Compile-Time Silent-Store Elimination for Energy Efficiency: an Analytic Evaluation for Non-Volatile Cache Memory

Rabab Bouziane, Erven Rohou, Abdoulaye Gamatié

Compile-Time Silent-Store Elimination for Energy Efficiency: an Analytic Evaluation for Non-Volatile Cache Memory

Rabab Bouziane
Univ Rennes, Inria, CNRS, IRISA
Campus de Beaulieu, 35042 Rennes Cedex, France
first.last@inria.fr

Erven Rohou
Univ Rennes, Inria, CNRS, IRISA
Campus de Beaulieu, 35042 Rennes Cedex, France
first.last@inria.fr

Abdoulaye Gamatié
CNRS, LIRMM, Univ. Montpllier
191 rue Ada, 34095 Montpellier,France
first.last@lirmm.fr

ABSTRACT

Energy-efficiency has become very critical in modern high-performance and embedded systems. In on-chip systems, memory consumes an important part of energy. Emerging non-volatile memory (NVM) technologies, such as Spin-Transfer Torque RAM (STT-RAM), offer power saving opportunities, while they suffer from high write latency.

In this paper, we propose a fast evaluation of NVM integration at cache level, together with a compile-time approach for mitigating the penalty incurred by the high write latency of STT-RAM. We implement a code optimization in LLVM for reducing so-called *silent stores*, i.e., store instruction instances that write to memory values that were already present there. This makes our optimization portable over any architecture supporting LLVM. Then, we assess the possible benefit of such an optimization on the Rodinia benchmark suite through an analytic approach based on parameters extracted from the literature devoted to NVMs. This makes it possible to rapidly analyze the impact of NVMs on memory energy consumption. Reported results show up to 42 % energy gain when considering STT-RAM caches.

Keywords

Silent stores, LLVM compiler, energy-efficiency, non volatile memory, embedded systems

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 [8]. 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 [23] [2].

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 [20]. 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.

In this paper, 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* [14], 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 [10, 21] 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 therefore 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 [12]. 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 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 tradeoff on the Rodinia benchmark suite [4]. 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.

In the rest of this paper, Section 2 discusses some related work. Then, Section 3 describes the general principle and implementation of our silent store elimination approach at compile-time. Section 4 applies this approach to the Rodinia benchmark suite for evaluating possible gains in energy on different applications. Finally, Section 5 gives concluding remarks and perspectives to the current work.

2. RELATED WORK

There are a number of studies devoted to energy-efficiency of NVM-based caches. Smullen et al. [24] 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.

Sun et al. [25] proposed a hybrid L2 cache consisting of MRAM and SRAM, and employed migration based policy to mitigate the drawbacks of MRAM. The idea is to keep as many write intensive data in the SRAM part as possible to reduce the number of write operations to the STT-RAM part. Hu et al. [9] targeted embedded chip multiprocessors with scratchpad memory (SPM) and non volatile main memory. They exploited data migration and re-computation in SPM so as to reduce the number of writes on main memory. Zhou et al. in [31] proposed a circuit-level technique, called Early Write Termination in order to reduce write energy. The basic idea is to terminate earlier a write transaction whenever detected as redundant.

Migration-based techniques require additional reads and writes for data movement, which penalizes the performance and energy efficiency of STT-RAM based hybrid cache. Li et al. [18] addressed this issue through a compilation method called migration-aware code motion. Data access patterns are changed in memory blocks so as to minimize the overhead of migrations. The same authors [16] also proposed a migration-aware compilation for STT-RAM based hybrid cache in embedded systems, by re-arranging data layout to reduce the number of migrations. They showed that the reduction of migration overheads improves energy efficiency and performance.

We also promote 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 [29], 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 [26] 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 [27], 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 [28] 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 [14]), which applies independently from any specific programming model. This optimization increases the benefits NVMs as much as possible, by mitigating their drawbacks.

3. SILENT-STORE ELIMINATION

3.1 General principle

Silent stores have been initially proposed and studied by Lepak et al. [13]. 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. [1] 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 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.

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

Figure 1: 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.

3.2 Some microarchitectural 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 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. For example, the ARM Cortex-A7 is a (partial) dual-issue processor. 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 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.

On the other hand, instruction predication, e.g., supported by ARM cores, is another helpful mechanism. The execution of predicated instructions is controlled 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. Figure 1 (c) shows how this helps save an instruction.

At compiler optimization levels, newly introduced instruction 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, our load is automatically eliminated by further optimization, resulting in additional benefits. We observed in our benchmarks that this situation is actually rather frequent.

3.3 Analysis of 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 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 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×.

3.4 Implementation

Our compilation process is a middle-end framework. We focused on the compiler intermediate representation (IR) shown in Figure 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 3.

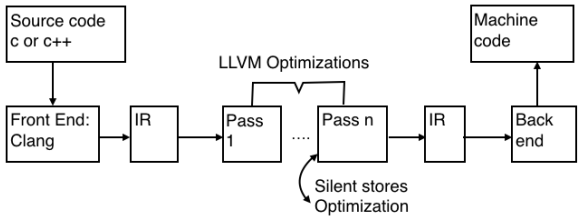


Figure 2: Implementation of the optimization in LLVM

Source	NVM parameters	Ratio
Wu et al. [30] Li et al. [16], [17]	Technology: 45 nm Read: 0.4 nJ Write: 2.3 nJ	5.75
Li et al. [15]	Technology: 32 nm Read: 174 pJ Write: 316 pJ	1.8
Cheng et al. [5]	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. [19]	Technology: 45 nm Read: 0.043 nJ Write: 3.21 nJ	75
Jog et al. [11]	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. [22]	Technology : 32 nm Read: 0.01 pJ/bit Write: 0.31 pJ/bit	31

Table 1: Relative energy cost of write/read in literature

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 if Figure 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 3.1 and illustrated in Figure 1.

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.

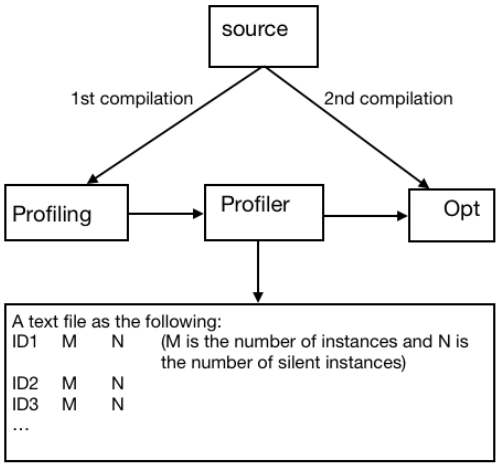


Figure 3: Design of the compilation framework

4. EVALUATION ON BENCHMARKS

In their seminal 2001 paper [14], 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 paper). Such silentness levels could typically benefit from our optimization.

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 [4], cross-compiled for ARM[1] 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.

4.1 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 stores's 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 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

[1]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.

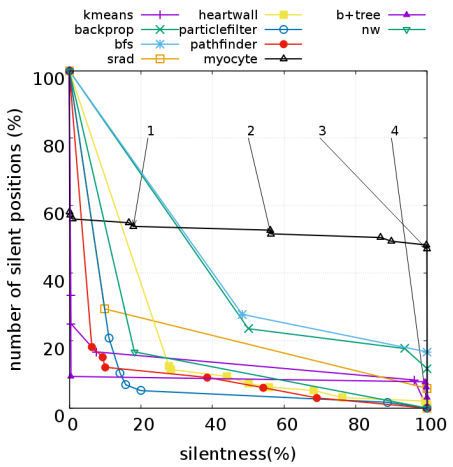


Figure 4: Percentage of silentness on the basis of the percentage of static positions in the code

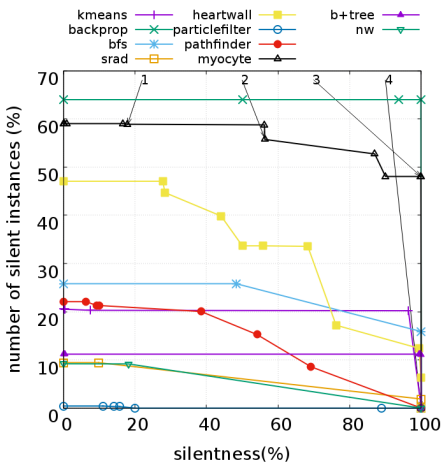


Figure 5: Percentage of silentness on the basis of the percentage of silent instances in the code

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 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).

In Figure 5, 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 5). 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.

4.2 Impact of the silent-store elimination

The impact of the silent-store elimination relies on different factors. As explained in Section 4.1, 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 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 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:

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

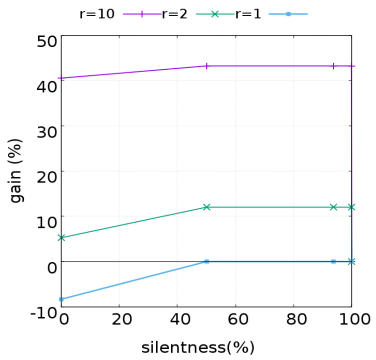
where P_i is the probability of silentness of different transformed stores. In order to get the effective gain, Δ is divided by $cost_{base}$ which is the energetic cost of all read and write operations before optimization:

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

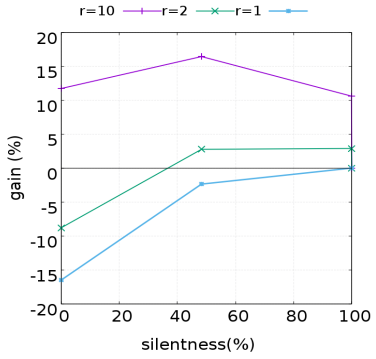
where N_R and N_W are respectively the number of load and write operations, obtained from the profiling. Considering $r = \frac{\alpha_W}{\alpha_R}$, then we obtain:

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

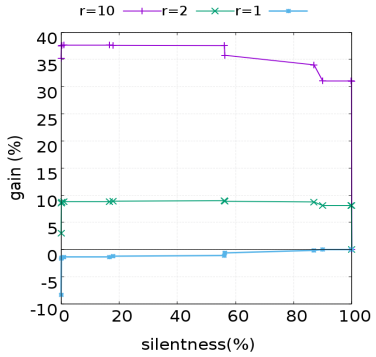
In Figure 6 and 7, we plot this energy gain for each benchmark and for three values of the ratio r: 10, 5, and 1.For each configuration, we plot the gain obtained when optimizing silent-stores whose silentness is greater than a given value (same x-axis as in previous figures).



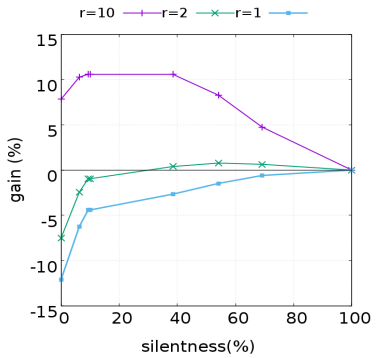
(a) backprop



(b) bfs



(c) myocyte



(d) pathfinder

Figure 6: Energy gain according to the silentness threshold of Rodinia applications, and their associated $r = \alpha_W/\alpha_R$ ratio: most sensitive applications.

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 6 are those which can benefit the most of silent store elimination given their silentness thresholds and considered r ratios. The remaining applications, displayed in Figure 7, 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 6 and 7).

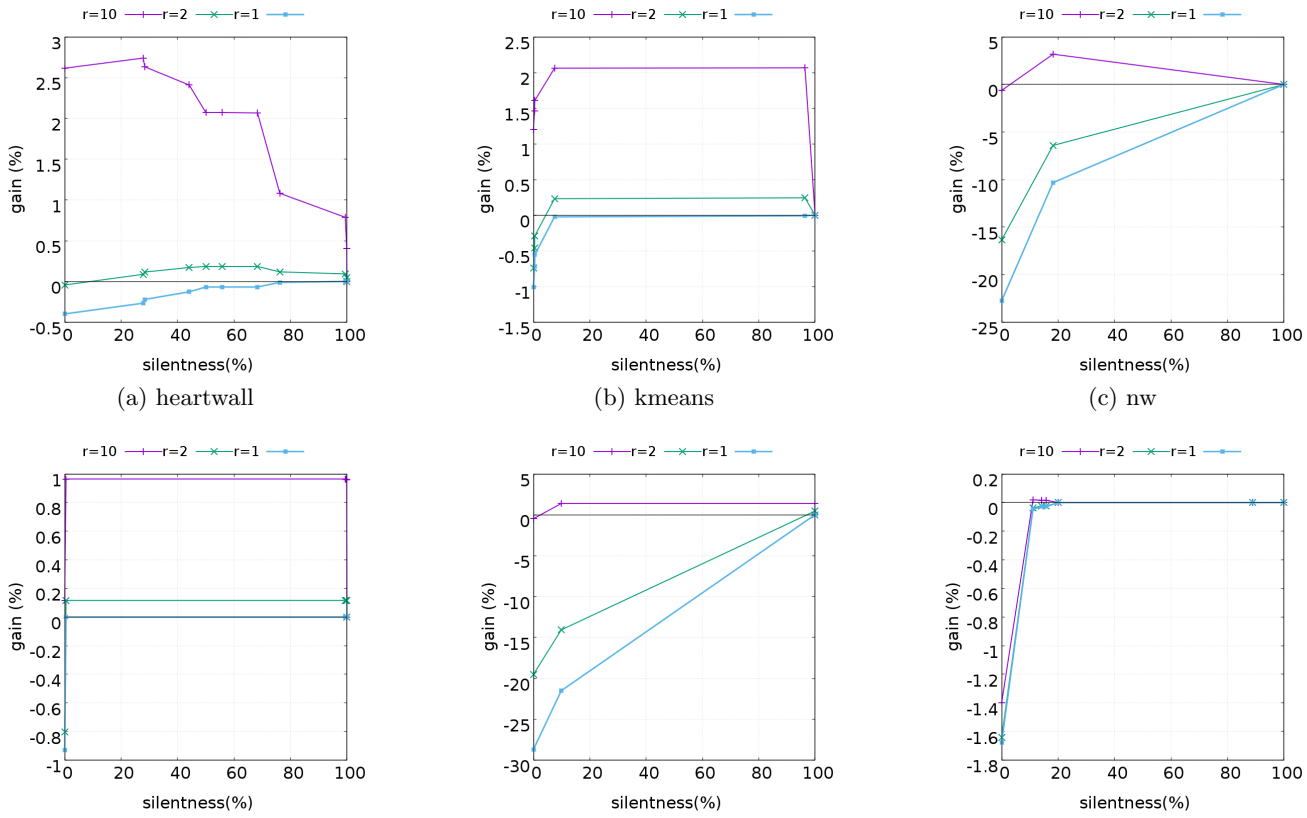
Depending on the silentness profile of the application, the gain can be fairly flat, as in *backprop*, *myocite* or *b+tree*, or vary significantly with the silentness threshold, as in *pathfinder*, or *heartwall*. In the latter case, the energy consumption critically depends on the choice of the threshold.

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.

5. CONCLUSION AND PERSPECTIVES

In this paper, we presented a rapid evaluation 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 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 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 tradeoff between the silentness threshold of stores in a given program



(a) heartwall (b) kmeans (c) nw

(d) b+tree (e) srad (f) particlefilter

Figure 7: Energy gain according to the silentness threshold of Rodinia applications, and their associated $r = \alpha_W/\alpha_R$ ratio: marginally sensitive applications.

and the energy cost ratio of memory accesses. Depending on the silentness of evaluated applications and typical ratios, the gain in memory consumed by memory can reach up to 42 %. While this paper 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.

This work will be extended by considering existing cycle-accurate simulation tools, combined with power estimation tools in order 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 [6], which is built on top of gem5 [3] and NVSim [7].

6. ACKNOWLEDGEMENTS

This work has been funded by the French ANR agency under the grant ANR-15-CE25-0007-01, within the framework of the CONTINUUM project.

7. REFERENCES

[1] 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.

[2] Luca Benini and Giovanni de Micheli. System-level power optimization: Techniques and tools. *ACM Trans. Des. Autom. Electron. Syst.*, 5(2), April 2000.

[3] 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. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[4] 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.

[5] 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. *Microprocessors and Microsystems*, 42, 2016.

[6] Thibaud Delobelle, Pierre-Yves Peneau, Abdoulaye Gamatie, Florent Bruguier, Gilles Sassatelli Sophiane Senni, and Lionel 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.

[7] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P. Jouppi. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 31(7):994–1007, 2012.

[8] Ricardo Gonzalez and Mark Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of solid-state circuits*, 31(9):1277–1284, 1996.

[9] 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.

[10] 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, 04 2014.

[11] 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.

[12] 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, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Kevin M Lepak, Gordon B Bell, and Mikko H Lipasti. Silent stores and store value locality. *Transactions on Computers*, 50(11), 2001.

[14] Kevin M. Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *International Symposium on Computer Architecture (ISCA)*, 2000.

[15] Jianhua Li, Chun Jason Xue, and Yinlong Xu. STT-RAM based energy-efficiency hybrid cache for CMPs. In *International Conference on VLSI and System-on-Chip (VLSI-SoC'11)*, 2011.

[16] 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 *International Symposium on Low Power Electronics and Design (ISLPED)*, 2012.

[17] 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. *Transactions on Very Large Scale Integration (VLSI) Systems*, 22(8), 2014.

[18] 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.

[19] Qingan Li, Yingchao Zhao, Jingtong Hu, Chun Jason Xue, Edwin Sha, and Yanxiang He. MGC: Multiple graph-coloring for non-volatile memory based hybrid scratchpad memory. *Workshop on Interaction between Compilers and Computer Architectures*, 2012.

[20] Sparsh Mittal and Jeffrey S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *Trans. Parallel Distrib. Syst.*, 27(5), 2016.

[21] Hiroki Noguchi, Kazutaka Ikegami, Keiichi Kushida, Keiko Abe, Shogo Itai, Satoshi Takaya, Naoharu Shimomura, Junichi Ito, Atsushi Kawasumi, Hiroyuki Hara, and Shigeji Fujita. 7.5 a 3.3ns-access-time 71.2w/mhz 1mb embedded stt-mram using physically eliminated read-disturb scheme and normally-off memory architecture, 02 2015.

[22] 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.

[23] 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. *Design & Test of Computers*, 18(3), 2001.

[24] 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 *International Symposium on High Performance Computer Architecture*, 2011.

[25] 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)*, 2009.

[26] Hung-Wei Tseng and Dean M. Tullsen. Data-triggered threads: Eliminating redundant computation. In *International Conference on High-Performance Computer Architecture (HPCA)*, 2011.

[27] Hung-Wei Tseng and Dean M. Tullsen. Software data-triggered threads. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2012.

[28] Hung-Wei Tseng and Dean M. Tullsen. CDTT: compiler-generated data-triggered threads. In *International Symposium on High Performance Computer Architecture HPCA*, 2014.

[29] 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.

[30] 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 Conference & Exhibition (DATE)*. IEEE, 2009.

[31] 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, 2009.