Software-level analysis and optimization to mitigate the cost of write operations on non-volatile memories

Rabab Bouziane

UNIVERSITE BRETAGNE LOIRE MATHSTIC

UNIVERSITÉ DE RENNES 1

THESE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *(voir liste des spécialités)* **Informatique**

Par

Rabab BOUZIANE

Software-level Analysis and Optimization to Mitigate the Cost of Write Operations on Non-Volatile Memories

Thèse présentée et soutenue à Rennes, le 07 décembre 2018
Unité de recherche : Inria
Thèse N° :

Rapporteurs avant soutenance :

Christine ROCHANGE Professeur, université Paul Sabatier, Toulouse
Pierre BOULET Professeur, université de Lille

Composition du Jury :

Président : Sandrine BLAZY Professeur, Université de Rennes 1
Examinateurs : Sandrine BLAZY Professeur, Université de Rennes 1
Kevin MARQUET Maître de Conférence, INSA Lyon

Dir. de thèse : Erven ROHOU Directeur de Recherche Inria, Centre Inria Rennes-Bretagne Atlantique
Co-dir. de thèse : Abdoulaye GAMATIÉ Directeur de Recherche CNRS, LIRMM, Montpellier

Invité(s)
Prénom Nom Fonction et établissement d'exercice

Abstract

Traditional memories such as SRAM, DRAM and Flash have faced during the last years, critical challenges related to what modern computing systems required: high performance, high storage density and low power. As the number of CMOS transistors is increasing, the leakage power consumption becomes a critical issue for energy-efficient systems. SRAM and DRAM consume too much energy and have low density and Flash memories have a limited write endurance. Therefore, these technologies can no longer ensure the needs in both embedded and high-performance computing domains. The future memory systems must respect the energy and performance requirements. Since Non Volatile Memories (NVMs) appeared, many studies have shown prominent features where such technologies can be a potential replacement of the conventional memories used on-chip and off-chip. NVMs have important qualities in storage density, scalability, leakage power, access performance and write endurance. Many research works have proposed designs based on NVMs, whether on main memory or on cache memories. Nevertheless, there are still some critical drawbacks of these new technologies. The main drawback is the cost of write operations in terms of latency and energy consumption. Ideally, we want to replace traditional technologies with NVMs to benefit from storage density and very low leakage but eventually without the write operations overhead.

The scope of this work is to exploit the advantages of NVMs employed mainly on cache memories by mitigating the cost of write operations. Obviously, reducing the number of write operations in a program will help in reducing the energy consumption of that program. Many approaches about reducing writes operations exist at circuit level, architectural level and software level. We propose a compiler-level optimization that reduces the number of write operations by eliminating the execution of redundant stores, called silent stores. A store is silent if it's writing in a memory address the same value that is already stored at this address. The LLVM-based optimization eliminates the identified silent stores in a program by not executing them.

Furthermore, the cost of a write operation is highly dependent on the used NVM and its non-volatility called retention time; when the retention time is high then the latency and the energetic cost of a write operation are considerably high and vice versa. Based on this characteristic, we propose an approach applicable in a multi-bank NVM where each bank is designed with a specific retention time. We analyze a program and we compute the worst-case lifetime of a store instruction. The worst-case lifetime will help to allocate data to the most appropriate NVM bank.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor M. Erven ROHOU for the continuous support of my Ph.D study. Beside my advisor, I would like to thank M. Abdoulaye GAMATIE for his co-supervision.

A special thanks to my parents, my brother, my sister and my friends for supporting me throughout writing this thesis. Thanks for all your encouragement!

I dedicate this work to my Dad who is my inspiration in life.

Contents

List of Figures

List of Tables

Chapter 1

Resumé

La consommation énergétique est devenue un défi majeur dans les domaines de l'informatique embarquée et haute performance. Différentes approches ont été étudiées pour résoudre ce problème, entre autres, la gestion du système pendant son exécution, les systèmes multicœurs hétérogènes et la gestion de la consommation au niveau des périphériques.

Cette étude cible les technologies de mémoire par le biais de mémoires non volatiles (NVMs) émergentes, qui présentent intrinsèquement une consommation statique quasi nulle. Cela permet de réduire la consommation énergétique statique, qui tend à devenir dominante dans les systèmes modernes. L'utilisation des NVMs dans la hiérarchie de la mémoire se fait cependant au prix d'opérations d'écriture coûteuses en termes de latence et d'énergie.

Dans un premier temps, nous proposons une approche de compilation pour atténuer l'impact des opérations d'écriture lors de l'intégration de STT-RAM dans la mémoire cache. Une optimisation qui vise à réduire le nombre d'opérations d'écritures est implémentée en utilisant LLVM afin de réduire ce qu'on appelle les *silent stores*, c'est-à-dire les instances d'instructions d'écriture qui écrivent dans un emplacement mémoire une valeur qui s'y trouve déjà. Cela rend notre optimisation portable sur toute architecture supportant LLVM. La validation expérimentale montre des résultats prometteurs selon le niveau de *silentness* des instructions d'écritures dans les benchmarks étudiés. Certaines considérations de conception sont analysées au niveau de la compilation et de la microarchitecture afin de tirer le meilleur parti de l'optimisation.

Le principe de l'implementation est inspiré par le travail initial de Lepak et al. [54] sur l'élimination des *silent stores* à travers des mécanismes matériels, pour améliorer le speedup sur une architecture monoprocesseur et réduire le trafic sur le bus de données sur une architecture multiprocesseur. Nous considérons une variante de compilation de cette optimisation. Notre approche consiste à transformer un code

donné en insérant des *vérification de silentness* avant chaque operation d'écriture qui a été identifiée comme potentiellement *silent*, comme décrit sur la Fig.1.1. La vérification comprend les instructions suivantes:

1. une instruction de lecture à l'adresse de l'écriture,
2. une instruction de comparaison, pour comparer la valeur écrite à la valeur déjà écrite,
3. une branche conditionnelle pour ignorer l'opération d'écriture si nécessaire.

```
1    store @x = val
```

(a) original

```
1    load y = @val
2    cmp val, y
3    bEQ next
4    store @x, val
5  next:
```

(b) transformé

FIGURE 1.1: Élimination des silent stores: le code d'origine ecrit `val` à l'adresse de `x`; Le code transformé charge d'abord la valeur à l'adresse `x` et la compare à la valeur à écrire. Si elle est égale, l'exécution du store est annulée.

Nous avons implémenté la transformation ci-dessus dans LLVM en utilisant la représentation intermédiaire du code, décrit sur la Fig.1.2. L'approche est accomplie en deux étapes : un profiling des *silent stores* basé sur les accès mémoire observés, suivi par l'application de l'optimisation uniquement sur les *silent stores* qui sont *silent* à une certaine *probabilité* donnée et qui exprime combien de fois une opération d'écriture est *silent*. Une opération d'écriture qui est *silent* et qui est souvent exécutée aura un haut niveau de *silentness* et sera donc bénéfique pour une telle optimisation.

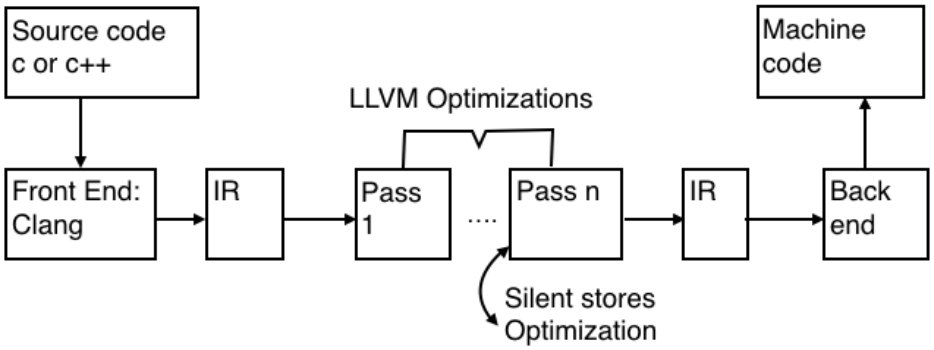
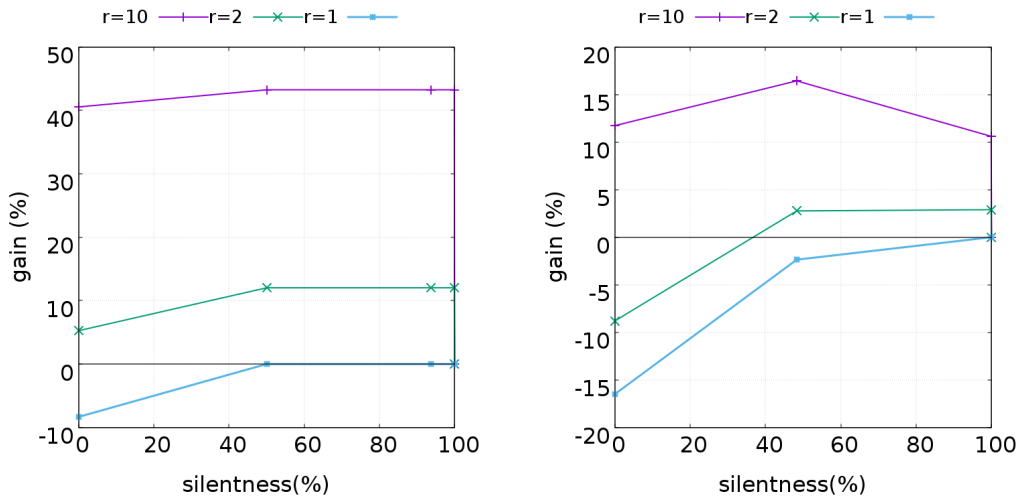


FIGURE 1.2: Implementation de l'optimization dans LLVM

Cette approche est validée dans la suite de benchmarks de Rodinia et des réductions significatives d'énergie dynamique de la mémoire (jusqu'à 42%) sont observées

en fonction des charactéristiques des programmes et de la technologie NVM. La figure 1.3 représente le gain énergétique de deux programmes Rodinia.



(A) backprop (B) bfs

FIGURE 1.3: Gain d'énergie en fonction du seuil de silentness des applications Rodinia et de leur rapport $r = \alpha_W/\alpha_R$ associé, où α_W et α_R représentent respectivement les couts des opérations d'écritures et de lectures.

Dans un second temps, nous proposons une approche qui s'appuie sur l'analyse des programmes pour estimer des *pire temps d'exécution partiaux* , dénomés δ-WCET. À partir de l'analyse des programmes, δ-WCETs sont déterminés et utilisés pour allouer en toute sécurité des données aux bancs de mémoire NVM avec des temps de rétention des données variables. L'analyse δ-WCET calcule le WCET entre deux endroits quelconques dans un programe, comme entre deux blocs de base ou deux instructions. Ensuite, les pires durées de vie des variables peuvent être déterminées et utilisées pour décider l'affectation des variables aux bancs de mémoire les plus appropriées.

L'implementation de cette partie a été faite sur Heptane, comme décrit sur la Fig.1.4, qui est un outil d'estimation statique de WCET. À partir d'un CFG, Heptane génère un problème ILP suite à une série d'analyses. Ce problème ILP est ensuite résolu en utilisant un solveur comme Cplex ou lp_solve. Le concept de notre approche se base sur l'utilisation du def-use (reaching definitions) qui permet d'identifier pour une opération d'écriture, toutes les opérations de lecture qui lui correspondent. Cela permet de construire un sous-graphe du CFG initial du programme. Ce sous-graphe représentera par la suite le graphe que Heptane va traiter et un nouveau problème ILP est généré pour le calcul du δ- WCET.

Notre approche est validée dans la suite de benchmarks de Mälardalen et des

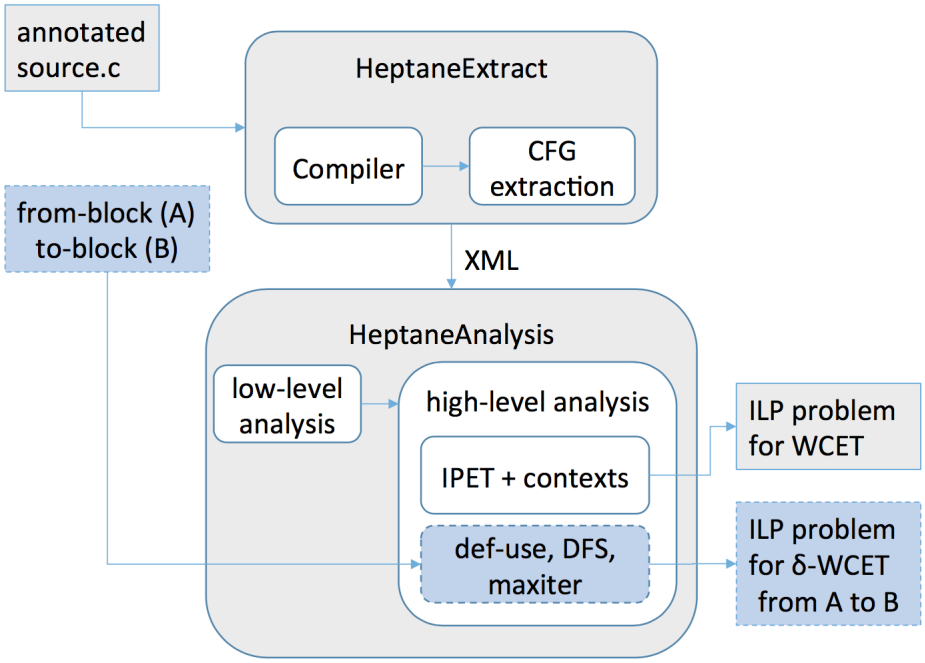


FIGURE 1.4: Outil Heptane étendu avec l'analyse δ-WCET.

réductions significatives d'énergie dynamique de la mémoire (jusqu'à 80% et 66% en moyenne, comme présentés sur Fig.1.5) sont observées en fonction des charactéristiques des programmes.

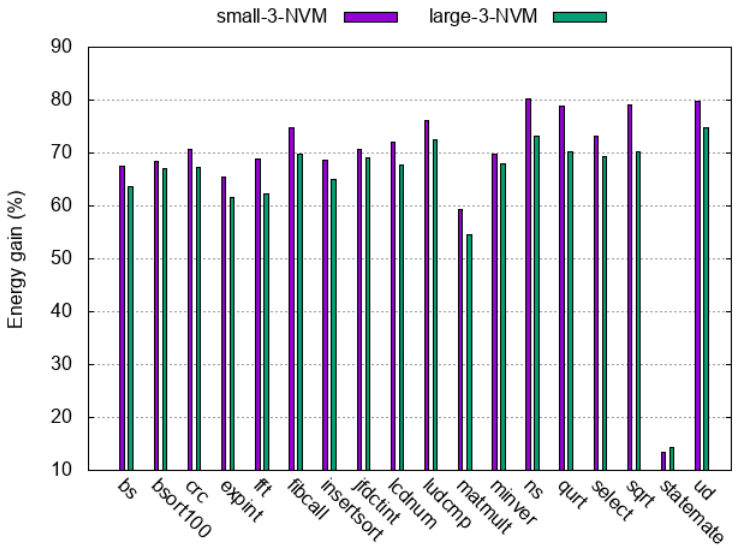


FIGURE 1.5: Économies d'énergie basées sur deux configurations différentes par rapport à une mémoire STT-RAM ayant une non-volatilité de 4,27 années.

Chapter 2

Introduction

2.1 Context and problematic

The trade-off between performance and energy consumption is one of the most challenging issues in embedded systems and high-performance systems. The ideal design would be a design providing the best computing performance while reducing as much as possible the energy consumption. As the integration of new functionalities requires high performance, the number of cores started increasing to meet the requirements. Advanced cache memory architectures including several levels have been a way to hide memory latency and to enable fast memory accesses, improving thus the performance. The current conventional memory technology used for both cache memory and registers is SRAM because of the short data access latency it provides compared to other technologies. However, as technology scales down, the leakage power in CMOS technology of the widely used SRAM-based cache gets increased, which is a major issue to energy-efficiency. **An energy-efficient system refers in this work to a system where the amount of energy (in Joules) used by a program to achieve a computation is minimized. Hence, we don't consider the conventional definition which is the number of flops/watt for HPC systems or MIPS/watt**.

Emerging Non-Volatile Memories (**NVMs**) are technologies that envisage addressing the energy consumption issue in future systems. In contrary to the traditional memory technologies that are volatile such as SRAM and DRAM, NVMs present prominent opportunities in favor of power consumption reduction by their almost zero leakage power and their non-volatility. Nevertheless, these emerging NVMs have shown critical drawbacks as well.

In the following, the challenges facing the conventional memory technologies are introduced. In addition, the challenges facing the emerging NVMs are presented. In Section 2.2, the objectives of this thesis are presented and in Section 2.3, the contributions are briefly presented.

2.1.1 Challenges facing traditional memory technologies

Traditional memories such as SRAM, DRAM and Flash have faced during the last years critical challenges related to what modern computing systems required: high performance, high storage density and low power. Memory system has become an important contributor in the overall energy consumption of modern on-chip architectures. As the traditional processor-memory gap is continuously increasing, which refers to the growing increase in number of processor cycles to get data from off-chip memory into the CPU, the imbalance between processor and memory speed is currently a drawback in modern computer system performance. The improvement of processor speed is much faster than the DRAM's one. The performance of the processor-memory interface is defined by two parameters: the latency and the bandwidth. The latency is the time between the memory request and its reply (request service), while the bandwidth, also called throughput, is the rate at which data is transferred to or from the memory system. The problem of the growing performance gap between the processor and DRAM is a latency problem. In order to deal with it, cache memory was introduced between the microprocessor and the memory based on SRAM, as a way to hide memory latency.

SRAM is widely used in on-chip memories close to the CPU, such as Scratchpad Memory (SPM), L1 cache and L2 cache. It is characterized by its rapid access latency less than 10 ns and by its endurance (number of write operations) which can go up to 10^{18}. Modern high performance Chip Multiprocessor (CMP) systems include large on-chip cache hierarchies, e.g., L1, L2, L3,... cache levels, as illustrated in Figure 2.1. Since SRAM suffers from the leakage power that makes it an energy hungry technology, an important part of the overall energy consumption comes from the multi-level on-chip cache hierarchy. In fact, as technology scales down, the leakage power in CMOS technology of the widely used SRAM-based cache gets increased, which degrades the system energy-efficiency. This motivates the need of efficient memory power management techniques able to address the issue.

Researches about the emerging NVM technologies have made a significant progress in recent years. With the attractive features of these emerging technologies, NVMs have the potential to replace the traditional memory technologies. Their high storage density, low leakage and scalability, make them very good candidates to replace the application of conventional SRAM, DRAM and Flash.

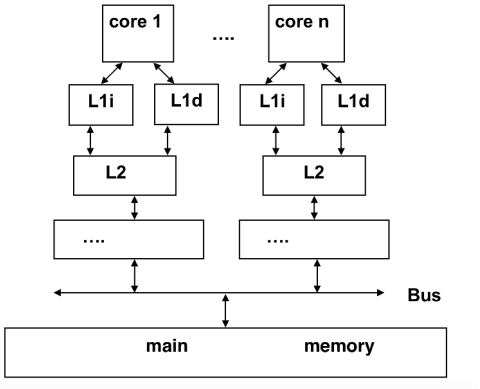


FIGURE 2.1: Memory hierarchy model

2.1.2 Challenges facing emerging NVMs technologies

Although many of the emerging NVM technologies are considered as mature technologies, they still suffer from an important downside which is their expensive write operations. Writing on an NVM is consuming both in latency and energy. Hence, this could neutralize the static energetic gain obtained thanks to the low leakage of such technologies. Therefore, the application of NVMs for on-chip memory needs effective techniques for mitigating their costly write operations in order to take advantages of NVMs features. Many previous works have proposed to employ new NVM technologies, such as PCM and STT-RAM, to replace the application of conventional SRAM, DRAM and Flash. Many approaches studied the application of STT-RAM and PCM in different ways: fist-level cache, middle or last level cache, hybrid caches with different technologies at the same cache level, e.g SRAM + STT-RAM, or at different cache levels, e.g SRAM(L2)+STT-RAM(L3). Different cache management techniques for energy saving have been proposed addressing the issue of expensive write operations: minimizing/avoiding the number of writes, cache partitioning, reducing refresh operations...

2.2 Thesis objectives

Through this thesis, we want to leverage the role of compiler optimizations and software analysis to take advantage of the NVMs features, mainly their low leakage. The current work aims to fill the gap between known compile-time techniques and successful NVM integration in upcoming energy-efficient computer architectures.

Compilers, traditionally, are not directly exposed to energy aspects. However, with all the weight given to energy consumption, it is worthy to investigate how the

compiler optimizations could have tremendous impacts on the overall energy consumption. Energy considerations are relatively recent in compilation, which is traditionally focusing more on size and speed. As the main factor for activity on processor and memory systems, software has a significant effect on the energy consumption of the entire system. Still, developing new power-aware compiler optimizations and combining them with performance-oriented compiler optimizations is the focus of several researches. As software execution corresponds to performing operations on hardware, as well as accessing and storing data, it requires power dissipation for computation, storage and communication. Beside the fact that the energy consumed during execution is very important, reducing the code size and thus, the storage cost of a program, is the classical goal of compilers. The energy cost of executing a program relies on the machine code and the target machine. Therefore, for a target architecture, energy cost is bound to machine code and consequently to compilation. Hence, it is the compilation process itself that strikes energy consumption.

Traditional optimizations on SRAM-based memories such as common subexpression elimination, partial redundancy elimination, silent stores elimination or dead code elimination, increase the performance of a program by reducing the computations during program execution [79, 2]. Memory optimizations play with latency and data placement. Optimizations such as loop tiling and register allocation try to keep data closer to the processor in order be accessed more faster. Memory optimizations contribute also in reducing energy consumption. Loop tiling tries to keep a value in an on-chip cache instead of an off-chip memory and register allocation optimization try to allocate data in a register instead of the cache. Both techniques allow to save power/energy due to reduced switching activities and switching capacitance [39].

In the wake of what was said above, the main directions involved in this thesis are related to compilation opportunities and software analysis, as the following:

- Evaluating the use of NVM-based cache memory: can we replace the SRAM-based caches by NVM-based caches? How traditional optimizations that have shown interesting results on SRAM-based caches, will behave with these new emerging technologies?.

- Energy-efficiency improvement by addressing the asymmetric nature of NVM's access: read and write latencies are asymmetric and write operations have an energetic cost higher than the SRAM's one. To mitigate this drawback, the number of write operations on NVM should be reduced.

- Reducing energy consumption via retention time relaxation: the cost of write operation (latency and energy) is related to the retention time which is the non volatility time. The higher is the retention time, the higher is the cost of a write operation. We studied a multi-retention NVM that has different banks with different retention times.

2.3 Contributions

In this work, we address the general question of improving system energy-efficiency by applying compile-time analysis and optimization techniques in presence of NVMs (here, Spin Transfer Torque RAM – STT-RAM, since STT-RAM are quite mature as test chips already exist [33, 81] and show reasonable performance at device level compared to SRAM) within memory hierarchy. The NVMs properties offer complementary advantages for meeting the performance and power consumption requirements.

We explore *compile-time analyses and optimization* and *software analysis*, as a possible alternative to leverage the low leakage current inherent to emerging NVM technologies for energy-efficiency. A major advantage is the flexibility and portability across various hardware architectures enabled by such an approach, compared to the hardware-oriented techniques found in literature. Our proposal is inspired by some existing techniques such as the silent store elimination technique introduced by Lepak et al. [55] and worst-case execution time analysis techniques [113].

The two main contributions are considered as follows:

- As writes on NVMs are generally more expensive than reads, we advocate a compile-time optimization by consistently reducing the number of writes on memory. Here, writes identified as redundant are eliminated, i.e.: when a strictly or approximately identical value is overwritten in the same memory location, respectively referred to as *strict and relaxed silent stores*. We discuss the effectiveness of this first direction from the architecture and compiler viewpoints, in "R. Bouziane, A. Gamatié, E. Rohou. How could compile-time program analysis help leveraging emerging NVM features?. EDIS 2017" [11], see Chapters 4 and 5 and "R.Bouziane, A.Gamatié, E.Rohou. Compile-Time Silent-Store Elimination for Energy Efficiency: an Analytic Evaluation for Non-Volatile Cache Memory. RAPIDO'18" [12], see Chapter 4.

- Given the possibility of relaxing the data retention time of NVMs, we leverage a design-time analysis on the lifetime of program variables so as to map them on NVM memory banks with customized retention capacities. This enables to

accommodate NVM features with program execution requirements while favoring energy-efficiency, in "R. Bouziane, A. Gamatié, E. Rohou. Energy-Efficient Memory Mappings based on Partial WCET Analysis and Multi-Retention Time STT-RAM. RTNS'18" [13], see Chapter 5.

2.4 Outline

The reminder of this thesis is organized as follows:

In Chapter 3, we review the issue of energy consumption in traditional technologies and in the emerging NVMs.

In Chapter 4, we present our silent stores elimination approach that eliminates the execution of silent stores in a program.

In Chapter 5, we present our worst-case data lifetimes methodology to allocate each store in a program to the appropriate bank in multi-retention and multi-bank NVMs.

Finally, in Chapter 6, we present a conclusion and some perspectives.

Chapter 3

Energy consumption in today's computing systems

3.1 The issue of energy consumption

The performance gap between processor's performance and memory speeds has been constantly growing in modern computer systems. State-of-the-art processors from both high-performance computing domain (e.g., Intel Xeon) and embedded computing domain (e.g., ARM Cortex-A57 processor) are able to provide their target applications with high on-chip compute power. At the same time, implemented memory systems do not show similar improvements as the cost of the access to usual off-chip main memory is hardly mitigated. This is often referred to as the *memory wall* problem in literature. The design and management of the memory system, from registers to remote memories hosted on input/output devices, is therefore an important challenge to address for enabling high system performance. Figure 3.1 illustrates the typical access latency growth across the memory hierarchy by considering the *lmbench* benchmark [72]. Here, as long as the memory accesses reach larger size memories (e.g., main memory), the duration of data retrieval increases accordingly.

Different approaches can be considered for addressing the performance issue related to the memory system. The organization of the memory hierarchy into multiple levels reduces as much as possible the access to the memory levels requiring high latency, e.g., the main memory. Software-level techniques can also mitigate the aforementioned memory access bottleneck. In particular, compile-time optimizations contribute in improving data locality in programs. Typically, it is the case of loop tiling or loop fusion transformations [4], which favor a majority of accesses to lower cache levels such as L1 or L2 caches. More generally, compilation techniques aim to improve the performance of programs by reducing as much as possible the usual

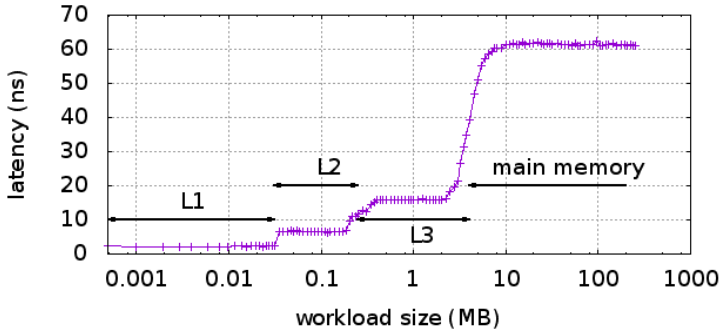


FIGURE 3.1: Typical memory access latency (Intel Xeon i7-5600U, 2.6 GHz).

costly memory accesses.

Beyond the performance issue pointed out previously regarding the memory access, it is obvious that the energy-efficiency of a system is also concerned. Indeed, higher access latency inevitably induces higher dynamic and static power consumption. The mitigation of the memory access bottleneck is therefore beneficial to the system energy-efficiency. On the other hand, as the static power consumption is becoming dominant over the dynamic power consumption in advanced chip technology nodes, an aggressive power reduction can be reached by considering memory technologies such as NVMs. A main advantage of NVMs is that they have a **quasi-zero** static power consumption thanks to their negligible leakage current. Therefore, potential optimization techniques for NVMs should mainly target their **dynamic activity**, characterized by their high read/write latency and power consumption, compared to classical SRAM and DRAM technologies. While it is a commonplace to consider quasi-similar costs for read and write accesses in the latter technologies, it is not the case with NVMs where the cost of a write often is several times bigger than that of a read. The gap about these costs varies according to the NVM technology parameters, which determine their degree of non volatility [101]. As a matter of fact, decreasing the data retention time of NVMs reduces the latency and energy related to their access. In addition, it decreases the gap between the read/write costs.

3.2 Dealing with energy consumption issue

The techniques for managing power consumption are involved at different levels. Although it has been studied for a long time in hardware, energy optimization is more recent in software. In the following, some memory and compiler optimizations are presented, followed by other techniques applied at the architectural level and circuit-level. For sake of brevity, only few works are cited here. The objective of this

section is to provide an overview of the existing techniques without covering the details.

3.2.1 Compiler optimizations

At the software-level, many directions have been conducted into compiler techniques. Yang et al. [117] studied the contribution of compiler optimizations to energy reduction. They investigated the impact of low-level loop optimizations; such as loop unrolling and software pipelining, and high-level loop optimizations; such as loop permutation and tiling, in terms of performance and power trade-offs. They showed that loop unrolling reduces execution time through effective exploitation of ILP from different iterations and results in energy reduction. Loop transformation such as loop permutation, loop tiling and loop fusion contribute significantly to energy reduction, by reducing the total execution time and the total main memory activities (due to improved cache locality).

Kandemir et al. [41] studied the effect of loop transformations in multi-bank memory architectures with low-power operating modes. They investigated how these transformations can be tuned to take into account the banked nature of the memory structure and the existence of low-power modes. They implemented three loop transformation techniques (loop fission/fusion, linear loop transformations and loop tiling) adapted to multi-bank memory. They showed that the modified loop transformations result in large energy savings in both cacheless and cache-based system.

Ştirb et al. [125] presented a new loop optimization called loop fusion designed for LLVM and studied its impact on both performance and energy consumption. Loop fusion merges two successive loops by adding the content of the second loop into the first loop and deleting the second loop. The two loops must have the same number of iterations and there should be no code between the two loops. Moreover, there should be no data dependencies between the two loops. They showed that the fusion of two loops cuts in half the number of threads that would be otherwise required to execute them. By decreasing the number of threads, the parallelization overhead is reduced, thus the energy consumption is decreased.

Mehta et al.[73] investigated the effect of loop unrolling, software pipelining and recursion elimination on CPU energy consumption. Then, they proposed a novel compiler technique that reduces energy consumption by proper register labeling during the compilation phase. The idea behind this technique is to reduce the energy of the processor by reducing the energy of the instruction register and the register file

decoder by encoding the register labels such that the amount of switching in those structures is minimized.

Ramanujam et al.[95] addressed the problem of estimating the amount of memory needed for transfers of data in embedded systems. Their technique estimates the number of distinct array accesses and the minimum amount of memory in nested loops, and reduces this number through loop-level transformations.

3.2.2 Dynamic voltage and frequency scaling (DVFS)

DVFS is a technique for altering the voltage and/or frequency of a computing system based on performance and power requirements. For CMOS circuits, dynamic power is related with voltage and frequency as $P = C \times V^2 \times f$ where C is the effective load capacitance,V the supply voltage and f the clock frequency. Thus, by changing the voltage and frequency, the power dissipation of the processor can be managed, according to workload at run-time to reduce dynamic power and save energy.

Hua et al. [31] studied saving energy in embedded systems using DVFS. Since supporting a large number of voltage levels causes overhead, their approach investigated the optimal number of voltage levels and their values to implement on the multiple-voltage system for achieving energy efficiency. They showed that systems with 3 or 4 voltages are almost as systems with more voltage levels. DVFS technique can be applied to scheduled applications for energy reduction. DVFS uses slack times in the schedule to slow down processes and save energy.

In real time systems, DVFS technique is strongly related to tasks scheduling. The timing deadlines must be respected. Since reducing the frequency will lead to an increase in the execution time, which may make a task not finishing its execution before its deadline. Hence, many works have studied power-aware scheduling based techniques. Kianzad et al. [47] presented a framework called CASPER (combined assignment, scheduling, and power-management) which is based on genetic algorithm to integrate task scheduling and voltage scaling under a single iterative optimization loop. Their technique generates a schedule such that deadline constraints are respected and the power consumption is minimized and distributes the slack time proportionally to different tasks.

3.2.3 Power modes

In embedded systems, a set of operating modes is provided in order to save energy. Different modes consume different amount of power and have different transition time to return back to the normal mode. The lower is the power mode, the higher is

the transition time and vice versa. Therefore, switching from one mode to another should be done carefully. Otherwise, the transition time may have a negative impact on systems with timing constraints.

Li et al. [56] proposed an approach for modeling and selecting the power modes for the optimal system-power management of embedded systems under timing and power constraints. Their approach determines the schedule of mode transitions such that the system can meet its power and timing constraints.

Niu et al. [80] proposed a technique to save both leakage and dynamic energy in hard real-time system scheduled by the earliest deadline first (EDF) strategy. To balance the dynamic and leakage energy consumption, higher-than-necessary processor speeds may be required when executing real-time tasks, which can result in a large number of idle intervals. In order to use these idle times, they proposed a technique that can effectively merge these scattered intervals into larger ones by making the coming tasks delayed as much as possible, without causing any deadline miss. Large idle intervals lead to reduced mode transition overhead, since the processor can stay in either idle or active state continuously for longer time.

3.2.4 Microarchitectural techniques

At the architectural level, Kaxiras et al.[44] presented a method to reduce cache leakage energy consumption by turning off cache lines that likely will not be used again. The authors were able to reduce L1 cache leakage energy by 5x for certain benchmarks with no overhead impact on performance. Zhang et al. [120] proposed a highly-configurable cache architecture for facilitating dynamic reconfiguration to save energy in embedded systems. The cache can be configured by software to be direct-mapped, two-way, or four-way set associative, using a technique they call way concatenation, having very little size or performance overhead. They showed that the proposed cache architecture reduces energy caused by dynamic power compared to a way-shutdown cache.

Hajimiri et al. [25] presented cache reconfiguration and code compression to improve both performance and energy efficiency of embedded systems. For a single-level cache hierarchy, their technique carries out an exhaustive research for the best configuration to minimize energy consumption though an exploration of the cache by varying different parameters such as line size, associativity and total size; and simulating each one of the resultant configuration. The code compression scheme is combined with cache reconfiguration, since code compression improves performance

by reducing memory traffic and bandwidth usage. Hence, the performance loss due to the cache reconfiguration can be neutralized.

Tsai et al.[107] proposed a memory structure called Trace Reuse (TR) Cache to serve as an alternative source for instruction delivery. Through an effective scheme to reuse the retired instructions from the pipeline back-end of a processor, the TR cache presents improvement both in performance and power efficiency by achieving a higher instruction rate.

Some researchers have used scratchpad memory for saving energy in embedded systems. Scratchpad memory refers to on-chip SRAM that is mapped into an address space disjoint from the off-chip memory but connected to the same address and data buses. Compared to off-chip memory, both cache and scratchpad allow much faster access. The main difference between the cache and scratchpad is that the cache access may lead to either hit or miss, while the scratchpad guarantees a single-cycle access time. Steinke et al. [103] proposed a new software technique which supports the use of an on-chip scratchpad memory by dynamically copying program parts into it and then executes the program from the scratchpad itself. This reduces the access to cache and therefore leads to saving of energy. The set of selected program parts are determined with an optimal algorithm using integer linear programming.

Benini et al. [7] presented their design of application specific memory to save energy in embedded systems. Their technique consists on mapping most frequently accessed locations onto a small memory (ASM). This small memory can be placed on-chip and be very close to the processor, thus, the access to it will cost less than accessing large memory. Their technique does not use cache. The use of such a memory instead of a traditional cache is advantageous because memory accesses become less energy consuming. The memory architecture that implements the proposed scheme exploits an ad hoc decoder that maps processor addresses onto ASM addresses. Such a decoder manages the communication with the processor by signaling whether or not the current address is accessing an ASM location. Their local memory and decoding logic ensure that most frequently accessed data are stored in a small number of contiguous memory addresses.

3.2.5 Circuit level techniques

At the circuit-level, Powel et al.[91] proposed an approach to reduce the leakage energy dissipation in instruction caches. They propose, gated-Vdd, a circuit-level technique to gate the supply voltage and reduce leakage in unused SRAM cells. Ye et al.[119] developed a method of transistor stacking in order to reduce leakage

energy consumption while maintaining high performance. Leakage reduction is achieved with minimal overheads in area, power and process technology. Lee et al.[53] proposed a new method that uses a combined approach of sleep-state assignment and threshold voltage (Vt) assignment in a dual-Vt process. They were the first to combine these two methods. By combining Vt and sleep-state assignment, leakage current is reduced since the circuit is in a known state in standby-mode and only transistors that are off need to be considered for high-Vt assignment.

So far, various directions have been investigated in order to reduce energy consumption on conventional memory technologies. However, the emerging NVMs technologies with the features of low leakage power and high density, present new ways of addressing the memory leakage power consumption issue.

3.3 Energy consumption issue in emerging NVMs

In this section, we introduce the characteristics of the emerging NVMs, in particular STT-RAM and we present some important research directions devoted for energy consumption in architectures using NVMs.

3.3.1 Quick introduction to NVMs basic features

For the last few decades, computer memory systems have been studied based on characteristics, such as technology, volatility and speed. The classical memory technologies that have been traditionally used for their high speed like SRAM for caches and DRAM for main memory are volatile memories. In the other hand, memories like magnetic disks for data storage and low speed/low energy flash memory that are used in embedded systems, are non-volatile memories

With the increasing concern about energy consumption in both embedded and high-performance systems, new memory technologies have appeared, such as Phase-Change RAM (PCRAM), Spin-Transfer Torque RAM (STT-RAM) and Resistive RAM (RRAM), that present prominent features and open new opportunities for improving the energy-efficiency of computer systems. Indeed, their very low leakage power, makes them very good candidates for optimized energy consumption, as studied in both academia and industry [77]. NVMs have zero standby power, high density and non-volatility. However, they also have the following limitations:

- Relatively long and asymmetric access latencies,
- High dynamic power consumption for write operations,

Feature	SRAM	DRAM	Disk	Flash	STT-RAM	PCRAM	RRAM
Cell size	>100F^2	6-8F^2	2/3F^2	4-5F^2	37F^2	8-16F^2	>5F^2
Read latency	<10 ns	10-50 ns	8.5ms	25 μs	<10 ns	48 ns	<10 ns
Write latency	<10 ns	10-60 ns	9.5 ms	200 μs	12.5 ns	40-150 ns	10 ns
Energy per bit access	>1 pJ	2 pJ	100-1000 mJ	10 nJ	2 pJ	100 pJ	0.02 pJ
Leakage power	High	Medium	High	Low	Low	Low	Low
Endurance	>10^{15}	>10^{15}	>10^{15}	10^4	>10^{15}	10^5-10^9	10^5-10^{11}
Non volatlitly	No	No	Yes	Yes	Yes	Yes	Yes
Scalability	Yes	Yes	Yes	Yes	Yes	Yes	Yes

FIGURE 3.2: Comparison of traditional memory and NVM technologies [122].

- Limited write endurance.

Current NVM technologies have different performance, energy and endurance properties as shown in Figure 3.2. For instance, PCRAM and RRAM, have limited endurance that is much lower than SRAM and DRAM. However, with current technology, STT-MRAM has an endurance similar to SRAM, making it conceivable to design cache hierarchies in non-volatile technologies, including the first level.

From a comparative view, we observe the following:

- Compared to SRAM and DRAM, PCM and STT-RAM have four main advantages: higher storage density, lower leakage power, better scalability and non-volatility,
- Compared to Flash, PCM and STT-RAM have better access performance and better write endurance,
- PCM has higher storage density and STT-RAM has better access performance and better write endurance.

Thus, using NVM can enable significantly larger caches which will lead to fewer cache misses and consequently reduced cache miss rate. Therefore, the NVMs performance will be better than the performance on SRAM-based caches. Using NVMs can also reduce the leakage energy, hence reducing the total energy consumption. Since PCM and STT-RAM are presenting prominent features to replace the traditional memory technologies such as SRAM, DRAM and Flash, many research works have been studying the usage of STT-RAM in cache memories and PCM in ScratchPad memories and in main memory.

Nevertheless, the emerging NVMs have one drawback which is their high write overhead in terms of latency and power consumption, especially compared with their counterpart SRAM. This makes NVMs a priori less favorable for write-intensive workloads. Therefore, a straightforward use of NVMs in place of SRAM would inevitably lead to a performance degradation and dynamic power increase that would in turn become energy-detrimental despite the leakage power saving.

In this work, we focus on a typical emerging NVM technology which is **STT-RAM**. STT-RAM uses a **Magnetic Tunnel Junction (MTJ)** as the memory storage. We focus on this technology to pursue the rest of this work. Nevertheless, as we mainly address a common feature to all types of NVMs which is the access asymmetry, we assume the applicability of our approaches to all NVMs.

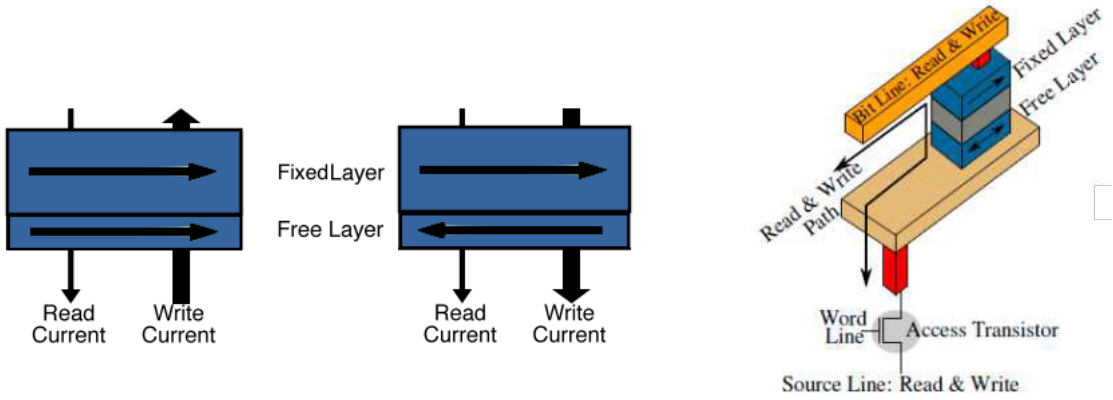
Magnetic Tunnel Junction Device

Unlike charge-based memories such as DRAM, STT-RAM uses Magnetic Tunnel Junction (MTJ) [123] as the information carrier and store data in the form of change in physical state. Each MTJ contains two ferromagnetic layer, *hard* layer and *free* layer, separated by an oxide barrier layer. The magnetization direction of the *hard* layer is fixed, while the one of the *free* layer can be changed by passing a spin-polarized current flowing through the MTJ.

When the polarities of the two layers are aligned, which represents the parallel state, electrons that have a polarity anti-parallel to the two layers (*fixed* and *free*) can move easily through the MTJ, while electrons with the same polarity as the two layers are dispersed. Inversely, when the two layers have anti-parallel polarities, which represents the anti-parallel state, electrons are largely dispersed by one of the two layers, whether their polarity is parallel or anti-parallel, leading to higher resistance. These low and high resistances are used to represent different logic value, [23].

Based on the current direction, i.e the spin of the electrons in the free layer, the spin polarity of the *free* layer will be either parallel or anti-parallel to the *hard* layer's one. Depending on whether the free layer is parallel or anti-parallel to the magnetic orientation of the hard layer, as shown in Figures 3.3(a) and 3.3(b), respectively, the applied voltage can lead to low or high current. This is due to the asymmetry of the current needed to switch the MTJ from the parallel state to the anti- parallel state and inversely [43].

The transition from the parallel to the anti-parallel state requires more current than the transition in the other way. Based on this feature, a value ('0' or '1') is stored in an STT-RAM cell, using the following logic, see Figure 3.3(c):



a) Low resistance b) High resistance c) Equivalent circuit

FIGURE 3.3: Magnetic tunnel junction (a) Parallel state, (b) Anti-parallel state, (c) Equivalent circuit

- when writing '1' into the STT-RAM cell, a positive voltage is applied between the source-line (SL) and the bit-line (BL), which will therefore produce a high-resistance state.
- when writing a '0' into the STT-RAM cell, a negative voltage is applied between the SL and the BL, which will therefore produce a low-resistance state.

As the *free* layer needs no current to maintain its state, MTJs have no inherent leakage current. The MTJ is usually modeled as a current-dependent resistor in the circuit schematic.

STT-RAM uses spin transfer torque to change the direction of the free layer by passing an important and directional write current through the MTJ. The switching process is managed by a random thermal process, meaning that the free layer could change state at any time. However, the MTJ magnetic properties and design are conceived in order to make this event improbable [102]. Performing a write requires that the write current is maintained for a certain amount of time, to guarantee that the free layer has changed state.

The thermal stability of the MTJ is the feature that determines the predicted time of a random bit-flip. The expected time is what we call the retention time. When the stability is high, the random bit-flips are unlikely to happen which makes write operations difficult and therefore high currents are required.

The stability is estimated by the thermal factor, Δ and the retention time T of a MTJ is determined by Δ, such as:

$$T = \frac{1}{f_0} e^{\Delta}$$

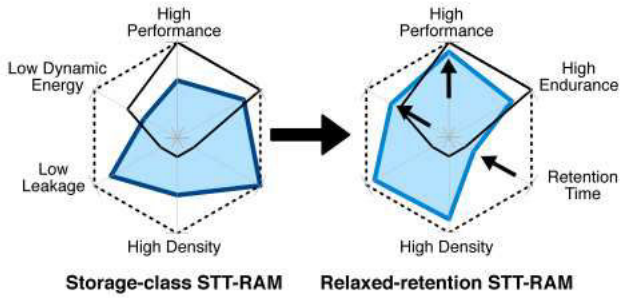


FIGURE 3.4: Relaxing the non-volatility of STT-RAM [102], dotted border is the optimal scenario and black line is representing the comparison to SRAM

where f_0 is the thermal attempt frequency.

For that reason, the thermal instability can lead to data loss which reduces the retention time. As wee see in Table 3.1, the lower is Δ, the lower is the retention time which means when the Δ decreases, the data loss is rapid.

In another part, studies have also shown that the working temperature has also an impact on data retention time. When the temperature increases, the retention time decreases exponentially [106]. Increasing the temperature from 300K to 350K allows to reduce the retention time to less than a month [96].

Figure 3.4 shows the impact of relaxing the retention time on energy and performance. Shorter retention times can be obtained by reducing the area of the *free* layer of the MTJ, making a significant write power saving as the cost of writing to a cell becomes less expensive [106].

Therefore, as a write operation to an NVM requires changing the physical state of the NVM, the cost of this write will be high since it consumes larger time and energy than a read operation, showing the read/write asymmetry of such technologies. In the same direction, the write-latency and energy cost of $1 \rightarrow 0$ transition are higher than that the ones of $0 \rightarrow 1$, leading to $0/1$ write asymmetry [9].

Data retention time	Δ
10 years	40.29
1 year	37.99
1 month	35.52
1 week	34.04
1 day	32.09
1 hour	38.91
1 minute	24.82
1 second	20.72

TABLE 3.1: Example thermal factors Δ for a range of retention times at 300K [102]

3.3.2 Dealing with energy consumption issues related to NVM caches

We discuss in this section some important approaches addressing energy consumption of caches designed with emerging NVMs. The common point between these approaches is to overcome the shortcomings of such technologies. We present here some compiler-level approaches, followed by architectural-level approaches. Then, some works that exploit reducing the retention time are presented. Tables 3.2, 3.3 and 3.4 present an overview of different approaches targeting NVMs at different level, software/compiler, architecture and circuit, respectively.

Software-level optimizations via compiler techniques

Chen et al. [17] proposed an SRAM-STT-RAM hybrid cache design which aims to minimize the number of writes to the STT-RAM. Their cache management technique uses the compiler to provide static hints to guide initial data placement such that the write-intensive data are placed into SRAM and the rest of data are placed into STT-RAM. Knowing that the compiler hints may be inaccurate due to lack of runtime information, the authors used hardware-support to correct the inaccurate hints provided by the compiler. Thus, if a write-intensive block is placed in the STT-RAM, it is moved to SRAM and replaced by an SRAM block that contains a less write intensive data.

In hybrid caches, many approaches considered employing migration based policy to mitigate the drawbacks of NVMs. Migration techniques employ additional reads and writes to ensure the data movement, therefore penalizing the performance and energy efficiency of STT-RAM based hybrid cache. For that purpose, Li et al. [62] proposed a migration-aware compilation for STT-RAM-based hybrid cache, by rearranging data layout to reduce the overhead of migrations. In [64], they addressed this issue through a compilation method called migration-aware code motion where data access patterns are changed in memory blocks in order to minimize the overhead of migrations.

Li et al. [65] proposed a compiler-assisted approach, called preferred caching, to reduce the overhead due to migration in hybrid caches. Their technique consists on giving migration-intensive memory blocks the preference for the SRAM part of the hybrid cache. The authors proposed a data assignment technique to improve the efficiency of preferred caching.

Li et al. [63] proposed two compilation-based approaches to improve the energy efficiency and performance of STT-RAM-based hybrid cache by reducing the migration overheads. The first approach, called migration-aware data layout, is proposed

to reduce the migrations by rearranging the data layout. The second approach, called migration-aware cache locking, is proposed to reduce the migrations by locking migration-intensive memory blocks into SRAM part of hybrid cache. The authors showed that the combination of these two techniques can reduce furthermore migrations.

Hu et al. [30] presented an approach based on region partitioning in order to generate optimized data allocation. A program is divided into regions and before executing each region, a data allocation is generated, which is suitable for the region. Li et al. [67] proposed a software dispatch, a cross-layer approach to distribute data to appropriate memory resources based on an application's data access characteristics. They presented an SRAM-STT-RAM hybrid cache design to increase cache capacity and mitigate the limited write endurance problem of STT-RAM. They used compiler and operating system support to migrate the write intensive data from STT-RAM to SRAM. The compiler identifies the write reuse patterns of heap data objects such as linked structures and arrays and inserts instructions to guide the hardware to perform the migration dynamically.

Hu et al. [29] targeted embedded chip multiprocessors with scratchpad memory (SPM) and non volatile main memory. They exploited data migration and recomputation in SPM so as to reduce the number of writes on main memory. Goswami et al. [22] implemented an SRAM-STT-RAM hybrid share memory in GPU. Using STT-MRAM in shared memory design, dynamic power, leakage power and area are reduced. They proposed an additional SRAM based shared memory area that can be configured using software level APIs to behave as cache or RAM to reduce STT-RAM access. Using compiler assistance, last SRAM cache eviction data is forwarded to the next level of memory avoiding STT-RAM.

Li et al. [69] presented a dual-mode STT memory cell to design a configurable L1 cache architecture termed C1C to mitigate read performance barriers with technology scaling. Guided by application access characteristics discovered through novel compiler analyses, the proposed cache adaptively switches between a high performance and a low-power access mode.

Recently, researchers proposed to improve the write performance of STT-RAM by relaxing its non-volatility property. To avoid data loss resulting from volatility, refresh schemes are proposed. However, refresh operations consume additional energy. Li et al. [61] proposed to reduce the number of refresh operations through re-arranging program data layout at compilation time. Rodriguez et al.in [97] presented the steps for designing an STT-RAM-based multi-retention scratchpad and a

customized compiler-based data allocation algorithm. Qiu et al.in [93] presented a refresh-aware loop scheduling for high performance low power volatile STT-RAM in which a loop scheduling technique traverse loops in a new direction such that data lifespan can be shortened.

Yang et al. [68] presented a cross-layer approaches that includes the compiler, operating system, and hardware layer. Cross-layer techniques can extract characteristics from the application that can be used to deliver the highest possible performance while minimizing power consumption for systems using NVMs. They described cross-layer approaches to detect data access and communication patterns within the applications and use this information to efficiently configure hardware that uses STT-RAM.

Table 3.2 summarizes the categories of approaches presented here.

Data migration in hybrid caches	[63, 30, 29, 105, 62, 64, 22, 17, 67, 65]
Reducing refresh operations	[61]
Cache reconfiguration	[69, 93, 97]
Cross-layer approach	[68]

TABLE 3.2: Some important approaches for minimizing/avoiding write operations

Architecture-level designs

There are a number of studies devoted to energy-efficiency of NVM-based caches. Reducing the impact of their expensive write operations leads to enhance the endurance of NVMs.

Some approaches addressed the energy-efficiency issue by considering hybrid cache memories using hardware mechanisms to migrate data between NVM and SRAM memory blocks. The idea is to keep as many write-intensive data in the SRAM blocks as possible in order to reduce the number of write operations to the NVM blocks. Xiaoxia et al. [114] evaluated two types of hybrid cache architectures: inter cache level hybrid cache architecture, in which the levels in a cache hierarchy can be made of different memory technologies; and intra cache level hybrid cache architecture, where a single level of cache can be partitioned into multiple regions where every region is designed with a certain type of memory technology. They studied hybrid caches made of combinations of SRAM, eDRAM, MRAM and PCM. Moreover, they proposed and evaluated low-overhead intra-cache data movement power-aware policies and their hardware support to both improve cache performance and reduce energy consumption.

Hybrid caches	[70, 104, 59, 114]
Cache management techniques	[111, 37, 35, 78, 75, 76, 94, 88]
Reducing number of writes to NVMs	[38, 118, 52, 1, 86, 23]
Minimizing refresh operations	[58, 106]
Addressing 0/1 write-asymmetry	[49]
Reducing retention time	[36, 106]
Cache reconfiguration	[16, 121]
Others	[84]

TABLE 3.3: Some important approaches for mitigating the cost of writes at architectural level

Li et al. [59] proposed a novel hybrid cache scheme called Dual Associative Hybrid Cache which, is capable of effectively mitigating the power dissipation of the emerging hybrid cache, through exploiting the set-level write non-uniformity. By organizing the SRAM blocks in the hybrid cache as a semi-independent set-associative cache, several hybrid cache sets can efficiently share and cooperatively use their SRAM blocks, along with the appropriate cache management policy, instead of exclusively using the SRAM blocks in each cache set.

Chen et al. [16] proposed a novel reconfigurable hybrid cache architecture, in which NVM is integrated in the LLC along with SRAM. The reconfigurable hybrid cache can be reconfigured by powering on/off SRAM and/or NVM arrays at way level. Furthermore, they provided hardware-based mechanisms to dynamically reconfigure reconfigurable hybrid cache based on the cache demand. Zhao et al. [121] proposed a bandwidth-aware reconfigurable cache hierarchy with hybrid memory technologies to improve the memory system with regard to bandwidth optimization. Their approach consists of an hybrid cache hierarchy that chooses different memory technologies to configure each level so that the bandwidth provided by the overall hierarchy is optimized. They also presented a reconfiguration mechanism to dynamically adapt the cache space of each level based on the predicted bandwidth demands of different applications, obtained by their prediction engine.

Depending on the behavior of memory access in a program and cache replacement strategies, a scenario of unbalanced writes to cache blocks may cause some memory cells to fail earlier than others which leads to shortening the cache lifetime. Since SRAM has a high write endurance, unbalanced writes do not have any impact. Hence, the traditional cache management policies cannot be applied directly to NVM-based on-chip caches because a large intra-set and inter-set variation may exist in the cache writes. Therefore, the existing wear-leveling techniques are not suitable for NVMs. Eliminating cache write variation is one of the important problems that was tackled in the literature to design reliable NVM-based caches. To address this, new wear-leveling

techniques have been proposed which aim to improve NVMs endurance by evenly distributing the writes to different cache blocks. Intra-set write variation increases if the associativity increases and it can be larger than inter-set write variation. Hence, many works have studied intra-set wear leveling techniques.

Wang et al. [111] proposed a new cache management policy that can reduce both inter and intra-set write variations. Joo et al. [37] developed a trace-driven PCM cache simulator that reports performance and energy as well as the accumulated number of writes of each PCM cell. They studied a set of techniques and a set of wear leveling schemes to design an energy and endurance-aware PCM cache. Jadidi et al. [35] proposed cache management policies to reduce the writes on an hybrid cache. Each cache set is composed of a limited SRAM lines and a large number of STT-RAM lines. SRAM lines are used for frequently-written data and the rarely-written or read-only ones are targeting the STT-RAM. Using intra-set swapping policies, the authors tried improving the write utilization of SRAM lines and uniforming the write distribution over STT-RAM lines within a set. Then by using inter-set policies, the rarely used SRAM lines of a set are exploited to handle write-intensive data that are kept in STT-RAM lines of another set.

Mittal et al. [78, 75] presented techniques to increase cache lifetime by reducing intra-set write variation. The idea behind is to change the physical cache-block location of a write-intensive data item within a set to achieve wear-leveling by periodically flushing a frequently-written data-item. Hence, next time the block can be made to load into a cold block in the set. Through this, the future writes to that data-item can be redirected from a hot block (i.e. frequently written) block to a cold block, which helps to achieve wear-leveling and improve cache lifetime. In another work [76], Mittal et al. presented a technique called WriteSmoothing for mitigating intra-set write variation in NVM caches. WriteSmoothing logically divides the cache-sets into multiple modules. The concept is based on the fact that if the intra-set write variation in a module is larger than a threshold, then the most frequently written cache ways in a module are made temporarily unavailable to transfer the write-intensity to other ways and this leads to wear-leveling which improves the cache lifetime.

Péneau et al. [88] presented an approach that mitigates the drawback of STT-RAM by exploiting its density to increase LLC cache size and by applying an improved cache replacement policy to reduce the LLC write-fill operations due to cache misses. The authors applied the Hawkeye replacement policy which is designed for reducing cache read misses. They showed that write-fill operations are undesirable sides of read misses and they are more important than write-back operations in order to

improve performance since they are on the critical path to main memory access.

Lin et al. [70] proposed a novel hybrid cache architecture that contains three types of cache banks: SRAM banks, STT-RAM banks and STT-RAM/SRAM hybrid banks, to consider the unbalanced write distribution among different cache partitions. Based on their proposed hybrid cache design, they presented two access-aware policies to mitigate unbalanced wearout of the STT-RAM region and a wearout-aware dynamic cache partitioning scheme to dynamically partition the hybrid cache. Quan et al. [94] proposed a new cache replacement and management policy for an STT-RAM and SRAM Hybrid L2 cache. The proposed technique replaces the dead cache lines in SRAM to improve the utilization ratio of SRAM and places frequently-written lines into SRAM and move out the less-written lines from SRAM as early as possible.

Sun et al.[104] presented a read-preemptive write technique which allows an SRAM-STT-RAM hybrid L2 cache to get performance improvements and power reductions. They proposed a write-buffer design to address the long write latency of L2 STT-RAM cache. The L2 may receive a request from both L1 and write buffer. The buffer uses a read-preemptive management policy, that makes sure that a read request gets a higher priority than a write request based on the fact that reads are performance-critical. They showed that combining the hybrid cache with read-preemptive write-buffer leads to additional energy saving and performance improvement.

On the other hand, Jung et al. [38] provided an architectural technique that exploits the fact that there are a significant number of zero-valued data in many applications. The authors proposed a cache design that appends additional flags in cache tag arrays and set these additional bits if the corresponding data in the cache line is equal to zero. In the proposed cache design, they leverage the predominance of zero data to reduce the large write energy of the STT-RAM cache by not executing write operations on STT-RAM when the to be written value is zero.

Park et al. [86] proposed a technique to reduce the write activity on a last level STT-RAM cache. Their approaches is based on dividing the cache line into multiple partial lines. At L1 cache, a history bit is associated to every partial line, to track which one has changed. Then, whenever a dirty L1 block is written to last level cache, only changed partial lines are written. Furthermore, they proposed a read energy reduction technique exploiting the non-volatility of STT-RAM, to reduce the dynamic energy. The technique is based on sequential tag-data access. A read operation will only read the tag values and if it is a hit, then only the selected cache line is accessed.

Azdanshenas et al.[118] proposed a novel coding scheme for STT-RAM last level cache based on the concept of value locality. Their technique consists on reducing

switching probability in cache by reducing the probability of memory cells being in logical "1" (high) state and then applying early write termination [124] at circuit level to eliminate redundant writes.

In order to mitigate the cost of writes, subbank buffers make possible to perform differential writes [52], where only bit positions that differ from their original contents are modified on a write. Partial writes enhance memory endurance, providing 5.6 years of lifetime for PCMs. Ahn et al. [1] proposed a novel technique called lower-bits caches for reducing write activities STT-RAM L2 caches. Based on the fact that upper bits of data are not changed as frequently as lower bits in many applications, their approach tries to hide frequent bit changes in lower bits from the L2 cache.

Guo et al. [23] proposed the use of STT-RAM to design combinational logic, register files and on-chip storage (I/D L1 caches, TLBs and L2 cache). They evaluated multiple cache designs where STT-RAM-based L1 and STT-RAM-based L2 caches can have the same capacity/same area as the SRAM caches. The L1 cache is optimized for write speed while L2 cache is optimized for density and access energy. In order to hide the write latency of STT-RAM, the authors propose sub-bank buffering that allows the write operations to complete within each sub-bank. Their results show that by carefully designing the pipeline, the STT-RAM based design can reduce the leakage power and maintain the performance level close to the CMOS design.

Li et al. [58] proposed an approach to minimize refresh operations for volatile STT-RAM, by novel modifications to cache coherence protocol. Jog et al. [36] proposed the *cache revive* scheme where instead of refreshing all the cache blocks, a small buffer is used to temporarily hold cache blocks that have been expired due to the retention time. Afterward, the most recently used cache blocks are copied back into the cache and refreshed.

Sun et al.in [106] proposed an L1 cache and LLC designed with STT-RAM bank with different retention times. The L1 cache is designed with a low retention time along with a refresh scheme called dynamic refresh scheme (DRS) (using counters to track the lifespan of cache data blocks) to maintain the data validity and prevent data loss, while the LLC is designed in a hybrid way with regular and non-volatility relaxed STT-RAM banks to migrate data from one bank to another. Since the refreshing involves multiple read/write operations to guarantee that the data is not evicted, an overhead will limit the impact of reducing the retention time.

To address 0/1 write-asymmetry, Kwon et al. [49] proposed a technique called AWARE, to reduce the write latency of STT-RAM cache by taking advantage of the asymmetric write characteristics. Their idea is based on the fact that if all the cells

Reducing the number of write operations	[124]
Reducing the retention time	[101, 102, 36]
Multi-retention NVMs	[106]

TABLE 3.4: Some important approaches at circuit-level

in a block are preset to 0, then the $1 \rightarrow 0$ transition is not required and the effective write latency can be reduced. The AWARE cache design introduces redundant blocks in each row, and they are preset to the initial state that enables the faster transition. Hence the write operations performed in these redundant blocks are much faster than the conventional write scheme. When a write operation occurs, the redundant block is checked and if it is preset to zero, instead of writing the data to the actual data block, the data is written to the redundant block and the redundant block is marked as the data block and the original data block is now marked as the the redundant block, thus changing the position of data block and the redundant block.

Pan et al.[84] proposed NVSleep, a low-power microprocessor framework that leverages STT-RAM to implement rapid shutdown of cores without loss of execution state. This kind of approaches allow to use idle times to save power by turning off frequently the cores. They presented two implementations of NVSleep: NVSleepMiss which will turn cores off when last level cache misses make the pipeline stalls for a certain time and NVSleepBarrier which will turn off the cores when they are blocked on barriers.

Table 3.3 summarizes the approaches presented here.

Circuit-level approaches

An advantage of using STT-RAM is that its retention time can be traded to improve the write energy and latency. In order to study the trade-offs between the write cost and the MTJ's non volatility, studies have investigated different ways to reduce the data retention time in order to mitigate the high cost write operations. However, sometimes, the retention time is shorter than the time for which data blocks must stay in the cache. Hence, refreshing scheme must take place.

Zhou et al. in [124] proposed a technique called Early Write Termination for reducing write energy with no performance penalty. It is a write process with the capability of early termination in case of a redundant write. It is implemented at the circuit level. Smullen et al. [101] investigated an approach that focuses on technology level to redesign STT-RAM memory cells. They lower the data retention time in STT-RAM, which induces the reduction of the write current on such a memory. This enables in turn to decrease the high dynamic energy and latency of writes.

Smullen et al.[102] proposed an approach to relax the retention time by shrinking the MTJ planar area. The found that redesigning the MTJ by using a smaller *free* layer will help improving the write performance and reduce dynamic energy. They showed how shrinking the cell surface area of the MTJ can reduce Δ, and consequently decreases the retention time (the retention time of an MTJ is primarily impacted by the thermal stability of its free layer, as discussed in Section 3.3.1).

Jog et al.in [36] presented an approach to reduce the retention time of STT-RAM from 10 years to 56 µs by decreasing the thickness of the free layer and lowering the saturation magnetization which reduces the thermal barrier of MTJ.

Table 3.4 summarizes the approaches presented here.

3.4 Empirical energy evaluation for STT-RAM caches with compiler optimizations

```
00: void kernel_syr2k(int N, int M,
01:        double C[N][N], double A[N][M],
02:        double B[N][M])
03: { int i, j, k;
04:   for (i = 0; i < N; i++) {
05:     for (k = 0; k < M; k++) {
06:       for (j = 0; j < N; j++) {
07:         C[i][j] += A[j][k] * B[i][k] +
08:                    B[j][k] * A[i][k];
09:       }
10:     }
11:   }
12: }
```

FIGURE 3.5: Extract of modified syr2k code.

In order to investigate the use of NVMs in caches, we led an empirical study about energy efficiency when integrating Spin Transfer Torque Magnetic Random Access Memory (STT-RAM) technologies [46] in embedded multicore architectures, while applying loop nest optimization [4] to application code. Magnetic memories are considered as promising technologies that combine desirable properties such as good scalability, low leakage, low access time and high density. On the other hand, loop nest optimization has been extensively studied for decades to improve code quality w.r.t. its performance. However, current compilers are not tuned for NVM.

In the context of energy consumption, different loop optimizations have been evaluated individually and collectively, taking into consideration the trade-off between performance and energy with SRAM cache memories [71, 42, 40]. In this work, we focused on two transformations that heavily impact the locality of memory access: loop tiling and loop permutation, and we study their impact on performance and energy consumption in the presence of NVM, compared to a **baseline** scenarios with full SRAM cache memories. The considered architecture scenarios rely on ARM Cortex-A15 cores.

3.4.1 Typical loop optimizations: tiling and permutation

We consider the initial code extract[1] described in Figure 3.5. This code is a typical loop nest that performs a dense linear algebra computation called symmetric rank 2k update. It mainly performs multiplications and additions of the elements from two input matrices denoted by `A` and `B`. The result is stored into the unique output matrix `C`.

Note that this code is a modified version of an original version defined in the Polybench benchmark [90] devoted to numerical computations. The modifications are deliberately not optimized w.r.t. the loop index ordering. This is solved by applying one of the loop transformations considered here.

From the code extract specified in Figure 3.5, we derive two variants by applying loop tiling and permutation to the loop nest defined in the `kernel_syr2k` function. Both loop optimizations are considered in parallel versions of this function as shown in Figure 3.6.

Loop tiling splits the iteration space of a loop into smaller chunks or blocks, in such a way that the data used in the loop remains in the cache until it is reused. This leads to splitting large arrays into fine grain blocks, which in turn enables the accessed array elements to fit into the cache size. For a given loop, finding an optimal tile size is not trivial as this depends on loop accessed array regions and on the cache size of the target machine [48]. Figure 3.6a denotes the optimized parallel (by using OpenMP pragmas) code resulting from loop tiling when applied to `kernel_syr2k` function. They are intended for single-core and multicore architectures respectively.

Loop permutation (or interchange) consists in exchanging the order of different iteration variables within a loop nest. It aims at improving the data locality by accessing the elements of a multidimensional array in an order according to which they are available in cache memory (the layout of arrays in C in row-major). Figure

[1] This code has been obtained from `http://perso.ens-lyon.fr/tomofumi.yuki/ejcp2015`.

3.6b features optimized parallel code corresponding to `kernel_syr2k` function after loop permutation.

To measure the effect of the considered optimizations, the execution time for the baseline version shown in Figure 3.5 must be relevant enough, i.e., not very small. Thus, we did an empirical analysis based on an Odroid-XU4 [26] board which integrates an Exynos-5422 System-on-Chip (SoC) [98]. This SoC contains two clusters of ARM cores, one with four Cortex-A7 and another with four Cortex-A15. For our experiments, we only used the latter. Given $(M, N) = (450, 450)$ as input parameter values, the execution time of `syr2k` is nearly 40 seconds, which is enough for measuring the optimization effects. For the tiling optimization, an exhaustive search yields 9 as the most efficient block size corresponding to the chosen input parameter values.

```
00:  #pragma omp parallel for private(i,j,k)
01:  for (ti = 0; ti < N; ti+=SI) {
02:    for (tk = 0; tk < M; tk+=SK) {
03:      for (tj = 0; tj < N; tj+=SJ) {
04:        for (i = ti; i <ti+SI; i++) {
05:          for (k = tk; k <tk+SK; k++) {
06:            for (j = tj; j <tj+SJ; j++) {
07:              C[i][j] += A[j][k] * B[i][k] +
08:                         B[j][k] * A[i][k];
```

(A) After loop tiling in OpenMP (named `syr2k-omp-ptiled`)

```
00:  #pragma omp parallel for private(i,j,k)
01:  for (i = 0; i < N; i++) {
02:    for (j = 0; j < M; j++) {
03:      for (k = 0; k < N; k++) {
04:        C[i][j] += A[j][k] * B[i][k] +
05:                   B[j][k] * A[i][k];
```

(B) After loop permutation in OpenMP (named `syr2k-omp-perm`)

FIGURE 3.6: Evaluated loop optimization and parallelization.

3.4.2 Experimental setup

This study is conducted with NVSim [20], a circuit-level model for memory performance and area. Beyond the classical SRAM, it supports a large variety of NVMs, including STT-RAM, PCRAM, ReRAM and NAND Flash. Its NVMs models have been validated against industrial prototypes. NVSim achieves within a reasonable time an estimation of electrical features of a complete memory chip, including read-/write access duration in seconds, read/write access energy, and static power. It is

used here to evaluate both SRAM and STT-RAM caches. We set the suitable size, associativity and technology parameters. In order to make the comparison between SRAM and STT-RAM more fair, we identified the relevant architectural parameters that have a significant impact on the efficient utilization of NVMs in a system-on-chip. In particular, we deal with the question about the choice of the suitable operating frequency at which data access should be considered in such a memory so as to minimize its latency penalty. The objective is to compare SRAM to STT-RAM given the same read latency, ideally. The results of the study have shown that the frequency that results in identical read latencies for both memory technologies is 1.0 GHz. For more details, please refer to [87].

Our case study is a 32 KB 4-way associative L1 cache, manufactured with 45 nm technology. We choose this parameter so as to reflect the state-of-the-art for STT-RAM technologies [51]. We therefore compare this cache with a 45 nm SRAM cache. The size and associativity values reflect the actual L1 cache in the Odroid-XU4 mentioned in Section 3.4.1.

3.4.3 Impact of code optimization on performance and energy

Since the first-level cache L1 is accessed very frequently, it should have reasonable latencies and write endurance and that is why it has a small size compared to the last-level cache (LLC) that is designed to reduce off-chip accesses and thus has a larger size. For this reason, many studies have preferred using NVMs for LLC because it is more suitable while L1 is designed using SRAM for performance reasons mainly. Therefore, a pure usage of NVM at L1 cache must be accompanied with low write latency that requires relaxing the retention time of the NVM.

The idea here behind the conducted experiments is to study the behavior of the `syr2k` code versions discussed in Section 3.4.1 while integrating STT-RAM at different cache hierarchy levels, i.e., L1-instruction cache, L1-data cache, and L2-cache. We compare the resulting behaviors with the same versions on a full SRAM cache hierarchy. In this study, we considered four cache hierarchy configurations in terms of memory technologies: SRAM_SRAM_SRAM, SRAM_SRAM_STT-RAM, STT-RAM_SRAM_STT-RAM and STT-RAM_STT-RAM_STT-RAM. They respectively denote a full SRAM cache hierarchy, a cache hierarchy with only L2 cache in STT-RAM, a cache hierarchy with both L1 instruction and L2 caches in STT-RAM, and full STT-RAM cache hierarchy. We refer to SRAM_SRAM_STT-RAM and STT-RAM_SRAM_STT-RAM as hybrid cache hierarchies.

Our experiments have been achieved by using an automated exploration flow, called MAGPIE [19]. It consists of a combination of three existing tools: NVSim, gem5 and McPAT. Odroid XU4 have been modeled in gem5 by Butko et al. [14]. We evaluate the performance and energy trade-off of the target code optimizations through the four cache hierarchy configurations by running programs with the gem5 cycle-approximate simulator [8] in its detailed mode. gem5 allows us to obtain detailed statistics of our simulations, e.g., number of cache accesses, cache misses, cache hits and branch misprediction penalties. The energy consumption is then estimated by using the NVSim [20] and McPAT [66] tools, based on the statistics collected from gem5 simulation. NVSim estimates the energy consumption related to non-volatile components, while McPAT covers the remaining part of the system.

Observations

As shown in Figure 3.7, the programs that take into account code optimizations show better results in terms of performance/energy trade-off. In full SRAM cache hierarchy, which is the **baseline** scenario, we observe in Figure 3.7 that the optimized versions are better as expected than the non-optimized version, in terms of execution time and Energy-to-Solution (or EtoS, i.e., the energy consumed for producing the execution results).

When considering a full STT-RAM cache hierarchy, Figure 3.7 shows the best EtoS. However, there is an important increase in the execution time of `syr2k` program. Again, this is due to the high write operation latency on L1-data and L2 caches. Yet, in the same figure, we observe that this drawback of NVM is eliminated in the optimized programs. This means that the application of efficient optimizations on an NVM-based cache enables better benefit from NVMs.

For hybrid configurations, an overhead is observed in execution time and EtoS, as depicted in Figure 3.7. Regarding the execution time, this is explained by the higher write operation latency in the L2 cache. When integrating STT-RAM in both L1-instruction and L2 caches, the execution time remains the same. This indicates that using a read-only L1-instruction cache in STT-RAM does not introduce additional penalty on execution time, but the energy consumption decreases thanks to the improved leakage power of STT-RAM.

This suggests that applying adequate code optimizations can result in very promising energy-efficient executions on multicore platforms.

This study shows that the difficulty known for finding the optimal set of optimizations for a piece of code on conventional technologies is also true in the presence

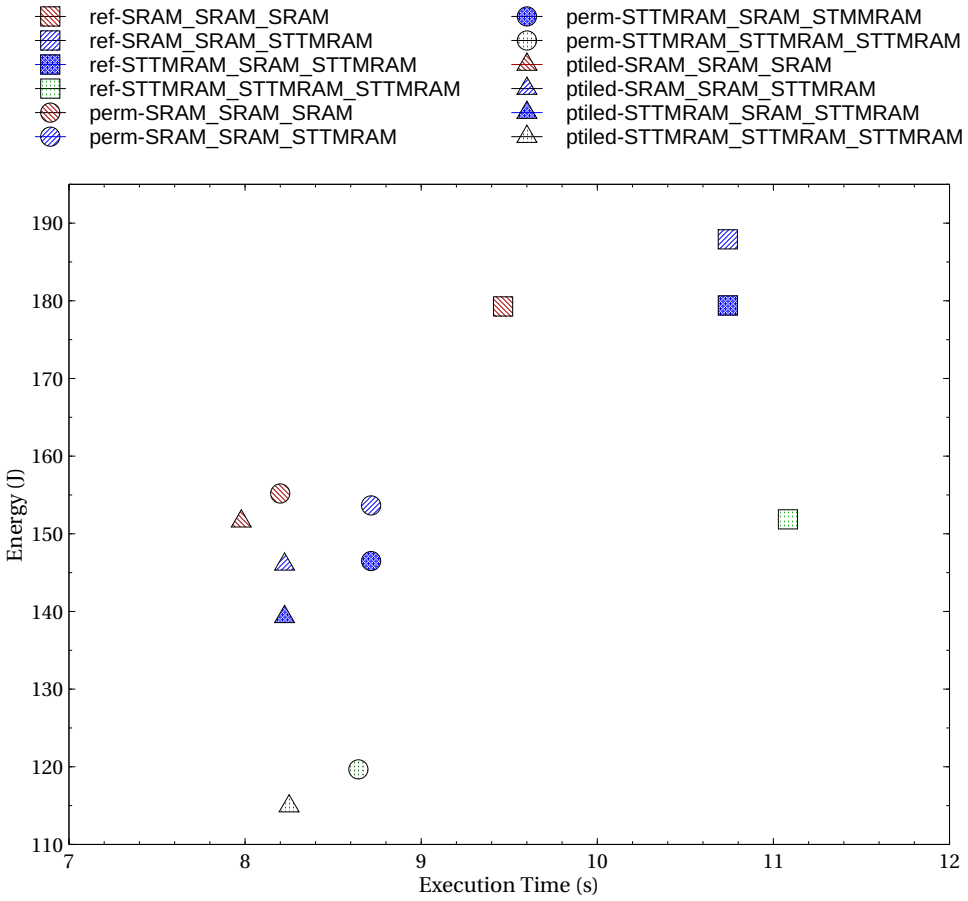


FIGURE 3.7: Performance/energy trade-off with 4 Cortex-A15 cores operating at 1.0 GHz. The reference case, denoted as ref-* is the `syr2k-omp` program. The other programs are optimized version of `syr2k-omp`

of NVM: first, the type of memory adds a new dimension to the search space; second, current compilers have been tuned to optimize for SRAM, not NVM. The produced code is hence sub-optimal in the presence of NVM.

At this stage of the work, we presented an empirical impact analysis about energy-efficiency when executing typical compiler-optimized codes on embedded multicore architectures integrating STT-RAM memory technology. We considered two loop nest optimizations (tiling and permutation) on ARM Cortex-A15 platforms to show performance improvements. At the same time, the significant reduction in leakage power enabled by STT-RAM compared to SRAM, decreases the overall energy consumption. At the same time, the performance observed with SRAM has been reasonably preserved, only 3.4 % less on the studied multicore platform compared to a system with a full SRAM cache hierarchy. This has been achieved through a detailed study on the most favorable core operating frequency. A key insight gathered in this study is that this gain in energy-efficiency depends on the correlation between NVM technologies and their operating frequencies. More details about this work is in [87]

3.5 Why advocating software-level techniques for mitigating the energy-issue on NVMs?

As stated in the previous section, we showed how compiler optimizations have a significant impact on energy efficiency and can help to exploit further more the advantages of NVMs while avoiding their disadvantages. Nevertheless, there are critical issues in replacing traditional memory technologies by NVMs. The approaches existing in the literature have shown a variety of works done at different levels: compiler/software, architectural and circuit levels, to mitigate the shortcomings of NVMs. Although the number of approaches at the architecture-level is significant, the compiler-level and software analysis approaches are more advantageous. As a matter of fact, unlike the architectural/circuit level approaches which require the architecture support, software-analysis and compiler optimizations are portable solutions that can be used on any type of architecture in the presence of the necessary components.

In this work, we focus on reducing the number of write operations by eliminating the redundant ones. Whether it is an inter cache level hybrid cache or intra cache level hybrid cache or a non-hybrid cache, reducing the number of write operations in a program running on an STT-RAM-based cache (or possibly on any NVM-based cache) will reduce the energy consumption. From the literature, compiler-based approaches

were often addressing hybrid caches, while reducing the number of write operations by eliminating them were tackled at the architectural-level and circuit-level.

Reducing the number of write operations by eliminating the redundant ones was addressed in [124] and implemented at the circuit-level (Early Write Termination) for STT-RAM. At the compiler-level, there is no such approach. Hence, this approach is limited by the required architecture support. Therefore, we want to implement it (in LLVM) to benefit from the portable aspect of a compiler optimization.

In another part, as relaxing the data retention time requires refreshing operations, some architecture-level and compiler-level approaches tried to reduce the overhead of refreshing in low retention and multi-retention NVM-based caches. However, from a software point of view, there is no approach addressing the worst-time required to retain a certain data. We want a software-analysis able to compute the worst-lifetime for data that will be written on an NVM. Based on such analysis, which can be used on any type of architectures designed with multi-banks NVMs with different retention, we can allocate a write operation to the most appropriate NVM bank, so that the energy consumed by that write operation is optimized. Although the trade-off between writing on a low-retention NVM with refresh operations and writing on high-retention NVMs is not addressed in this work, we suppose having a multi-retention STT-RAM designed with different retention times and the objective is to allocate the write operations to a specific bank based on its worst-lifetime obtained from the software-analysis.

In summary, both approaches are at the compiler/software level. Hence, this thesis favors a software vision to mitigate the overhead of writes, which is from our point of view more flexible and portable in terms of applicability since there is no hardware requirement.

Chapter 4

Redundant memory write elimination

4.1 Introduction

Memory system plays a very important role in performance and power consumption of computing devices. Previous work already pointed out that the energy related to the cache hierarchy in a system can reach up to 40% of the overall energy budget of corresponding chip [21]. As technology scales down, the leakage power in CMOS technology of the widely used SRAM cache memory increases, which degrades the system energy-efficiency. Existing memory system management techniques offer various ways to reduce the related power consumption. For instance, a technology such as SDRAM has the ability to switch to lower power modes upon a given inactivity threshold is reached. Further approaches, applied to embedded systems, deal with memory organization and optimization [85] [6].

With the increasing concern about energy consumption in both embedded and high-performance systems, emerging non-volatile memory (NVM) technologies, such as Phase-Change RAM (PCRAM), Spin-Transfer Torque RAM (STT-RAM) and Resistive RAM (RRAM), have gained high attention as they open new power saving opportunities [77]. Indeed, their very low leakage power, makes them good candidates for energy-efficiency improvement of computer systems. NVMs have variable performance, energy and endurance properties. For instance, PCRAM and RRAM have limited endurance compared to usual SRAM and DRAM technologies. STT-RAM has an endurance property that is close to that of SRAM, making it an attractive candidate for cache level integration. Nevertheless, a main limitation of NVMs at cache level is their high write latency compared to SRAM. This can be penalizing, especially for write-intensive workloads, as it leads to performance degradation, with a possible increase in global energy consumption despite the low leakage power.

We investigate an effective usage of STT-RAM in cache memory such that its inherent overhead in write latency can be mitigated for better energy-efficiency. For this purpose, we revisit the so-called *silent store elimination* [55], devoted to system performance improvement by eliminating redundant memory writes. We target STT-RAM because it is considered as more mature than similar emerging NVM technologies. Test chips with STT-RAM already exist [33, 81] and show reasonable performance at device level compared to SRAM. More generally, the silent store elimination presented here can be applied to all NVMs for addressing their asymmetric access latencies. It may be less beneficial for technologies with less asymmetric access latencies, e.g., SRAM.

An instance (or occurrence) of a store instruction is said to be silent if it writes to memory the value that is already present at this location, i.e., it does not modify the state of the memory. A given store instruction may be silent on some instances and not silent on others. The *silentness* percentage of a store instruction therefore characterizes the ratio between its silent and non-silent instances: high silentness hence means a larger number of silent store instances.

While silent store elimination has been often developed in hardware, here we rather adopt a software approach through an implementation in the LLVM compiler [50]. A high advantage is that the optimization becomes portable for free, to any execution platform supporting LLVM: a program is optimized once, and run on any execution platform while avoiding silent stores. This is not the case of the hardware-level implementation. Thanks to this flexible implementation, we evaluate the profitability of silent store elimination for NVM integration in cache memory. In particular, we show that energy-efficiency improvements highly depend on the silentness percentage in programs, and on the energy consumption ratio of read-/write operations of NVM technologies. We validate our approach by exploring this tarde-off on the Rodinia benchmark suite [15]. Up to 42 % gain in energy is reported for some applications. The described validation approach is quite fast and relies on an analytic evaluation considering typical NVM parameters extracted from the literature.

4.2 Related Work

There are a number of studies devoted to energy-efficiency of NVM-based caches. We evoked some of them that are implemented at circuit-level and architectural-level, as presented in Section 3.3.2. In addition, we present here some profiling-based approaches and an static analysis approach.

4.2.1 Profiling-based approaches

We promote in this work a *compile-time* optimization by leveraging redundant writes elimination on memory. While this is particularly attractive for NVMs, a few existing works already addressed the more general question about code redundancy for program optimization. In [112], the REDSPY profiler is proposed to pinpoint and quantify redundant operations in program executions. It identifies both temporal and spatial value locality and is able to identify floating-point values that are approximately the same. The data-triggered threads (DTT) programming model [109] offers another approach. Unlike threads in usual parallel programming models, DTT threads are initiated when a memory location is changed. So, computations are executed only when the data associated with their threads are modified. The authors showed that a complex code can exploit DTT to improve its performance. In [110], they proposed a pure software approach of DTT including a specific execution model. The initial implementation required significant hardware support, which prevented applications from taking advantages of the programming model. The authors built a compiler prototype and runtime libraries, which take C/C++ programs annotated with DTT extensions, as inputs. Finally, they proposed a dedicated compiler framework [108] that automatically generates data-triggered threads from C/C++ programs, without requiring any modification to the source code.

While the REDSPY tool and the DTT programming model are worth-mentioning, their adoption in our approach has some limitations: the former is a profiling tool that provides the user with the positions of redundant computations for possible optimization, while the latter requires a specific programming model to benefit from the provided code redundancy elimination (note that the DTT compiler approach built with LLVM 2.9 is no longer available, and is not compatible with the latest versions of LLVM). Here, we target a compile-time optimization in LLVM, i.e., silent store elimination (introduced at hardware level by [55]), which applies independently from any specific programming model. This optimization increases the benefits NVMs as much as possible, by mitigating their drawbacks.

4.2.2 Static Analysis approaches

Pereira et al. [89] proposed an approach to statically predict silent stores. The authors developed a technique that classifies write operations based on syntactic features of programs. Each store operation is associated to a vector of features. Such features are then used to develop different types of predictors, which determine if a store instruction might be silent during its execution.

4.3 Silent-store elimination

In this section, we introduce the definition of silent stores and we present the principle of our implementation. Then, we analyze the factors that help to achieve the best outcomes of such implementation. We focus on micro-architectural and compilation features that are in favor of our approach.

4.3.1 Definition and general principle

Silent stores have been initially proposed and studied by Lepak et al. [54]. They suggested new techniques for aligning cache coherence protocols and microarchitectural store handling techniques to exploit the value locality of stores. Their studies showed that eliminating silent stores helps to improve uniprocessor speedup and reduce multiprocessor data bus traffic. The initial implementation was devised in hardware, and different mechanisms for store squashing have been proposed. Methods devoted to removing silent stores are meant to improve the system performance by reducing the number of write-backs. Bell et al. [5] affirmed that frequently occurring stores are highly likely to be silent. They introduced the notion of critical silent stores and show that all of the avoidable cache write-back can be removed by removing a subset of silent stores that are critical.

In our study, the silent store elimination technique is leveraged at compiler-level for reducing the energy consumption of systems with STT-RAM caches. This favors portability and requires no change to the hardware. We remind that this technique is not dedicated only to STT-RAM but to all NVMs. Here, STT-RAM is considered due to its advanced maturity and performance compared to other NVM technologies. Our approach concretely consists in modifying the code by inserting *silentness* verification before stores that are identified as likely silent. As illustrated in Figure 4.1, the verification includes the following instructions:

1. a load instruction at the address of the store;
2. a comparison instruction, to compare the to-be-written value with the already stored value;
3. a conditional branch instruction to skip the store if needed.

```
1    store @x = val
```

(a) original

```
1    load y = @val
2    cmp val, y
3    bEQ next
4    store @x, val
5  next:
```

(b) transformed

FIGURE 4.1: Silent store elimination: original code stores `val` at address of `x`; transformed code first loads the value at address of `x` and compares it with the value to be written, if equal, the branch instruction skips the store execution.

4.3.2 Implementation

Our compilation process is a middle-end framework. We focused on the compiler intermediate representation (IR) shown in Figure 4.2, where we implemented the silent stores optimization. While this figure describes a generic decomposition into basic steps, its instantiation in our case only contains two LLVM "passes", as illustrated in Figure 4.3.

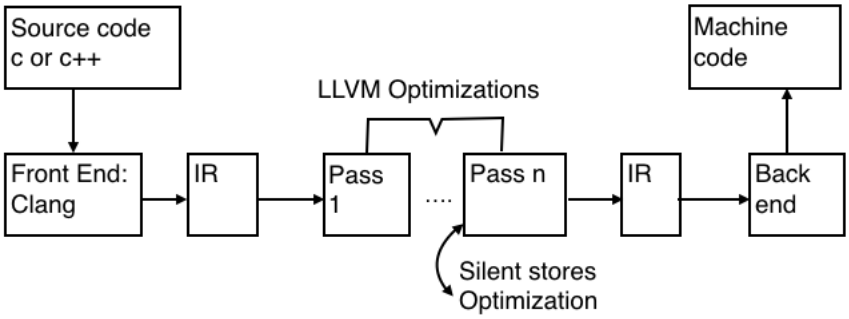


FIGURE 4.2: Implementation of the optimization in LLVM

Through the first step, we get information about all store instructions that are present in a program (note that the current version of the optimization handles stores on integer and floating-point data). For that, we insert new instructions in the IR in front of every store to check whether the stored valued is equal to the already stored value. If it is the case, we increment a counter related to this particular store. Once the first pass is done, we run the application on representative inputs to obtain a summary reporting how many times the stores have been executed and how many times they have been silent. We also save their positions in the program so that we can identify them in the next pass. The output of the first pass is a file that contains all the characteristics of a store as described in Figure 4.3.

In a second step, where we apply the main part of the optimization: we compile again the source code, taking into account the profiling data. For each store instruction whose silentness is greater than a predefined threshold, we insert verification code, as described in Section 4.3.1 and illustrated in Figure 4.1.

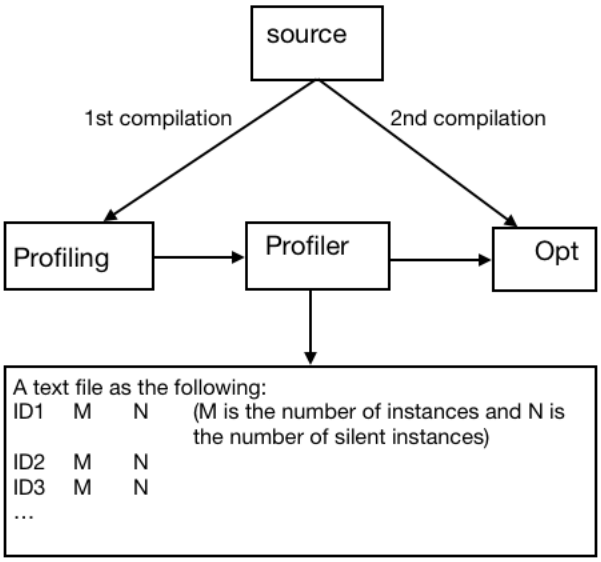


FIGURE 4.3: Design of the compilation framework

By working at the IR level we achieve three goals: 1) the source code of the application remains unmodified; 2) we do not need to be concerned by the details of the target processor and instruction set: the compiler back-end and code generator will select the best instruction patterns to implement the code; and 3) our newly introduced instructions may be optimized by the compiler, depending on the surrounding code.

4.3.3 Micro-architectural and compilation considerations

While reducing the cost of cache access, the new instructions introduced in the above transformation may also incur some performance overhead. Nevertheless, specific (micro)architectural features of the considered execution platforms play an important role in mitigating this penalty.

Superscalar and out-of-order (OoO) execution capabilities are now present in embedded processors. In many cases, despite the availability of hardware resources, such processors are not able to fully exploit the parallelism because of data dependencies, therefore leaving empty *execution slots*, i.e., wasted hardware resources. When the instructions added by our optimization are able to fit in these unexploited slots, they do not degrade the performance. Their execution can be scheduled earlier by the

```
      (original)              (transformed)

add  r1=r2+r3          add  r1=r2+r3 || load  r0=@val
add  r4=r1+2           add  r4=r1+2
mul  r5=r4*r3          mul  r5=r4*r3
add  r6=r5+3           add  r6=r5+3
store @x=r6            sub  r4=r2-r3 || cmp r6, r0
sub  r4=r2-r3          sub  r5=r4-1  || bEQ next
sub  r5=r4-1           store @x=r6
...                    next:
```

FIGURE 4.4: A superscalar execution: the original sequential code is transformed to exploit parallelism by hiding the new instructions added by our transformation

compiler/hardware so as to maximize instruction overlap. The resulting code then executes in the same number of cycles as the original one. This instruction rescheduling is typical in OoO cores. Let us consider the example in Figure 4.4. In the original code, each of the first five instructions depends on its predecessor. Execution is necessarily purely sequential, even on a superscalar processor. Our newly introduced instructions can be executed in the empty slots, as shown on the right hand side of the figure.

Note that the independent subtraction instructions in Figure 4.4 have been scheduled earlier by the compiler in order to maximize instruction overlap. The resulting code executes in the same number of cycles as the original one. Compared to in-order cores, OoO cores enable to execute instructions according to an order, which differs from the one in the original program, as long as instruction dependencies hold. When a high-latency instruction delays the execution of its successors, the processor can select an independent instruction from a window of nearby instructions in order to accelerate the code execution.

Finally, *branch prediction* and *speculative execution* are complementary features that can contribute to improve the performance of cores. At each branch instruction, the processing of instructions is interrupted and the processor must fetch new instructions from a non-contiguous memory location. Due to the pipelined execution, this requires the processor to stall for several cycles, until the target is known. Branch prediction minimizes this penalty by guessing the target of a branch. When correctly predicted, the processor incurs no penalty. If the store's *silentness* is difficult to predict, the added **bEQ** instruction in Figure 4.4 will penalize the application[1]. In this case, it is better to leave the store unchanged.

On the other hand, instruction predication, e.g., supported by ARM cores, is another helpful mechanism. The execution of predicated instructions is controlled

[1]Modern branch predictors, e.g., TAGE [100], can detect complex patterns.

via a condition flag that is set by a predecessor instruction. Whenever the condition is false, the effect of the predicated instructions is simply canceled, i.e., no performance penalty. This is shown in Figure 4.5 where the store instruction is executed only if the preceding compare instruction returns `Not Equal (NE)`, i.e. variables `x` and `y` are not equal. To eliminate silent stores, LLVM automatically generates this code pattern for ARM.

Thanks to the predication mechanism, the obtained code results in one instruction less, thus limiting the risk to increase execution time. As a side effect, it also lowers the number of possible branch mispredictions by eliminating the branch altogether. Note that Thumb extension does not provide predication.

```
load y = @val
cmp x, y
strNE @x, val; executes if cmp returned false
...
```

FIGURE 4.5: Example of predicated execution

(a) original code	**store** @x = val
load y = @x **cmp** val, y **bEQ** next **store** @x, val next:	**load** y = @x **cmp** val, y **strne** @x, val
(b) transformed code	(c) with predication

FIGURE 4.6: Silent store elimination: (a) original code stores `val` at address of `x`; (b) transformed code first loads the value at address of `x` and compares it with the value to be written, if equal, the branch instruction skips the store execution; (c) when the instruction set supports predication, the branch can be avoided and the store made conditional.

At compiler optimization levels, newly introduced instructions may cause two phenomena.

1. Since the silentness-checking code requires an additional register to hold the value to be checked, there is a risk to increase the register pressure beyond the number of available registers. Additional spill-code could negatively impact the performance of the optimized code. We observed that this sometimes happens in benchmarked applications. This is easily mitigated by either assessing the register pressure before applying the silent-store transformation; or by deciding to revert the transformation when the register allocation fails to allocate all values in registers.

2. It can happen that the load we introduce is redundant because there already was a load at the same address before and the compiler can prove the value has not changed in-between. In this favorable case, the added load instruction is automatically eliminated by further optimization, resulting in additional benefits. We observed in our benchmarks that this situation is actually rather frequent.

4.4 Profitability threshold

Since our approach includes a verification phase consisting of an extra load (memory read) and compare. This overhead may be penalizing if the store is not silent often enough. Therefore, we use a pre-optimization process to identify the silent stores, and especially the "promising" silent stores in the light of the profitability threshold. Hence, the compilation framework consists of two steps as follows:

1. silent store profiling, based on memory profiling, collects information on all store operations to identify the silent ones;

2. apply the optimization pass on the stores that are silent often enough.

From the data cache viewpoint (considered in isolation), the silent store optimization transforms a write into a read, possibly followed by a write. The write must occur when the store happens to not be silent. In the most profitable case, we replace a write by a read, which is beneficial due to the asymmetry of STT-RAM. On the contrary, a never-silent store results in write being replaced by a read and a write. Thus, the profitability threshold depends on the actual costs of memory accesses. In terms of energy cost, we want:

$$\alpha_{read} + (1 - P_{silent}) \times \alpha_{write} \leq \alpha_{write}$$

where α_X denotes the cost of operation X and P_{silent} is the probability of this store to be silent. This is equivalent to:

$$P_{silent} \geq \frac{\alpha_{read}}{\alpha_{write}}$$

Relative costs vary significantly depending on the underlying memory technology. Table 4.1 reports some values from literature. In our survey, the ratio in energy consumption between α_{write} and α_{read} varies from 1.02× to 75×.

Source	NVM parameters	Ratio
Wu et al. [115] Li et al. [62], [63]	Technology: 45 nm Read: 0.4 nJ Write: 2.3 nJ	5.75
Li et al. [60]	Technology: 32 nm Read: 174 pJ Write: 316 pJ	1.8
Cheng et al. [18]	Technology: not mentioned High retention Read: 0.083 nJ Write: 0.958 nJ Low retention Read: 0.035 nJ Write: 0.187 nJ	11.5, 5.3
Li et al. [92]	Technology: 45 nm Read: 0.043 nJ Write: 3.21 nJ	75
Jog et al. [36]	Technology: not mentioned Retention time = 10 years Read: 1.035 nJ Write: 1.066 nJ Retention time = 1 s Read: 1.015 nJ Write: 1.036 nJ Retention time = 10 ms Read: 1.002 nJ Write: 1.028 nJ	1.03, 1.02, 1.025
Pan et al. [84]	Technology : 32 nm Read: 0.01 pJ/bit Write: 0.31 pJ/bit	31

TABLE 4.1: Relative energy cost of write/read in literature

4.5 Preliminary evaluation based on a cycle-accurate simulator

In this section, we present an study accomplished with the cycle-accurate simulator Gem5, through MAGPIE framework, with the configurations shown in Table 4.2:

TABLE 4.2: 32 KB SRAM and STT-RAM memory estimation with NVSim

Technology	Latency		Dynamic Energy		Leakage
	Read (ns)	Write (ns)	Read (pJ)	Write (pJ)	(mW)
SRAM	1.31	1.19	24	6	44.70
STT-RAM	1.96	10.94	109	174	6.72

The numbers given in Table 4.2 show a ration $Ratio = 4.5$ which is a low ratio. Nevertheless, the presented results show interesting aspects of this study. This preliminary evaluation aims to identify the potential gain obtained with the silent stores elimination. We first present the application of our optimization on a simple example. Then, we analyze the profitability based on specific characteristics of the used configuration.

4.5.1 Application to a simple example

As a motivational example, we analyze the possible impact of memory on the energy consumed by an on-chip system. For the sake of simplicity, we consider a simple on-chip system model comprising a CPU, an L1 data denoted by L1d, an L1 instruction cache denoted by L1i, a bus and a memory controller. No other cache level (e.g. L2 or L3 cache) is taken into account, while the main memory is off-chip. In order to design and evaluate this model, we use a number of simulation and power modeling tools already integrated in the MAGPIE automated design evaluation framework [19]. These tools include the gem5 cycle-approximate simulator, combined with the NVSim and McPAT power, area and latency modeling tools, which respectively target non volatile and CMOS technologies. Here, the gem5 minor CPU model is used to define an ARM Cortex-A7 core running at 1 GHz, associated with L1i and L1d caches, both 32 KB and 4-way associative.

In order to assess the possible gain that one could expect from different memory technology candidates, we consider a simple program illustrated in Figure 4.7 (C and assembly codes). Since we are focusing on store instructions, we decided to carry out a preliminary experimental analysis on such a dedicated kernel with different execution scenarios. This simple code shows one store operation that is executed N times. Later on, we will identify this store as a silent store.

```
(original C code)                (corresponding assembly)

                                 ; i in r3, N in r4
                                 .L3
volatile int x = 0;                str  r2, [r1, #0]
for (i=0; i<N;i++) {               add r3, r3, #1 ; i++
  x=0; // silent                   cmp r3, r4 ; i==N?
}                                  bNE.L3
```

FIGURE 4.7: Dedicated kernel. The `volatile` keyword forces the compiler to keep the memory access on `x`, while other variables may be promoted to registers. The assignment in the loop repeatedly writes the same value 0 to the same memory location

Let us evaluate the cost of this simple code on a Cortex-A7 in terms of execution time, energy consumption and energy-delay product (EDP). In this work, EDP is considered as the main figure of merit for energy-efficiency as it combines both execution time and energy. By executing 50 million iterations of the simple program (i.e., $N = 50 \times 10^6$), we obtain the result reported in Table 4.3 and referred to as "full-SRAM". In this reference evaluation scenario, both L1d and L1i caches are in SRAM.

Now, by re-executing the same program while considering the L1 cache is entirely in STT-RAM instead of SRAM, we obtain the result displayed in Table 4.3, referred to as "full-STTRAM". Here, we observe a reduction of the energy consumption thanks to the low static power inherent to the used non volatile memory. A very marginal increase is observed in the execution time, which is due to the penalty expected from higher write latency of STT-RAM compared to SRAM. Fortunately, this marginal increase is not important enough to cancel the gain in static power enabled by STT-RAM. As a result the EDP is improved.

While the previous system evaluation motivates the potential benefit of STT-RAM in reducing the overall energy consumption of a system, we can however note a

TABLE 4.3: Evaluations of dedicated kernel: execution of original code with full SRAM L1 cache (ref.); execution of original and optimized codes with full STT-RAM L1 cache.

	Exec. time (ms)	Energy (mJ)	EDP
full-SRAM (ref)	305.13	79.5	24257.83
full-STTRAM	305.31 (+0.06 %)	67.8 (-14.7 %)	20700.01
full-STTRAM + silent store opt	305.31 (+0.06 %)	64.6 (-18.7 %)	19723.02

Benchmarks	Reads		
	Original (N_R)	Optimized	δ_R
kmeans	548 491 642	548 809 242	+317 600
backprop	46 853 587	47 152 381	+298 794
heartwall	1 920 044 615	1 921 988 609	+1 943 994
srad	76 371 774	76 526 801	+155 027
b+tree	213 650 781	213 938 589	+287 808
bfs	181 175 711	180 697 730	-477 981
pathfinder	205 090 257	206 350 923	+1 260 666

TABLE 4.4: Number of loads in Rodinia benchmarks profiled with gem5 before and after optimization (only stores with 60 % silentness probability are transformed).

possible limitation factor that could hamper the expected gains: the high write latency of NVMs. Indeed, as shown above, this can increase system execution time, which in turn increases the resulting energy of the system components, except for L1 cache. On the other hand, as this high write latency also comes with a higher write energy consumption (see Table 4.2), the overall dynamic energy consumption of the L1 cache will be increased as well. This may ultimately cancel the benefit of NVM integration at cache level. In order to reduce this risk, we investigate the opportunity offered by the silent store elimination in order to mitigate the potential negative impact of writes on the system in presence of NVMs. Let us consider again the simple dedicated kernel shown in Figure 4.7, to which the silent store elimination has been applied. Table 4.3 reports the resulting evaluation according to an L1 cache memory in full STT-RAM, referred to as "full-STTRAM + silent store opt". The energy consumption is reduced while the execution time is preserved compared to the original program running with the same memory configuration. Here, the gain mainly comes from the dynamic energy reduction through the removal of silent stores (i.e. redundant writes) and their replacement with reads.

Globally, we note that the increase in execution time observed in Table 4.3 with STT-RAM is quite marginal. This is central here as a huge penalty in execution time would significantly increase the static power consumption of the overall on-chip system, therefore making the optimization less beneficial. In fact, there is a trade-off between a number of system design considerations that should be analyzed in order to make the proposed compiler-level optimization effective when using STT-RAM technology in cache memory.

4.5.2 Optimization profitability w.r.t. number of silent stores

From the point of view of the data cache (considered in isolation), the silent store optimization transforms a write into a read, possibly followed by a write. The write

Benchmarks	Writes		
	Original (N_W)	Optimized	δ_W
kmeans	44 606 552	43 341 904	-1 264 648
backprop	26 354 954	24 257 754	-2 097 200
heartwall	20 198 643	14 983 887	-5 214 756
srad	49 122 503	48 892 569	-229 934
b+tree	24 094 567	23 894 250	-200 317
bfs	112 051 515	111 854 905	-196 610
pathfinder	138 368 817	137 682 697	-686 120

TABLE 4.5: Number of stores in Rodinia benchmarks profiled with gem5 before and after optimization (only stores with 60 % silentness probability are transformed).

must occur when the store happens to not be silent.

As stated in Section 4.4, we want to satisfy the following equation:

$$P_{silent} \geq \frac{\alpha_{read}}{\alpha_{write}}$$

Given the values of Table 4.2, we obtain a threshold of:

$$P_{silent} \geq 109/174 \approx 0.63, \text{i.e., a silentness percentage of } 63\,\%.$$

The above reasoning only considers cache memory accesses, and holds as long as the rest of computations does not heavily impact the system energy consumption compared to those cache memory accesses. We confirm it experimentally by considering the simple program shown in Figure 4.8, where N represents the total number of store operations and M represents the number of silent stores. We run the program with different values of M, so as to vary the silentness percentage. In Figure 4.8, we observe, as expected, that higher probability of silentness reduces energy consumption of the L1 data cache. The silent store elimination is beneficial when the silentness percentage is above 63 %.

```
volatile int x;
for(i=0; i<N; i++)
{
    y=0;
    if (i>M)
        y=i;
    x=y;   // store
}
```

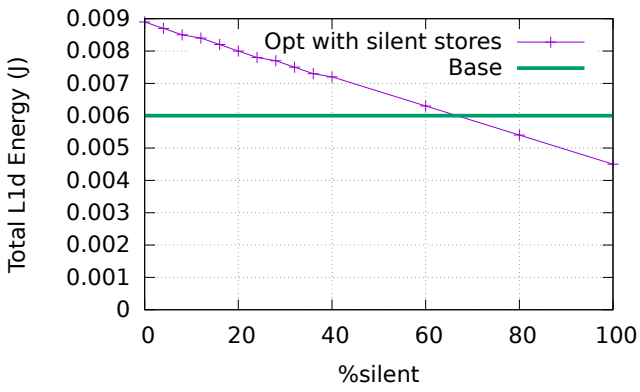


FIGURE 4.8: Determination of the profitability threshold of silent store elimination.

Benchmarks	Energy gain (%)			
	$r = 1$	$r = 5$	$r = 10$	$r = 75$
kmeans	0.16	0.78	1.24	2.43
backprop	2.46	5.70	6.66	7.76
heartwall	0.17	1.19	2.37	11.33
srad	0.06	0.31	0.38	0.45
b+tree	-0.04	0.21	0.38	0.73
bfs	0.23	0.20	0.19	0.18
pathfinder	0.17	0.24	0.35	0.47

TABLE 4.6: Energy gain projection w.r.t. specific NVM technologies.

In their seminal 2001 study [55], Lepak and Lipasti studied the SPEC95 benchmark suite in which high silentness percentages have been exposed. For instance, `vortex` and `m88ksim` reach respectively 64 % and 68 % of silent stores overall on PowerPC architecture (there was no per-store characterization in that study). Such silentness levels could typically benefit from our optimization.

Table 4.4, 4.5 and 4.6 report an evaluation of the suggested optimization on a subset of the Rodinia benchmark suite [15]. Only store operations with 60% silentness probability are transformed here. The table indicates the profiled read and write operations before and after program optimization by considering an execution on a single ARM Cortex-A7 core, using the MAGPIE framework [19]. The expected dynamic energy gain out of all memory access activity is computed, according to NVM technologies associated with different ratios r in terms of read/write energy costs α_{write} and α_{read}, as follows:

$$r = \frac{\alpha_{write}}{\alpha_{read}}$$

These energy costs vary significantly depending on the underlying memory technology. Table 4.1 reports some values from literature. In this survey, the ratio in energy consumption between writes and read varies from 1.02× to 75×. Considering the values computed in Table 4.4 and 4.5, a projection of energy gain can be computed as: $-(r \times \delta_W + \delta_R)/(N_R + r \times N_W)$, where N_R and N_W are respectively the total numbers of reads and writes in an original program; and δ_R and δ_W respectively indicate the number of reads and writes added or reduced after the optimization of a program. It is important to notice that the energy gain projection reported here is only a partial gain as it only results from an optimization of stores with 60 % silentness probability for each program. Indeed, optimizing the stores with less than 60 % silentness probability can improve the energy gain. Programs such as *kmeans*, *heartwall*, *srad* and *pathfinder* contain more such kinds of stores.

On the other hand, looking at δ_R and δ_W, we observe that the applied optimization

improves differently the ratio between the number of reads and writes in considered applications. Beyond the application-specific features, there are also a number of micro-architectural and compilation features that have an impact on the results, as discussed in the next.

4.6 Comprehensive evaluation based on an analytic approach

In this section, we present a similar analysis, uncovering silent-stores in applications, and analytically assessing the impact of our proposal, based on their characteristics. We study some applications from Rodinia benchmark [15], cross-compiled for ARM[2] and we execute them on a single core. Rodinia is composed of applications and kernels from various computing domains such as bioinformatics, image processing, data mining, medical imaging and physics simulation. In addition, it provides simple compute-intensive kernels such as LU decomposition and graph traversal. Rodinia targets performance benchmarking of heterogeneous systems.

4.6.1 Distribution of silent stores

The impact of eliminating a given store depends on two factors: (1) its *silentness*, i.e. how often this particular store is silent when it is executed; and (2) how many times this store is executed (obviously highly silent but infrequently executed store are not of much interest). We first study the distribution of silent-stores across applications, and then we analyze the dynamic impact. A *static* distribution represents silentness through store positions in the code, i.e., store instructions in the assembly code file, while a *dynamic* distribution represents silentness throughout all the store instances occurring during the program execution.

Figure 4.9 represents the cumulative distribution of (static) silent stores in each of our applications. The plots show, for a given silentness (x-axis), what fraction of static stores achieves this level of silentness. When x increases, we are more selective on the degree of silentness, which is why all curves are decreasing. For x=100 %, we select stores that are *always* silent. This is extremely rare because, typically, at least the first instance of a store initializes a piece of data that is not already present. This explains why almost all curves reach the value y=0 when x=100 %. Conversely, when x=0, we select stores that are required to be silent at least 0 % of the time, i.e., all stores: the curves start from 100 % when x=0. The points in the curves identify the silentness of

[2]Here, we choose ARMv7 instruction set architecture (ISA), e.g., supported by Cortex-A7 and Cortex-A15 cores, for illustration purpose. Further ISAs could be straightforwardly targeted as well, e.g., X86. This makes our code optimization portable on different processor architectures.

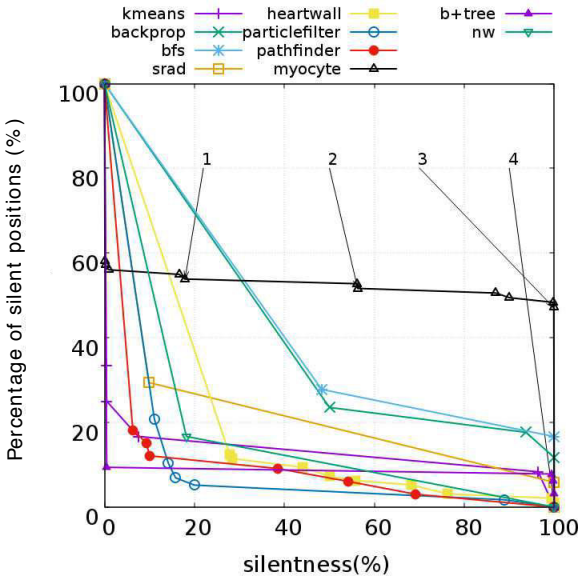


FIGURE 4.9: Percentage of silentness on the basis of the percentage of static positions in the code

the stores in each application. For example, we observe that in the *myocyte* program, there are 53.8%, 51.6%, 47% and 0% of store instructions that are respectively 17.9%, 56.3%, 99.9% and 100% silent (see labels 1, 2, 3 and 4 in Figure 4.9).

In Figure 4.10, we take into account the weight of each store in an execution. Stores executed many times contribute more than rarely executed ones. While the x-axis still denotes the same threshold filter for silentness, the y-axis now represents the fraction of total executed stores that are silent given this threshold. For the *myocite* program, we observe that 58.8%, 56%, 48% and 0% of the silent instances are respectively 17.9%, 56.3%, 99.9% and 100% silent (see labels 1, 2, 3 and 4 in Figure 4.10). Also observe the case of *particlefilter* where the silentness of a number of store instructions can be high, but with a very low impact.

After studying the distribution of silent stores in a program, we can obtain an overview of the optimization benefit. If the heaviness is not significant then the gain will be marginal and even negative. In the next section, we describe how we formulate the gain based on the output of the profiling of each application.

4.6.2 Impact of the silent-store elimination

The impact of the silent-store elimination relies on different factors. As explained in Section 4.6.1, the level of silentness (high/low) and its heaviness are important. Moreover, the relative cost of read/write operation (in Joules) is critical. The ratio between the cost of a read denoted as α_R and the cost of a write denoted as α_W, can

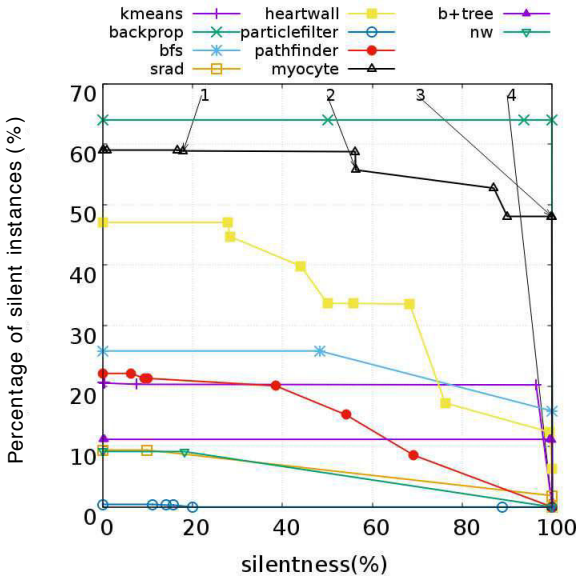


FIGURE 4.10: Percentage of silentness on the basis of the percentage of silent instances in the code

change the direction of the results. Indeed, given a ratio, the optimization outputs may vary from very bad to very good. As mentioned in Table 4.1, different ratios are presented in the literature. Based on that, we did a study to analyze how the impact of silent stores transformation depends drastically on the used ratio. We assume that $\alpha_W = 10$ and we vary α_R from 1 to 10 (as shown below in the formulas, only the ratio matters, hence our choice of synthetic values).

In order to formulate the gain obtained in energy after transformation, we define Δ which is the difference between the energetic cost before optimization and after optimization, denoted respectively as $cost_{base}$ and $cost_{opt}$. In other words, Δ is the expected benefit from the transformation. Since we replace a store with a read and maybe a store if the store is not always silent, then: Δ_i is defined as follows:

$$\Delta_i = cost_{base} - cost_{opt} = \alpha_W - (\alpha_R + \alpha_W \times (1 - P_i))$$

where P_i is the probability of silentness. After transforming all the silent stores, then Δ will be:

$$\Delta = \sum_{i \in \{silent\}} (\alpha_W - (\alpha_R + \alpha_W \times (1 - P_i)))$$

where P_i is the probability of silentness of different transformed stores. In order to get the effective gain, Δ is divided by $cost_{base}$ which is the energetic cost of all read

and write operations before optimization:

$$Gain = \frac{\sum_{i \in \{silent\}} (\alpha_W - (\alpha_R + \alpha_W \times (1 - P_i)))}{\alpha_R \times N_R + \alpha_W \times N_W}$$

where N_R and N_W are respectively the number of load and write operations, obtained from the profiling. Considering $r = \frac{\alpha_W}{\alpha_R}$, then we obtain:

$$Gain = \frac{\sum_{i \in \{silent\}} (P_i - 1/r)}{N_W/r + N_R}$$

In Figure 4.11 and 4.12, we plot this energy gain for each benchmark and for three values of the ratio r: 10, 5, and 1.For each configuration, we plot the gain obtained when optimizing silent-stores whose silentness is greater than a given value (same x-axis as in previous figures).

First, we observe that, without exception, the higher is the ratio, the higher is the gain. In other words, the more asymmetric is the non volatile memory, the more the transformation is beneficial. This is expected, and confirms the validity of our approach.

Second, a ratio $r = 1$ means symmetric memory accesses. For this technological node, our optimization cannot be beneficial. This is confirmed graphically: the gain represented by the blue curve is always negative, reaching 0 only when all stores are 100 % silent.

Generally speaking, the maximum value indicates the best threshold for the silent-store optimization. For a given non volatile memory technology, characterized by the r value, application developers and system designers can plot such curves and identify the best threshold.

The four Rodinia applications reported in Figure 4.11 are those which can benefit the most from silent store elimination given their silentness thresholds and considered r ratios. The remaining applications, displayed in Figure 4.12, only show a marginal benefit. *backprop* and *myocite* can deliver large energy gains up to 42 % when $r = 10$, while *bfs* can reach 16 %, and *pathfinder* 10 %.

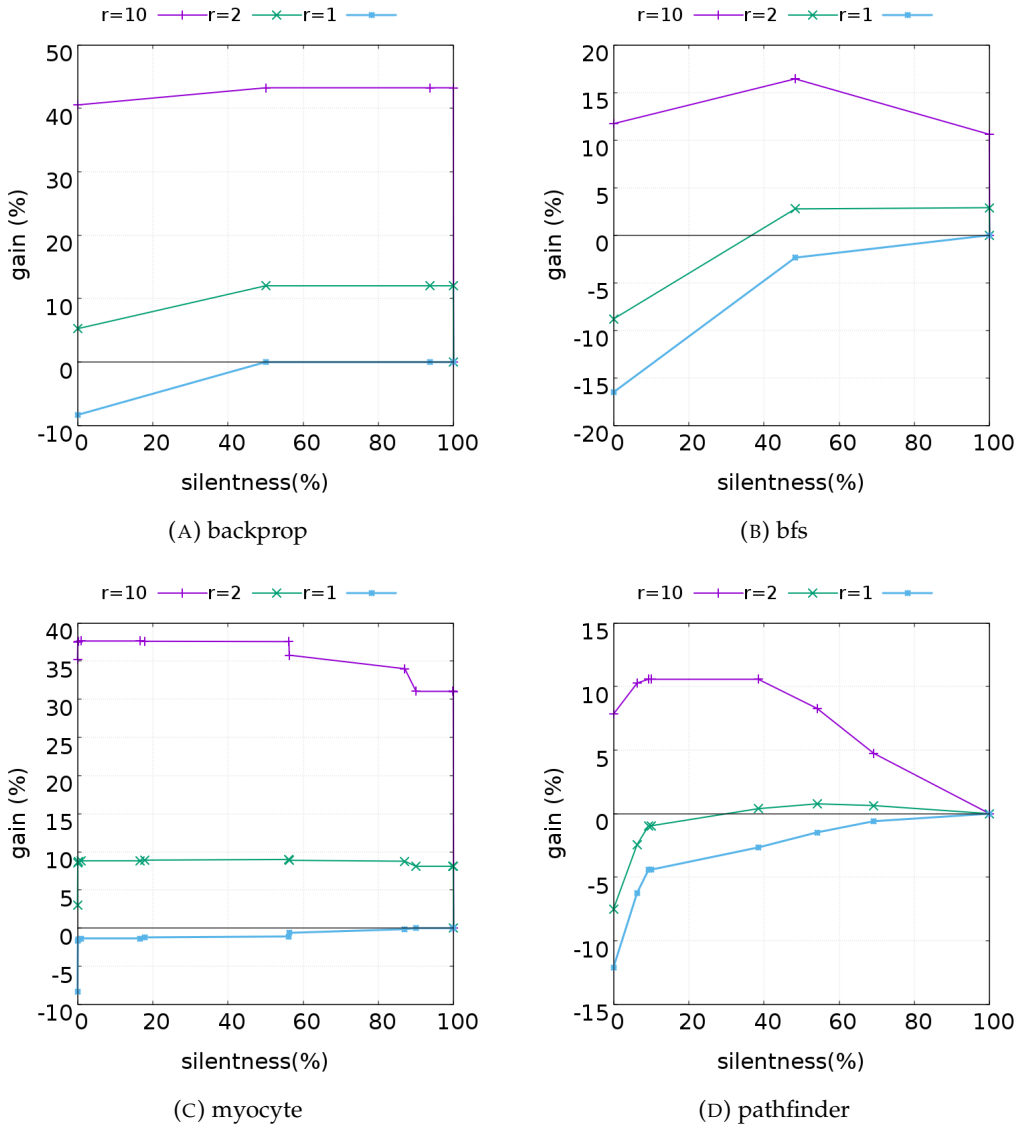
In the intermediate case $r = 2$, the behavior basically depends on applications. The *bfs* program shows a negative gain with low silentness and positive gain with high silentness (from 48 %). The *srad* program shows negative gain for all silentness percentages, while the *myocyte* program shows an interesting gain through all silentness percentages (see Figure 4.11 and 4.12).

Depending on the silentness profile of the application, the gain can be fairly flat, as in *backprop*, *myocite* or *b+tree*, or vary significantly with the silentness threshold, as

in *pathfinder*, or *heartwall*. In the latter case, the energy consumption critically depends on the choice of the threshold.

Finally, curves typically show an ascending then descending phase. This derives from the following phenomenon. Consider the value $x = 1$, i.e. the code is the original not optimized (except for the extremely rare case where a store is silent in exactly 100 % of the cases). When lowering x, we increase the number of store instructions that are optimized, and we increase the gain because we add highly silent stores. But when x keeps decreasing, we start adding stores that may not be silent enough and start causing degradation.

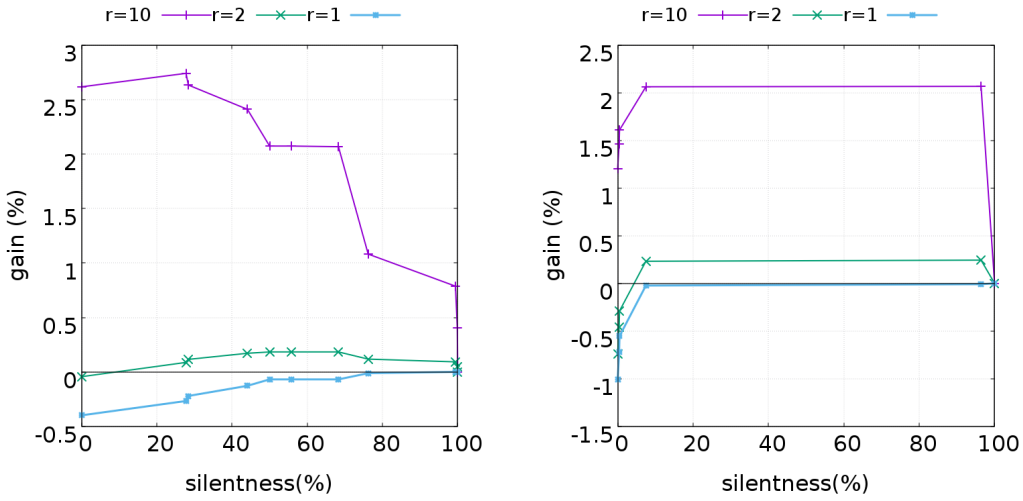
As a final note, remember that loads may be eliminated by compiler optimization. This opportunity is not captured by our above analytical model. It is hence pessimistic, and actual results should be better than our findings.



(A) backprop (B) bfs

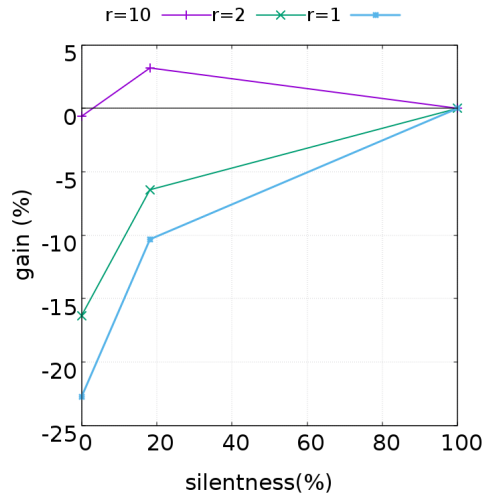
(C) myocyte (D) pathfinder

FIGURE 4.11: Energy gain according to the silentness threshold of Rodinia applications, and their associated $r = \alpha_W/\alpha_R$ ratio: most sensitive applications.

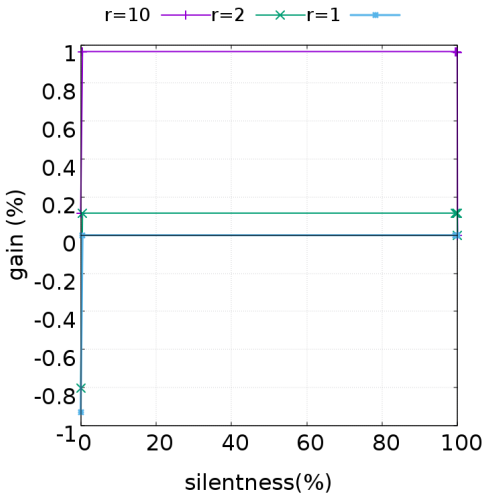


(A) heartwall

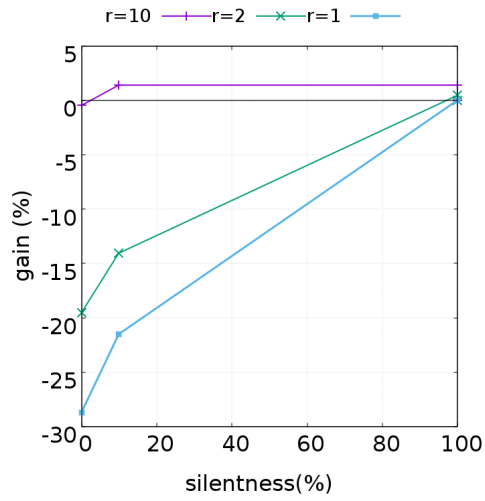
(B) kmeans



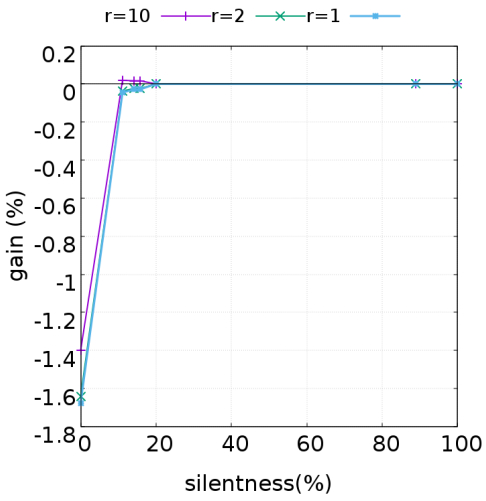
(C) nw



(D) b+tree



(E) srad



(F) particlefilter

FIGURE 4.12: Energy gain according to the silentness threshold of Rodinia applications, and their associated $r = \alpha_W/\alpha_R$ ratio: marginally sensitive applications.

4.7 Towards relaxed silent-stores

Previous sections consider strict silent-stores, where the to-be-written value is exactly the already-stored value. The recent field of approximate computing offers new opportunities [116]. In many cases, floating-points values are not strictly equal, but very close to each other. We claim that, in some cases, it may be beneficial to skip storing a value when it is very close to the previous value.

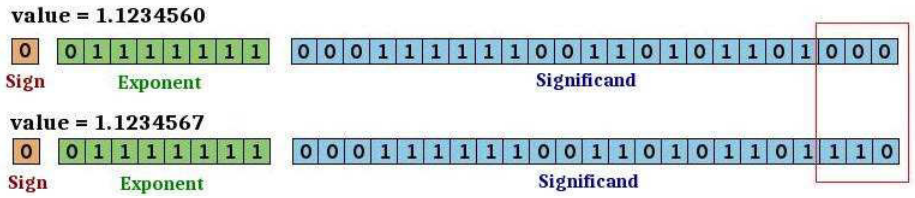


FIGURE 4.13: Binary representation of floating-point numbers as defined by IEEE 754 – 32-bit floats. The mantissa (or significant) is 24-bit long. The two values differ only by their last 3 bits.

This can be easily done as an extension of the previous strict silent-store elimination. A compiler knows the type of the data it processes, and it is straightforward to focus on floating-point values. The transformation code shown in Figure 4.1 is easily extended to loosen the value comparison. The representation of floating-point numbers is precisely defined by the IEEE 754-2008 standard. Let us consider Figure 4.13 for illustrating the representation of the widely used data type `float` (32 bits). With the notations s for the sign bit, m for the mantissa, e for the exponent and bias a constant, these sequences of bits represent the number: $(-1)^s \times 1.m \times 2^{e-bias}$.

Ignoring the last bits of the mantissa is straightforward, as it typically requires only two machine instructions. Let us consider the code in Figure 4.14: an `xor` (exclusive-or) instruction only keeps the bits that differ in `x` and `y`. It is followed by a masking `and` operation that drops the least significant bits (4 bits, in this example). Some instruction sets do not require an explicit `cmp` instructions when comparing to the special value 0, thus reducing the overhead to a single additional instruction.

```
        load y = @val
        xor z = x, y
        and z, 0xfffffff0
        cmp z, 0
        bEQ next
        store @x, val
    next:
```

FIGURE 4.14: Checking for Relaxed Silent-Stores

silentness threshold	Strict	Relaxed	
		4 bits masked	12 bits masked
0	59.03	61.08	61.14
20	58.89	61.07	61.13
56	55.74	61.07	61.12
87	52.72	55.02	55.03
90	48.06	50.35	50.36
99.9	48.00	50.30	50.35

TABLE 4.7: Fraction of silent stores (strict and relaxed) at selected silentness thresholds – Rodinia/*myocyte*)

As a proof of concept of the validity of the approach, we experimented with the Rodinia benchmarks, and we hereby report the case of *myocyte* application. When checking for silentness when storing floating-point values, we tolerate a slight difference between the two values. In practice, we tried ignoring the least significant 4 and 12 bits. In Figure 4.15, we observe that this relaxed approach increases the number of stores to be skipped, which will further reduce the energy penalty related to NVM accesses. For *myocyte*, the number of masked bits has a very marginal impact on the obtained number of candidate stores for optimization (see Table 4.7).

Note that relaxed silent-store elimination could also be applied on variables of integer type. For example, when dealing with image processing, the grey-level can probably be changed by one unit (out of 256) without any sensitivity to human eyes. Integers, however, are more risky because they intrinsically represent discrete values. A programmer annotation may be required to point at candidate variables, but this is not in the scope of the present study.

More generally, it is important to keep in mind that the results presented in this study regarding the silent store elimination are (pessimistic) worst-case scenarios. Actual results are likely to be improved over our current baseline for the following

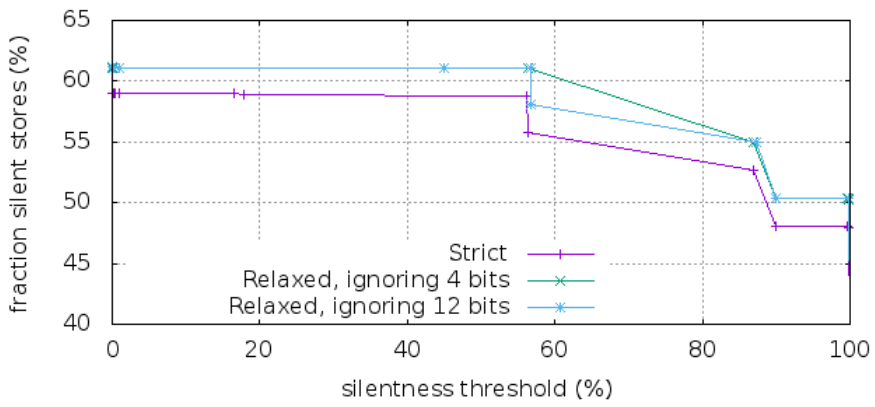


FIGURE 4.15: Strict vs. relaxed silent-stores – Rodinia/*myocyte*

reasons: first, we deliberately set the silentness threshold of 60 % for illustration purpose, which is overly conservative for large values of the ratio r. This misses a further optimization opportunities (e.g., addressing lower silentness thresholds can also contribute in reducing more the global energy); second, the used gem5 simulation considers memory statistics beyond those related only to the user code itself, by including also the C runtime and libraries, while our optimization only targets silent stores in the user code. The "actual" gain should therefore be measured over reads/writes induced by this code (unfortunately this information cannot be extracted from gem5 statistics). Finally, the expected gain in energy is also application-dependent: the more the user code contains silent stores, the better will be the energy savings.

4.8 Conclusion

In this work, we presented an approach for addressing the effective usage of STT-RAM emerging non volatile memory technology in cache memory. We proposed a software implementation of silent store elimination through the LLVM compiler, in order to mitigate the costly write operations on STT-RAM memory when executing programs. A store instruction is said to be silent if it writes to a memory location a value that is already present there. An important property of our approach is its portability to different processor architectures, contrarily to the previous hardware-level approach. We conducted a comprehensive evaluation of our proposal on the Rodinia benchmark. For that, we applied an analytic evaluation of the trade-off between the silentness threshold of stores in a given program and the energy cost ratio of memory accesses. Depending on the silentness of evaluated applications and typical ratios, the gain in energy consumed by memory can reach up to 42 %. While this study mainly targeted the STT-RAM technology (due to its maturity), the proposed silent store elimination applies as well to other NVMs with asymmetric access latencies.

From the conducted analysis, a successful application of our compiler-level optimization requires a number of features for a better outcome. It provides better energy-efficiency results when the cost of newly introduced instructions can be hidden adequately. This requires superscalar capability for exploiting empty instruction slots. We also rely on the compiler to schedule these instructions at the most suitable locations in the original sequence of instructions. This is critical for in-order cores, while OoO cores are able to find an optimal sequence by themselves. Moreover, OoO also generates a denser schedule, with fewer empty slots, which improves execution

performance. The use of predicated instructions is another interesting feature that enables to mitigate the possible execution time overhead related to the extra instructions introduced by our optimization.

Chapter 5

Variable retention times in a multi-bank NVM

5.1 Introduction

Energy-efficiency has become one major challenge in several computing domains, including embedded and high-performance computing domains. Examples of approaches for addressing the issue are system runtime management, heterogeneous multicores and device-level power management.

In the particular case of embedded domain, e.g., the Internet of Things (IoT), devices are increasingly considering non-volatile memories (NVMs) for their ability to favor normally-off computing that aggressively powers off devices. Thanks to their inherent non-volatility property, the information stored within the NVM can be safely recovered when the system is powered on later. During the switch-off period of the system, neither leakage nor refresh power are involved to preserve the information. This property of NVMs makes them highly attractive w.r.t. classical memory technologies such as SRAM or DRAM. In the current study, we consider emerging non-volatile memory technologies [10], such as spin torque transfer random access memory (STT-RAM), phase-change memory (PCM) or resistive random access memory (ReRAM). While these technologies have quasi-null leakage, one major challenge concerns the mitigation of their expensive write operations from both latency and energy points of view. Indeed, depending to the NVM technology, writes are several times more costly compared to SRAM or DRAM.

Non-volatility refers to extremely long retention time. Current non volatile RAMs (NVRAMs) are designed with 10+ years of data retention in order to favor higher memory reliability [101]. Yet, many other NVM technology designs are possible, which enable to build multi-bank memories with variable retention times [106]. This opens an interesting opportunity because memories with shorter retention times have

much cheaper latency and energy costs, and typical variables in programs often have a much shorter lifetime. The next section motivates this opportunity.

5.2 Exploiting variable data retention times

Data retention time is another parameter one can consider when designing a "non-volatile" memory. Standard STT-RAM cells usually target several years data retention period, e.g., 10-years [101] for reliability concern. However, different designs can target shorter retention times as illustrated in Figures 5.1 and 5.2 extracted from literature [106]. Typically, a 32 KB L1 cache with a retention time of 3.24 s leads to half expensive write operations than an L1 cache with 4.27 years retention time, i.e., shorter retention time comes with a gain in access latency and energy. We envision a system with memory banks designed for various energy/retention time trade-offs. Knowing at design-time the amount of time a data is needed would let a compiler allocate data to appropriate memory banks.

As a proof-of-concept, we implemented, through a program profiling software (a *pin tool* running on x86), a functionality for measuring the lifetimes of all data written to memory. Here, the lifetime is defined as the time between a write and the last read to the same location before another write occurs. Figure 5.3 illustrates the distribution of lifetimes in a run of three Rodinia benchmarks: *backprop*, *kmeans* and *myocyte*. It clearly appears that the different lifetimes are not evenly distributed, but grouped rather into clusters. We claim that this property can be leveraged to reduce energy consumption by allocating memory accesses to appropriate banks.

	32KB (L1)						**256KB (L2)**
	SRAM	lo1	lo2	lo3	md	hi	md1
Cell size (F^2)	125	20.7	27.3	40.3	22	23	22
MTJ sw time (ns)	/	2	1.5	1	5	10	5
Retention Time	/	26.5μs			3.24s	4.27yr	3.24s
Read Lat (ns)	1.113	0.778	0.843	0.951	0.792	0.802	2.118
Read Lat (cycles)	3	2	2	2	2	2	5
Write Lat (ns)	1.082	2.359	1.912	1.500	5.370	10.378	6.415
Write Lat (cycles)	3	5	4	4	11	21	13
Read Dyn. Eng (nJ)	0.075	0.031	0.035	0.043	0.032	0.083	0.083
Write Dyn. Eng (nJ)	0.059	0.174	0.187	0.198	0.466	0.958	0.932
Leakage pow (mW)	57.7	1.73	1.98	2.41	1.78	1.82	14.24

FIGURE 5.1: Examples of cache memory configurations comparing SRAM and STT-RAM [106].

	4MB (L2 or L3)					
	SRAM	lo	md1	md2	md3	hi
Cell size (F^2)	125	20.7	22	15.9	14.4	23
MTJ sw time (ns)	/	2	5	10	20	10
Retention Time	/	26.5μs	3.24s			4.27yr
Read Lat (ns)	4.273	2.065	2.118	1.852	1.779	2.158
Read Lat (cycles)	9	5	5	4	4	5
Write Lat (ns)	3.603	3.373	6.415	11.203	21.144	11.447
Write Lat (cycles)	8	7	13	23	43	23
Read Dyn. Eng (nJ)	0.197	0.081	0.083	0.070	0.067	0.085
Write Dyn. Eng (nJ)	0.119	0.347	0.932	1.264	2.103	1.916
Leakage pow (mW)	4107	96.1	104	69.1	61.2	110

FIGURE 5.2: Examples of cache memory configurations comparing SRAM and STT-RAM [106].

Through Figure 5.3, we observe that the maximum lifetimes are 0.16 s, 0.4 s and 0.9 s respectively for *backprop*, *kmeans* and *myocyte* applications. In other words, a 1 s retention time memory bank configuration can meet the non-volatility requirements here. To assess the impact on the energy consumed by the memory hierarchy, we ran the obtained address traces through the Dinero cache simulator[1], configured for the hierarchy shown in Figure 5.2 (using column *md2* for the L3 cache). We collected the number of reads and writes on each cache level and weighted them by their respective energy cost. The results show that the configuration with a retention time of 3.24 s saves approximately half of the energy consumed by the one with a retention time of 4.27 years. More precisely, for *backprop*, *kmeans*, and *myocyte*, the savings are respectively of 44.9 %, 54.3 % and 52.7 %.

For a safe deployment of the above memory system configurations, a refresh memory mechanism should be taken into account in case a data is unexpectedly required beyond the corresponding lifetime estimated at design-time.

5.3 Motivation for δ-WCET Estimates

Real-time systems are composed of tasks that must deliver their results within a well-defined time-frame. Designers of such systems compute an upper bound of the worst-case execution time of the tasks of a system such that any execution of the task takes less time than the estimate (the bound is said to be *safe*). In addition, to be useful, the bound shall be as close as possible to the actual worst-case (the bound is *tight*).

Traditional WCET estimates are computed at the granularity of a function (or task). This is convenient for both computation and exploitation of the results. Functions are well-defined code fragments with a single entry and few exits, a few parameters,

[1] http://pages.cs.wisc.edu/~markhill/DineroIV/

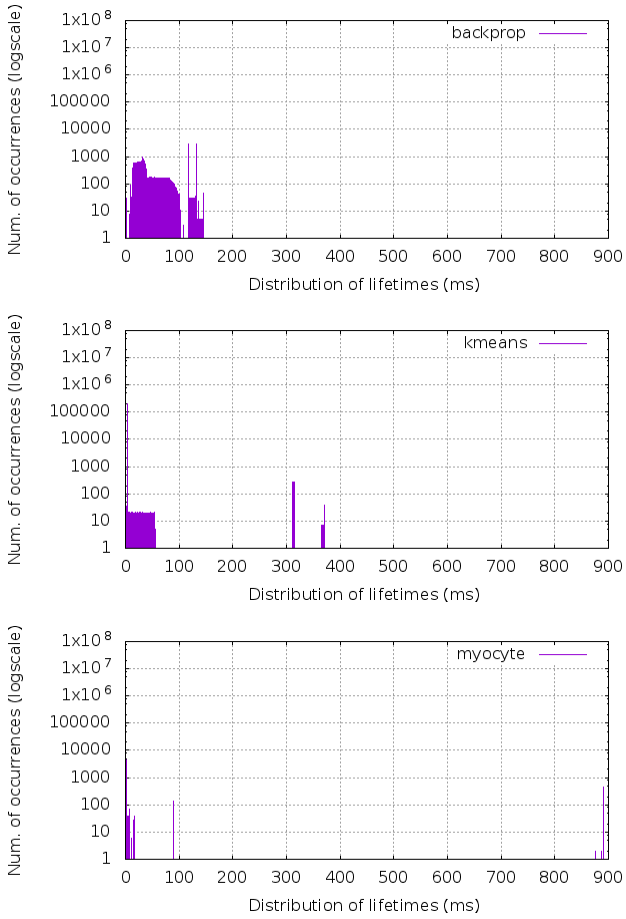


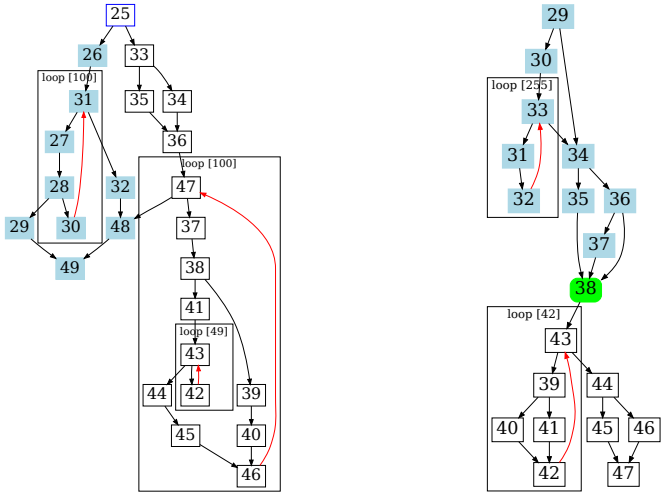
FIGURE 5.3: Data lifetime distributions for *backprop*, *kmeans* and *myocyte*.

and one or no return value. Compilers and analysis tools have developed extensive theory and a wealth of tools based on basic blocks, control flow graphs, and graph theory to deal with functions.

Yet, designing computing systems requires careful attention to a number of aspects such as performance, energy consumption, or security concerns. Computing partial worst case execution time (δ-WCET) estimates proves useful in many contexts as discussed below.

Performance debugging

Many tools exist to measure the actual (average) execution time of regions of interest in code and identify performance bottlenecks. Pinpointing fragments with high WCET is more difficult because the worst-case may occur in rare circumstances and may not be easy to actually expose, hence usual profiling techniques do not apply. δ-WCET is useful for developers to identify potential (worst-case) performance bottlenecks in their applications and focus their effort in the relevant code fragments. In multicore systems, performance can be limited by contention on shared resources



(a) *expint* benchmark (b) *crc* benchmark

FIGURE 5.4: Motivation for δ-WCET on sample control-flow graphs.

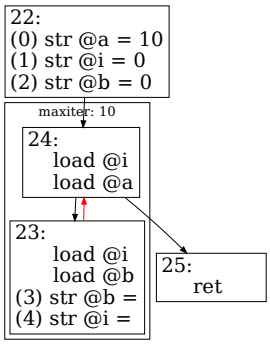
(caches, bus...), and interactions between tasks must be taken into account when estimating WCET. Without precise knowledge about the occurrence of competing events in the different cores, pessimistic values must be considered. δ-WCETs are useful to split a task in finer grain fragments, whose executions can be proven to not overlap. This results in fewer contentions and tighter overall WCET. Execution time also matters for security: some software attacks consist in injecting new code in a target application. A protection may consist in computing the WCET of a task and verifying at runtime that the actual time does not exceed the computed value. Any timing anomaly suggests an intrusion. Hence, if we consider the example of Figure 5.4(a) (taken from the Mälardalen benchmarks [24]) which consists of an unbalanced if-statement: the left branch, starting at block 26, has a δ-WCET of 78 900 cycles and the right branch, starting at block 33, has 629 524 cycles, then bounding the execution time of the function to the latter value would be overly pessimistic when the left branch executes. δ-WCET for each branch is much tighter.

Energy-efficiency

Energy consumption is another major concern driving the design of a computing system. It led to the development of many techniques such as power gating, DVFS, non-volatile memories... Knowing the WCET of the remainder of a computation makes it possible to apply optimizations. Consider Figure 5.4(b), and assume that the execution has reached block 38 in advance with respect to its (pessimistic) bound. This slack time combined with the knowledge of the rest of the computation makes it possible to apply energy saving techniques, such as reducing the clock frequency or

```
#define N 10
int main(void)
{
  int a, b, i;
  a = N;
  b = 0;
  for (i=0; i<a; i++) {
    ANNOT_MAXITER(N);
    b=b+i;
  }
}
```

(a) motivational example



(b) control-flow graph and memory access instructions

FIGURE 5.5: A sample program (a), with its associated Control Flow Graph - CFG (b). The loop iterates ten times. **ANNOT_MAXITER** is a macro that stores this information in a dedicated ELF section of the binary, retrieved by the Heptane tool [27] and attached to the CFG.

switching the task to as less power hungry core (as in Arm's big.LITTLE architecture). In the context of Internet of Things (IoT), many systems do not have any exchangeable battery. They harvest energy from physical phenomena (light, vibration...) into a capacitor, and run as long as there is energy left. For these intermittently powered systems, it is crucial to guarantee that the system stops at predefined locations. In other words, when a fragment starts executing, we must guarantee that execution will reach the next checkpoint where the state can be safely stored. Combining δ-WCET estimates with an energy model of the system is a promising way to ensure that the system makes forward progress and never runs out of power at unwanted locations. Designers also stated incorporating non-volatile memories in their products. On standard STT-RAM, non-volatility refers to a 10-year retention period. However, whenever shorter retention is acceptable, cheaper designs are possible [106]. We envisioned a system with several memory banks designed at various energy/retention points [11]. By computing the WCET between a memory write and its subsequent reads, we can assign writes to the most appropriate bank.

5.4 Worst-Case stores lifetimes

In this section, we present a simple code as a motivational example to show lifetimes variations. Then we present some WCET-related works and we introduce our proposed methodology

5.4.1 Motivational Example

Consider the example of Figure 5.5. The instructions using the three variables a, b and i result in five different memory writes (store instructions) as shown in the control flow graph. The first three initialize the variables in block 22, the last two update the values of i (i++) and b (b=b+i) in block 23. The lifetime of a value is defined as the duration between its *definition* (i.e., a write) and its last *use* (i.e., a load) before it is redefined. The lifetimes created by all the stores are very different. Store (0) in block 22 creates a lifetime that spans the entire duration of the program because it is defined at the beginning and read in each iteration of the loop. Conversely, the lifetimes defined by stores (1) and (2) in block 22 only reach the first iteration of block 23 in which they are redefined. The same holds for the data lifetimes defined at stores (3) and (4) in block 23, alive only across one iteration. Hence, even with a large value of N, most values written to memory only require a short retention time. This is shown in Figure 5.6 (c): only one value increases proportionally to N.

As an example, let us assume a flat memory (i.e., no cache), with the read/write energy costs from Khoshavi et al. [45], also summarized in Table 5.1. Let α_t^R (resp. α_t^W) be the cost a single read (resp. write) to a memory bank with retention time t, the cost of executing the program with a 10-years retention NVM memory is:

$$E_0 = N \times (4\alpha_{10yr}^R + 2\alpha_{10yr}^W) + 3\alpha_{10yr}^W$$

If a 10 ms retention memory bank is available, stores (1), (2), (3), and (4) can be assigned to it. For large values of N, store (0) must be assigned to the longer bank (for $N \geq 66655$, the lifetime exceed 400,002 cycles, i.e. 10 ms at 40 MHz as in our

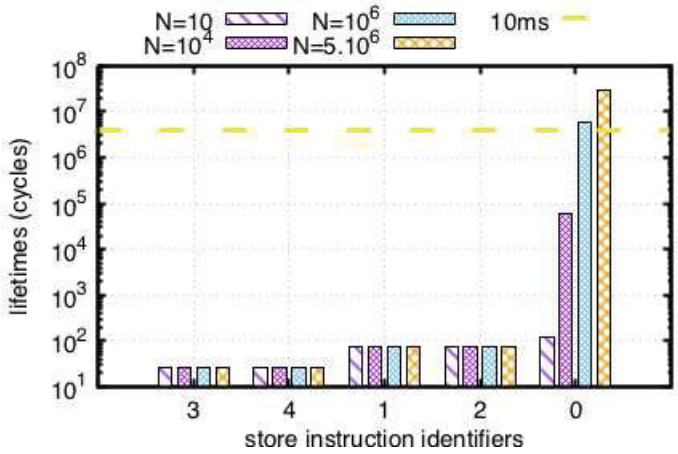


FIGURE 5.6: Duration of the lifetimes created by each store instruction, for different numbers of iterations of the loop, and 10 ms threshold, for the example shown in Figure 5.5.

Retention time	Read energy (nJ): α^R	Write energy (nJ): α^W
10 years	0.233	0.601
10 ms	0.233	0.269

TABLE 5.1: Two 512 KB NVM memory retention times [45]

experimental setup). The energy cost therefore becomes:

$$E_1 = N \times (3\alpha^R_{10ms} + \alpha^R_{10yr} + 2\alpha^W_{10ms}) + \alpha^W_{10yr} + 2\alpha^W_{10ms}$$

The energy gain is $\frac{E_0 - E_1}{E_0} \simeq 31\,\%$.

The above simple example motivates the potential energy gain from the design compromise enabled by NVM technologies with variable retention time. This feature can be leveraged through multi-bank memory systems. Mapping variables to the most appropriate memory bank will result in significant reduction of overall memory energy. However the mapping must guarantee that the lifetimes of data stored in memory will not exceed the retention time of their allocated memory bank, in any circumstances.

In this work, we exploit static analysis and worst-case execution time techniques to estimate an upper bound of the lifetime of every store instruction in programs. Given, this information, we map each store to the most appropriate memory bank according to the most suitable NVM retention time available. Our results show that significant energy reduction can be obtained with only a few banks, favoring energy-efficient memory designs.

We summarize our contribution as follows:

- an extension of program WCET analysis with the possibility to compute the partial worst-case execution time (δ-WCET) of any portion of a program,
- exploitation of the δ-WCET analysis to compute the worst-case lifetime for variables defined in store instructions of a program,
- multi-bank NVM memory allocation of variables based on their worst-case lifetime characterizations and memory data retention times,
- validation of the proposed approach on the Mälardalen [24] benchmark-suite (composed of 18 workloads) to show up to 80 % (and 66 % on average) dynamic energy reduction on memory, while considering design calibration parameters from NVM literature.

5.4.2 Related work

We first review existing studies devoted to energy-efficiency of NVM-based systems. Then, we discuss the WCET estimation approach in general.

WCET estimation techniques

WCET estimation of programs provides an upper bound of task execution time, used for guaranteeing that real-time requirements of a system are met. Traditionally, WCETs are estimated at the granularity of a function. Many tools of WCET estimation are proposed in the literature. Wilhelm et al. [113] presented an overview of methods and existing tools. Two classes of methods are distinguished: static methods and measurement-based methods. Static methods do not rely on real hardware executions. They analyze the code itself, combine the control flow graph with a model of the hardware architecture, and produce an upper bound of this combination. On the other hand, measurement-based methods execute the code on real hardware or a simulator for certain inputs. Then, based on the measured times, the minimal and maximal execution times are derived.

The notion of *partial* WCET (δ-WCET) considered in the current work is recent to the field. It has been recently addressed by Jacobs et al. [34], where they focus on interference of concurrent tasks sharing a bus, and they compute for how many cycles concurrent cores may be granted access to the resource in any time interval of a given length. Avila et al. [3] also used the concept of δ-WCET, associated with the gain time. The difference between the estimated WCET and the actual execution time is known as *gain time*. Early identification of gain time requires to obtain δ-WCETs of the code instead of considering the code as a whole. The authors placed gain points in the program where they measured the actual execution time. These measurements are used to identify all sources of pessimism in the WCET analysis. Oehlert et al. [82] presented a compiler-based extraction of event arrival curves using CFGs. In order to determine the maximum/minimum number of events in a specific time interval, all possible paths in this CFG have to be considered. Their approach is an extension of the work presented by Jacobs et al. [34]. Then, they suggested an alternative idea where the objective function is set to maximize the number of events on a sub-path to be chosen in a CFG. Our work differs from this idea in that it rather aims to maximize the WCET in a given sub-path of the CFG. Ozaktas et al. [83] introduced two approaches that play part in tightening the computation of WCET of a parallel program. Stall times have an impact on WCETs and contribute to making it more pessimistic. Parallel threads face synchronization-related delays, which challenges the timing analysis.

Hence, the authors proposed a refined analysis of the stall times in sequences of critical sections (known to be a performance bottleneck in parallel programs) and a new strategy for lock granting that is implemented in a set of primitives and aims to minimize stall times on the worst-case path. Therefore, the authors presented a partial WCET estimation of a sequence of the code that is an critical section. Thus, it is not a general implementation that can be applied from any point and to any point in a program. It is mainly an implementation dedicated for parallel programs.

There are various static WCET tools that are available. AbsInt's aiT tool[2] and Bound-T [28] are probably the most famous and commercially successful ones. However, in this work, we consider Heptane, which is a tool implemented by some of our team's members.

Background: the Heptane Tool

We consider Heptane [27], a static WCET estimation tool. The aim of Heptane is to produce upper bounds of the execution times of applications. It targets applications with hard real-time requirements (automotive, railway, aerospace domains). It computes WCETs using static analysis at the binary code level. It is divided in two parts: HeptaneExtract and HeptaneAnalysis. HeptaneExtract generates the control flow graph G from a program compiled from C language. Then, it identifies the different loops, attaches the loop bounds information provided by the programmer and attaches the instruction addresses based on the binary file. Heptane does not include the analysis of maximum number of loops iterations, which are not always statically computable in the general case. Thus, loops must be annotated by the user with their maximum number of iterations (*maxiter*)[3]. Afterwards, HeptaneAnalysis implements IPET (Implicit Path Enumeration Technique) along with cache analysis techniques for several cache architectures [113]. Static WCET estimation methods are divided into two steps: *high-level* analysis and *low-level* analysis (see Figure 5.7). The *high-level* analysis consists of determining the longest execution path. The *low-level* analysis takes into consideration the micro architecture.

For the *high-level* analysis, Heptane performs an IPET analysis, based on Integer Linear Programming (ILP) formulation of the WCET estimation problem. The program flow is mapped into a set of graph flow constraints. An upper bound of the program's WCET is then obtained by maximizing the following objective function: $max \sum_i n_i \times w_i$ where w_i is the timing information of the basic block i (constant in

[2] https://www.absint.com/ait/

[3] External tools such as oRange [74] are able to provide loops upper bounds of C programs in some cases.

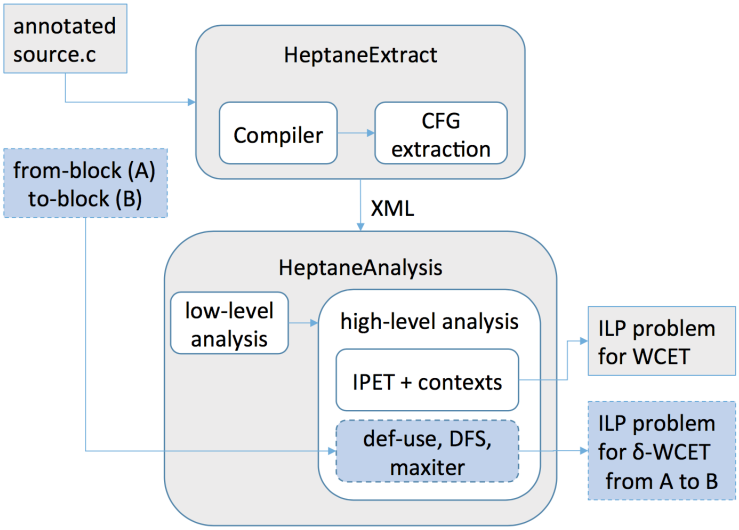


FIGURE 5.7: Heptane tool extended with δ-WCET analysis (blue dashed-box components added by current work).

the ILP problem) determined by the low-level analysis, and n_i is the number of times the basic block i is executed (variable in the ILP problem). A basic block is a set of sequential instructions. For the *low-level* analysis, Heptane performs data address analysis, cache analysis and pipeline analysis. The pipeline analysis for all supported architectures currently considers a simple in-order pipeline. Note that Heptane performs context-sensitive analysis, which means that every call path of a function is analyzed separately, e.g $main \rightarrow foo \rightarrow bar$; and $main \rightarrow bar$, where bar is called directly from main but also from foo. Therefore, the objective function will be like the following: $max \sum_i n_i\_c \times w_i$, where w_i is the timing information of the basic block i in the context c and n_i_c is the number of times the basic block i is executed in the context c. The ILP problem is solved by a solver such as *lp_solve* or *cplex* by maximizing the objective function and identifying the paths that lead to the estimated WCET.

The WCET estimation depends on the structure of the program's CFG and the number and type of instructions inside each block, but also on the capability of Heptane to apply address and cache analysis to bound the number of cache misses.

5.4.3 Proposed Methodology

Our methodology is a two-step process, illustrated on Figure 5.8. First, we identify the *def-use* chains in the program (step referred to as "Reaching loads"). In other words, for each store instruction, we determine all the loads that can read the value previously written. Second, we compute the worst-case execution time between the store and all subsequent loads (step referred to as "δ-WCET"). For this purpose we

Algorithm 1 General δ-WCET algorithm

procedure BETWEENBLOCKS(A,B)
  $L = DFS(A, B)$ ▷ nodes in DFS stack at completion
  for all nodes N visited in the DFS and $N \notin L$ **do**
    $L = L \cup DFS(N, B)$
  return L ▷ the list of nodes encountered in possible paths
procedure GENERATION OF THE ILP PROBLEM FROM A TO B
  $L = BetweenBlocks(A, B)$
  $max \sum_i n_i\_c \times w_i$ where $i \in L$ ▷ the rest of the basic blocks are excluded by setting their w_i to 0
  for all calls in L **do**
    add to the objective function the callee nodes with their callee context
  Modify w_i based on whether A and/or B are inside a loop or not ▷ Maxiter analysis

have developed a method to compute partial WCET estimates which we present in this section. We then present how we apply it to the case of worst-case lifetimes. Both steps are entirely static analyses.

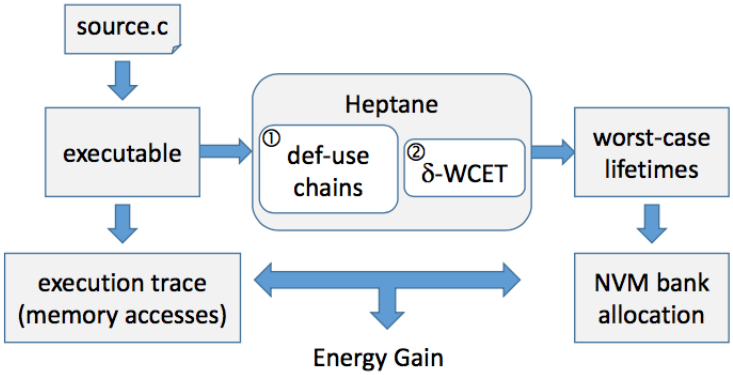


FIGURE 5.8: Sketch of our framework (input: C code).

δ-WCET Estimation

A program is given as a regular executable (both ARM and MIPS instruction sets are supported), in binary format, and an entry point (function `main`). Considering two basic blocks A and B, we are interested in estimating the WCET from block A to block B, which is the δ-WCET from A to B (item (2) of Figure 5.8).

The δ-WCET estimation is based on the WCET estimation. Considering a subgraph of the CFG, the objective will be to make Heptane derive the δ-WCET from its analysis for the whole CFG. Thus, we first compute the WCET estimate for the entire program, which consists in computing for each basic block its WCET estimate, and its worst-case execution frequency. Through this step, we obtain the system constraints of

the original ILP that we want to use later for δ-WCET estimations, as well as the contextual analysis. Secondly, given two blocks A and B, we compute the set of blocks and edges that can be traversed in any path from A to B. Consider the example on Figure 5.6, where we selected block A=22 and B=25. All blocks 22, 23, 24, and 25 must be considered, which means that all the blocks in the different paths leading to B must be considered. We achieve this by iterating a depth-first search (DFS) on the CFG, starting from node A, until we reach B or a block that reaches B. Note that the WCET path from A to B is not necessarily on the overall WCET, i.e, the path from A to B may not be a part of the longest path in the overall program, thus, the WCET from A to B is not a sub-WCET of the program's WCET.

Then, we compose a new ILP problem for the subgraph G' obtained from the DFS, to compute the WCET from A to B (see Algorithm 1 for the details). Therefore, the objective function will include the nodes in G' along with the callee nodes (with the callee context) if there is a function call. Moreover, in order to tighten the WCET (make it less pessimistic), we analyze the *maxiter* annotations. The potential for improvement comes from the fact that the user annotation applies to the execution of the entire function, while we consider only a subgraph. We take into account the following cases, as illustrated in Figure 5.9.

1. The backedge is part of the subgraph (see example in Figure 5.9 (a)), hence the loop may execute its maximum number of iterations on a path from A to B. This is the case of the store instruction with ID 0 in Figure 5.6 (b). We keep the value of *maxiter* unmodified.

2. The backedge is not part of subgraph (see example in Figure 5.9 (b)), as examplified by the store with ID=1 in Figure 5.6 (b): the lifetime created in block 22 (initialization of variable i) is killed in block 23 by store ID=4 (i++). We set maxiter=0 so that the ILP formulation for the subgraph does not consider pessimistic frequencies due to the loop structure.

3. The backedge is part of subgraph, but it does not contribute to any cycle (see example in Figure 5.9 (c)). Consider for example the lifetime created by store with ID=3 (storing the result of b=b+i). Hence, we set maxiter=1.

So, the new ILP problem, presented as: $max \sum_i n_i\_c \times w_i$ where $n_i \in BetweenBlock(A, B)$, has a new system constraints, slightly different from the original one to consider the above cases. Note that the maxiter modification is applied to all backedges that are not part of the subgraph. The whole implementation of the δ-WCET algorithm has been done inside Heptane, as shown in Figure 5.7. Given a start node A and an end

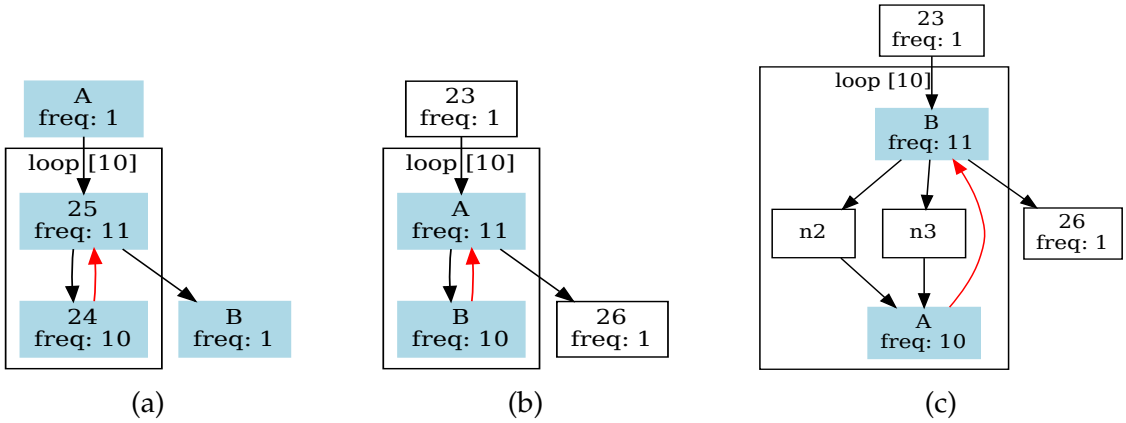


FIGURE 5.9: Handling of maxiter

node B, the high-level analysis of Heptane performs IPET analysis along with the contextual analysis to estimate WCET and then performs the depth-first search and creates a new ILP problem using the contextual analysis related to the output of the DFS and the maxiter analysis. The contextual analysis here consists of including in the ILP objective function, the callee nodes with their callee context if there is any function call in the DFS output nodes.

Linking δ-WCET to NVM Allocation

Data retention time is another parameter one can consider when designing a "non-volatile" memory. Standard STT-RAM cells usually target several years data retention period, e.g., 10-years [101] for reliability concern. However, different designs can target shorter retention times, coming with gains in memory access latency and energy as shown in Table 5.1.

We envision a system with memory banks designed for various energy/retention time trade-offs. Knowing at design-time the amount of time a data is needed would let a compiler allocate data to appropriate memory banks.

Assigning a write with a certain lifetime to the appropriate memory bank will help us reduce the energy consumption by avoiding expensive write energy for writes with low lifetimes. Identifying the subsequent reads for each write is done through a dataflow analysis called *reaching definitions* which statically determines which *definitions* may reach a given point in the code (item (2) of Figure 5.8).

Reaching definitions are used to compute *use-def* chains and *def-use* chains:

- *use-def* chain: consists of a use, U, of a variable, and all the *definitions*, D, of that variable that can reach that *use* without any other intervening definitions

- *def-use* chain: consists of a *definition*, D, of a variable and all the *uses*, U, reachable from that *definition* without any other intervening *definition*.

We use the *def-use* chain where a *definition d* is a store and the reachable *uses u* are the loads. A *definition d1* reaches point *u1* if there is a path from *d1* to *u1* such that *d1* is not killed along that path. In Figure 5.10, *d0* is a *definition* that is never killed and its *use* is *u2*, which explains why its lifetime is the highest one. However, *d1* is killed by *d4*. The *uses* of *d1* are *u1* and *u3*; hence, after *d4*, *d1* is no longer available.

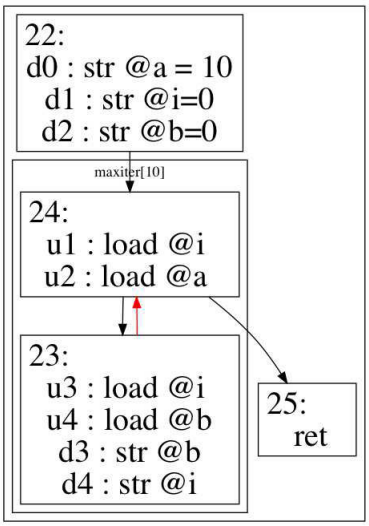


FIGURE 5.10: Example of reaching definitions

We implemented the computation of *def-use* chains in Heptane in order to compute for every write instruction S the subsequent load instructions $L1$, $L2$..., Ln and to define the subgraphs. Once this step is performed in Heptane, we create an ILP problem for every combination $S, \{Li\}$ in order to estimate the δ-WCET p_i from S to Li as shown in the Algorithm 1. Then, the lifetime of the store S is the maximum value of p_i. We repeat this step to compute the lifetimes of all stores in the given program.

Regarding the concrete implementation of variables mapping to memory banks, both software and hardware approaches could be envisioned, as applied previously for scratchpad memories [32]. For instance, a simple solution consists in extending the instruction set with specialized load and store instructions, one for each memory bank (alternatively, the load/store could be extended with an extra parameter to specify the bank). Given the worst-case lifetimes, it is a fairly simple compiler job – or even post-processor on the assembly – to optimize each instruction. It is important that the code size does not change in order to maintain the validity of the analysis carried out by Heptane. The implementation of the above mapping solution belongs to short-term perspectives of the current work.

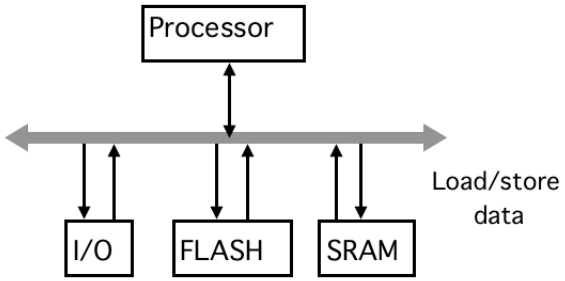


FIGURE 5.11: MCU containing a processor, SRAM, embedded Flash, and programmable I/O peripherals

5.5 Validation on Mälardalen Benchmark-suite

We first present the experimental setup, chosen to evaluate our approach. Then, we discuss the a potential type of architecture where our proposed methodology can provide prominent results. After that, we detail our gain evaluation based on the computed lifetimes.

5.5.1 Experimental setup

We experimented with the Mälardalen Benchmarks, a typical suite for WCET-related experiments. Benchmarks were compiled for ARM using GCC. We slightly modified the code to increase execution times which are otherwise too small: all lifetimes would be far below our lowest threshold, making gains artificially high. As customary with real-time systems, no optimization is applied (optimization level `-O0`). The reason for this is the need to keep source-level annotations consistent with the binary representation. Compiler optimizations heavily restructure the program representation, to the point that the CFG representation at binary level cannot be matched with the source level, making annotations invalid[4]. The ILP problems are solved by *cplex*. To assess the impact on the energy consumed by the memory, we need the number of reads and writes to all memory locations. Thus, we instrumented the benchmarks using *DynamoRIO*, a runtime code manipulation system that supports code transformations on any part of a program, while it executes [5], to obtain traces of execution. Through the traces, we count the number of occurrences for each store as well as for its subsequent loads, to estimate the energy. The overall setup is shown on Figure 5.8.

5.5.2 Architectural setup for IoT domain

In Internet of Things (IoT) systems, connected devices are able to sense and collect data from the environment. These devices are typically battery powered. To maintain a long period of autonomy, they must be able to manage their energy as long as possible. Therefore, the energy consumption is a critical constraint in the design of an IoT device. An IoT device is most of the time inactive, switching to low-power mode (sleep mode) and waiting for the next task. Consequently, sleep mode power will consumes an important part of energy and battery life, as the transition to the active mode will required additional energy to exit the sleep mode and become fully operational. Several Microcontrollers (MCUs) targeting low-power applications implement several power-down modes with different transition times, depending on from which low-power mode the MCU returns to the active mode. The most popular choice of NVM in an MCU is an embedded-flash memory which can be used for both code-storage and data storage applications. However, as for all types of NVM, characterized by their density and low leakage, write operations are much expensive than read operations. By relaxing its retention time, we can reduce the energy cost of a write operation. Partitioning a flash memory into different blocks where each set of blocks has a retention time can help to save an important amount of the energy consumption.

In this work, we target an IoT architecture (see Figure 5.11), with a frequency of 40 MHz, where the read and write latency is 1 cycle and where the flash memory has a high retention time (up to 10 years). We mainly consider the memory sub-system, as in low-power designs, the core consumption of the core is extremely low. As an example, the Cortus APS25s+ core [99] is rated at 17.9 µW/MHz, i.e. 716 µW at 40 MHz. The memory systems we consider consume 2 nJ (resp. 1 nJ) per write. Assuming a write every 10 instructions at 40 MHz, the power would be $(40.10^6 \times 2.10^{-9})/10 = 8.10^{-3}W$, i.e. 8 mW (resp. 4 mW), an order of magnitude larger than the core.

[4] Li et al. [57] have successfully traced annotations throughout compiler optimizations, but this requires heavy compiler machinery and it is not the focus of this work.

[5] http://www.dynamorio.org/

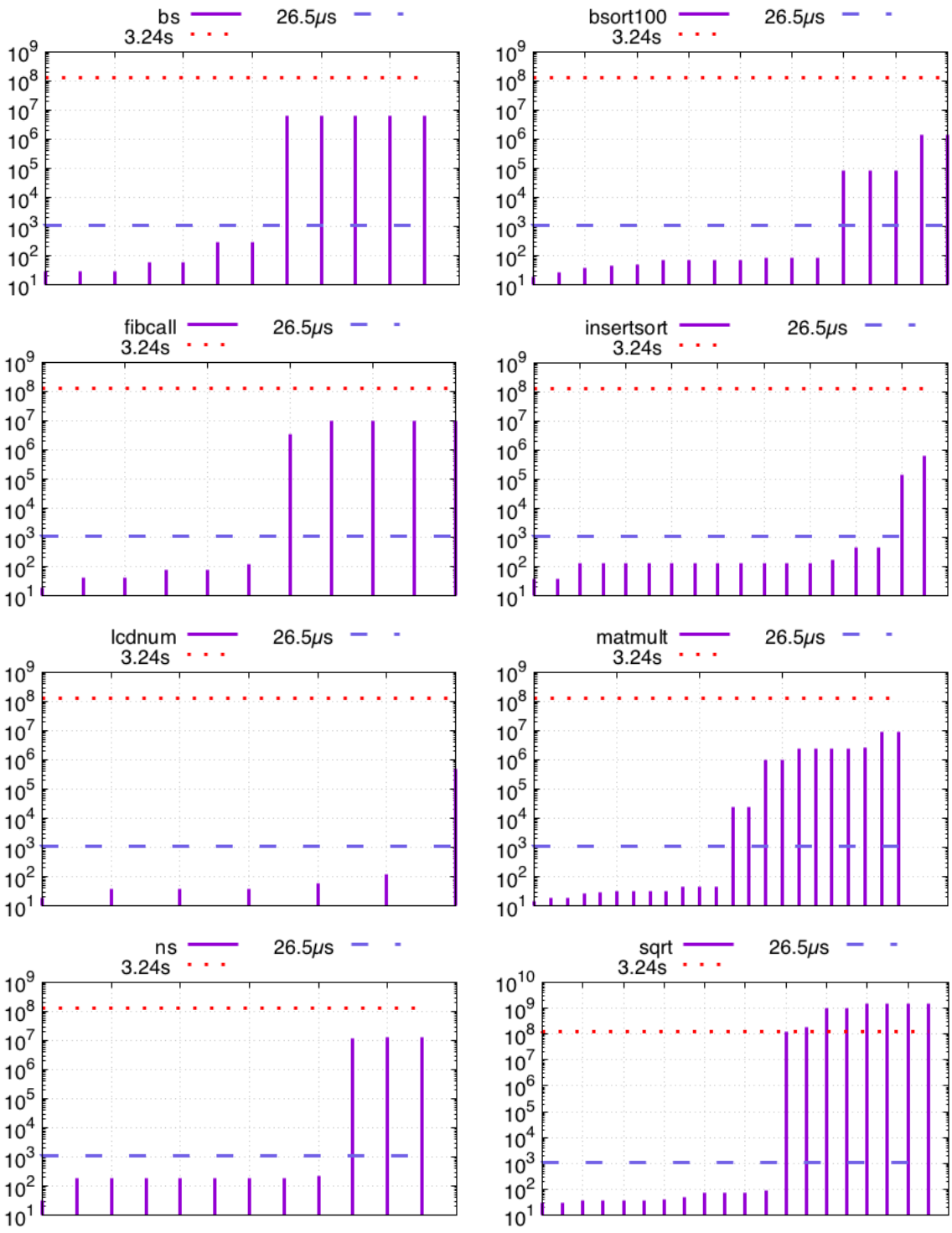


FIGURE 5.12: Worst-case lifetimes of static store instructions for write-light workloads from the Mälardalen benchmark-suite: the X and Y axes respectively represent the write occurrences and their corresponding worst-case lifetime in cycles at 40 MHz. Two duration thresholds are made explicit along the Y axis (26.5 µs and 3.24 s).

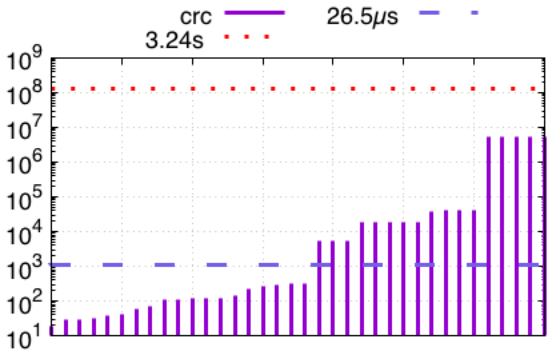


FIGURE 5.13: Worst-case lifetimes of static store instructions for write-light workloads from the Mälardalen benchmark-suite: the X and Y axes respectively represent the write occurrences and their corresponding worst-case lifetime in cycles at 40 MHz. Two duration thresholds are made explicit along the Y axis (26.5 µs and 3.24 s).

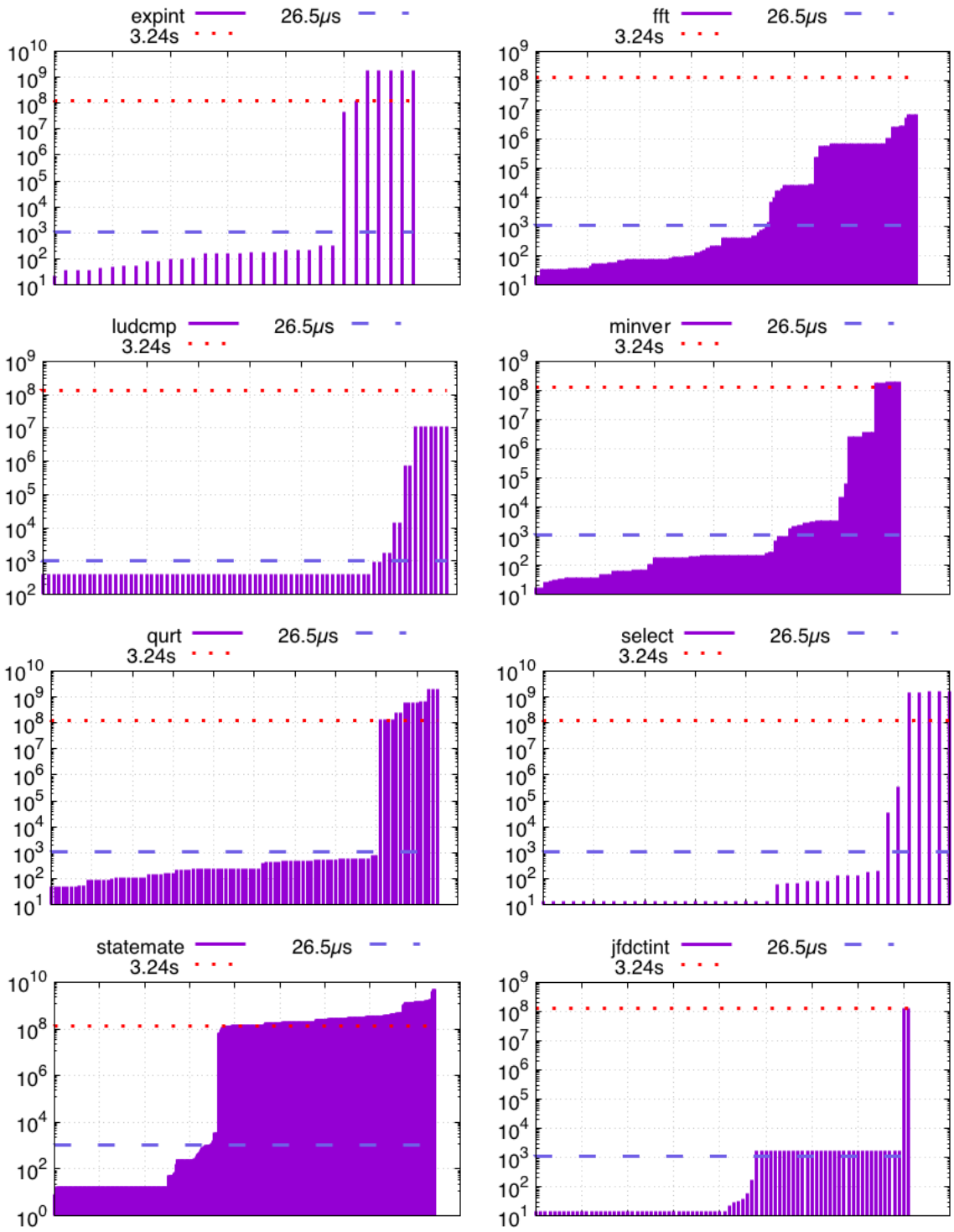


FIGURE 5.14: Worst-case lifetimes of static store instructions for write-intensive workloads from the Mälardalen benchmark-suite: the X and Y axes respectively represent the write instructions and their corresponding worst-case lifetime in cycles at 40 MHz. Two duration thresholds are made explicit along the Y axis (26.5µs and 3.24s).

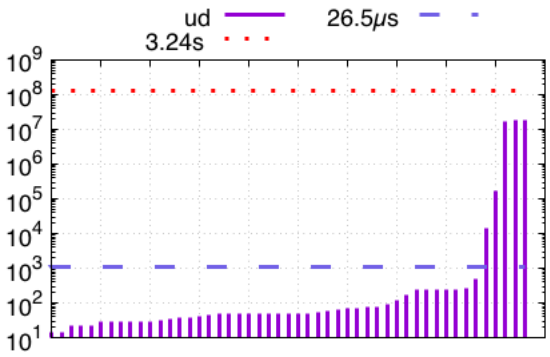


FIGURE 5.15: Worst-case lifetimes of static store instructions for write-intensive workloads from the Mälardalen benchmark-suite: the X and Y axes respectively represent the write instructions and their corresponding worst-case lifetime in cycles at 40 MHz. Two duration thresholds are made explicit along the Y axis (26.5µs and 3.24s).

5.5.3 Lifetimes Evaluation on Benchmarks

As a prior step to data allocation in multi-retention memory, we apply the proposed δ-WCET-based analysis to the different workloads of the Mälardalen benchmark-suite. The expected results are the lifetimes distributions according to the benchmarks. Due to the diversity of these benchmarks, the corresponding distributions in terms of store instructions vary from one benchmark to another. We can categorize the distributions according to the write-intensiveness of the benchmarks.

Figures 5.14 and 5.15 summarize the worst-case lifetimes of static store instructions found in the subset of write-intensive workloads from the Mälardalen benchmark-suite. The X and Y axes respectively denote the write instructions and their corresponding worst-case lifetime in clock cycles, while operating at a frequency of 40 MHz. We note that half of the benchmark-suite falls into this category. In order to build a few clusters of store instructions on the base of their estimated worst-case lifetimes, we consider three duration thresholds featuring three memory retention times: 26.5 µs, 3.24 s and 4.27 years, taken from Sun et al. [106] (see Table 5.2). It clearly appears that the identified store instructions can be partitioned into different groups w.r.t. the three lifetime thresholds. For instance, in the *fft* benchmark, all store instruction lifetimes fall into two clusters: either below 26.5 µs threshold or below 3.24 s threshold; in the *qurt* benchmark, the identified lifetimes are partitioned in three clusters: either below 26.5 µs threshold or below 3.24 s threshold or above 3.24 s threshold. Therefore, this information can be leveraged to reduce energy consumption by allocating the stored data memory accesses to appropriate NVM banks.

Figures 5.12 and 5.13 shows the subset of benchmarks that contain fewer static store instructions compared to those profiled in Figures 5.14 and 5.15. While the expected energy gain may a priori sound limited for write-light benchmarks, it is

Retention time	Read energy (nJ)	Write energy (nJ)
4.27 yr	0.085	1.916
3.24 s	0.083	0.932
26.5 µs	0.081	0.347

TABLE 5.2: 4 MB NVM memory retention times [106]

not necessarily true since a small number of static store instructions executed several times, e.g., in a loop, could have a non negligible impact on the overall memory energy consumption.

According to its lifetime, each store instruction will be associated with a dynamic energy cost corresponding to the memory retention threshold targeted for this instruction. From the execution traces of benchmarks, we determine how many times such instructions are executed, and we finally estimate the energy for each benchmark.

Evolution of lifetimes with program duration. As discussed in Section 5.4.1, not all lifetimes are equal. Some are constant, regardless of the number of iterations (such as lifetimes related to variables b and i of Figure 5.6(b), others vary (such as variable a). We confirmed that the same applies beyond our motivational example. Figure 5.16 shows how lifetimes evolve when the number of iterations of the main loop grows in *sqrt*. Clearly the first ten lifetimes remain unchanged, while the last six increase proportionally, suggesting optimization potentials, even for large run times.

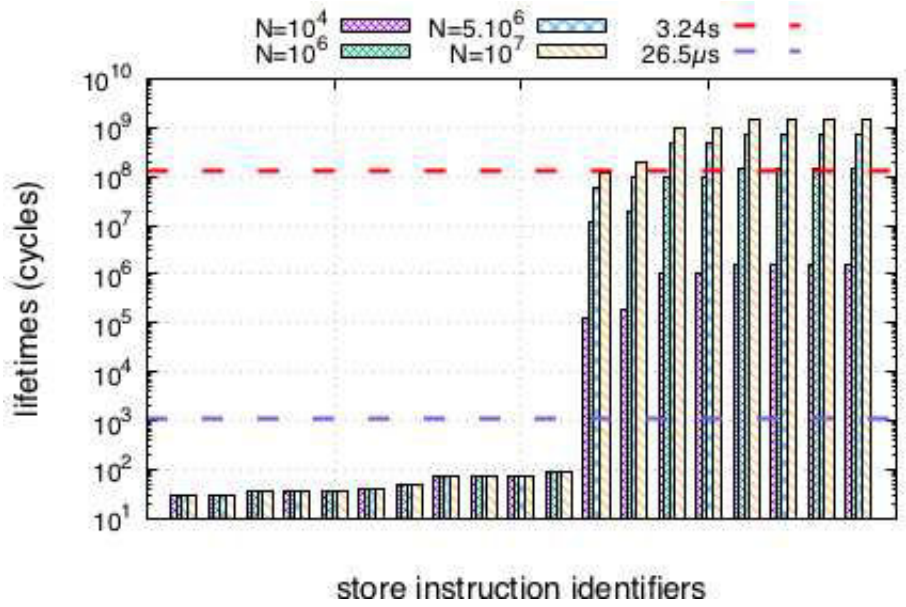


FIGURE 5.16: Lifetime distribution in *sqrt* for different values of MAXITER (i.e., N)

Retention time	Read energy (nJ)	Write energy (nJ)
4.27 yr	0.083	0.958
3.24 s	0.032	0.466
26.5 µs	0.031	0.174

TABLE 5.3: 32 KB NVM memory retention times [106]

5.5.4 Gain evaluation

To evaluate how NVMs with variable retention times have prominent impact on energy consumption, we consider different memory setups depending on the target memory bank sizes. We already introduced one possible setup in Table 5.2 featuring a 4 MB STT-RAM memory size. Now, let us consider another memory setup featuring smaller memory banks with a size of 32 KB (based on the same technology [106]). The energy consumption for this new setup according to the same retention times is summarized in Table 5.3. Note that write energy costs overall are smaller, and read costs are also reduced for shorter retention times.

Given the above setups, we evaluate the energy gain, which mainly comes from the assignment of store operations to appropriate banks based on their worst lifetimes, hence reducing the corresponding write energy. Since the leakage of NVMs is almost null, we only focus on their dynamic energy, i.e., induced by read and write operations. We formulate the energy consumed by loads and stores as:

$$E = N^R \times \alpha^R + N^W \times \alpha^W \tag{5.1}$$

where N^R and N^W are respectively the number of read and write executions and α^R and α^W are respectively the dynamic energies of a read and a write operation. In a system where we have three different banks, we have three different α^R and α^W. Therefore, the energy consumed by loads and stores will become as follows:

$$E = \sum_{i \in \{\text{NVM retention times}\}} N_i^R \times \alpha_i^R + N_i^W \times \alpha_i^W \tag{5.2}$$

where N_i^R and N_i^W are respectively the number of read and write executions on the NVM memory bank i and α_i^R and α_i^W are respectively the dynamic energies of a read and a write operation on the memory bank i.

By applying the above formulas, we compute the dynamic energy gain, as illustrated in Figure 5.17 for the Mälardalen benchmark-suite, w.r.t. 4 MB and 32 KB STT-RAM memory setups respectively. Here, the reported gain is computed against a baseline setup consisting of an STT-RAM memory with a retention time of 4.27 years.

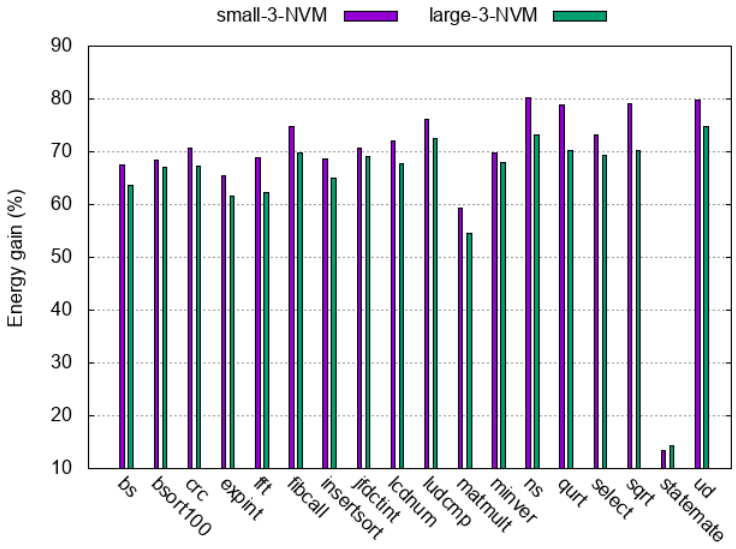


FIGURE 5.17: Energy savings based on Tables 5.3 and 5.2 setup w.r.t. a 4.27 yr STT-RAM memory

Figure 5.17 shows that we achieved with the small memory configuration up to 80 % of energy gain compared to the baseline (small memory with 4.27 years retention time) and up to 75 % with the large memory configuration. In fact, dynamic energy of read and write operations on a small size memory are less expensive compared to the large size memory.

Generally speaking, the energy gain depends on how many times a store is executed on a specific bank of memory. For illustration, Figure 5.18 shows for three benchmarks how many times each static store instruction has been executed (NW) and how many times the stored value has been read (NR), before getting rewritten. The trend observed for these benchmarks is representative of the overall benchmark-suite. Please check Figure 5.19, 5.20 and 5.21 for the whole benchmark-suite.

In the *ud* benchmark, the major part of the store instructions is executed more than 10^4 times. More than half of these instructions have low lifetimes. This is beneficial for energy saving.

The *statemate* benchmark, which shows more lifetimes requiring higher retention times (falling strongly in the memory region with a retention time of 4.27 years), has less energy savings among the three benchmarks. As expected, with both small and large memory configurations, the lowest energy gain is for this benchmark.

In the *matmult* benchmark, we can notice the possible detrimental impact of high NR values despite an important number of store instructions with short lifetimes. This benchmark shows a gain of 60 % as shown in Figure 5.17, for the small memory setup. However, this gain grows down to 55 % when considering the large memory setup, in which the cost of read energy is multiplied by more than two and half.

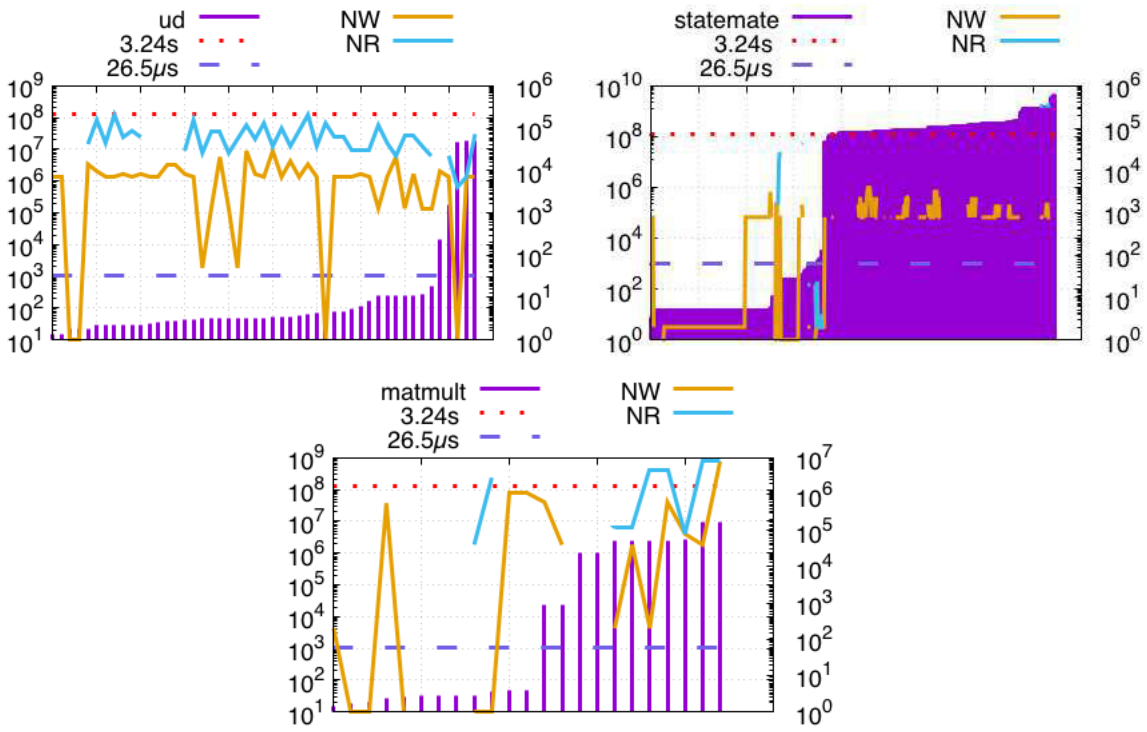


FIGURE 5.18: Plots showing how many times each static store instruction has been executed (NW) and how many times the stored value has been read (NR), before getting rewritten. X, Y-left and Y-right axes respectively correspond to store identifiers, lifetimes in cycles and thresholds for NR and NW.

5.6 Conclusion

In this work, we applied a partial worst-case execution time (δ-WCET) analysis to programs in order to determine the worst-case lifetimes of program variables involved in store instructions. This information is then used to safely allocate these variables in appropriate NVM memory banks, according to their data retention time. We validated our approach on the Mälardalen benchmark-suite, by showing a significant reduction of memory dynamic energy (up to 80%, with an average of 66%). This contributes to answer the energy-efficiency challenge faced in both embedded and high-performance computing domains.

The short-term perspective to this work concerns the implementation of variables mapping to memory banks. While both software and hardware approaches could be considered, we plan to focus on the former approach. More precisely, we will explore a compiler-oriented approach to post-process the assembly for deciding the mappings based on the pre-evaluated worst-case lifetimes. Further research directions address cache-based architectures. Indeed, the case study presented in this paper, features a cache-less system. More generally, NVMs can be exploited at different levels of the memory hierarchy. As discussed in Section 5.4.2, they can be leveraged through

hybrid designs where they are combined with traditional memory technologies, e.g., SRAM or DRAM. Data layout guided by worst-case lifetimes could be key to drive each piece of data to the appropriate memory bank.

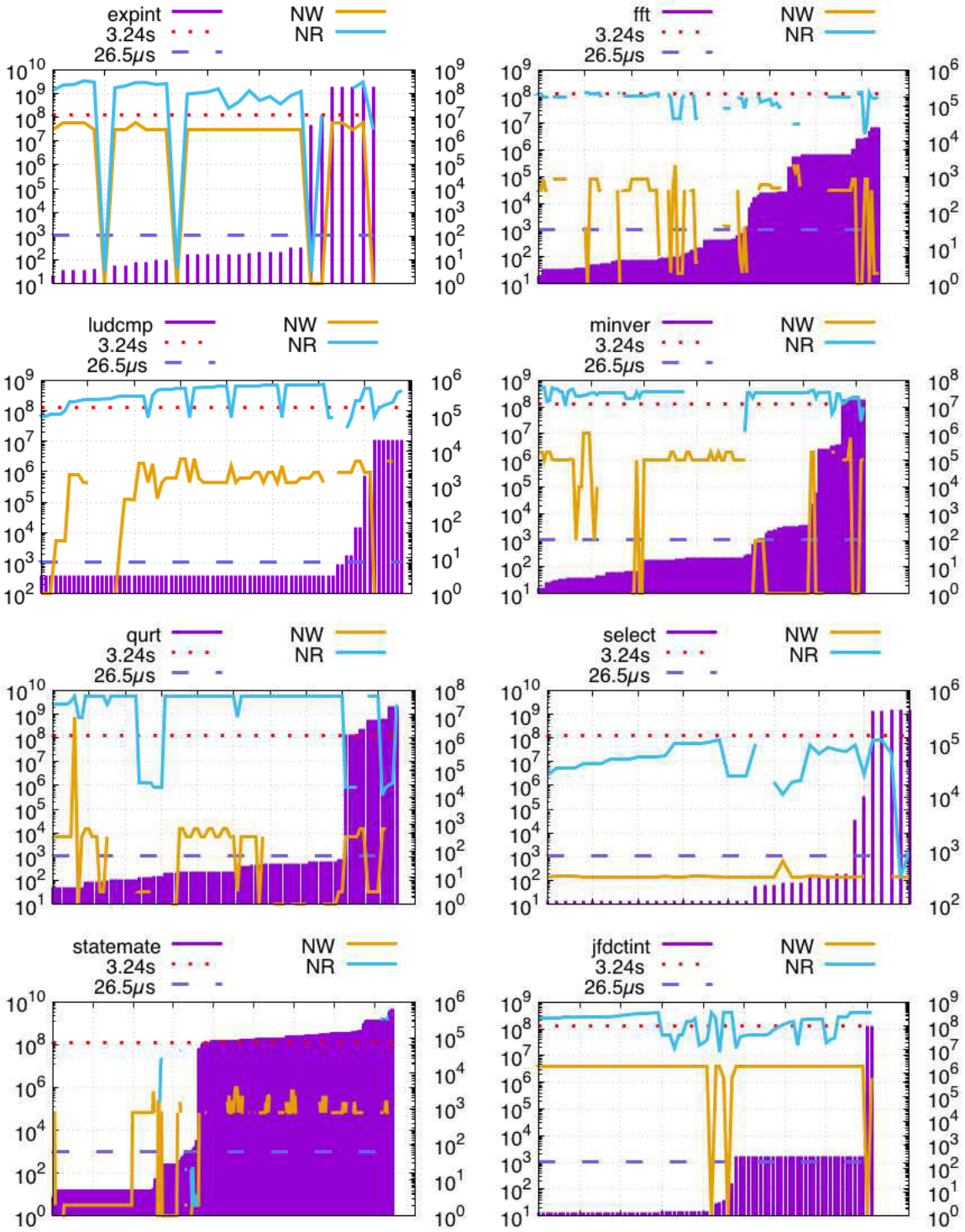


FIGURE 5.19: Plots showing how many times each static store instruction has been executed (NW) and how many times the stored value has been read (NR), before getting rewritten. X, Y-left and Y-right axes respectively correspond to store identifiers, lifetimes in cycles and thresholds for NR and NW.

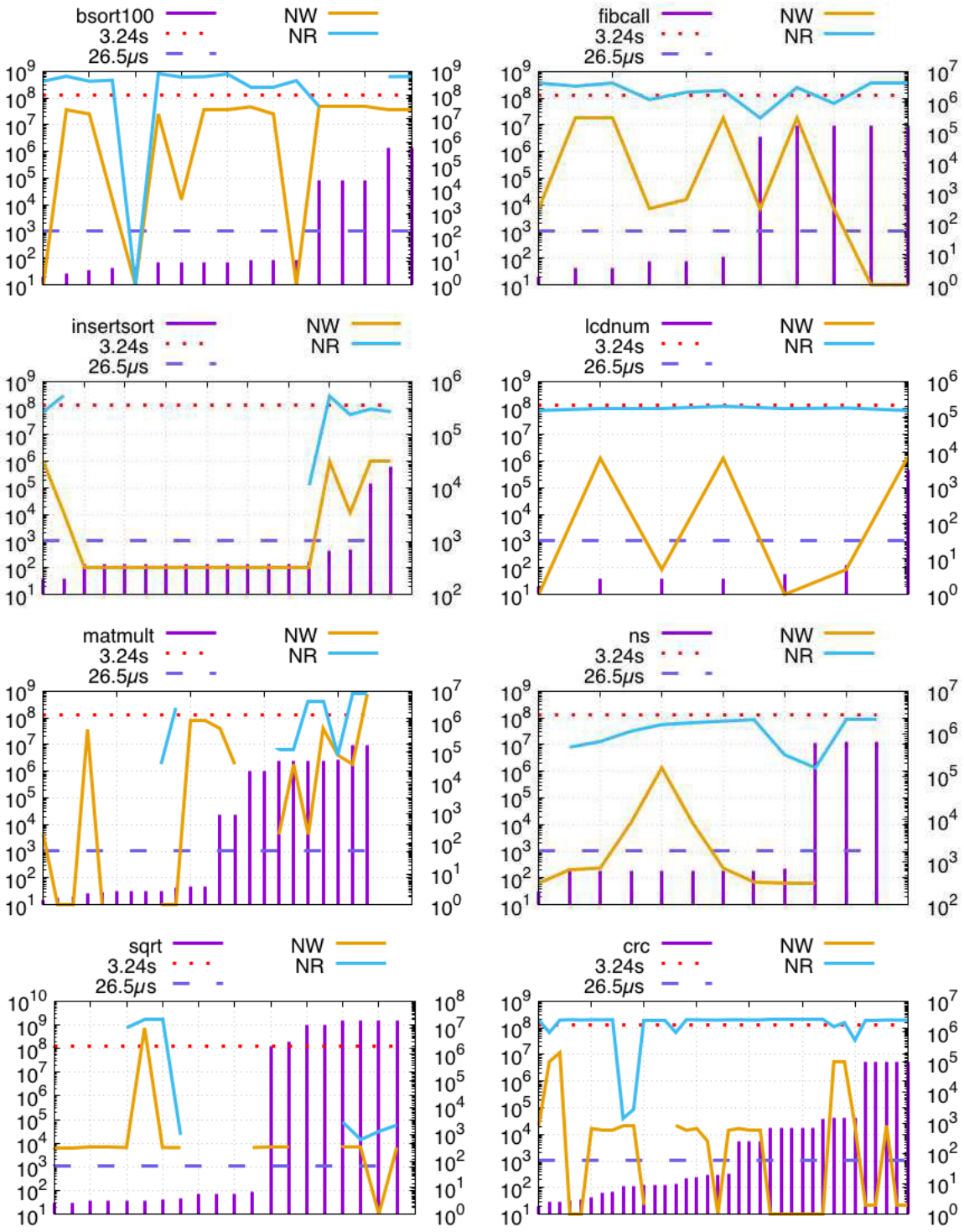


FIGURE 5.20: Plots showing how many times each static store instruction has been executed (NW) and how many times the stored value has been read (NR), before getting rewritten. X, Y-left and Y-right axes respectively correspond to store identifiers, lifetimes in cycles and thresholds for NR and NW.

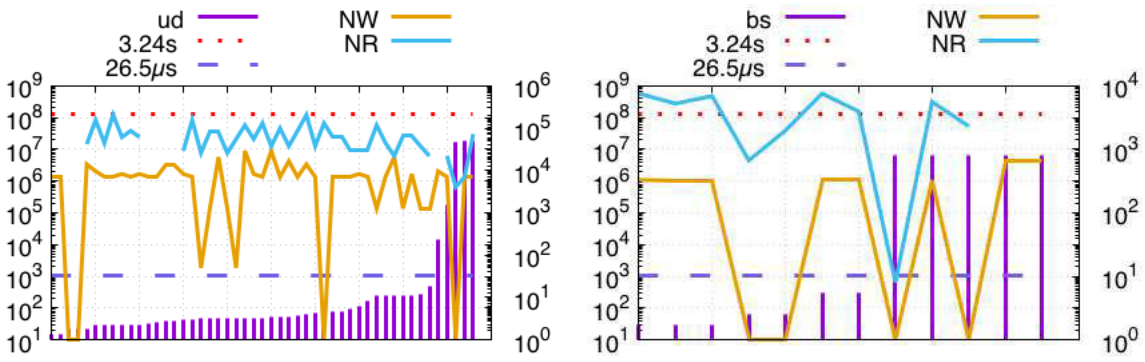


FIGURE 5.21: Plots showing how many times each static store instruction has been executed (NW) and how many times the stored value has been read (NR), before getting rewritten. X, Y-left and Y-right axes respectively correspond to store identifiers, lifetimes in cycles and thresholds for NR and NW.

Chapter 6

Conclusion

As power-efficient design becomes more important, spin-transfer torque RAM (STT-RAM) has drawn a lot of attention due to its ability to meet both high performance and low power consumption. However, its high write energy incurs an increase of dynamic power consumption and may offset power saving due to its low static power. For decades, the memory technology of target systems has consisted in SRAM at cache level, and DRAM for main memory. Emerging non-volatile memories (NVMs) open up new opportunities (they have a quasi-zero static power consumption thanks to their negligible leakage current), along with new design challenges. In particular, the asymmetric cost of read/write accesses calls for adjusting existing techniques in order to efficiently exploit NVMs. In addition, this technology makes it possible to design memories with cheaper accesses at the cost of lower data retention times. These features can be exploited at compile time to derive better data mappings according to the application and data retention characteristics.

6.1 Achieved results

We considered the case of the reduction of the number of writes, and we presented a compiler-based approach to eliminate silent stores. We showed how redundant write elimination can be leveraged in memory systems including NVM, so as to reduce the energy and performance penalty induced to NVMs. Another advantage is the flexibility and portability across various hardware architectures enabled by such compiler-level approaches, compared to the hardware-oriented techniques found in literature. An important advantage of our approach is its portability to any execution platform. This is not the case of the usual hardware implementation, which only integrates the dedicated hardware mechanisms. We addressed a number of system design considerations so as to maximize the overall energy efficiency of target systems. This comprised the analysis of the impact of the application characteristics, the compiler, and various microarchitectural features.

Furthermore, we analyzed data lifetimes through the analysis of variables lifetimes for memory bank assignment of program variables. We applied a partial worst-case execution time (δ-WCET) analysis to programs in order to determine the worst-case lifetimes of program variables involved in store instructions. This information is then used to safely allocate these variables in appropriate NVM memory banks, according to their data retention time. We validated our approach on the Mälardalen benchmark-suite, by showing a significant reduction of memory dynamic energy (up to 80%, with an average of 66%). This contributes to answer the energy-efficiency challenge faced in both embedded and high-performance computing domains.

The approaches presented in this study can be positioned along with the existing architectural approaches and circuit-level ones in a complementary way. As they are portable, our contributions can enhance the proposed architectural designs found in the literature and thus further mitigate the addressed drawbacks and improve the energy consumption of NVM-based systems.

6.2 Perspectives

Future work includes a full validation of identified opportunities within an experimental full-system architecture simulation environment, by considering existing cycle-accurate simulation tools, combined with power estimation tools to evaluate more precisely the gain expected from memory configurations identified as the most energy-efficient with the present analytic approach. A candidate framework is MAGPIE [19], built on top of gem5 [8] and NVSim [20].

The approximate silent stores identification and elimination can be detailed and evaluated in a futur work. In this work, we showed the potential to eliminate such stores and a further analysis may reveal interesting sides. Moreover, we can enhance the elimination of silent stores by combining the transformation with a static prediction of silent stores as proposed by Peireia et al. in [89]. By predicting silent stores, we can apply our transformation without the profiling step, since we already identified the stores that tend to be silent. The asymmetry of $0/1$ writes is also a perspective of this work. As we focused on the asymmetry of read/write, another side of NVMs's asymmetry can be addressed which is the $1/0$ transitions.

Another perspective to the second part of this work (δ-WCET) concerns addressing cache-based architectures. Indeed, the case study presented in this work, features a cache-less system. More generally, NVMs can be exploited at different levels of the memory hierarchy. They can be leveraged through hybrid designs where they are combined with traditional memory technologies, e.g., SRAM or DRAM. Data

layout guided by worst-case lifetimes could be key to drive each piece of data to the appropriate memory bank. Furthermore, our δ-WCET approach can be also used for other reasons. It can be used to improve the performance of a program in a heterogeneous architecture such as ARM big.LITTLE where there are a low-power cluster and high-performance cluster. By computing different δ-WCETs, the user can decide to migrate the execution of the program from one cluster to another based on some constraints (timing, energy, ...). More generally, the computation of different δ-WCETs can be used to help the user to take some decisions related to performance/energy or debugging.

Bibliography

[1] J. Ahn and K. Choi. "Lower-bits cache for low power STT-RAM caches". In: *2012 IEEE International Symposium on Circuits and Systems*. 2012, pp. 480–483. DOI: 10.1109/ISCAS.2012.6272069.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[3] Mathieu Avila, Maxime Glaizot, and Isabelle Puaut. "Impact of Automatic Gain Time Identification on Tree-Based Static WCET Analysis". In: *Proceedings of 3rd International Workshop on WCET 2003- a Sattelite Event to ECRTS 2003, Polytechnic Institute of Porto, Portugal, July1 1, 2003*. 2003.

[4] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. "Compiler Transformations for High-performance Computing". In: *ACM Comput. Surv.* 26.4 (Dec. 1994), pp. 345–420. ISSN: 0360-0300. DOI: 10.1145/197405.197406.

[5] Gordon B Bell, Kevin M Lepak, and Mikko H Lipasti. "Characterization of silent stores". In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2000.

[6] L. Benini, L. Macchiarulo, A. Macii, and M. Poncino. "Layout-driven memory synthesis for embedded systems-on-chip". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 10.2 (2002), pp. 96–105. ISSN: 1063-8210. DOI: 10.1109/92.994985.

[7] L. Benini, A. Macii, E. Macii, and M. Poncino. "Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation". In: *IEEE Design Test of Computers* 17.2 (2000), pp. 74–85. ISSN: 0740-7475. DOI: 10.1109/54.844336.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7.

[9] R. Bishnoi, M. Ebrahimi, F. Oboril, and M. B. Tahoori. "Asynchronous Asymmetrical Write Termination (AAWT) for a low power STT-MRAM". In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2014, pp. 1–6. DOI: 10.7873/DATE.2014.193.

[10] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. "Emerging NVM: A Survey on Architectural Integration and Research Challenges". In: *ACM Trans. Design Autom. Electr. Syst.* 23.2 (2018), 14:1–14:32. DOI: 10.1145/3131848. URL: http://doi.acm.org/10.1145/3131848.

[11] R. Bouziane, E. Rohou, and A. Gamatié. "How could compile-time program analysis help leveraging emerging NVM features?" In: *2017 First International Conference on Embedded Distributed Systems (EDiS)*. 2017, pp. 1–6. DOI: 10.1109/EDIS.2017.8284031.

[12] Rabab Bouziane, Erven Rohou, and Abdoulaye Gamatié. "Compile-Time Silent-Store Elimination for Energy Efficiency: an Analytic Evaluation for Non-Volatile Cache Memory". In: *Proceedings of the RAPIDO 2018 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, Manchester, UK, January 22-24, 2018*. 2018, 5:1–5:8. DOI: 10.1145/3180665.3180666. URL: http://doi.acm.org/10.1145/3180665.3180666.

[13] Rabab Bouziane, Erven Rohou, and Abdoulaye Gamatié. "Energy-Efficient Memory Mappings based on Partial WCET Analysis and Multi-Retention Time STT-RAM". In: *RTNS 2018 - 26th International Conference on Real-Time Networks and Systems*. Poitiers, France, Oct. 2018, pp. 1–11. DOI: 10.1145/3273905.3273908. URL: https://hal.inria.fr/hal-01871320.

[14] A. Butko, F. Bruguier, A. Gamatié, G. Sassatelli, D. Novo, L. Torres, and M. Robert. "Full-System Simulation of big.LITTLE Multicore Architecture for Performance and Energy Exploration". In: *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*. 2016, pp. 201–208. DOI: 10.1109/MCSoC.2016.20.

[15] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads". In: *International Symposium on Workload Characterization (IISWC'10)*. 2010. ISBN: 978-1-4244-9297-8.

[16] Y. Chen, J. Cong, H. Huang, B. Liu, C. Liu, M. Potkonjak, and G. Reinman. "Dynamically reconfigurable hybrid cache: An energy-efficient last-level cache

design". In: *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2012, pp. 45–50. DOI: 10.1109/DATE.2012.6176431.

[17] Yu-Ting Chen, Jason Cong, Hui Huang, Chunyue Liu, Raghu Prabhakar, and Glenn Reinman. "Static and dynamic co-optimizations for blocks mapping in hybrid caches". In: *Proceedings of the 2012 international symposium on Low power electronics and design*. ACM. 2012, pp. 237–242.

[18] Wei-Kai Cheng, Yen-Heng Ciou, and Po-Yuan Shen. "Architecture and data migration methodology for L1 cache design with hybrid SRAM and volatile STT-RAM configuration". In: *Microprocessors and Microsystems* 42 (2016).

[19] T. Delobelle, P. Péneau, A. Gamatié, F. Bruguier, S. Senni, G. Sassatelli, and L. Torres. "MAGPIE: System-level Evaluation of Manycore Systems with Emerging Memory Technologies". In: *Workshop on Emerging Memory Solutions - Technology, Manufacturing, Architectures, Design and Test at Design Automation and Test in Europe (DATE'2017), Lausanne, Switzerland*. 2017.

[20] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P Jouppi. "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 31.7 (2012), pp. 994–1007.

[21] Ricardo Gonzalez and Mark Horowitz. "Energy dissipation in general purpose microprocessors". In: *IEEE Journal of solid-state circuits* 31.9 (1996), pp. 1277–1284.

[22] Nilanjan Goswami, Bingyi Cao, and Tao Li. "Power-performance co-optimization of throughput core architecture using resistive memory". In: *High Performance Computer Architecture (HPCA), IEEE 19th International Symposium on*. 2013, pp. 342–353.

[23] Xiaochen Guo, Engin Ipek, and Tolga Soyata. "Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM Based Computing". In: *SIGARCH Comput. Archit. News* 38.3 (June 2010), pp. 371–382. ISSN: 0163-5964. DOI: 10.1145/1816038.1816012. URL: http://doi.acm.org/10.1145/1816038.1816012.

[24] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. *The Mälardalen WCET benchmarks: Past, present and future*. Jan. 2010.

[25] H. Hajimiri, K. Rahmani, and P. Mishra. "Synergistic integration of dynamic cache reconfiguration and code compression in embedded systems". In: *2011 International Green Computing Conference and Workshops*. 2011, pp. 1–8. DOI: 10.1109/IGCC.2011.6008580.

[26] HardKernel. *Odroid XU4*. http://www.hardkernel.com.

[27] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. "The Heptane Static Worst-Case Execution Time Estimation Tool". In: *17th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Vol. 8. Dubrovnik, Croatia, June 2017, p. 12. DOI: 10.4230/OASIcs.WCET.2017.8. URL: http://hal.upmc.fr/hal-01590444.

[28] Niklas Holsti and Sami Saarinen. "Status of the Bound-T WCET tool". In: (Jan. 2002).

[29] Jingtong Hu, Chun Jason Xue, Wei-Che Tseng, Yi He, Meikang Qiu, and Edwin H.-M. Sha. "Reducing Write Activities on Non-volatile Memories in Embedded CMPs via Data Migration and Recomputation". In: *Design Automation Conference (DAC'10)*. 2010.

[30] Jingtong Hu, Chun Jason Xue, Qingfeng Zhuge, Wei-Che Tseng, and Edwin Hsing-Mean Sha. "Data Allocation Optimization for Hybrid Scratch Pad Memory With SRAM and Nonvolatile Memory". In: *Trans. VLSI Syst.* (2013).

[31] Shaoxiong Hua and Gang Qu. "Approaching the maximum energy saving on embedded systems with multiple voltages". In: *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No.03CH37486)*. 2003, pp. 26–29. DOI: 10.1109/ICCAD.2003.159666.

[32] Maha Idrissi Aouad and Olivier Zendra. "A Survey of Scratch-Pad Memory Management Techniques for low-power and -energy". In: *2nd ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007)*. Ed. by Olivier Zendra, Eric Jul, and Michael Cebulla. July 2007, pp. 31–38. URL: https://hal.inria.fr/inria-00170210.

[33] K Ikegami, H Noguchi, C Kamata, M Amano, K Abe, K Kushida, E Kitagawa, T Ochiai, N Shimomura, A Kawasumi, H Hara, J Ito, and S Fujita. "A 4ns, 0.9V write voltage embedded perpendicular STT-MRAM fabricated by MTJ-Last process". In: *Procedings of Technical Progarm- 2014 International Symposium on VLSI Technology, Systems and Application* (Apr. 2014), pp. 1–2.

[34] Michael Jacobs, Sebastian Hahn, and Sebastian Hack. "WCET Analysis for Multi-core Processors with Shared Buses and Event-driven Bus Arbitration". In: *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. RTNS. Lille, France: ACM, 2015, pp. 193–202. ISBN: 978-1-4503-3591-1. DOI: 10.1145/2834848.2834872. URL: http://doi.acm.org/10.1145/2834848.2834872.

[35] A. Jadidi, M. Arjomand, and H. Sarbazi-Azad. "High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement". In: *IEEE/ACM International Symposium on Low Power Electronics and Design*. 2011, pp. 79–84. DOI: 10.1109/ISLPED.2011.5993611.

[36] Adwait Jog, Asit K. Mishra, Cong Xu, Yuan Xie, Vijaykrishnan Narayanan, Ravishankar Iyer, and Chita R. Das. "Cache revive: architecting volatile STT-RAM caches for enhanced performance in CMPs". In: *Annual Design Automation Conference DAC*. 2012. DOI: 10.1145/2228360.2228406.

[37] Yongsoo Joo, Dimin Niu, Xiangyu Dong, Guangyu Sun, Naehyuck Chang, and Yuan Xie. "Energy- and endurance-aware design of phase change memory caches". In: *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)* (2010), pp. 136–141.

[38] Jinwook Jung, Y. Nakata, M. Yoshimoto, and H. Kawaguchi. "Energy-efficient Spin-Transfer Torque RAM cache exploiting additional all-zero-data flags". In: *International Symposium on Quality Electronic Design (ISQED)*. 2013, pp. 216–222. DOI: 10.1109/ISQED.2013.6523613.

[39] Mahmut Kandemir, N. Vijaykrishnan, and Mary Jane Irwin. "Power Aware Computing". In: ed. by Robert Graybill and Rami Melhem. Norwell, MA, USA: Kluwer Academic Publishers, 2002. Chap. Compiler Optimizations for Low Power Systems, pp. 191–210. ISBN: 0-306-46786-0. URL: http://dl.acm.org/citation.cfm?id=783060.783071.

[40] Mahmut Kandemir, N. Vijaykrishnan, Mary Jane Irwin, and Wu Ye. "Influence of Compiler Optimizations on System Power". In: *IEEE Trans. Very Large Scale Integr. Syst.* 9.6 (Dec. 2001), pp. 801–804. ISSN: 1063-8210.

[41] Mahmut T. Kandemir, Ibrahim Kolcu, and Ismail Kadayif. "Influence of Loop Optimizations on Energy Consumption of Multi-bank Memory Systems". In: *Proceedings of the 11th International Conference on Compiler Construction*. CC '02. London, UK, UK: Springer-Verlag, 2002, pp. 276–292. ISBN: 3-540-43369-4. URL: http://dl.acm.org/citation.cfm?id=647478.727932.

[42] Mahmut T. Kandemir, Ibrahim Kolcu, and Ismail Kadayif. "Influence of Loop Optimizations on Energy Consumption of Multi-bank Memory Systems". In: *Proceedings of the 11th International Conference on Compiler Construction*. Vol. 2304. Lecture Notes in Computer Science. Springer International Publishing, 2002, pp. 276–292. DOI: 10.1007/3-540-45937-5_20.

[43] S.H. Kang and K. Lee. "Emerging materials and devices in spintronic integrated circuits for energy-smart mobile computing and connectivity". English. In: *Acta Materialia* 61.3 (2013), pp. 952–973. DOI: 10.1016/j.actamat.2012.10.036.

[44] S. Kaxiras, Zhigang Hu, and M. Martonosi. "Cache decay: exploiting generational behavior to reduce cache leakage power". In: *Proceedings 28th Annual International Symposium on Computer Architecture*. 2001, pp. 240–251. DOI: 10.1109/ISCA.2001.937453.

[45] Navid Khoshavi, Xunchao Chen, Jun Wang, and Ronald F. DeMara. "Read-Tuned STT-RAM and eDRAM Cache Hierarchies for Throughput and Energy Enhancement". In: *CoRR* abs/1607.08086 (2016). arXiv: 1607.08086. URL: http://arxiv.org/abs/1607.08086.

[46] A V Khvalkovskiy, D Apalkov, S Watts, R Chepulskii, R S Beach, A Ong, X Tang, A Driskill-Smith, W H Butler, P B Visscher, D Lottis, E Chen, V Nikitin, and M Krounbi. "Basic principles of STT-MRAM cell operation in memory arrays". In: *Journal of Physics D: Applied Physics* 46.7 (2013).

[47] V. Kianzad, S. S. Bhattacharyya, and Gang Qu. "CASPER: an integrated energy-driven approach for task graph scheduling on distributed embedded systems". In: *2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*. 2005, pp. 191–197. DOI: 10.1109/ASAP.2005.23.

[48] Toru Kisuki, Peter M. W. Knijnenburg, and Michael F. P. O'Boyle. "Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation." In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, 2000, pp. 237–248. ISBN: 0-7695-0622-4.

[49] K. Kwon, S. H. Choday, Y. Kim, and K. Roy. "AWARE (Asymmetric Write Architecture With REdundant Blocks): A High Write Speed STT-MRAM Cache Architecture". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.4 (2014), pp. 712–720. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2013.2256945.

[50] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. ISBN: 0-7695-2102-9.

[51] Christophe Layer, Laurent Becker, Kotb Jabeur, Sylvain Claireux, Bernard Dieny, Guillaume Prenat, Gregory Di Pendina, Stephane Gros, Pierre Paoli, Virgile Javerliac, Fabrice Bernard-Granger, and Loic Decloedt. "Reducing System Power Consumption Using Check-Pointing on Nonvolatile Embedded Magnetic Random Access Memories". In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 12.4 (2016), p. 32.

[52] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. "Architecting Phase Change Memory As a Scalable Dram Alternative". In: *SIGARCH Comput. Archit. News* 37.3 (June 2009), pp. 2–13. ISSN: 0163-5964. DOI: 10.1145/1555815.1555758. URL: http://doi.acm.org/10.1145/1555815.1555758.

[53] Dongwoo Lee and David Blaauw. "Static Leakage Reduction Through Simultaneous Threshold Voltage and State Assignment". In: *Proceedings of the 40th Annual Design Automation Conference*. DAC '03. Anaheim, CA, USA: ACM, 2003, pp. 191–194. ISBN: 1-58113-688-9. DOI: 10.1145/775832.775881. URL: http://doi.acm.org/10.1145/775832.775881.

[54] Kevin M Lepak, Gordon B Bell, and Mikko H Lipasti. "Silent stores and store value locality". In: *IEEE Transactions on Computers* 50.11 (2001).

[55] Kevin M. Lepak and Mikko H. Lipasti. "On the Value Locality of Store Instructions". In: *International Symposium on Computer Architecture (ISCA'2000), Vancouver, British Columbia, Canada*. 2000, pp. 182–191.

[56] Dexin Li, P. H. Chou, and N. Bagherzadeh. "Mode selection and mode-dependency modeling for power-aware embedded systems". In: *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design*. 2002, pp. 697–704. DOI: 10.1109/ASPDAC.2002.995016.

[57] Hanbing Li, Isabelle Puaut, and Erven Rohou. "Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation". In: *RTNS - 22nd International Conference on Real-Time Networks and Systems*. Versailles, France, Oct. 2014. DOI: 10.1145/2659787.2659805. URL: https://hal.inria.fr/hal-01072138.

[58] J. Li, L. Shi, Q. Li, C. J. Xue, Y. Chen, and Y. Xu. "Cache coherence enabled adaptive refresh for volatile STT-RAM". In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2013, pp. 1247–1250. DOI: 10.7873/DATE.2013.258.

[59] J. Li, L. Shi, C. J. Xue, C. Yang, and Y. Xu. "Exploiting set-level write non-uniformity for energy-efficient NVM-based hybrid cache". In: *2011 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia*. 2011, pp. 19–28. DOI: 10.1109/ESTIMedia.2011.6088521.

[60] Jianhua Li, Chun Jason Xue, and Yinlong Xu. "STT-RAM based energy-efficiency hybrid cache for CMPs". In: *Int. Conf. on VLSI and SoC (VLSI-SoC'11)*. Kowloon, Hong Kong, China, 2011.

[61] Qing'an Li, Yanxiang He, Jianhua Li, Liang Shi, Yiran Chen, and Chun Jason Xue. "Compiler-Assisted Refresh Minimization for Volatile STT-RAM Cache." In: *IEEE Trans. Computers* 64.8 (2015), pp. 2169–2181.

[62] Qingan Li, Jianhua Li, Liang Shi, Chun Jason Xue, and Yanxiang He. "MAC: Migration-aware Compilation for STT-RAM Based Hybrid Cache in Embedded Systems". In: *Int. Symp. on Low Power Electronics and Design (ISLPED)*. 2012.

[63] Qingan Li, Jianhua Li, Liang Shi, Mengying Zhao, Chun Jason Xue, and Yanxiang He. "Compiler-assisted STT-RAM-based hybrid cache for energy efficient embedded systems". In: *Transactions on Very Large Scale Integration (VLSI) Systems* 22.8 (2014).

[64] Qingan Li, Liang Shi, Jianhua Li, Chun Jason Xue, and Yanxiang He. "Code Motion for Migration Minimization in STT-RAM Based Hybrid Cache". In: *Computer Society Annual Symposium on VLSI*. 2012.

[65] Qingan Li, Mengying Zhao, Chun Jason Xue, and Yanxiang He. "Compiler-assisted Preferred Caching for Embedded Systems with STT-RAM Based Hybrid Cache". In: *ACM International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES '12)*. 2012.

[66] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures". In: *Proceedings of the 42nd Annual International Symposium on Microarchitecture*. MICRO 42. New York, New York: ACM, 2009, pp. 469–480. ISBN: 978-1-60558-798-1.

[67] Yong Li, Yiran Chen, and Alex K. Jones. "A Software Approach for Combating Asymmetries of Non-volatile Memories". In: *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED '12. Redondo Beach, California, USA: ACM, 2012, pp. 191–196. ISBN: 978-1-4503-1249-3. DOI: 10.1145/2333660.2333708. URL: http://doi.acm.org/10.1145/2333660.2333708.

[68] Yong Li and Alex K Jones. "Cross-layer techniques for optimizing systems utilizing memories with asymmetric access characteristics". In: *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on*. IEEE. 2012, pp. 404–409.

[69] Yong Li, Yaojun Zhang, Hai LI, Yiran Chen, and Alex K. Jones. "C1C: A Configurable, Compiler-guided STT-RAM L1 Cache". In: *ACM Trans. Archit. Code Optim.* 10.4 (Dec. 2013), 52:1–52:22. ISSN: 1544-3566. DOI: 10.1145/2555289.2555308. URL: http://doi.acm.org/10.1145/2555289.2555308.

[70] I. Lin and J. Chiou. "High-Endurance Hybrid Cache Design in CMP Architecture With Cache Partitioning and Access-Aware Policies". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.10 (2015), pp. 2149–2161. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2014.2361150.

[71] Paul Marchal, José Ignacio Gómez, and Francky Catthoor. "Optimizing the Memory Bandwidth with Loop Fusion". In: *Proceedings of the 2nd International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '04. Stockholm, Sweden, 2004, pp. 188–193. ISBN: 1-58113-937-3.

[72] Larry McVoy and Carl Staelin. "Lmbench: Portable Tools for Performance Analysis". In: *USENIX Annual Technical Conference*. San Diego, CA: USENIX Association, 1996, pp. 23–23.

[73] R. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. "Techniques for low energy software". In: *Proceedings of 1997 International Symposium on Low Power Electronics and Design*. 1997, pp. 72–75. DOI: 10.1145/263272.263286.

[74] M. de Michiel, A. Bonenfant, C. Ballabriga, and H. Cassé. "Partial Flow Analysis with oRange". In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. LNCS 6416. 2010, pp. 479–482.

[75] S. Mittal, J. S. Vetter, and D. Li. "LastingNVCache: A Technique for Improving the Lifetime of Non-volatile Caches". In: *2014 IEEE Computer Society Annual Symposium on VLSI*. 2014, pp. 534–540. DOI: 10.1109/ISVLSI.2014.69.

[76] Sparsh Mittal, Jeffrey Vetter, and Dong Li. "WriteSmoothing: Improving Lifetime of Non-volatile Caches Using Intra-set Wear-leveling". In: *Proceedings of the ACM Great Lakes Symposium on VLSI*. ACM, 2014, pp. 139–144.

[77] Sparsh Mittal and Jeffrey S. Vetter. "A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems". In: *Trans. Parallel Distrib. Syst.* 27.5 (2016).

[78] Sparsh Mittal and Jeffrey S. Vetter. "EqualChance: Addressing Intra-set Write Variation to Increase Lifetime of Non-volatile Caches". In: *2nd USENIX Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*. Broomfield, CO, United States, Oct. 2014. URL: https://hal.archives-ouvertes.fr/hal-01104645.

[79] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4.

[80] Linwei Niu and Gang Quan. "Reducing Both Dynamic and Leakage Energy Consumption for Hard Real-time Systems". In: *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. CASES '04. Washington DC, USA: ACM, 2004, pp. 140–148. ISBN: 1-58113-890-3. DOI: 10.1145/1023833.1023854. URL: http://doi.acm.org/10.1145/1023833.1023854.

[81] Hiroki Noguchi, Kazutaka Ikegami, Keiichi Kushida, Keiko Abe, Shogo Itai, Satoshi Takaya, Naoharu Shimomura, Junichi Ito, Atsushi Kawasumi, Hiroyuki Hara, and Shigeji Fujita. "A 3.3ns-access-time 71.2µW/MHz 1Mb embedded STT-MRAM using physically eliminated read-disturb scheme and normally-off memory architecture". In: *ISSCC*. IEEE, Feb. 2015, pp. 1–3.

[82] Dominic Oehlert, Selma Saidi, and Heiko Falk. "Compiler-based Extraction of Event Arrival Functions for Real-Time Systems Analysis". In: *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*. 2018, 4:1–4:22. DOI: 10.4230/LIPIcs.ECRTS.2018.4. URL: https://doi.org/10.4230/LIPIcs.ECRTS.2018.4.

[83] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. "Minimizing the Cost of Synchronisations in the WCET of Real-time Parallel Programs". In: *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '14. Sankt Goar, Germany: ACM, 2014, pp. 98–107. ISBN: 978-1-4503-2941-5. DOI: 10.1145/2609248.2609261. URL: http://doi.acm.org/10.1145/2609248.2609261.

[84] Xiang Pan and Radu Teodorescu. "NVSleep: Using non-volatile memory to enable fast sleep/wakeup of idle cores". In: *International Conference on Computer Design, ICCD*. 2014. DOI: 10.1109/ICCD.2014.6974712.

[85] Preeti Ranjan Panda, Nikil D. Dutt, Alexandru Nicolau, Francky Catthoor, Arnout Vandecappelle, Erik Brockmeyer, Chidamber Kulkarni, and Eddy de Greef. "Data Memory Organization and Optimizations in Application-Specific Systems". In: *IEEE Design & Test of Computers* 18.3 (2001), pp. 56–68.

[86] S. P. Park, S. Gupta, N. Mojumder, A. Raghunathan, and K. Roy. "Future cache design using STT MRAMs for improved energy efficiency: Devices, circuits and architecture". In: *DAC Design Automation Conference 2012*. 2012, pp. 492–497.

[87] PierreYves Péneau, Rabab Bouziane, Abdoulaye Gamatié, Erven Rohou, Florent Bruguier, Gilles Sassatelli, Lionel Torres, and Sophiane Senni. "Loop optimization in presence of STT-MRAM caches: A study of performance-energy tradeoffs". In: *26th International Workshop on Power and Timing Modeling, Optimization and Simulation, PATMOS 2016, Bremen, Germany, September 21-23, 2016*. 2016, pp. 162–169. DOI: 10.1109/PATMOS.2016.7833682. URL: https://doi.org/10.1109/PATMOS.2016.7833682.

[88] Pierre-Yves Péneau, David Novo, Florent Bruguier, Lionel Torres, Gilles Sassatelli, and Abdoulaye Gamatié. "Improving the Performance of STT-MRAM LLC through Enhanced Cache Replacement Policy". In: *ARCS: Architecture of Computing Systems*. Vol. LNCS. 10793. Braunschweig, Germany, Apr. 2018, pp. 168–180. DOI: 10.1007/978-3-319-77610-1_13. URL: https://hal-lirmm.ccsd.cnrs.fr/lirmm-01669254.

[89] Fernando Magno Quintão Pereira, Guilherme Vieira Leobas, and Abdoulaye Gamatié. "Static Prediction of Silent Stores". In: URL: https://doi.org/10.1145/.

[90] *Polybench Benchmark*. Ohio State University. URL: https://sourceforge.net/projects/polybench.

[91] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories". In: *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*. ISLPED '00. Rapallo, Italy: ACM, 2000, pp. 90–95. ISBN: 1-58113-190-9. DOI: 10.1145/344166.344526. URL: http://doi.acm.org/10.1145/344166.344526.

[92] Qingan Li et al. “MGC: Multiple graph-coloring for non-volatile memory based hybrid Scratchpad Memory”. In: *Workshop on Interaction between Compilers and Computer Architectures (INTERACT)* (2012).

[93] Keni Qiu, Junpeng Luo, Zhiyao Gong, Weigong Zhang, Jing Wang, Yuanchao Xu, Tao Li, and Chun Jason Xue. “Refresh-aware loop scheduling for high performance low power volatile STT-RAM”. In: *34th IEEE International Conference on Computer Design, ICCD 2016, Scottsdale, AZ, USA, October 2-5, 2016*. 2016, pp. 209–216. DOI: 10.1109/ICCD.2016.7753282. URL: https://doi.org/10.1109/ICCD.2016.7753282.

[94] Baixing Quan, Tiefei Zhang, Tianzhou Chen, and Jianzhong Wu. “Prediction table based management policy for STT-RAM and SRAM hybrid cache”. In: *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*. 2012, pp. 1092–1097.

[95] J. Ramanujam, J. Hong, M. Kandemir, and A. Narayan. “Reducing memory requirements of nested loops for embedded systems”. In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. 2001, pp. 359–364. DOI: 10.1145/378239.378523.

[96] N. D. Rizzo, M. DeHerrera, J. Janesky, B. Engel, J. Slaughter, and S. Tehrani. “Thermally activated magnetization reversal in submicron magnetic tunnel junctions for magnetoresistive random access memory”. In: *Appl. Phys. Lett.* 80 (2002). DOI: https://doi.org/10.1063/1.1462872.

[97] Gabriel Rodríguez, Juan Touriño, and Mahmut T. Kandemir. “Volatile STT-RAM Scratchpad Design and Data Allocation for Low Energy”. In: *TACO* 11.4 (2014), 38:1–38:26. DOI: 10.1145/2669556. URL: http://doi.acm.org/10.1145/2669556.

[98] Samsung. *Exynos Octa 5422*. 2015.

[99] Cortus SAS. *APS25s+ – Enhanced Performance Embedded Microcontroller With Leading Code Density*. Product flyer – Online http://www.cortus.com/index.php/ip/. 2017.

[100] André Seznec and Pierre Michaud. “A case for (partially) TAgged GEometric history length branch prediction”. In: *Journal of Instruction Level Parallelism* 8 (2006).

[101] Clinton W. Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R. Stan. "Relaxing Non-volatility for Fast and Energy-efficient STT-RAM Caches". In: *Int. Symp. on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2011. ISBN: 978-1-4244-9432-3.

[102] Clinton W. Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R. Stan. "Relaxing Non-volatility for Fast and Energy-efficient STT-RAM Caches". In: *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*. HPCA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 50–61. ISBN: 978-1-4244-9432-3. URL: http://dl.acm.org/citation.cfm?id=2014698.2014895.

[103] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. "Reducing energy consumption by dynamic copying of instructions onto onchip memory". In: *15th International Symposium on System Synthesis, 2002.* 2002, pp. 213–218. DOI: 10.1145/581199.581247.

[104] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen. "A novel architecture of the 3D stacked MRAM L2 cache for CMPs". In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 2009, pp. 239–249. DOI: 10.1109/HPCA.2009.4798259.

[105] Guangyu Sun, Xiangyu Dong, Yuan Xie, Jian Li, and Yiran Chen. "A novel architecture of the 3D stacked MRAM L2 cache for CMPs". In: *International Conference on High-Performance Computer Architecture (HPCA'09)*. Raleigh, North Carolina, USA, 2009, pp. 239–249.

[106] Zhenyu Sun, Xiuyuan Bi, Hai (Helen) Li, Weng-Fai Wong, Zhong-Liang Ong, Xiaochun Zhu, and Wenqing Wu. "Multi Retention Level STT-RAM Cache Designs with a Dynamic Refresh Scheme". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. Porto Alegre, Brazil: ACM, 2011, pp. 329–338. ISBN: 978-1-4503-1053-6. DOI: 10.1145/2155620.2155659. URL: http://doi.acm.org/10.1145/2155620.2155659.

[107] Y. Tsai and C. Chen. "Energy-Efficient Trace Reuse Cache for Embedded Processors". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 19.9 (2011), pp. 1681–1694. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2010.2055908.

[108] Hung-Wei Tseng and Dean M. Tullsen. "CDTT: Compiler-generated data-triggered threads". In: *International Symposium on High Performance Computer Architecture HPCA*. 2014. DOI: 10.1109/HPCA.2014.6835973.

[109] Hung-Wei Tseng and Dean M. Tullsen. "Data-triggered threads: Eliminating redundant computation". In: *International Conference on High-Performance Computer Architecture (HPCA)*. 2011. DOI: 10.1109/HPCA.2011.5749727.

[110] Hung-Wei Tseng and Dean M. Tullsen. "Software data-triggered threads". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*. 2012. DOI: 10.1145/2384616.2384668.

[111] J. Wang, X. Dong, Y. Xie, and N. P. Jouppi. "WAP: Improving non-volatile cache lifetime by reducing inter- and intra-set write variations". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 2013, pp. 234–245. DOI: 10.1109/HPCA.2013.6522322.

[112] Shasha Wen, Milind Chabbi, and Xu Liu. "REDSPY: Exploring Value Locality in Software". In: *International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS*. 2017. DOI: 10.1145/3037697.3037729.

[113] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. "The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools". In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 36:1–36:53. ISSN: 1539-9087. DOI: 10.1145/1347375.1347389. URL: http://doi.acm.org/10.1145/1347375.1347389.

[114] Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, Ram Rajamony, and Yuan Xie. "Hybrid Cache Architecture with Disparate Memory Technologies". In: *International Symposium on Computer Architecture (ISCA)*. 2009.

[115] Xiaoxia Wu, Jian Li, Lixin Zhang, Evan Speight, and Yuan Xie. "Power and performance of read-write aware hybrid caches with non-volatile memories". In: *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*. 2009.

[116] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. "Approximate Computing: A Survey". In: *IEEE Design & Test* 33.1 (2016), pp. 8–22. DOI: 10.1109/MDAT.2015.2505723.

[117] Hongbo Yang, Guang R. Gao, Andres Marquez, George Cai, and Ziang Hu. "Power and Energy Impact by Loop Transformations". In: *In Proceedings of the Workshop on Compilers and Operating Systems for Low Power 2001, Parallel Architecture and Compilation Techniques*. 2000.

[118] Sadegh Yazdanshenas, Marzieh Ranjbar Pirbasti, Mahdi Fazeli, and Ahmad Patooghy. "Coding Last Level STT-RAM Cache for High Endurance and Low Power". In: *IEEE Comput. Archit. Lett.* 13.2 (July 2014), pp. 73–76. ISSN: 1556-6056. DOI: 10.1109/L-CA.2013.8. URL: http://dx.doi.org/10.1109/L-CA.2013.8.

[119] Yangyang Ye, S Borkar, and V De. "A new technique for standby leakage reduction in high-performance circuits". In: *Symposium on VLSI Circuits. Digest of Technical Papers* (July 1998), pp. 40 –41.

[120] C. Zhang, F. Vahid, and W. Najjar. "A highly configurable cache architecture for embedded systems". In: *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* 2003, pp. 136–146. DOI: 10.1109/ISCA.2003.1206995.

[121] J. Zhao, C. Xu, and Y. Xie. "Bandwidth-aware reconfigurable cache design with hybrid memory technologies". In: *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2011, pp. 48–55. DOI: 10.1109/ICCAD.2011.6105304.

[122] Jishen Zhao, Cong Xu, Ping Chi, and Yuan Xie. "Memory and Storage System Design with Nonvolatile Memory Technologies". In: *IPSJ Transactions on System LSI Design Methodology* 8 (2015), pp. 2–11. DOI: 10.2197/ipsjtsldm.8.2.

[123] W. Zhao, E. Belhaire, Q. Mistral, C. Chappert, V. Javerliac, B. Dieny, and E. Nicolle. "Macro-model of Spin-Transfer Torque based Magnetic Tunnel Junction device for hybrid Magnetic-CMOS design". In: *2006 IEEE International Behavioral Modeling and Simulation Workshop*. 2006, pp. 40–43. DOI: 10.1109/BMAS.2006.283467.

[124] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. "Energy reduction for STT-RAM using early write termination". In: *International Conference on Computer-Aided Design*. ICCAD. San Jose, CA, USA, 2009.

[125] I. Ştirb and H. Ciocârlie. "Improving performance and energy consumption with loop fusion optimization and parallelization". In: *2016 IEEE 17th International Symposium on Computational Intelligence and Informatics (CINTI)*. 2016, pp. 000099–000104. DOI: 10.1109/CINTI.2016.7846386.