches for if-else trees does not hold all along.

To further illustrate the impact of this condition, a state-of-the-art implementation of if-else trees from Chen et al. [CSH+22] is investigated. Specifically, two possible implementations of the same logic model are studied in the following: 1) a naive implementation of if-else trees, where every node in the logic tree structure becomes an

[1] This observation is not necessarily bounded to the X86 architecture, the ARMv8 architecture neither does offer such a feature.

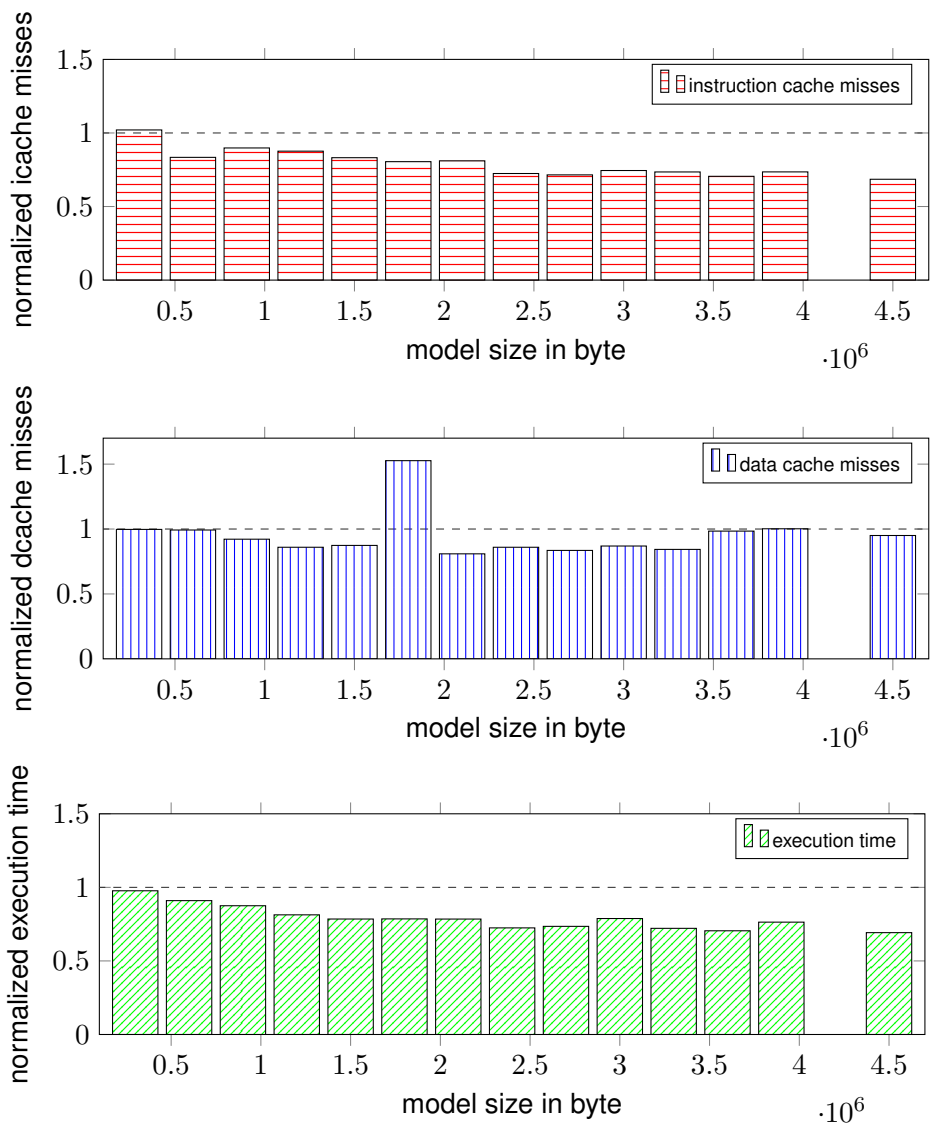


Figure 7.1: Execution time, icache misses and dcache misses for if-else tree optimization

if-else block and the left and right subtree is placed within the corresponding if or else block. 2) the generated trees with the state-of-the-art optimization [CSH+22], where the tree is reordered in regard to the branch probability within every node. This reordering aims to optimize cache prefetching and minimize the amount of cache misses. A large set of random forests is generated for data sets, which result in floating-point split values for the naive and the optimized if-else tree implementation. As datasets, parts from the UCI machine-learning repository [AN07] are chosen: the *EEG Eye State Data Set* (eye), the *Gas Sensor Array Drift Data Set* (gas), the *MAGIC Gamma Telescope Data Set* (magic), the *Sensorless Drive Diagnosis Data Set* (sensorless) and the *Wine Quality Data Set* (wine), which are all classification data sets. All datasets are divided into 75%

training data and 25% test data. Hyperparameter tuning is not performed but rather the maximal depth of the decision trees is tuned in order to derive different sized models. These models are executed on a X86 server machine (2x AMD EPYC 7742, 32kB L1 i/dcache, 256GB RAM) and are compared with respect to their execution time, their amount of misses in the level 1 instruction cache and the amount of misses on the level 1 data cache.

Figure 7.1 depicts the recorded results from the execution of the implementations with the performance analysis tools for Linux (Perf). The results of the optimized implementation are normalized (applying the optimization method from Chen et al. [CSH+22]) to the naive implementation. Two knobs are tuned: The maximal depth of single trees and the amount of trees within the ensemble. The resulting size of the model is based on the measurement of the binary size of the implementation after the compilation, which is illustrated along the x-axis. It should be noted that the binary size of the model is only indirectly controlled by the maximal depth and the amount of trees, hence not for every size on the x-axis also a model is generated. The optimized implementation is considered, even if it may result in a different binary size, to the original binary size from the naive implementation. Hence, even if optimizing the model increases the binary size, the performance still is compared to the corresponding naive implementation of the same tree structure. An increase in the binary size potentially causes a higher amount of cache misses, which is then reported in the normalized data. The models are further grouped in size groups (0kB-300kB, 300kB-600kB, ...) and the geometric mean of the normalized improvements is computed. This value is the ultimately depicted in the figure. The green bars with diagonal lines indicate the reduction in total execution time, the red bars with horizontal lines indicate the reduction in L1 icache misses, and the blue bars with vertical lines indicate the reduction in L1 dcache misses.

It can be observed that, although the optimization reduces the total execution time and amount of icache misses for large models[2], the amount of L1 dcache misses is not reduced similarly. This stems from the fact that the if-else tree optimization proposed by Chen et al. [CSH+22] only modifies the sequence of the source code in order to reduce the amount of L1 icache misses. The placement and loading of the split values is not considered and thus not handled in the optimization. When the dcache misses can be reduced as well, a further reduction of the execution time can be possible. Furthermore, loads can be released from the instruction memory, which may comfort other applications within the system.

Observing this shortcoming in the existing optimization motivates the development of a new optimization technique, which specifically focuses on the optimization of dcache misses by handling the loading of the split values in a dedicated manner. One trivial method is to round the floating-point split values to integer values and subsequently encode them in the immediate field of the instructions itself, such that they do not need to be loaded from data memory at all. This, however, potentially induces a loss in

[2] The optimization targets to optimize the memory layout, such that cache misses are reduced. Thus, effects likely only can be observed when the model size exceeds the cache capacity, which is only for larger models the case.

accuracy due to the rounding of the split values. Alternatively, this section presents an implementation, where the full floating-point split value can be encoded in the immediate field of instructions and therefore also omits the need to load the split values from data memory.

7.3.3 Immediate Encoding

As mentioned before, when it comes to the optimization of the cache behavior of if-else implementations of DTs, both cache types, i.e. the instruction cache and the data cache, need to be handled. In general, optimization methods profile the execution of the DT on the training dataset and determine empirical branch probabilities. These probabilities are used subsequently to shape the tree implementation in an optimized manner. When the total model size exceeds the capacity of a cache, which likely happens for kilobyte sized level 1 caches, cache misses cannot be avoided during execution of the tree.

Hence, the optimization target is to reduce the amount of cache misses in order to improve the total execution time of the DTs. Such optimizations usually can exploit two aspects: 1) the tree is shaped in a way that frequently accessed parts of the decision tree are less likely evicted from the cache as in a naive implementation and therefore do not cause cache misses on access, and 2) the tree is shaped in a way that automatic prefetching of (spatial) local memory contents is utilized to load parts of the tree into caches before they are accessed and thus omit cache misses at the access time itself. To shape the tree itself, data memory and instruction memory needs to be distinguished. Data memory is usually used to store variables and arrays. If a tree implementation uses large arrays, changing the layout of the array allows shaping the tree. Instruction memory is used to store the instruction sequence of the tree itself. If the tree implementation uses many instructions, changing the sequence of instructions allows shaping the tree regarding the behavior of instruction caches.

As motivated before, the naive implementation of an if-else tree in C++ uses data memory to load both feature values and split values. Access to the feature values cannot be omitted and hardly be optimized, since the input tuple is not created by the tree implementation itself. Thus, data memory accesses to the feature values are compulsory. In consequence, optimization of the data memory accesses for the split values is challenging, since these accesses are necessarily interleaved with the accesses to the feature values. Therefore, the implementation, illustrated in this section, alters the loading of the split value from data memory to instruction memory. Subsequently, the tree is shaped by ordering the instruction sequence with respect to the behavior in the instruction cache.

Based on the arch-forest framework[3], used in [CSH+22], a new code generator module is implemented for the generation of the optimized if-else tree. The code generator does not generate C or C++ code, but rather directly generates X86 or ARMv8 assembly code, which is embedded by inline assembly to the rest of the framework. In

[3] https://github.com/tudo-ls8/arch-forest

```
1 __rtitt_lab_27_0:
2 movss    12(%1),          %%xmm1
3 //0x3fc00000=1.5
4 mov      $0x3fc00000,     %%eax
5 movd     %%eax,           %%xmm2
6 comiss   %%xmm1,          %%xmm2
7 jnb      __rtitt_lab_29_0
```

Listing 7.3: Optimized assembly implementation (X86)

```
1 "__rtitt_lab_27_0:"
2 ldr      s1,              [%1, 12]
3 //0x3fc00000=1.5
4 movz     w2,              #0x0000
5 movk     w2,              #0x3fc0,     lsl 16
6 fmov     s2,              w2
7 fcmp     s1,              s2
8 b.le     __rtitt_lab_29_0
```

Listing 7.4: Optimized assembly implementation (ARMv8)

order to explain the assembly implementation, the example node from Listing 7.1 is illustrated in the following.

Listing 7.3 illustrates the output of our code generator for the example node. In line 2, similarly as in the compiler generated code, the feature value is loaded from data memory, which cannot be omitted. Afterwards, the split value (1.5) is converted to IEEE-754 32 bit representation in line 4 and loaded as a bit mask to a general purpose register[4]. The `movd` instruction subsequently copies the register content without conversion to a floating-point register and in line 6 and 7 the according comparison and jumps are executed.

[4]The generator also supports double precision floating-points; the code is generated accordingly on demand.

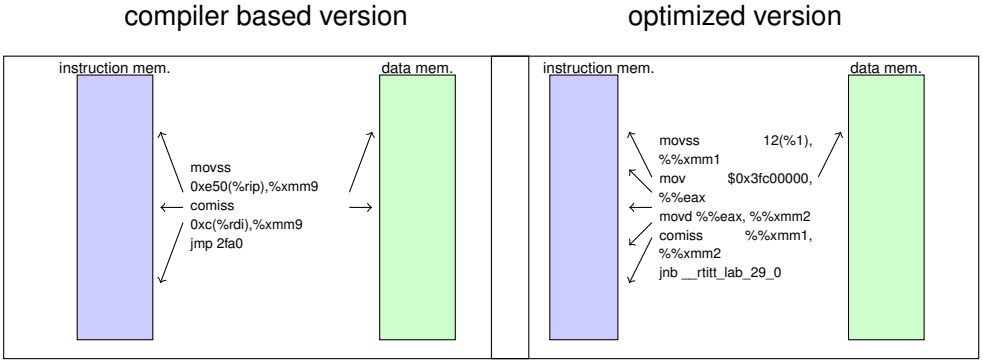


Figure 7.2: Optimized loading of constants from memory

Listing 7.4 similarly depicts an example of the ARMv8 code, which is generated by the code generator. The key difference is that ARMv8 does not offer pseudo instructions to load 32 or 64 bit immediate values, thus these are decomposed into a set of `movz` (move and zero contents before) and `movk` (move and keep contents) instructions with according bit shifts. The `fmov` instruction in ARMv8 is the respective instruction to move contents from a general purpose register to a floating-point register without conversion.

Figure 7.2 Illustrates the difference in memory accesses between the compiler generated code and the explicit generated code for X86. All instructions, by default, access instruction memory, since the instruction has to be loaded from instruction memory. In the compiler generated version, two out of three instructions in addition access data memory, in the optimized version only one out of five instructions additionally accesses data memory. Despite moving the split value entirely to the instruction memory, also the code sequence optimization is inherited from [CSH+22] in the code generator. For every node, the relative probability to visit the left or right child is compared, and the more probable child is placed as the subsequent instructions. The less probable child hence is labeled and targeted by the jump / branch instruction. Implementation wise, this requires a swap of the branch condition, since the branch must be taken either on the $\leq$ or on the $>$ condition. This is achieved by either generating a `jnb` (jump if not below) / `b.le` (branch if less or equal) or a `jb` (jump if below) / `b.gt` (branch if greater than) instruction.

In order to integrate the code generation in a generally applicable shape, all possible combinations for data types within if-else trees are implemented in the code generator. This includes various combinations of data types for the feature and for the split values, since the comparison has to be realized accordingly. The code generator allows generating if-else trees for 32 and 64 bit floating-point split values, including the optimization from [CSH+22], in assembly code and eliminates a large portion of data memory loads, at the cost of few additional instructions, which are used to encode the data directly in the immediate field. Thus, the data cache misses are likely reduced when employing this implementation.

Beyond the concrete implementation, this method is applicable to other models and structures as well. Floating-point constants are required for a large set of machine learning models, e.g. neural networks or simple regression models. Such models are usually trained by adjusting a set of constants (weights, parameters, etc.), which are then incorporated for computation during inference. Since the computation is implemented as code execution in a CPU based variant, constants can be similarly immediately encoded and possibly allow a performance improvement of other models.

7.3.4 Evaluation

In order to evaluate the implementation of encoding the split values in the immediate fields of integer instructions, this section focus on two central aspects: 1) the reduction of data cache misses and 2) the effect of the reduction on the total execution time. For the evaluation, the data sets from the UCI machine-learning repository [AN07] are again

	CPU	L1 icache	L1 dcache	Memory
X86 Server	2x AMD EPYC 7742	32 kB	32 kB	256 GB DDR4
X86 Embedded	Intel Atom x5-Z8350	32 kB	24 kB	2GB DDR3
ARMv8 Server	2x Cavium Thunder X2	32 kB	32 kB	256 GB DDR4
ARMv8 Embedded	Amlogic S9052	32 kB	32 kB	2 GB DDR3

Table 7.1: Test system details

investigated: The *EEG Eye State Data Set* (eye), the *Gas Sensor Array Drift Data Set* (gas), the *MAGIC Gamma Telescope Data Set* (magic), the *Sensorless Drive Diagnosis Data Set* (sensorless) and the *Wine Quality Data Set* (wine). These data sets are all classification data sets. The arch-forest framework is used together with the custom code generator to generate ensembles of different amount of trees and tree sizes for all data sets. Subsequently, three implementations for every tree are generated: 1) a naive if-else tree implementation without any optimization, 2) the optimized if-else tree implementation from Chen et al. [CSH+22] as the state of the art and 3) the assembly-based implementation, as presented in Section 7.3.3. As test platforms, four different systems are chosen, two server systems with X86 and ARMv8 architectures and two embedded systems with X86 and ARMv8 architectures. The system details can be found in Table 7.1. All generated ensembles are executed on all the systems and the performance analysis tools for Linux (Perf) are used to record instruction cache misses, data cache misses and the total execution time for every configuration. The model size is determined in bytes after compilation to compare the different configurations regarding their final size. The optimized implementations are considered for the binary size of the naive implementation and size groups are built, which are used to compute the geometric mean and present the results. Thus, even if the model size is increased by the optimization, the normalized ratio still is depicted for the same logic model structure.

Figure 7.3 depicts the icache misses for the server systems, Figure 7.4 depicts the dcache misses, and Figure 7.5 depicts the execution time, respectively. Figure 7.6 depicts the icache misses for the embedded systems and Figure 7.7 the dcache misses, respectively. The normalized ratio between the optimized and the naive implementation is computed again. Thus, a number larger than 1 indicates worse performance in comparison to the naive implementation. Each figure includes results for the X86 architecture and for the ARMv8 architecture. Comparing the reduction for the optimization from the state of the art and the encoding optimization leads to another, relative improvement, which is illustrated in Table 7.2. The #IMPROVED and #IMPROVED($> 900k$) values describe in how many of the tested models of the encoding optimization perform better regarding instruction cache misses, data cache misses or execution time than

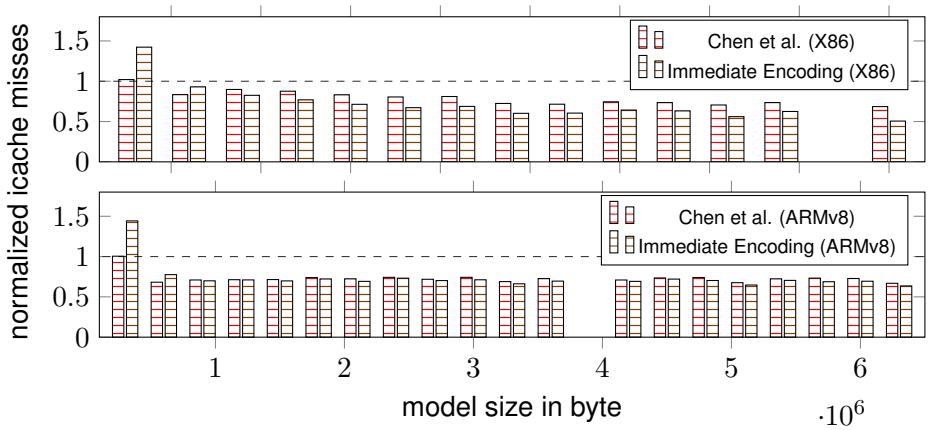


Figure 7.3: Instruction cache misses of immediate encoded split values - server

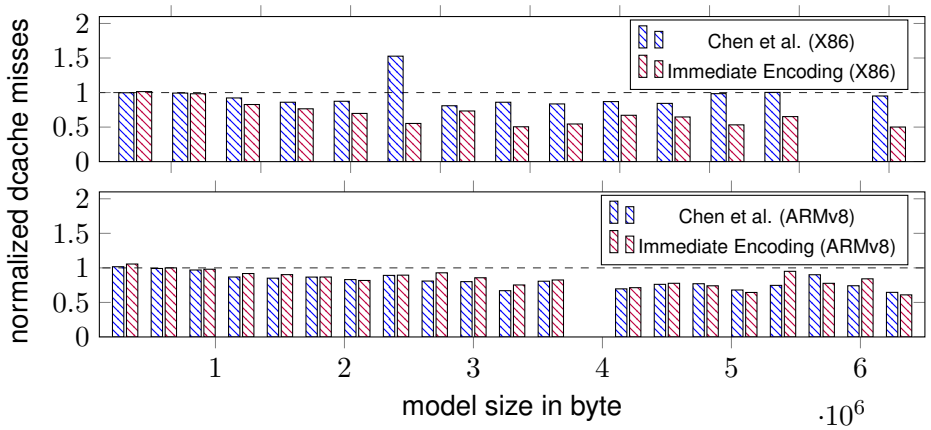


Figure 7.4: Data cache misses of immediate encoded split values - server

the optimization from Chen et al. The latter value only considers models, which lead to a binary size of more than $900kB$. The improvement ratio is further computed by $1 - \frac{\text{Immediate Split}}{\text{Chen et al.}}$. Hence, a number of $+100\%$ for the cache misses would mean that the encoding optimization eliminates all cache misses, which are left after the optimization from the state of the art. This improvement is computed for all data sets and the geometric mean for models larger than $900kB$ and the peak value is reported in the table.

Discussion

Generally, it can be observed that for rather small ensembles (up to $\approx 900kB$) a diminished performance can be observed for most configurations. If a small model anyway can be held entirely in the level 1 cache, there is no requirement for any optimization. The optimization, however, induces certain overheads by introducing more instructions, which leads to an ultimate performance decrease. In consequence, this draws the conclusion that the optimization should only be applied in meaningful scenarios, where the ensemble size exceed the level 1 cache size and necessarily

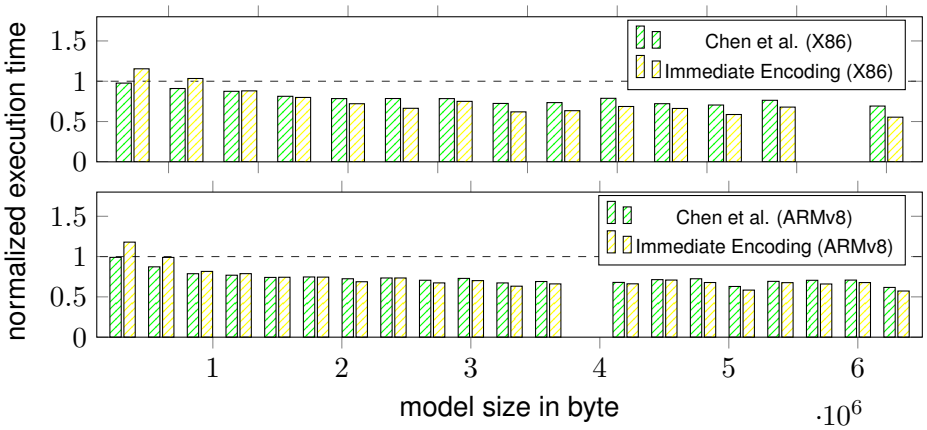


Figure 7.5: Execution time of immediate encoded split values - server

	X86		ARMv8	
	Server	Embd.	Server	Embd.
Time				
#IMPROVED	33.7%	09.6%	34.6%	06.1%
#IMPROVED(>900k)	80.9%	10.7%	68.3%	06.5%
GEOMEAN(>900k)	+08.4%	−16.4%	+01.5%	−17.8%
Peak	+39.5%	+36.7%	+29.7%	+38.7%
ICache				
#IMPROVED	60.0%	33.7%	55.5%	-
#IMPROVED(>900k)	95.2%	60.7%	76.9%	-
GEOMEAN(>900k)	+14.7%	+00.3%	+02.6%	-
Peak	+90.4%	+26.8%	+61.8%	-
DCache				
#IMPROVED	64.1%	76.8%	41.6%	37.5%
#IMPROVED(>900k)	84.5%	100.0%	39.4%	27.8%
GEOMEAN(>900k)	+26.1%	+96.1%	−4.7%	+4.6%
Peak	+92.3%	+99.8%	+65.5%	+87.7%

Table 7.2: Average and peak improvements compared to [CSH+22][5] for server and embedded systems

produces cache misses. Therefore, a focus is put on these meaningful scenarios in the following.

Focusing on the instruction cache misses only, it can be seen that for most configurations with large model sizes the amount of icache misses is further decreased by the proposed optimization, compared to the state of the art (on the X86 server system in 95% of the relevant cases in geomean by 14.7%). Considering the data cache misses, considerable reductions can be observed for larger ensembles in comparison to the

[5]The geomean values in this table are computed for models only, which are larger than 900 kB.

state of the art as well. For the X86 server, the amount of data cache misses for large ensembles is even reduced in 84% of the relevant cases by up to $\approx 92\%$ in peak. In case of the ARMv8 server, a slighter reduction of dcache misses can be observed, up to $\approx 65\%$ in peak and even an increase of $\approx 4\%$ in geomean for large ensembles. Focusing on the embedded systems, similar behavior can be observed for the data cache, the behavior for the icache misses contrarily differs[6]. Data cache misses are reduced by up to $\approx 99\%$ in peak and $\approx 96\%$ in average for X86. Instruction cache misses, however, are not significantly reduced on the X86 embedded system. For the ARMv8 embedded system, the improvement of dcache misses as well is comparably lower to the X86 embedded system.

Despite reducing icache and dcache misses, the allover execution time of the optimized implementation matters. In general, it can be observed that a high reduction in dcache misses does not necessarily result in a high reduction in execution time. For large ensemble sizes on the server machines, a consistent reduction of execution time can be however observed for the proposed optimization. The majority of relevant cases (more than 65%) yields an improvement in execution time on the X86 and ARMv8 servers. The improvement is up to $\approx 40\%$ in peak for X86 and ARMv8, compared to the state of the art. For small ensemble sizes, it can be observed that the execution time is increased beyond the naive implementation with the optimized implementation. In these cases, the additional overheads due to the immediate encoding cannot be leveraged by the improvement. Investigating the embedded systems, the execution time can only be improved for few cases ($\approx 10\%$ on the X86 and $\approx 6\%$ on the ARMv8 system). In geomean, the execution time is enlarged for the relevant cases, although the amount of dcache misses is drastically reduced for X86. This suggests that dcache misses are not the limiting factor for the execution in this scenario. Furthermore, this also implies that the CPU architecture is an important factor to the intended reduction of dacache misses with the encoding optimization.

Although the results reveal that the proposed optimization cannot improve performance unconditionally, especially for small model sizes and embedded systems, scenarios with a massive reduction of dcache misses and also a reduction of icache misses can be reported. Such a reduction can be useful to comfort parallel running applications. In several cases, the reduction of cache misses further directly relates to reduction of total execution time. When generating implementations, various versions can be profiled on the training data set, so the best implementation can be chosen. Thus, for the cases where a worse result is achieved by immediate encoding, the implementation of Chen et al. [CSH+22] can still be chosen. Similarly, for small models, where the optimized implementation induces a high overhead, the native implementation can be chosen.

[6] The ARMv8 system used does not allow tracking of icache misses with perf.

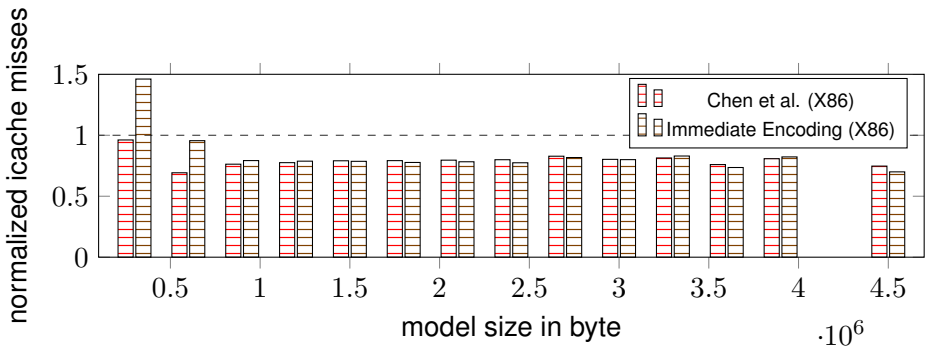


Figure 7.6: Instruction cache misses of immediate encoded split values - embedded

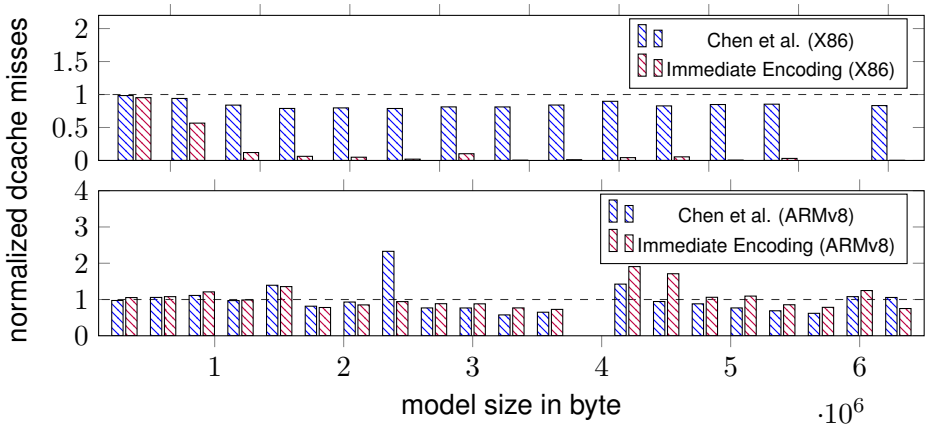


Figure 7.7: Data cache misses of immediate encoded split values - embdedded

7.3.5 Wrap-Up

Tackling the impact of RANDOM FOREST (RF) execution on the memory subsystem, especially the distribution of pressure on the instruction and data memory, this section discusses a method to move the encoding of floating-point constants in RFs inference to immediate field in the instruction memory, which is not possible by default. These methods exploit the binary encoding of floating-points in an offline manner and transfer them as integers, encoding the binary value of the floating-point, into the program code. A direct conversion at runtime then makes the floating-point value available, without loading it from data memory. Since such procedures are not supported by default, the implementation of the RF is realized by direct assembly code generation. Since only the way of loading the split value is modified, the optimization is orthogonal to existing state-of-the-art performance optimization. Experimental evaluation shows that the encoding optimization can reduce the amount of data cache misses by up to 99% upon the state-of-the-art and can even lower the allover execution time by up to 40% on server systems. It can be further concluded that the overheads, which are introduced by the encoding optimization, can only be leveraged for model sizes,

which exceed the size of the level 1 caches. Thus, the optimization should be only applied in these cases. On embedded systems, the execution time is overall not significantly lowered, although the amount of cache misses can be drastically reduced. Hence, different aspects should also be explored. The implementation of the code generation fully supports X86 and ARMv8 architectures with different width integer and floating-point data types. The source code is available at https://github.com/tu-dortmund-ls12-rt/arch-forest/tree/immediatesplittrees.

7.4 FLInt: Exploiting Floating-Point Enabled Integer Arithmetic for Efficient Random Forest Inference

As mentioned in the opening of this chapter, the specific design of the comparison operation in RFs can have a performance impact on the memory subsystem, as well as on the way of numeric computation directly. The previous section discusses the immediate encoding of split values, targeting the impact on the memory subsystem. Although the way of storing and loading the floating numbers is modified in this method, the constants are still available in a floating-point register afterward and the numeric operation is a normal floating-point comparison. This does not take a direct impact on the numeric computation.

Consequently, this section provides a rather drastic approach towards performance optimization of the comparison operation. On the one hand, the previously presented target of leveling the pressures on the instruction and data caches is maintained and even improved. On the other hand, the need to use floating-point comparisons is omitted at all, without a loss in accuracy. Hence, the floating-point unit is not further used, and comparison operations only operate in the integer unit. This method comes on the cost of being specific only for comparison operations.

7.4.1 Scope

In this section, the motivation of modifying the usage of floating-point operations in RF execution is picked up from the previous section. While the previous sections modifies the loading of floating-point values and allows arbitrary computation with the loaded value, in this section a more drastic approach is introduced. The computation of the comparison operation, i.e. computing $\leq$ between two floating-point numbers is modified in a way, that it only uses integer and logic operations. This section provides a full proof, this transformation delivers correct comparison results for all possible cases. In order to achieve this, the relation between the binary encoding of floating-point numbers and the encoding of integer numbers is deeply investigated. As the outcome of the proof, the computation can only be correctly performed with a case distinction between positive and negative numbers. Since this would be costly in terms of performance, a specific solution for RF inference is introduced, where the case distinction can be done offline in the

published work: [HCC22a] and as an extended abstract in [HCC24]

implementation phase of the ensemble. The resulting operator can be applied directly in C code without further overheads, i.e. a line like `if(pX[3]<=(float)10.074347)` becomes `if((*(((int*)(pX))+3))<=((int)(0x41213087)))`. In short, the following contributions are presented in this section:

- FLInt: A two's complement and logic operation based comparison operator for floating-point numbers, where the correctness is formally proven.
- An efficient implementation of FLInt in RFs with if-else tree implementations, where the special case handling is resolved offline during the implementation time.
- Experimental evaluation on X86 and ARMv8 server and desktop class systems to study the reduction of overall execution time when using FLInt instead of floating-points.

7.4.2 Problem Analysis and Statement

In this section, the studied problem is to compute correct floating-point arithmetic in RFs without the need for hardware floating-point support. By only using standard integer and logic operations, this can 1) enable floating-point-based RFs on devices without floating-point hardware and 2) eliminate the overheads to use the floating-point unit. Following question is answered: How floating-point comparisons can be correctly computed based on integer and logic operations? This problem is solved by specifically investigating the binary floating-point format [ECS85] and consider the binary ordering in relation to two's complement signed integer interpretation [PH17]. While it is formally proven that positive floating-point numbers are order preserving, the handling of negative numbers and a few special cases requires dedicated handling. Such a handling is integrated into a single operator, which is called FLInt.

7.4.3 Providing Correct Floating-Point Comparisons with Integer and Logic Arithemtic

Floating-point arithmetic includes several operations, which are required to process floating-point numbers. In this section, the comparison operation is only studied (i.e. $\geq$), since this is the only operation needed during random forest inference. To eliminate the use of hardware floating-point support or software float implementations, a comparison operation by only using signed integer arithmetic and logic operations is realized. In this section, the binary layout of floating-point numbers and two's complement numbers is presented, and the comparison operator between them is constructed. Every step is formally proven and the correctness of the final operator is concluded.

Binary Floating-Point and Signed Integer Format

In order to illustrate the relation between the binary representation of floating-point numbers and signed integer numbers, the state-of-the art formats is laid out in the

following. Almost all computer systems nowadays use two's complement [PH17] for signed integer numbers and IEEE 754-1985 [ECS85] for single or double precision floating-point numbers. Both formats support, among others, 32 bit and 64 bit types. For the rest of this section, the floating-point and two's complement format is defined independent of the total bit length. 32 and 64-bit numbers in two's complement and in IEEE 754-1985 then can be interpreted as an instance of the defined format. Later in this section, real implementation on common hardware is discussed, where 32 and 64 bit two's complement and IEEE 754-1985 numbers are used.

Both formats, two's complement and floating-point, define an interpretation of a fixed length bit vector. Thus, an arbitrary bit vector can be either interpreted, among other options, as a signed integer, an unsigned integer or a floating-point. Furthermore, the binary representation of a floating-point number can be interpreted as a signed integer and vice versa.

Definition 7. Let $B \in \{0,1\}^k$ be a k bit wide vector, then these bits can be interpreted either as a floating-point or as signed integer number in two's complement. $FP : \{0,1\}^k \to \mathbb{Q}$ denotes the floating-point interpretation $FP(B)$ of B, $SI : \{0,1\}^k \to \mathbb{Z}$ denotes the signed integer (two's complement) interpretation $SI(B)$ of B and $UI : \{0,1\}^k \to \mathbb{N}$ denotes the unsigned integer interpretation $UI(B)$ of B.

For the interpretation of signed and unsigned integers, every bit within the bit vector is assigned a fixed value (2^i). If the bit is set to 1, the value of the position is counted, otherwise it is ignored. Negative numbers for the signed two's complement format always have the most significant bit (MSB) set to 1. The signed value is then interpreted by assigning a negative value to the MSB and interpreting the other bits similar to the unsigned format.

Definition 8. The two's complement interpretation (signed integer) of a bit vector $B = (b_{k-1}, ..., b_0)$ is defined as

$$SI(B) = \begin{cases} \sum_{i=0}^{k-1} b_i \cdot 2^i & b_{k-1} = 0 \\ -2^{k-1} + \sum_{i=0}^{k-2} b_i \cdot 2^i & b_{k-1} = 1 \end{cases} \tag{7.1}$$

where the unsigned integer interpretation is the same as the two's complement for positive numbers:

$$UI(B) = \sum_{i=0}^{k-1} b_i \cdot 2^i \tag{7.2}$$

One key advantage of this format is that the binary ordering (i.e. the interpretation as an unsigned integer) is the same for negative and positive numbers. In addition, the border between positive and negative numbers allows for unchanged arithmetic, when the overflow bit is ignored. In detail, the representation of -1 in two's complement is $(1,1,1,...)$ and the representation of 0 is $(0,0,0,...)$. When adding $+1$ to the representation of -1 in unsigned arithmetic, all bits switch to 0 and an overflowing 1 goes

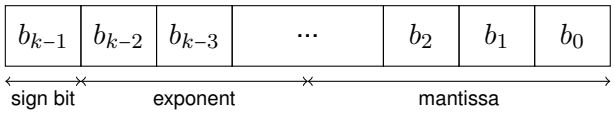


Figure 7.8: Illustration of the binary floating-point representation

to position k. Since the overflow can be safely ignored (which is intended in this case [PH17]), the computation is correct.

The floating-point format differs from the binary representation of integers. In this format, the binary representation is interpreted as three components: 1) a sign bit at the position of the most significant bit $(k-1)$, 2) a biased exponent of j bit length where the bias is $2^{j-1}-1$, and 3) a mantissa, filling the remaining bits, which is interpreted with an implicit 1. Figure 7.8 illustrates the layout of these three components within the bit vector. The interpretation of the floating-point format differs in large parts from the interpretation of the two's complement. While the sign bit is interpreted as a factor ($\times -1$ or $\times 1$), the mantissa is interpreted as a decimal number between 1 and 2, which is scaled by the exponent.

Definition 9. The floating-point interpretation of a bit vector $B = (b_{k-1}, ..., b_0) = (s, e_{j-1}, ..., e_0, m_{x-1}, ..., m_0)$ for j bit exponent and x bit mantissa is defined as

$$FP(B) = (-1)^s \cdot 2^{UI(e_{j-1},...,e_0)-bias} \cdot (1 + \sum_{i=0}^{x-1} m_i \cdot 2^{-x+i}) \tag{7.3}$$

where the bias for the interpretation of the exponent is $bias = 2^{j-1}-1$. It should be noted that the commonly used IEEE 754-1985 format is exactly an instance of this format for $j = 8, x = 23$ (single precision) and $j = 11, x = 52$ (double precision) [ECS85].

In addition to the normal interpretation of numbers (Definition 9), the floating-point format includes a few exceptional cases for special numbers. The special encoding for positive and negative infinity and the encoding for *not a number* is not further discussed in this section, since the usage of these numbers does not occur in random forests. If positive or negative infinity should occur anyway, they are encoded as the smallest and largest representable number and thus make no difference for comparison.

Since the normal interpretation cannot encode a 0 (due to the implicit 1 added to the mantissa), the special encoding for the representation of 0.0 is all bits set to 0. In addition, the format allows also the encoding of -0.0, when the sign bit is set to 1 and all other bits are set to 0. In this section, it is assumed that $-0.0 < 0.0$, which differs from the definition $-0.0 = 0.0$ of the IEEE 754-1985 standard[7]. As the mantissa is always interpreted as a number between 1 and 2, the smallest representable absolute value would be limited to 2^{-bias}. To extend this, the floating-point format includes a *denormalized* format, which is indicated by an exponent of all 0s. In this format, the

[7] -0 can be the result of rounding a not representable negative number. Extending the presented method to handle -0.0 equals to 0.0 is quite straightforward by including one additional scenario during code generation.

exponent is interpreted as $-bias + 1$ and the mantissa is interpreted without implicit 1 (i.e. as a number between 0 and 1). This essentially makes the representation of 0.0 also a valid denormalized number.

Ordering Between Floating-Points and Signed Integers

Now it is shown that the floating-point format (when the bit vector is interpreted as two's complement) preserves the order of numbers for positive numbers and inverses the order for negative numbers. This is also illustrated in Figure 7.9, where the signed integer values (respectively, corresponding floating-point values) of all combination of 32 bit vectors B are plotted on the x-axis (respectively, y-axis).

As the intention of this method is to evaluate the $\geq$ relation of floating-point numbers in two's complement arithmetic, the equality of numbers has to be considered first.

Lemma 10. Given two arbitrary bit vectors $X, Y \in \{0,1\}^k$, then the floating-point interpretation is the same for both numbers, if and only if also the signed integer representation in two's complement and the bit vector itself is the same.

$$FP(X) = FP(Y) \Leftrightarrow X = Y \Leftrightarrow SI(X) = SI(Y) \tag{7.4}$$

Proof. Both formats, floating-point and two's complement, are bijective for the mapping of the bit vector to a number[8]. The counted weight for the single bits is a power of 2 in floating-point and in two's complement. Hence, the weight of one bit cannot be constructed as a sum of other bits. Furthermore, numbers with a positive sign bit are always positive in both formats, numbers with a negative sign bit are always negative in both formats. Therefore, the bit vector of X and Y must be the same in both formats. □

As already explained in the beginning of this section, the interpretation of signed integer numbers uses the same binary ordering as the interpretation of unsigned integer numbers for both, positive and negative numbers. The floating-point interpretation, in contrast, uses the same encoding of the exponent and mantissa for both, positive and negative numbers, and only distinguishes them by the sign bit. Therefore, the sign bit of the floating-point format can be ignored in order to obtain the absolute value of a floating-point number:

Definition 10. Given a bit vector $B = (b_{k-1}, ..., b_0) = (s, e_{j-1}, ..., e_0, m_{x-1}, ..., m_0)$, the absolute value of the floating-point interpretation for j bit exponent and x bit mantissa is defined as

$$|FP(B)| = 2^{UI(e_{j-1},...,e_0)-bias} \cdot (1 + \sum_{i=0}^{x-1} m_i \cdot 2^{-x+i}) \tag{7.5}$$

Since the two's complement interpretation is order preserving for negative and positive numbers, the absolute value of the floating-point interpretation follows the same order:

[8] The definition of the floating-point format implies that $-0 \neq +0$, which ensures bijectivity. To accommodate for the definition of $-0 = +0$ (as in IEEE 754-1985), this case would need to be excluded here and added as a case distinction.

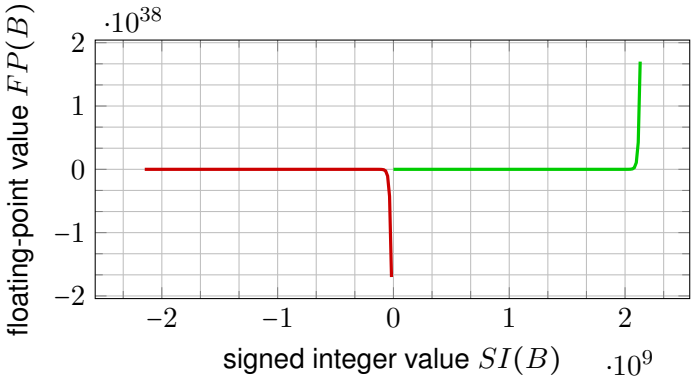


Figure 7.9: Illustration of signed integer (x axis) and floating-point (y axis) space for all combination of 32 bit vectors B

Lemma 11. Given two arbitrary bit vectors $X, Y \in \{0,1\}^k$ with the same sign bit $x_{k-1} = y_{k-1}$, the interpretation as the absolute value of the floating-point and as signed integer numbers is strictly monotonically increasing:

$$|FP(X)| > |FP(Y)| \Leftrightarrow SI(X) > SI(Y) \tag{7.6}$$

Proof. Independent of the interpretation as two's complement or as floating-point, the bit vectors can be divided into the three sections of sign bit, exponent and mantissa (Definition 9). Since the sign bits are the same, the numbers have to differ in either the exponent and the mantissa bits according to Lemma 10. Consequently, two cases have to be distinguished:

- **Case 1**: Both numbers have the same exponent: $\Rightarrow$: Then both numbers are either in normal or in denormalized floating-point format, thus the mantissa is either interpreted with implicit one or without for both numbers in floating-point interpretation. Thus, the term $\sum_{i=0}^{x-1} m_i \cdot 2^{-x+i}$ from Definition 10 must evaluate to a larger number for Y as for X. In two's complement the bits of the mantissa are evaluated the same, just with another weight. From Definition 8, the mantissa bits contribute with the term $\sum_{i=0}^{x-1} m_i \cdot 2^{i} = 2^{x} \cdot \sum_{i=0}^{x-1} m_i \cdot 2^{i-x}$ to the total sum, thus with the same value as to the floating-point format, weighted with a constant factor. Hence, this part of the sum also has to evaluate to a larger number in two's complement. Since the exponents are the same and the sign bit is the same, the remaining part of the sum in Equation ((7.1)) is the same for X and Y. Consequently, X evaluates to a larger number in two's complement as Y. $\Leftarrow$: For X to be larger than Y in two's complement while the exponent bits and the sign bit are the same, the part of the sum of the mantissa bits must evaluate to a larger value for X, which also increases the interpretation of the mantissa in floating-point interpretation, since the bits contribute with another constant weight, as discussed before.

- **Case 2**: The two numbers have different exponent bits: $\Rightarrow$: X must have a larger exponent than Y since the interpretation of the mantissa m ranges between $1 \leq m < 2$. If the exponent of X would be smaller than the exponent of Y, the factor, the exponent contributes to Equation ((7.5)) would be at least smaller by a factor of 2. This could only be compensated by the mantissa, if Y would be in denormalized format, which leads to a contradiction, since the denormalized format is encoded by the smallest possible exponent, hence X cannot have a smaller exponent. Consequently, for the exponent of X to be larger, $UI(e_{j-1}, ..., e_0)$ must evaluate to a larger number for X. Thus, the same bits must evaluate to a larger number also in Equation ((7.1)). According to Equation ((7.1)), the total contribution of the mantissa bits is smaller than any contribution of an exponent bit (as the weight for every higher significant bit is larger than the weight of all lower significant bits summed). Thus, X must also evaluate to a larger number than Y in two's complement. $\Leftarrow$: If the exponent bits are different, the part of the sum for the exponent bits in Equation ((7.1)) must be larger, since the mantissa bits cannot compensate a smaller sum due to their lower total weight. Then, the interpreted exponent in floating-point must be larger as well, which cannot be compensated by the mantissa, as explained before.

Since the representation of 0 is covered by the denormalized format in floating-point, all cases are considered. □

Next, the cases for comparing two positive and two negative numbers have to be distinguished.

Lemma 12. Given two arbitrary bit vectors $X, Y \in \{0,1\}^k$ with **positive** sign bit $x_{k-1} = y_{k-1} = 0$, the interpretation as floating-point and as signed integer numbers is strictly monotonically increasing:

$$FP(X) > FP(Y) \Leftrightarrow SI(X) > SI(Y) \tag{7.7}$$

Proof. When the sign bit of both numbers is 0, the term $(-1)^s$ in Definition 9 evaluates to 1 and has no contribution. Then, the interpretation of the floating-point number is exactly the same as in Definition 10. Thus Lemma 11 holds. □

Lemma 13. Given two arbitrary bit vectors $X, Y \in \{0,1\}^k$ with **negative** sign bit $x_{k-1} = y_{k-1} = 1$, the interpretation as floating-point and signed integer is monotonically decreasing:

$$FP(X) \geq FP(Y) \Leftrightarrow SI(X) \leq SI(Y) \tag{7.8}$$

Proof. When the sign bit of both numbers is 1, the term $(-1)^s$ in Definition 9 evaluates to -1. Then, the interpretation of the floating-point number is exactly the same as in Definition 10 with a constant factor of -1. Therefore, $-1 \cdot FP(X) = |FP(X)|$ and $-1 \cdot FP(Y) = |FP(Y)|$ holds. Starting from Lemma 11, the following can be derived:

$$-1 \cdot FP(X) > -1 \cdot FP(Y) \Leftrightarrow SI(X) > SI(Y) \tag{7.9}$$

which can be transformed into

$$FP(X) < FP(Y) \Leftrightarrow SI(X) > SI(Y) \tag{7.10}$$

and further into

$$FP(X) \geq FP(Y) \Leftrightarrow SI(X) \leq SI(Y) \tag{7.11}$$

□

A comparison between two floating-point numbers can, in addition to the considered cases, also operate on the mixed case of a positive and negative number. Since the sign bit in floating-points at the position of the most significant bit also serves in two's complement as a sign bit, these cases are covered as well.

Lemma 14. Given two arbitrary bit vectors $X, Y \in \{0,1\}^k$ with **different** sign bits $x_{k-1} \neq y_{k-1}$, the interpretation as floating-point and as signed integer numbers is strictly monotonically increasing:

$$FP(X) > FP(Y) \Leftrightarrow SI(X) > SI(Y) \tag{7.12}$$

Proof. Negative numbers in floating-point are indicated by the sign bit set to 1, which also indicates a negative number in two's complement. Positive numbers in floating-point are indicated by the sign bit set to 0, which also indicates a positive number in two's complement. Hence, numbers are interpreted as negative and positive similarly in floating-point and in two's complement. If one number is positive and the other is negative, the interpreted absolute value is irrelevant. □

Next, Lemma 10 can be used to extend Lemma 13:

Lemma 15. Given two arbitrary bit vectors $X, Y \in \{0,1\}^k$ with **negative** sign bit $x_{k-1} = y_{k-1} = 1$, the interpretation as floating-point and as signed integer numbers is strictly monotonically decreasing:

$$FP(X) > FP(Y) \Leftrightarrow SI(X) < SI(Y) \tag{7.13}$$

Proof. From Lemma 10 it is known that the interpretation as two's complement can only be the same if and only if the interpretation in floating-point is the same. Thus, $FP(X) = FP(Y) \Leftarrow SI(X) \neq SI(Y)$ or $FP(X) \neq FP(Y) \Rightarrow SI(X) = SI(Y)$ cannot happen. Thus, the equality cases can be excluded from Lemma 13. □

Design of the FLInt Operator

Leveraging the previous lemmata, an evaluation of the $\geq$ relation for floating-point numbers can be constructed, which only evaluates the $\geq$ and the $<$ (which is the logic negation of $\geq$) relation of two's complement signed integer numbers.

Corollary 3. Given two arbitrary bit vectors $X, Y \in \{0,1\}^k$, the $\geq$ relation can be computed between the floating-point interpretation of these bit vectors, using only two's complement signed integer arithmetic when distinguishing two cases:

$$\begin{aligned} & FP(X) \geq FP(Y) \\ \Leftrightarrow & \\ & \begin{cases} SI(X) < SI(Y) & \text{if } FP(X) < 0 \wedge FP(Y) < 0 \\ & \wedge FP(X) \neq FP(Y) \\ SI(X) \geq SI(Y) & \text{otherwise} \end{cases} \end{aligned} \tag{7.14}$$

Proof. The first case (both numbers negative) is discussed in Lemma 15. It should be noted that this only covers the case that both numbers are negative, but not equal. For the case that the numbers are equal, either positive or negative, Lemma 10 shows that the second case holds. Also, for the case that both numbers are positive, but not equal, Lemma 12 shows that the second case holds. For the case that only one number is positive, Lemma 14 shows that also the second case holds. Since the second case consists of the latter three cases, all cases are covered. It should be also noted that the condition, whether the first or second case is needed, also can be evaluated on the signed integer representation, according to Lemma 14, Lemma 12 and Lemma 10. The evaluation of the first and the second number is negative could also be done independent of the format interpretation by only extracting the sign bits x_{k-1} and y_{k-1}. □

Theorem 3. *Given two arbitrary bit vectors $X, Y \in \{0,1\}^k$, the $\geq$ relation can be computed between the floating-point interpretation of these bit vectors, using only two's complement signed integer arithmetic, with the following operation:*

$$\begin{aligned} & FP(X) \geq FP(Y) \\ \Leftrightarrow & \\ & (SI(X) \geq SI(Y)) \oplus \\ & ((SI(X) < 0 \wedge SI(Y) < 0 \wedge SI(X) \neq SI(Y)) \end{aligned} \tag{7.15}$$

Here, the XOR function $\oplus$ is used to achieve negation in case the second input is $true$. Let $u = (SI(X) \geq SI(Y))$ and $v = ((SI(X) < 0 \wedge SI(Y) < 0 \wedge SI(X) \neq SI(Y))$. Applying XOR to the value u while the second input v is false, evaluates to the identity function ($u \oplus false = u$), applying XOR to the value u while the second input v is true, evaluates to the negation ($u \oplus true = \neg u$).

Proof. Since $\neg(SI(X) \geq SI(Y))$ is $SI(X) < SI(Y)$, it is known from Corollary 3 that only u needs to be computed and negated the result when the first case applies. Hence, the condition is evaluated for the first or second case v based on signed integer arithmetic, which delivers true when the condition holds and false when the condition does not hold. In order to achieve negation in case the condition holds, the exclusive or (XOR) function $\oplus$ is applied here. □

Towards efficient computation: Previously, a method to perform floating-point comparisons by only using signed integer arithmetic and logic operations is presented. In many CPU instructions sets (including X86 and ARMv8), there is no dedicated operation to compute the $<$ or $\neq$ relation. Instead, a comparison instruction needs to be called and a subsequent conditional set or even a conditional branch is required. Hence, the method from Theorem 3 would require in total four comparisons and conditional set or branch instructions. Depending on the CPU architecture, it may be more efficient to check only if $SI(X) < 0$ and exchange and invert X and Y:

Theorem 4. *Given two arbitrary bit vectors $X, Y \in \{0,1\}^k$ where the positiveness of $FP(X)$ (equivalently $SI(X)$) is known a priori, the $\geq$ relation can be computed between the floating-point interpretation of these bit vectors, using only two's complement signed integer arithmetic:*

$$\begin{aligned} & FP(X) \geq FP(Y) \\ \Leftrightarrow & \\ & \begin{cases} -1 \cdot SI(Y) \geq -1 \cdot SI(X) & \text{if } SI(X) < 0 \\ SI(X) \geq SI(Y) & \text{otherwise} \end{cases} \end{aligned} \tag{7.16}$$

Proof. Following Corollary 3, the second case is needed when X is positive. If X is negative, the comparison $SI(X) \geq SI(Y)$ can be directly transformed to $-1 \cdot SI(X) \leq -1 \cdot SI(Y) \Leftrightarrow -1 \cdot SI(Y) \geq -1 \cdot SI(X)$, which is then a comparison with at least one positive operand, thus the second case from Corollary 3 applies again. □

It should be noted that in Theorem 4 always one operand is ensured to be positive for the comparison. Hence, the equivalence $FP(X) \geq FP(Y) \Leftrightarrow SI(X) \geq SI(Y)$ or $FP(X) \geq FP(Y) \Leftrightarrow -1 \cdot SI(Y) \geq -1 \cdot SI(X)$ holds. This also implies that all other relations ($\leq$, $>$, $<$) hold in the same manner. Especially for integrating FLInt into program code, this allows the usage of arbitrary comparison constructs.

7.4.4 Evaluation

Previously, a method how to eliminate floating-point operations entirely from random forest inference is discussed. This method does not change the result of the model at all. Although there may be unavoidable motivations to eliminate the use of floating-points from a system (e.g. no presence of a hardware floating-point unit or high energy consumption of the floating-point unit), a more general motivation is studied in the following: the reduction of *execution time*. To comprehensively study the impact on the allover performance in terms of execution time for random forests of using FLInt, experiments are conducted on multiple data sets, machine classes and CPU architectures in the following.

Evaluation Setup

Scikit-learn is used to train multiple random forest configurations on a subset of data sets from the UCI machine-learning repository [AN07]: The *EEG Eye State Data Set* (eye), the *Gas Sensor Array Drift Data Set* (gas), the *MAGIC Gamma Telescope Data Set* (magic), the *Sensorless Drive Diagnosis Data Set* (sensorless) and the *Wine Quality Data Set* (wine). All these data sets contain floating-point values, thus scikit-learn inherently creates floating-point split values for the trained random forests and decision trees.

For every data set, random forests with $\{1, 5, 10, 15, 20, 30, 50, 80, 100\}$ trees are trained. For every random forest size, the maximal depth of all trees is limited to $\{1, 5, 10, 15, 20, 30, 50\}$ layers. It should be noted that this is only a maximal depth, the training may thus lead to smaller trees, which was not under control. Furthermore, no tuning of hyperparameters is performed, instead scikit-learn is employed in the standard configuration, since the optimal creation of random forests is out of scope. Consequently, data sets are split into 75% training data and 25% test data and the execution time of the random forests is measured only on the formerly unseen test data.

To evaluate the impact of the omission of the use of floating-point units on the execution time, the random forests are executed on X86 and ARMv8 systems. The target platforms differ from Table 7.1. For each architecture, a server class (S) (AMD EPYC 7742 for X86 and Cavium ThunderX2 99xx for ARM) and a desktop class (D) (Intel Core i7-10700 for X86 and Apple Silicon M1 for ARM) system is considered. All systems run Linux without any underlying hypervisor or simulation system.

To compare the achievement in terms of execution time reduction, multiple implementations are considered for every random forest, including the state-of-the-art [CSH+22]:

1. A standard if-else tree, where tree nodes are straightforward translated into nested if-else blocks and normal floating-point numbers are used

2. A cache-aware if-else tree implementation[CSH+22], called CAGS (cache-aware grouping and swapping), where if-else blocks are cache-aware repositioned

3. The C implementation of the standard if-else tree with FLInt

4. An Implementation of CAGS with FLInt integrated

For the latter three implementations, the normalized execution time to the standard implementation is computed, by which a fraction of the execution time of the naive version is derived, which indicates the gained improvement. All configurations are further grouped with the same maximal tree depth together and are presented by their mean normalized execution time and the corresponding variance across all data-sets and number of trees within the ensemble.

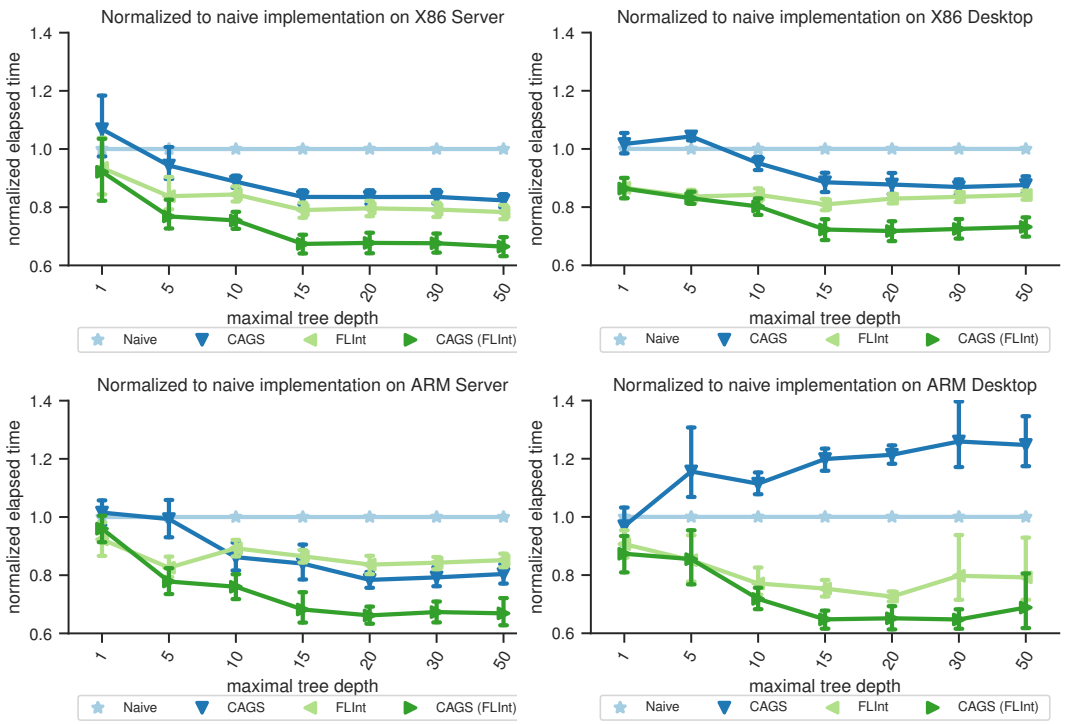


Figure 7.10: Normalized execution time for increasing maximal tree depth

Evaluation Results

The average (geometric mean) normalized execution time across all data sets and ensemble sizes is illustrated for a specific maximal depth of the single trees in Figure 7.10 for all considered systems. The naive standard implementation is illustrated in light blue with star tick marks as the baseline. The cache-aware implementation (CAGS) for if-else trees from Chen et al. [CSH+22] is shown in dark blue with down arrow tick marks. The FLInt results for standard trees are then depicted by a light green line with left arrow tick marks. The CAGS implementation with integrated FLInts is depicted by a dark green line and right arrow tick marks. While the x-axis depicts a growing maximal depth of trees, the y-axis presents the fraction of execution time from the naive baseline version. A value of e.g. 0.75 thus indicates 25% improvement in execution time. A value larger than 1 consequently describes an increase of the execution time. Each point in the plots is also associated with the computed variance across all data sets and ensemble sizes.

In addition to the graphical illustration, the average (geometric mean) normalized execution time is provided in Table 7.3. The average over two sets is computed: 1) all tree configurations for all benchmarks for one implementation and 2) all tree configurations where the maximal depth is limited to more than 20 for all benchmarks for one implementation.

Results in general: From the presented results, several observations can be made. First, it can be observed for almost all systems and configurations that the gained

Table 7.3: Average (geometric mean) normalized execution time: ($D \geq 20$): Average of ensembles with a maximal tree depth of more than 20

	X86 S	X86 D	ARMv8 S	ARMv8 D
CAGS	0.88×	0.92×	0.85×	1.14×
CAGS ($D \geq 20$)	0.83×	0.87×	0.79×	1.22×
FLInt	0.81×	0.83×	0.85×	0.77×
FLInt ($D \geq 20$)	0.79×	0.83×	0.84×	0.74×
CAGS (FLInt)	0.71×	0.76×	0.72×	0.70×
CAGS (FLInt) ($D \geq 20$)	0.66×	0.72×	0.66×	0.64×

execution time improvement varies much for small trees and reaches a more constant value for deeper trees. For small trees, a normalized short time is spent for every feature vector for executing the tree, which imposes a higher contribution of overheads (e.g. creating of data structures and function calls). For higher maximal depths of the trees, single trees do not reach the maximal depth at a certain point (when the data set requires no further splitting to gain accuracy), hence trees can have a similar shape for high maximal depths. Second, it can be observed that the FLInt implementation improves the execution time for almost all evaluated cases for the standard tree, as well as for the CAGS implementation.

Impact of compilation: Additional experiments with disabled compiler optimization, which are only presented here partially (Figure 7.11), show that the assembly implementation causes a similar execution time for small trees as the naive version without compiler optimization and does not degrade the performance much. It further shows that the assembly implementation achieves a similar performance improvement for deep trees over the naive version without compiler optimization, compared to the naive version with compiler optimization. This suggests the conclusion, that compiler optimization has more benfits for smaller trees. Compiler optimization cannot be performed on the directly generated assembly code, while deeper trees are less affected by compiler optimization. Considering the experiments with compiler optimization again, it can be further observed that the assembly implementation of FLInt achieves a better reduction in terms of execution time for large trees than the C implementation on all systems except the ARM Desktop system. This supports our design principle that explicit control over the value loading and interpretation can be gained by directly generating assembly code.

Integration into CAGS: In order to assess the range of improvement in terms of execution time with other state-of-the-art optimization approaches for decision trees and evaluate how FLInt can work together with such optimizations, FLInt is compared to CAGS from Chen et al. [CSH+22]. For all systems, except the ARM server system, FLInt on its own achieves a similar or larger improvement as CAGS does. For smaller trees, the improvement is even consequently larger. For all evaluated cases for large trees and machines, FLInt achieves a higher reduction in terms of execution time by

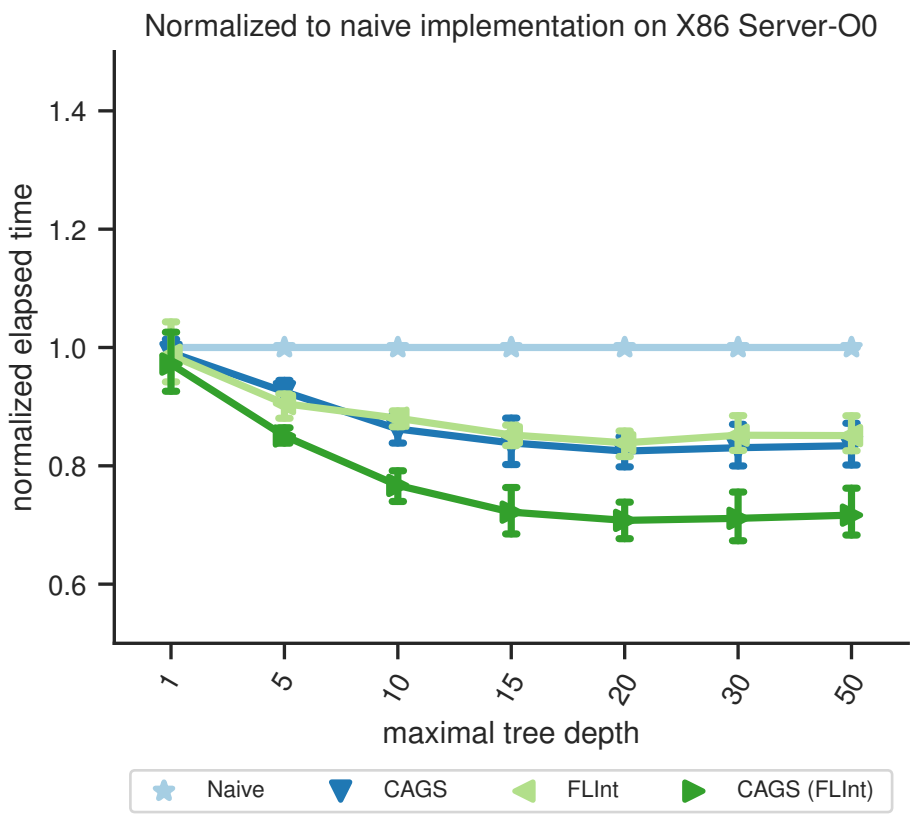


Figure 7.11: Execution time improvement with disabled compiler optimization

at least $\approx 10\%$ more than CAGS. For smaller trees, the assembly implementation performs worse, which seemingly is caused by the missing compiler optimization. The C implementation, however, consequently achieves a better execution time improvement. For the ARMv8 based systems, two interesting observations can be made: 1) the C-based implementation performs worse on the ARMv8 Server than CAGS, although the assembly implementation performs better. 2) CAGS increases the execution time in almost all cases for the ARMv8 Desktop system, while FLInt achieves consistent improvement compared to the other systems.

Basically, FLInts can be also integrated into CAGS directly. The approach however, explicitly considers instruction and data caches for the implementation. Since floating-point constants are usually loaded from data memory, but encoded as immediate values in FLInt and thus are loaded from instruction memory, the optimization algorithm has to be redesigned to properly work together with FLInts. Ignoring this for a moment and investigating the results of the straightforward integration of FLInt into CAGS, it can be seen that the performance is improved significantly in almost all cases. Furthermore, the improvement seems to be almost constant over different sized trees for almost all systems. This suggests the conclusion that FLInt is an orthogonal optimization to

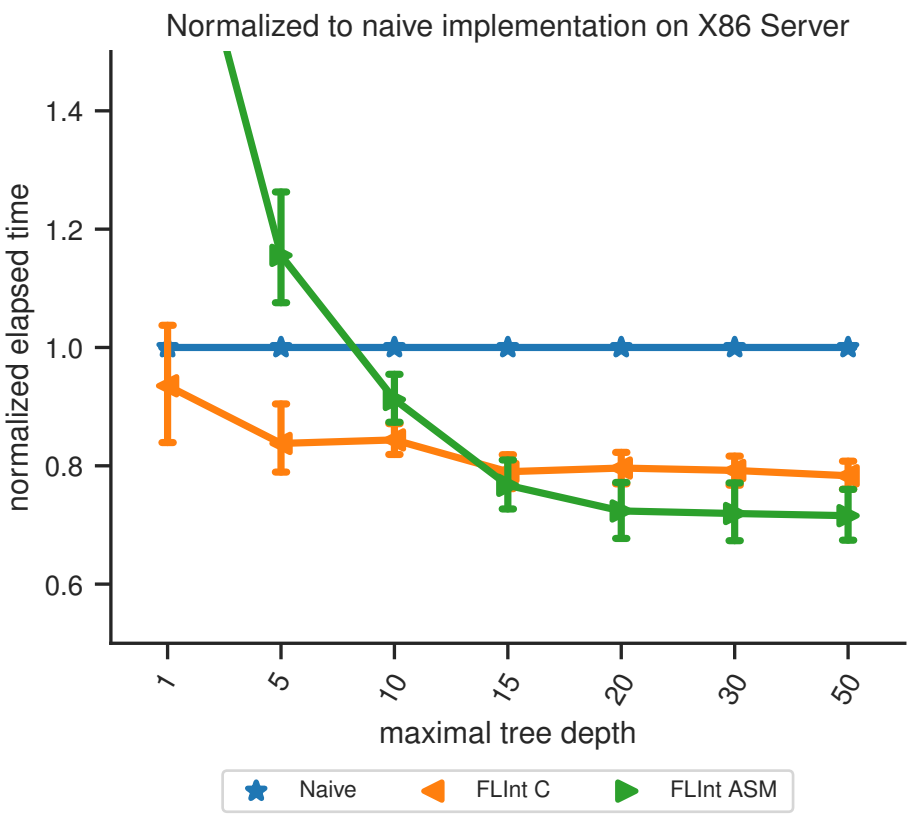


Figure 7.12: Normalized execution time for the assembly and C implementation

CAGS and optimizes another performance bottleneck, working well together with CAGS. This furthermore suggests that even further performance improvement could be reached by integrating the assembly based implementation of FLInt into CAGS.

Direct Assembly Implementation: The direct assembly implementation of FLInt is motivated by eliminating language related overheads for the reinterpretation of floating-point values. Therefore, the direct assembly based implementation of FLInt is also compared with the C-based implementation of standard trees. It should be noted that the assembly based implementation could also be combined with CAGS but requires rewriting of the entire CAGS algorithm. Since this imposes methodological changes in the algorithm itself, which open another design space, it is out scope. Figure 7.12 highlights the normalized execution time for the direct assembly implementation (orange line with left arrow ticks) in relation to the C-based implementation (green line with right arrow ticks) for the X86 server system. In the measured execution time, it can be observed that although the assembly version performs worse for small tree sizes due to the missing compiler optimization, it can outperform the C-based implementation for larger trees. This is also consistent with the other tested systems (Table 7.4). This

Table 7.4: Average normalized execution time for the assembly implementation:

	X86 S	X86 D	ARMv8 S	ARMv8 D
FLInt C	0.81×	0.83×	0.85×	0.77×
FLInt C ($D \geq 20$)	0.79×	0.83×	0.84×	0.74×
FLInt ASM	0.89×	0.95×	0.83×	0.89×
FLInt ASM ($D \geq 20$)	0.70×	0.75×	0.69×	0.72×

suggests the conclusion that the assembly implementation could gain even higher performance improvements, when integrated for deep trees.

Overall, it can be observed that the integration of FLInts into random forests can reduce the execution time in comparison to a naive implementation by up to $\approx 30\%$. Integrating FLInts into other existing optimization methods, even reduces the execution time by up to $\approx 35\%$. All these modifications do not impact the model output, nor the accuracy. While it makes sense to utilize the C-based implementation for small trees, the assembly based implementation can achieve higher performance gains for deeper trees due to the explicit control over the value loading and interpretation. At the implementation time, a simple threshold (e.g. $D \geq 20$) on the maximal depth can decide weather the C-based or the assembly-based implementation should be used.

7.4.5 Wrap-Up

This section picks up the previously discussed problem of missing support for immediate encoding of floating-point constants (Section 7.3) and provides a more specialized solution to the problem. The need of explicit floating-point computations is entirely eliminated by providing a computationally correct evaluation of the comparison operation, which only bases on integer arithmetic with a few logic extensions. This leads to the design of the FLInt operator, where the computational correctness is formally proven. Despite formal correctness, considerations about implementation effort and implementation overheads are presented, and a version is proposed, which requires an evaluation of numbers during implementation time and hence a reduced overhead at runtime. Experimental evaluation of the various proposed versions is conducted on X86 and ARMv8 test systems, which shows a possible performance improvement in terms of execution time by up to $\approx 30\%$ on its own. It should be clearly noted at this point, that performance improvement is not the only positive outcome, but the ability to deploy floating-point based RFs to hardware without floating-point support is provided as well. Another insight from the evaluation is that the discussed method works largely orthogonal to other optimization strategies and can gain a total improvement of up to $\approx 35\%$. Although the implementation in this section is specific and can only be applied to comparison operations of floating-point number and, in addition, can be only be efficiently realized when one number is constantly known to be negative or positive upfront, the integration can be achieved much more simple, compared to Section 7.3. The changes can be realized by a one by one replacement of the C code. The entire implementation is also published: `https://github.com/tu-dortmund-ls12-rt/arch-forest/tree/flintcomparison`.

7.5 Concluding CPU Optimization for Random Forests

This chapter is motivated by enabling the option for immediate floating-point operations, allowing low level optimization of RFs. Traditional ISAs do not provide such an option and require handing of floating pint values from memory. The possible impact on the memory subsystem can be significant. Towards this, this chapter discusses two approaches, one to reinterpret immediately loaded integer bits to a floating-point number and process it accordingly, another to perform floating-point arithmetic directly on the reinterpreted integer representation for the comparison operation in RFs. Both of these methods are purely software based exploitation of existing ISA means, allowing the immediate processing of floating-point values. The positive effect on the performance optimization is studied in this chapter and significant performance improvements are shown.

CHAPTER 8

Conclusion and Future Work

Contents

8.1 Conclusion 166
8.1.1 Summary 166
8.1.2 Outreach 167
8.2 Future Work 168

8.1 Conclusion

This thesis, especially in the topical chapters (Chapter 4, Chapter 5, Chapter 6 and Chapter 7), studies possibilities for the software exploitation of traditional interfaces for modern technologies. This means, if a system is equipped with modern hardware technologies, but the interfaces to communicate with these technologies remain on a traditional level, creative solutions need to be applied to manage the applied technology efficiently and effectively. Software exploitation, with software modifications employed at various levels from the application to the system software, is one way to exploit the existing interfaces and enrich them by newly exploited features. Although such software exploited interface should not be considered as the only or superior solution to operate modern technologies, they offer the possibility to operate them on traditional interfaces.This gains large freedom in the design of systems and allows putting a focus to other components. Consequently, allowing software exploited interfaces is crucial to effective system design.

8.1.1 Summary

Since the software exploitation for modern technologies inherently comes with being specific to these technologies, this thesis does not aim for general solutions, but rather presents various approaches to software manage different modern technologies.

In Chapter 4, the lifetime property of NON-VOLATILE MEMORY (NVM) technologies is managed by software wear-leveling. The information about the current memory usage is collected by a sampling mechanism, which captures a histogram of memory access frequencies. MEMORY MANAGEMENT UNIT (MMU) based transparent relocation, together with transparent relocation on the stack memory and on the text memory, a wear-leveling strategy is deployed, which can leverage massive shortcomings in the memory usage, which would cause extremely low memory lifetimes, if not properly managed. Furthermore, a specific focus is put to read-destructive memories, which require a read wear-leveling solution. In order to not only provide effective, but also efficient wear-leveling through the software exploited interface, a semantic memory tracing is presented that allows a specific application of wear-leveling techniques to memory regions, which can highly profit from wear-leveling.

While the presented method in Chapter 4 is realized in an application-transparent manner inside the operating system and the system software, Chapter 5 shifts the action space towards the application and presents application-cooperative wear-leveling techniques. The wear-leveling of the stack segment can consume massive overheads and allows a great potential to reduce overheads, when actions are performed at suitable points. An analysis tool is presented, which can automatically determine such smart points and deploy them back with the application implementation in order to trigger efficient wear-leveling actions for the stack. In addition, a specific wear-leveling mechanism for B^+ trees is presented, where memory usage information is internally

collected during operations on the tree. This information then is used to determine a lifetime optimized checkpoint layout to the NVM.

Chapter 6 keeps focusing on NVM technologies, but instead of limited lifetime, the latency and energy properties of RACETRACK MEMORY (RTM) are considered. RANDOM FOREST (RF) ensembles are studied as a specific application, which allows effective software optimization for RTM. The layout of the single DECISION TREE (DT) models with minimal latency and energy cost on RTM is an optimization problem, which is infeasible to compute for larger models. This chapter shows a simple and fast algorithm, where an upper bound to the optimal solution is formally proven. Two organization approaches of the single DTs on RTM are distinguished.

Chapter 7, as the last topical chapter, focuses on CPU internal arithmetic and the effect on the memory hierarchy. Two possible solutions of immediate encoding of floating-point constants are presented, which allow gaining active control over the distribution of data and instruction memories. One solution is kept generic and allows all possible floating-point arithmetic operation, by loading the floating-point value to a floating-point register from immediate encoded fields. The other solution allows only the specific comparison operation but does not require any transformation or conversion of data, which causes a very low overhead.

Throughout the entire thesis, experimental evaluation studies the effectiveness of the presented methods. For all solutions, the results justify the application of the presented software exploitation techniques in a way, that an improved usage of the considered modern technology is achieved. This makes the presented methods considerable candidates when designing a system with these modern technologies.

8.1.2 Outreach

When reading this thesis and observing how specific software solutions have to designed and which creative and partially complex solutions need to be applied, one may ask a question of how relevant software exploitation of traditional interface can be realistic. To provide at least an intuition for the interest of the research community, it should be noted that the author of this thesis participated actively and for major parts in two successful accepted DFG project proposals in exactly the scope of software solutions for modern technologies. One of these projects is the second phase of the *Design and Optimization of Non-Volatile One-Memory Architecture (NVM-OMA)*[1] project. In the second phase, a specific focus is put to tailored solutions for dedicated NVM technologies. The parts of the project, which are led by the research group in Dortmund, are almost entirely software-based and provide creative solutions for different NVM technologies. The second project is the *Memory Diplomat*[2] project, in which a central software agent negotiated between hardware management requirements and software needs. The core idea of this project is to apply creative software-based management solutions in a most efficient and effective way.

[1] https://gepris.dfg.de/gepris/projekt/405422836
[2] https://gepris.dfg.de/gepris/projekt/502384507

8.2 Future Work

Since the research results of this thesis for parts led to two accepted DFG projects, parts of the direct future work naturally are embedded in the execution of these projects. The further research for software management of NVM technologies should focus on memory technologies with unique and special properties. Skyrmion memory, for instance, leads to interesting properties due to the generation and storing of skyrmions. This offers a large space for the design of memory devices, imposing characteristics on the software, which need to be efficiently exploited. Beyond the specific focus on NVM technologies, a broader focus on disruptive memory technologies is part of the future work. In-memory computing and near-memory computing impose further interesting properties to the software and the software-based management. Technologies like high bandwidth memory further require active software-exploited management.

Out of the scope of the executing future research projects, another directions of research, which partially stems from the results of this thesis and therefore forms a straightforward future work, is the WORST-CASE EXECUTION TIME (WCET) analysis and optimization of specific software applications. Modern technologies impose interesting impacts on the WORST-CASE EXECUTION TIME (WCET) model. Especially when focusing on certain applications, software exploitation can assist the analysis and even the optimization of the WCET. A major focus for this are RF ensembles, as also widely studied in this thesis. A starting point towards this is presented in a paper for deriving WCET surrogate models for DT ensembles [HHC+23].

Beyond straightforward continuation of research directions, started by this thesis, a broader exploration of the relation of traditional interfaces and modern technologies forms a crucial part of the future work. Whenever modern technologies emerge, it must be expected that they are operated on traditional interfaces. Consequently, the components of a system have to be well-equipped to exploit these interfaces. While this thesis and the direct future work focus mainly on modern memory technologies, other emerging components have to be considered as well. One of these components is formed by emerging computing facilities in systems. Vectorization, dedicated accelerators, and computational memory belong to many computing systems nowadays. Especially for dedicated accelerators, standard interfaces, such as memory mapped registers are used. Equipping software with means to efficiently exploit such computation devices is one direction to further study.

Apart from computation facilities, modern security technologies are another emerging field. Components, such as secure execution units, trusted code and encryption devices are under active development. The integration of applications into this domain is not straightforward. Especially for data-based methods, such as machine learning, security guarantees can be of large interest. Despite the major functional property of providing security aspects, efficiency can be considered as a second major target. Optimizing the performance of emerging security technologies while keeping the security guarantees forms a field of future research.

Another aspect, which should be studied in future work, is computational disaggregation, which can be considered as a crosscutting modern technology. Computational jobs may not be executed on the machine, where they are created. In the simplest form, they may be executed on another core, a coprocessor or on an accelerator. In a more complex shape, the job may be sent over a communication network to another computation machine. Exploiting such disaggregated facilities from a software perspective can open opportunities for efficient execution and optimization.

Acronyms

CART	Classification and Regression Trees 27
CMOS	Complementary Metal-Oxide-Semiconductor 22
DBC	Domain Block Cluster 43, 44, 104, 106, 107, 115–124, 126, 129, 130
DRAM	Dynamic Random Access Memory 18–20, 25, 36, 37, 40, 43, 54, 90
DT	Decision Tree ix, 6, 7, 12–16, 26–28, 44–46, 101, 104–109, 111, 113, 115–119, 121, 123, 125, 127, 129, 130, 133, 135–137, 140, 167, 168
FeRAM	Ferroelectric RAM 18–22, 25, 31, 36, 38, 40, 59
GA	Genetic Algorithm 10, 11, 83, 86, 88–90
GOT	Global Offset Table 65, 70
HBM	High Bandwidth Memory 24
HDD	Hard Disk Drive 18, 19
ILP	Integer Linear Program 13, 105
ISA	Instruction Set Architecture 2, 7, 49, 132, 135, 164
ML	Machine Learning 7, 12, 104
MMU	Memory Management Unit 8, 9, 56, 58, 60, 63–65, 78, 82, 96, 99, 166
NVM	Non-Volatile Memory iii, vii, 1, 4–9, 11, 15, 17–19, 22, 24, 25, 28, 31, 32, 36–41, 43, 54, 55, 58, 66, 69, 78, 80, 89–96, 98, 99, 103, 115, 166–168
OS	Operating System 78, 81, 83

PC PROGRAM COUNTER 9, 61, 63–66, 73, 78, 84, 85
PCM PHASE CHANGE MEMORY 18–20, 22, 36, 37
PIC POSITION INDEPENDENT CODE 65
PLT PROCEDURE LINKAGE TABLE 65, 70

ReRAM RESISTIVE RAM 18, 19, 36
RF RANDOM FOREST iii, vii, 12, 14, 15, 17, 18, 24, 26, 28, 33, 44–46, 48, 104–106, 132–137, 147–149, 163, 164, 167, 168
RTM RACETRACK MEMORY iii, vii, 1, 6, 7, 11–13, 16, 17, 22, 28, 32, 33, 35, 36, 38, 43, 44, 102–107, 112, 115–118, 125, 129, 130, 133, 167

SLC SINGLE LEVEL CELL 40
SP STACK POINTER 63–65, 85
SRAM STATIC RANDOM ACCESS MEMORY 18, 19, 25, 40, 54, 105
SSD SOLID STATE DRIVE 2
STTM SPIN-TORQUE TRANSFER MAGNETORESISTIVE RAM 18, 19, 21, 22, 36

WCET WORST-CASE EXECUTION TIME 168

Bibliography

[AH73] D. Adolphson and T. C. Hu. “Optimal linear ordering”. In: *SIAM Journal on Applied Mathematics* 25.3 (1973), pp. 403–423 (Cited on pages 32, 107 sqq., 111, 122).

[ALD13] N. Asadi, J. Lin, and A. P. De Vries. “Runtime optimizations for tree-based machine learning models”. In: *IEEE transactions on Knowledge and Data Engineering* 26.9 (09/2013), pp. 2281–2292. ISSN: 1041-4347 (Cited on pages 33, 46, 137).

[ALM+18] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson. “BzTree: A high-performance latch-free range index for non-volatile memory”. In: *Proceedings of the VLDB Endowment* 11.5 (2018), pp. 553–565 (Cited on page 31).

[Amp19] I. V. Amplifier. *Intel Vtune Amplifier*. 2019 (Cited on page 30).

[AN07] A. Asuncion and D. Newman. *UCI machine learning repository*. 2007 (Cited on pages 49, 112, 138, 142, 158).

[Ato15] E. Atoofian. “Reducing Shift Penalty in Domain Wall Memory Through Register Locality”. In: *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES ’15. Amsterdam, The Netherlands, 2015, pp. 177–186. ISBN: 978-1-4673-8320-2 (Cited on page 32).

[AXY+14] H. Aghaei Khouzani, Y. Xue, C. Yang, and A. Pandurangi. “Prolonging PCM Lifetime Through Energy-efficient, Segment-aware, and Wear-resistant Page Allocation”. In: *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*. ISLPED ’14. La Jolla, California, USA: ACM, 2014, pp. 327–330. ISBN: 978-1-4503-2975-0. DOI: 10.1145/2627369.2627667. URL: http://doi.acm.org/10.1145/2627369.2627667 (Cited on pages 28 sq., 31).

[Bay72] R. Bayer. “Symmetric binary B-Trees: Data structure and maintenance algorithms”. In: *Acta Informatica* 1.4 (12/1972), pp. 290–306. ISSN: 1432-0525. DOI: 10.1007/BF00289509. URL: https://doi.org/10.1007/BF00289509 (Cited on page 90).

[BBB+11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. “The Gem5 Simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (08/2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: http://doi.acm.org/10.1145/2024716.2024718 (Cited on pages 29, 41, 58).

[BCC+18] S. Buschjäger, K.-H. Chen, J.-J. Chen, and K. Morik. “Realization of Random Forest for Real-Time Evaluation through Tree Framing”. In: *IEEE International Conference on Data Mining (ICDM)*. 2018, pp. 19–28. DOI: 10.1109/ICDM.2018.00017 (Cited on page 33).

[BÇP+98] R. E. Burkard, E. Çela, P. M. Pardalos, and L. S. Pitsoulis. “The Quadratic Assignment Problem”. In: *Handbook of Combinatorial Optimization: Volume1–3*. Ed. by D.-Z. Du and P. M. Pardalos. 1998, pp. 1713–1809 (Cited on pages 32, 107).

[BCR+08] Y. Bao, M. Chen, Y. Ruan, L. Liu, J. Fan, Q. Yuan, B. Song, and J. Xu. “HMTT: a platform independent full-system memory trace monitoring system”. In: *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. SIGMETRICS ’08. Annapolis, MD, USA: ACM, 2008, pp. 229–240. ISBN: 978-1-60558-005-0. DOI: 10.1145/1375457.1375484. URL: http://doi.acm.org/10.1145/1375457.1375484 (Cited on page 29).

[BFO+] L. Breiman, J. Friedman, R. A. Olshen, and C. J. Stone. “Classification and Regression Trees. 1984. Monterey, CA: Wadsworth & Brooks”. In: *Cole Advanced Books & Software Google Scholar* () (Cited on pages 26 sq., 44).

[Bix07] B. Bixby. “The gurobi optimizer”. In: *Transp. Re-search Part B* (2007) (Cited on pages 114, 125).

[BKF+20] R. Bläsing, A. A. Khan, P. C. Filippou, C. Garg, F. Hameed, J. Castrillón, and S. S. P. Parkin. “Magnetic Racetrack Memory: From Physics to the Cusp of Applications Within a Decade”. In: *Proceedings of the IEEE* (2020) (Cited on pages 6, 22, 36, 43).

[BL18] S. Byma and J. R. Larus. “Detailed heap profiling”. In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*. 2018, pp. 1–13 (Cited on page 30).

[Bli97] J. F. Blinn. “Floating-point tricks”. In: *IEEE Computer Graphics and Applications* 17.4 (1997), pp. 80–84 (Cited on page 33).

[BM18] S. Buschjäger and K. Morik. “Decision Tree and Random Forest Implementations for Fast Filtering of Sensor Data”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 65.1 (2018), pp. 209–222. ISSN: 1549-8328. DOI: 10.1109/TCSI.2017.2710627 (Cited on page 33).

[Bra] J. Bramley. *Condition Codes 4: Floating-point comparisons using VFP*. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/condition-codes-4-floating-point-comparisons-using-vfp (Cited on pages 33, 134).

[BRC+17] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao. “Emerging NVM: A survey on architectural integration and research challenges”. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 23.2 (2017), pp. 1–32 (Cited on pages 18 sq., 36 sqq., 54).

[Bre01] L. Breiman. “Random forests”. In: *Machine learning* 45 (2001), pp. 5–32 (Cited on pages 26, 44).

[BSH+17] S. Bhatti, R. Sbiaa, A. Hirohata, H. Ohno, S. Fukami, and S. Piramanayagam. “Spintronics based random access memory: a review”. In: *Materials Today* 20.9 (2017), pp. 530–548 (Cited on pages 21, 36).

[CHK+12] C.-H. Chen, P.-C. Hsiu, T.-W. Kuo, C.-L. Yang, and C.-Y. M. Wang. “Age-based PCM Wear Leveling with Nearly Zero Search Cost”. In: *Proceedings of the 49th Annual Design Automation Conference*. DAC ’12. San Francisco, California: ACM, 2012, pp. 453–458. ISBN: 978-1-4503-1199-1. DOI: 10.1145/2228360.2228439. URL: http://doi.acm.org/10.1145/2228360.2228439 (Cited on pages 28 sq., 31).

[CJ15] S. Chen and Q. Jin. “Persistent B+-trees in Non-Volatile Main Memory”. In: *Proceedings of the VLDB Endowment* 8.7 (2015), pp. 786–797 (Cited on page 31).

[CL09] S. Cho and H. Lee. “Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance”. In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 42. New York, New York: ACM, 2009, pp. 347–357. ISBN: 978-1-60558-798-1. DOI: 10.1145/1669112.1669157. URL: http://doi.acm.org/10.1145/1669112.1669157 (Cited on page 29).

[CLX15] P. Chi, W.-C. Lee, and Y. Xie. “Adapting B+-Tree for Emerging Nonvolatile Memory-Based Main Memory”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.9 (2015), pp. 1461–1474 (Cited on page 31).

[Cou08] T. P. P. Council. “TPC-H benchmark specification”. In: *Published at http://www. tcp. org/hspec. html* 21 (2008), pp. 592–603 (Cited on page 74).

[CPA+19] A. Colaso, P. Prieto, P. Abad, J. A. Gregorio, and V. Puente. “Architecting Racetrack Memory Preshift through Pattern-Based Prediction Mechanisms”. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 273–282. DOI: 10.1109/IPDPS.2019.00037 (Cited on page 32).

[CS07] T. Colburn and G. Shute. “Abstraction in computer science”. In: *Minds and Machines* 17 (2007), pp. 169–184 (Cited on page 2).

[CSH+22] K.-H. Chen, C. Su, C. Hakert, S. Buschjäger, C.-L. Lee, J.-K. Lee, K. Morik, and J.-J. Chen. “Efficient Realization of Decision Trees for Real-Time Inference”. In: *ACM Trans. Embed. Comput. Syst.* 21.6 (10/2022). ISSN: 1539-9087. DOI: 10.1145/3508019. URL: https://doi.org/10.1145/3508019 (Cited on pages 33, 48 sq., 112, 135, 137–140, 142 sq., 145 sq., 158 sqq.).

[CST+10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154 (Cited on page 95).

[CSZ+16] X. Chen, E. H.-M. Sha, Q. Zhuge, C. J. Xue, W. Jiang, and Y. Wang. “Efficient Data Placement for Improving Data Access Performance on Domain-Wall Memory”. In: *IEEE Trans. Very Large Scale Integr. Syst.* (2016) (Cited on pages 32, 112, 114, 125).

[DDR95] J. W. Demmel, I. Dhillon, and H. Ren. “On the correctness of some bisection-like parallel eigenvalue algorithms in floating point arithmetic”. In: *Electronic Trans. Num. Anal* 3 (1995), pp. 116–140 (Cited on page 33).

[DLN+16] D. Dato, C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini. “Fast ranking with additive ensembles of oblivious and non-oblivious regression trees”. In: *ACM Transactions on Information Systems* (2016). ISSN: 15582868 (Cited on page 33).

[DPS02] J. Díaz, J. Petit, and M. Serna. “A Survey of Graph Layout Problems”. In: *ACM Comput. Surv.* 34.3 (09/2002), pp. 313–356. ISSN: 0360-0300. DOI: 10.1145/568522.568523. URL: https://doi.org/10.1145/568522.568523 (Cited on page 107).

[DZH+11] J. Dong, L. Zhang, Y. Han, Y. Wang, and X. Li. “Wear rate leveling: Lifetime enhancement of PRAM with endurance variation”. In: *Proceedings of the 48th Design Automation Conference*. ACM. 2011, pp. 972–977 (Cited on pages 28 sqq.).

[ECS85] I. of Electrical, E. E. C. S. S. Committee, and D. Stevenson. “IEEE Standard for Binary Floating-Point Arithmetic”. In: *ANSI/IEEE Std 754-1985* (1985), pp. 1–20. DOI: 10.1109/IEEESTD.1985.82928 (Cited on pages 136, 149 sqq.).

[ETA14] T. Eshita, T. Tamura, and Y. Arimoto. “Ferroelectric random access memory (FRAM) devices”. In: *Advances in non-volatile memory and storage technology*. Elsevier, 2014, pp. 434–454 (Cited on pages 20 sq.).

[FZB+10] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé. “Increasing PCM Main Memory Lifetime”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE ’10. Dresden, Germany: European Design and Automation Association, 2010, pp. 914–919. ISBN: 978-3-9810801-6-2. URL: http://dl.acm.org/citation.cfm?id=1870926.1871147 (Cited on pages 28 sq.).

[GJ79] M. R. Garey and D. S. Johnson. “Computers and intractability”. In: 1979 (Cited on page 107).

[GMS+21] N. Gupta, A. Makosiej, H. Shrimali, A. Amara, A. Vladimirescu, and C. Anghel. “Tunnel FET negative-differential-resistance based 1T1C refresh-free-DRAM, 2T1C SRAM and 3T1C CAM”. In: *IEEE Transactions on Nanotechnology* 20 (2021), pp. 270–277 (Cited on pages 20, 36).

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. “MiBench: A Free, Commercially Representative Embedded Benchmark Suite”. In: *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. WWC ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. ISBN: 0-7803-7315-4. DOI: 10.1109/WWC.2001.15. URL: https://doi.org/10.1109/WWC.2001.15 (Cited on pages 58, 86).

[GWD+19] V. Gogte, W. Wang, S. Diestelhorst, A. Kolli, P. M. Chen, S. Narayanasamy, and T. F. Wenisch. “Software Wear Management for Persistent Memories”. In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, 02/2019, pp. 45–63. ISBN: 978-1-939133-09-0. URL: https://www.usenix.org/conference/fast19/presentation/gogte (Cited on pages 28–31).

[HCC22a] C. Hakert, K.-H. Chen, and J.-J. Chen. “FLInt: Exploiting Floating Point Enabled Integer Arithmetic for Efficient Random Forest Inference”. In: *arXiv preprint arXiv:2209.04181* (2022) (Cited on pages 16, 148).

[HCC22b] C. Hakert, K.-H. Chen, and J.-J. Chen. “Immediate Split Trees: Immediate Encoding of Floating Point Split Values in RandomForests”. In: *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases* (2022) (Cited on pages 16, 136).

[HCC24] C. Hakert, K.-H. Chen, and J.-J. Chen. “FLInt: Exploiting Floating Point Enabled Integer Arithmetic for Efficient Random Forest Inference”. In: *Design, Automation and Test in Europe Conference* (2024) (Cited on page 148).

[HCK+20] C. Hakert, K.-H. Chen, S. Kuenzer, S. Santhanam, S.-H. Chen, Y.-H. Chang, F. Huici, and J.-J. Chen. “Splitn Trace NVM: Leveraging Library OSes for Semantic Memory Tracing”. In: *9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 2020 (Cited on pages 16, 41, 71).

[HCS+21] C. Hakert, K.-H. Chen, H. Schirmeier, L. Bauer, P. R. Genssler, G. von der Brüggen, H. Amrouch, J. Henkel, and J.-J. Chen. “Software-Managed Read and Write Wear-Leveling for Non-Volatile Main Memory”. In: *ACM Transactions on Embedded Computing Systems Special Issue on Memory and Storage Systems for Embedded and IoT Applications*. 2021 (Cited on pages 16, 58).

[HCY+20] C. Hakert, K.-H. Chen, M. Yayla, G. v. d. Brüggen, S. Bloemeke, and J.-J. Chen. “Software-Based Memory Analysis Environments for In-Memory Wear-Leveling”. In: *25th Asia and South Pacific Design Automation Conference ASP-DAC 2020, Invited Paper*. Beijing, China, 2020 (Cited on pages 41, 58).

[HDW+16] Y. Han, J. Dong, K. Weng, Y. Wang, and X. Li. “Enhanced Wear-Rate Leveling for PRAM Lifetime Improvement Considering Process Variation”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.1 (01/2016), pp. 92–102. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2015.2395415 (Cited on pages 28 sq.).

[HHC+23] C. Hakert, N. Hölscher, K.-H. Chen, J. Reineke, and J.-J. Chen. “Fallen Leafs: Towards WCET and ACET Performance Surrogate Models for Decision Trees”. In: *Workshop on OPtimization for Embedded and ReAl-time systems (OPERA)*. 2023 (Cited on page 168).

[HKC+21a] C. Hakert, A. A. Khan, K.-H. Chen, F. Hameed, J. Castrillon, and J.-J. Chen. “BLOwing Trees to the Ground: Layout Optimization of Decision Trees on Racetrack Memory”. In: *58th ACM/IEEE Design Automation Conference (DAC), accepted*. 2021 (Cited on pages 16, 104).

[HKC+21b] C. Hakert, R. Kühn, K.-H. Chen, J.-J. Chen, and J. Teubner. “OCTO+: Optimized Checkpointing of B+Trees for Non-Volatile Main Memory Wear-Leveling”. In: *The 10th IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2021 (Cited on pages 16, 90).

[HKC+22] C. Hakert, A. A. Khan, K.-H. Chen, F. Hameed, J. Castrillon, and J.-J. Chen. “ROLLED: Racetrack Memory Optimized Linear Layout and Efficient Decomposition of Decision Trees”. In: *IEEE Transactions on Computers* (2022) (Cited on pages 16, 116).

[HMH20] K. Huang, Y. Mei, and L. Huang. “Quail: Using NVM write monitor to enable transparent wear-leveling”. In: *Journal of Systems Architecture* 102 (2020), p. 101658. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2019.101658. URL: http://www.sciencedirect.com/science/article/pii/S1383762119304655 (Cited on page 31).

[HP18] J. Hennessy and D. Patterson. “A new golden age for computer architecture: domain-specific hardware/software co-design, enhanced”. In: *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018 (Cited on pages 2 sq.).

[HSS+16] Q. Hu, G. Sun, J. Shu, and C. Zhang. “Exploring Main Memory Design Based on Racetrack Memory Technology”. In: *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*. 05/2016, pp. 397–402 (Cited on page 32).

[HYC+19] C. Hakert, M. Yayla, K.-H. Chen, G. v. d. Brüggen, J.-J. Chen, S. Buschjäger, K. Morik, P. R. Genssler, L. Bauer, H. Amrouch, and J. Henkel. “Stack Usage Analysis for Efficient Wear Leveling in Non-Volatile Main Memory Systems”. In: *1st ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*. Alberta, Canada, 2019 (Cited on pages 16, 83).

[IYZ+19] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, et al. “Basic performance measurements of the intel optane DC persistent memory module”. In: *arXiv preprint arXiv:1903.05714* (2019) (Cited on page 40).

[JZH+14] T. Jiang, Q. Zhang, R. Hou, L. Chai, S. A. Mckee, Z. Jia, and N. Sun. “Understanding the behavior of in-memory computing workloads”. In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2014, pp. 22–30 (Cited on page 30).

[KB57] T. C. Koopmans and M. Beckmann. “Assignment Problems and the Location of Economic Activities”. In: *Econometrica* 25.1 (1957), pp. 53–76. ISSN: 00129682, 14680262. URL: http://www.jstor.org/stable/1907742 (Cited on page 32).

[KCS+10] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, and P. Dubey. “FAST: Fast architecture sensitive tree search on modern CPUs and GPUs”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010 (Cited on page 33).

[KHB+19] A. A. Khan, F. Hameed, R. Bläsing, S. S. P. Parkin, and J. Castrillon. “ShiftsReduce: Minimizing Shifts in Racetrack Memory 4.0”. In: *ACM Transactions on Architecture and Code Optimization* 16.4 (12/2019). ISSN: 1544-3566. DOI: 10.1145/3372489. URL: https://doi.org/10.1145/3372489 (Cited on pages 32, 103 sq., 112, 114, 125).

[KK16] O. Kramer and O. Kramer. “Scikit-learn”. In: *Machine learning for evolution strategies* (2016), pp. 45–53 (Cited on page 45).

[Kle05] A. Kleen. “A numa api for linux”. In: *Novel Inc* (2005) (Cited on page 54).

[KRH+19] A. A. Khan, N. A. Rink, F. Hameed, and J. Castrillon. “Optimizing Tensor Contractions for Embedded Devices with Racetrack Memory Scratch-Pads”. In: *International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2019. 2019, pp. 5–18 (Cited on page 32).

[KSV+19a] S. Kuenzer, S. Santhanam, Y. Volchkov, F. Schmidt, F. Huici, J. Nider, M. Rapoport, and C. Lupu. “Unleashing the Power of Unikernels with Unikraft”. In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. SYSTOR ’19. Haifa, Israel: Association for Computing Machinery, 2019, p. 195. ISBN: 9781450367493. DOI: 10.1145/3319647.3325856. URL: https://doi.org/10.1145/3319647.3325856 (Cited on pages 58, 71).

[KSV+19b] S. Kuenzer, S. Santhanam, Y. Volchkov, F. Schmidt, F. Huici, J. Nider, M. Rapoport, and C. Lupu. “Unleashing the power of unikernels with unikraft”. In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. 2019, pp. 195–195 (Cited on page 41).

[KXM+15] D. Kline, H. Xu, R. Melhem, and A. K. Jones. “Domain-wall memory buffer for low-energy NoCs”. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6. DOI: 10.1145/2744769.2744826 (Cited on page 32).

[LeC98] Y. LeCun. “The MNIST database of handwritten digits”. In: (1998) (Cited on page 112).

[LHC+14] Q. Li, Y. He, Y. Chen, C. J. Xue, N. Jiang, and C. Xu. “A wear-leveling-aware dynamic stack for PCM memory in embedded systems”. In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2014, pp. 1–4 (Cited on pages 30 sq.).

[LNO+15] C. Lucchese, F. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini. “Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees”. In: *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM. 2015, pp. 73–82 (Cited on page 33).

[LPN+16] C. Lucchese, R. Perego, F. M. Nardini, N. Tonellotto, S. Orlando, and R. Venturini. “Exploiting CPU SIMD extensions to speed-up document scoring with tree ensembles”. In: *SIGIR 2016 - Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2016. ISBN: 9781450342902. DOI: 10.1145/2911451.2914758 (Cited on page 33).

[LR03] K. Lyytinen and G. M. Rose. “The disruptive nature of information technology innovations: the case of internet computing in systems development organizations”. In: *MIS quarterly* (2003), pp. 557–596 (Cited on page 2).

[LSX+19] W. Li, Z. Shuai, C. J. Xue, M. Yuan, and Q. Li. “A Wear Leveling Aware Memory Allocator for Both Stack and Heap Management in PCM-based Main Memory Systems”. In: *Proceedings of the 2019 Design, Automation & Test in Europe (DATE)*. IEEE. 2019, pp. 228–233 (Cited on pages 28, 30 sq.).

[LW08] A. L. Lacaita and D. J. Wouters. “Phase-change memories”. In: *Physica status solidi (a)* 205.10 (2008), pp. 2281–2297 (Cited on pages 19, 36).

[LWW+13] D. Liu, T. Wang, Y. Wang, Z. Shao, Q. Zhuge, and E. Sha. “Curling-PCM: Application-specific wear leveling for phase change memory based embedded systems”. In: *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 01/2013, pp. 279–284. DOI: 10.1109/ASPDAC.2013.6509609 (Cited on page 28).

[Man19] L. Man-Pages. *ptrace(2) Linux Programmer’s Manual*. 10/2019 (Cited on page 84).

[MIG14] S. Motaman, A. Iyengar, and S. Ghosh. “Synergistic Circuit and System Design for Energy-efficient and Robust Domain Wall Caches”. In: *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED ’14. 2014, pp. 195–200 (Cited on page 32).

[MIG15] S. Motaman, A. S. Iyengar, and S. Ghosh. “Domain Wall Memory-Layout, Circuit and Synergistic Systems”. In: *IEEE Transactions on Nanotechnology* 14.2 (2015), pp. 282–291. DOI: 10.1109/TNANO.2015.2391185 (Cited on page 32).

[MJK+19] J. Multanen, P. Jääskeläinen, A. A. Khan, F. Hameed, and J. Castrillon. “SHRIMP: Efficient Instruction Delivery with Domain Wall Memory”. In: *International Symposium on Low Power Electronics and Design (ISLPED)*. 2019, pp. 1–6. DOI: 10.1109/ISLPED.2019.8824954 (Cited on page 32).

[MWZ+14] M. Mao, W. Wen, Y. Zhang, Y. Chen, and H. Li. “Exploration of GPGPU Register File Architecture Using Domain-wall-shift-write Based Racetrack Memory”. In: *Proceedings of the 51st Annual Design Automation Conference on Design Automation Conference*. 2014, 196:1–196:6 (Cited on page 32).

[MXM+16] M. Moeng, H. Xu, R. Melhem, and A. Jones. “ContextPreRF: Enhancing the Performance and Energy of GPUs With Nonuniform Register Access”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24 (2016), pp. 343–347 (Cited on page 32).

[MZS+15] H. Mao, C. Zhang, G. Sun, and J. Shu. “Exploring Data Placement in Racetrack Memory Based Scratchpad Memory”. In: *2015 IEEE Non-Volatile Memory System and Applications Symposium*. 08/2015, pp. 1–5. DOI: 10.1109/NVMSA.2015.7304358 (Cited on page 32).

[New04] C. Newman. *SQLite (Developer’s Library)*. USA: Sams, 2004. ISBN: 067232685X (Cited on page 74).

[NHH+17] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. “An analysis of persistent memory use with WHISPER”. In: *ACM SIGPLAN Notices* 52.4 (2017), pp. 135–148 (Cited on page 30).

[NS07] N. Nethercote and J. Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100 (Cited on pages 29 sq., 56).

[NSY+20] S. Nakandala, K. Saur, G.-I. Yu, K. Karanasos, C. Curino, M. Weimer, and M. Interlandi. “A Tensor Compiler for Unified Machine Learning Prediction Serving”. In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 2020, pp. 899–917 (Cited on page 33).

[OJR+19] S. Ollivier, D. K. Jr., K. A. Roxy, R. G. Melhem, S. Bhanja, and A. K. Jones. “Leveraging Transverse Reads to Correct Alignment Faults in Domain Wall Memories”. In: *DSN*. IEEE, 2019, pp. 375–387 (Cited on pages 32, 117).

[OLN+16] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. “FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory”. In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 371–386 (Cited on page 31).

[PCM+13] R. Prenger, B. Chen, T. Marlatt, and D. Merl. *Fast MAP Search for Compact Additive Tree Ensembles (CATE)*. Tech. rep. Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2013 (Cited on page 33).

[PGG19] I. B. Peng, M. B. Gokhale, and E. W. Green. “System evaluation of the intel optane byte-addressable nvm”. In: *Proceedings of the International Symposium on Memory Systems*. 2019, pp. 304–315 (Cited on page 25).

[PH17] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128122757 (Cited on pages 23, 149 sqq.).

[Phi96] E. M. Philofsky. “FRAM-the ultimate memory”. In: *Proceedings of Nonvolatile Memory Technology Conference*. IEEE. 1996, pp. 99–104 (Cited on page 38).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (Cited on page 136).

[PYL+14] E. Park, S. Yoo, S. Lee, and H. Li. “Accelerating Graph Computation with Racetrack Memory and Pointer-assisted Graph Representation”. In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 03/2014, pp. 1–4. DOI: 10.7873/DATE.2014.172 (Cited on page 32).

[PZX15] M. Poremba, T. Zhang, and Y. Xie. “NVMain 2.0: A User-Friendly Memory Simulator to Model (Non-)Volatile Memory Systems”. In: *IEEE Computer Architecture Letters* 14.2 (07/2015), pp. 140–143. ISSN: 1556-6056. DOI: 10.1109/LCA.2015.2402435 (Cited on pages 29, 41, 58).

[QFJ+] M. K. Qureshi, M. M. Franceschini, A. Jagmohan, and L. A. Lastras. “PreSET: Improving performance of phase change memories by exploiting asymmetry in write times”. In: *ACM SIGARCH Computer Architecture News* () (Cited on page 40).

[QKF+09] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. “Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling”. In: *2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE. 12/2009, pp. 14–23. DOI: 10.1145/1669112.1669117 (Cited on page 29).

[RRV+15] A. Ranjan, S. G. Ramasubramanian, R. Venkatesan, V. Pai, K. Roy, and A. Raghunathan. “DyReCTape: A dynamically reconfigurable cache using domain wall memory tapes”. In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015, pp. 181–186. DOI: 10.7873/DATE.2015.0838 (Cited on page 32).

[SBJ+14] Z. Sun, X. Bi, A. K. Jones, and H. Li. “Design Exploration of Racetrack Lower-Level Caches”. In: *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 2014, pp. 263–266. DOI: 10.1145/2627369.2627651 (Cited on page 32).

[SBW+16] Z. Sun, X. Bi, W. Wu, S. Yoo, and H. (Li. “Array Organization and Data Management Exploration in Racetrack Memory”. In: *IEEE Transactions on Computers* 65.4 (04/2016), pp. 1041–1054 (Cited on page 32).

[Shi79] Y. Shiloach. “A Minimum Linear Arrangement Algorithm for Undirected Trees”. In: *SIAM Journal on Computing* 8.1 (1979), pp. 15–32. DOI: 10.1137/0208002 (Cited on page 32).

[SNM+15] Songping Yu, Nong Xiao, Mingzhu Deng, Yuxuan Xing, Fang Liu, Zhiping Cai, and Wei Chen. “WAlloc: An efficient wear-aware allocator for non-volatile main memory”. In: *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*. 12/2015, pp. 1–8. DOI: 10.1109/PCCC.2015.7410326 (Cited on pages 28 sq., 31).

[SWL13] Z. Sun, W. Wu, and H. Li. “Cross-layer Racetrack Memory Design for Ultra High Density and Low Power Consumption”. In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 05/2013, pp. 1–6 (Cited on page 32).

[SZL+15] G. Sun, c Zhang, H. Li, Y. Zhang, W. Zhang, Y. Gu, Y. Sun, J. Klein, D. Ravelosona, Y. Liu, W. Zhao, and H. Yang. “From Device to System: Cross-Layer Design Exploration of Racetrack Memory”. In: *DATE*. ACM, 2015, pp. 1018–1023 (Cited on page 32).

[VKS+16] R. Venkatesan, V. J. Kozhikkottu, M. Sharad, C. Augustine, A. Raychowdhury, K. Roy, and A. Raghunathan. “Cache Design with Domain Wall Memory”. In: *IEEE Trans. on Computers* 65.4 (2016), pp. 1010–1024 (Cited on page 32).

[VMG+12] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger. “Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA?” In: *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE. 2012, pp. 232–239 (Cited on page 33).

[VMS+17] A. Vahid, G. Mappouras, D. J. Sorin, and A. R. Calderbank. “Correcting Two Deletions and Insertions in Racetrack Memory”. In: *CoRR* abs/1701.06478 (2017). arXiv: 1701.06478. URL: http://arxiv.org/abs/1701.06478 (Cited on page 32).

[WA17] A. Waterman and K. Asanovic. “The RISC-V instruction set manual, volume I: User-level ISA, version 2.2”. In: *1SiFive Inc., CS Division, EECS Department, University of California, Berkeley* (2017) (Cited on page 25).

[WC20] C. Wang and S. Chattopadhyay. “Isle-Tree: A B+-Tree with Intra-Cache Line Sorted Leaves for Non-volatile Memory”. In: *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE. 2020, pp. 573–580 (Cited on page 31).

[XAM+16] H. Xu, Y. Alkabani, R. Melhem, and A. K. Jones. “FusedCache: A Naturally Inclusive, Racetrack Memory, Dual-Level Private Cache”. In: *IEEE Trans. on Multi-Scale Computing Systems* 2.2 (04/2016), pp. 69–82. ISSN: 2332-7766. DOI: 10.1109/TMSCS.2016.2536020 (Cited on page 32).

[XLM+15] H. Xu, Y. Li, R. Melhem, and A. K. Jones. “Multilane Racetrack caches: Improving efficiency through compression and independent shifting”. In: *Asia and South Pacific Design Automation Conference (ASP-DAC)*. 01/2015, pp. 417–422. DOI: 10.1109/ASPDAC.2015.7059042 (Cited on page 32).

[YWW+15] J. Yang, Q. Wei, C. Wang, C. Chen, K. L. Yong, and B. He. “NV-Tree: A Consistent and Workload-adaptive Tree Structure for Non-volatile Memory”. In: *IEEE Transactions on Computers* 65.7 (2015), pp. 2169–2183 (Cited on page 31).

[YZZ+18] T. Ye, H. Zhou, W. Y. Zou, B. Gao, and R. Zhang. “RapidScorer: Fast tree ensemble evaluation by maximizing compactness in data level parallelization”. In: *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*. 2018. ISBN: 9781450355520. DOI: 10.1145/3219819.3219857 (Cited on page 33).

[ZHN+06] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. “Super-Scalar RAM-CPU Cache Compression.” In: *Icde*. Vol. 6. 2006, p. 59 (Cited on page 89).

[ZJZ+14] M. Zhao, L. Jiang, Y. Zhang, and C. J. Xue. “SLC-enabled wear leveling for MLC PCM considering process variation”. In: *Proceedings of the 51st Annual Design Automation Conference*. 2014, pp. 1–6 (Cited on page 37).

[ZL09] W. Zhang and T. Li. “Characterizing and mitigating the impact of process variations on phase change based memory systems”. In: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 12/2009, pp. 2–13 (Cited on page 29).

[ZSY+14] M. Zhao, L. Shi, C. Yang, and C. J. Xue. “Leveling to the last mile: Near-zero-cost bit level wear leveling for PCM-based main memory”. In: *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. IEEE. 10/2014, pp. 16–21. DOI: 10.1109/ICCD.2014.6974656 (Cited on page 29).

[ZSZ+15a] C. Zhang, G. Sun, W. Zhang, F. Mi, H. Li, and W. Zhao. “Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power”. In: *The 20th Asia and South Pacific Design Automation Conference*. 01/2015, pp. 100–105. DOI: 10.1109/ASPDAC.2015.7058988 (Cited on page 32).

[ZSZ+15b] C. Zhang, G. Sun, X. Zhang, W. Zhang, W. Zhao, T. Wang, Y. Liang, Y. Liu, Y. Wang, and J. Shu. “Hi-fi playback: Tolerating position errors in shift operations of racetrack memory”. In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 2015, pp. 694–706. DOI: 10.1145/2749469.2750388 (Cited on page 32).

[ZZY+09] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. “A durable and energy efficient main memory using phase change memory technology”. In: *ACM SIGARCH computer architecture news* 37.3 (2009), pp. 14–23 (Cited on pages 28 sq., 40).