How Could Compile-Time Program Analysis help Leveraging Emerging NVM Features?

Rabab Bouziane, Erven Rohou, Abdoulaye Gamatié

▶ **To cite this version:**

Rabab Bouziane, Erven Rohou, Abdoulaye Gamatié. How Could Compile-Time Program Analysis help Leveraging Emerging NVM Features?. EDiS: Embedded and Distributed Systems, Dec 2017, Oran, Algeria. pp.1-6, 10.1109/EDIS.2017.8284031 . hal-01655195

HAL Id: hal-01655195

https://inria.hal.science/hal-01655195

Submitted on 4 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

How Could Compile-Time Program Analysis help Leveraging Emerging NVM Features?

Rabab Bouziane*, Erven Rohou* and Abdoulaye Gamatié†
*Univ Rennes, Inria, CNRS, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France – Email: first.last@inria.fr
†CNRS, LIRMM, Univ. Montpllier, 191 rue Ada, 34095 Montpellier, France – Email: first.last@lirmm.fr

***Abstract*—Beyond the fact of generating machine code, compilers play a critical role in delivering high performance, and more recently high energy efficiency. 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, 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. We review a number of compile-time analysis and optimization techniques, and how they could apply to systems in presence of NVMs. In particular, we consider the case of the reduction of the number of writes, and the analysis of variables lifetime for memory bank assignment of program variables.**

I. Introduction

The performance gap between the compute and memory parts 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 1 illustrates the typical access latency growth across the memory hierarchy by considering the *lmbench* benchmark [1]. 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 be also mitigate the aforementioned memory access bottleneck. In particular, compile-time optimization contributes in improving data locality in programs. Typically, it is the case of loop tiling or loop fusion transformations [2], 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 costly memory accesses.

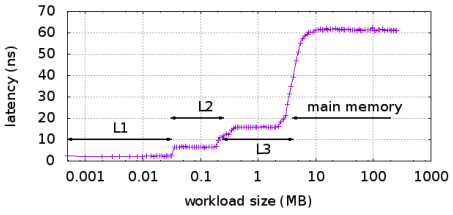

Fig. 1. Typical memory access latency (Intel Xeon i7-5600U, 2.6 GHz).

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 emerging non volatile memories (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 [3]. 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.

In this paper, 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) within memory hierarchy. As discussed above, both ingredients offer complementary advantages for meeting the performance and power consumption

requirements. Two main directions 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* in the current paper. We discuss the effectiveness of this first direction from the architecture and compiler viewpoints.
- 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.

More generally speaking, the current work aims to fill the gap between known compile-time techniques and successful NVM integration in upcoming energy-efficient computer architectures. Note that there are different types of NVMs, each with specific endurance and performance/energy characteristics. The silent stores optimization can be used with all NVMs. But since STT-RAM are quite mature as test chips already exist [4], [5] and show reasonable performance at device level compared to SRAM, we consider STT-RAM in this work.

II. Related work

There are several works that studied energy efficiency on NVMs-based hybrid cache using hardware based mechanisms to migrate data between NVM and SRAM memory blocks. 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. [6] addressed this issue through a compilation method called migration-aware code motion. This mechanism is designed to change the data access patterns in memory blocks so as to minimize the overhead of migrations. The same authors [7] 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. Hu et al. [8] 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.

Zhou et al. in [9] 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. [10] proposed an alternative approach for redesigning STT-RAM memory cells to reduce the high dynamic energy and write latencies. They lower the data retention time in NVM, which further enables to reduce the corresponding write current.

In this paper, we explore *compile-time* analyses and optimization as a possible alternative to leverage the low leakage current inherent to emerging non volatile memory 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. [11] and worst-case execution time analysis techniques [12].

III. Redundant write access elimination

A. Strict silent store elimination

Inspired by the initial work of Lepak et al. [13] on silent stores elimination through hardware mechanisms, for improving monoprocessor speedup and reduce multiprocessor data bus traffic, we consider a compile-time variant of this optimization. Our approach consists in transforming a given code by inserting *silentness* verification before each store that has been identified as potentially silent. As illustrated in Fig. 2, the verification includes the following instructions:

1) a load instruction at the store address;
2) a comparison instruction, to compare the written value to the already stored value;
3) a conditional branch 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

Fig. 2. 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.

We implemented the above transformation within the LLVM compiler at its intermediate code representation, through a two-step approach as follows: a profiling of silent stores based on observed memory accesses, followed by an optimization pass on the stores that are silent up to a given *silentness probability*, which expresses how often a store operation is silent. A frequently executed store operation that is often silent will have a high silentness probability, and therefore will be beneficial for an optimization.

Table I reports an evaluation of the suggested optimization on a subset of the Rodinia benchmark suite [14]. 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 [15]. 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 c_{write} and c_{read}, as follows:

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

These energy costs vary significantly depending on the underlying memory technology. Table II reports some values from literature. In this survey, the ratio in energy consumption

Benchmarks	Reads			Writes			Energy gain (%)			
	Original (N_R)	Optimized	δ_R	Original (N_W)	Optimized	δ_W	$r=1$	$r=5$	$r=10$	$r=75$
kmeans	548 491 642	548 809 242	+317 600	44 606 552	43 341 904	-1 264 648	0.16	0.78	1.24	2.43
backprop	46 853 587	47 152 381	+298 794	26 354 954	24 257 754	-2 097 200	2.46	5.70	6.66	7.76
heartwall	1 920 044 615	1 921 988 609	+1 943 994	20 198 643	14 983 887	-5 214 756	0.17	1.19	2.37	11.33
srad	76 371 774	76 526 801	+155 027	49 122 503	48 892 569	-229 934	0.06	0.31	0.38	0.45
b+tree	213 650 781	213 938 589	+287 808	24 094 567	23 894 250	-200 317	-0.04	0.21	0.38	0.73
bfs	181 175 711	180 697 730	-477 981	112 051 515	111 854 905	-196 610	0.23	0.20	0.19	0.18
pathfinder	205 090 257	206 350 923	+1 260 666	138 368 817	137 682 697	-686 120	0.17	0.24	0.35	0.47

TABLE I
NUMBER OF LOADS/STORES IN RODINIA BENCHMARKS PROFILED WITH GEM5 BEFORE AND AFTER OPTIMIZATION (ONLY STORES WITH 60 % SILENTNESS PROBABILITY ARE TRANSFORMED) AND ENERGY GAIN PROJECTION W.R.T. SPECIFIC NVM TECHNOLOGIES.

Source	NVM parameters	Ratio r
Wu et al. [16]	Technology: 45 nm Read: 0.4 nJ Write: 2.3 nJ	5.75
Li et al. [17]	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. [19]	Technology: 45 nm Read: 0.043 nJ Write: 3.21 nJ	75

TABLE II
RELATIVE ENERGY COST OF WRITE/READ IN LITERATURE

between writes and read varies from 1.8× to 75×. Considering the values computed in Table I, 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.

B. Micro-architectural and compilation considerations

While reducing the cost of memory accesses, the new instructions introduced by the optimization 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. Let us consider the example in Fig. 3. 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.

```
(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:
```

Fig. 3. A superscalar execution: the original sequential code is transformed to exploit parallelism by hiding the new instructions added by our transformation

Note that the independent subtraction instructions in Fig. 3 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. This instruction rescheduling is typical in *out-of-order* (OoO) cores. 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 Fig. 3 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 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 Fig. 4 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.

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

Fig. 4. Example of predicated execution

Thanks to the predication mechanism, the obtained code is one instruction less, thus limiting the risk to increase execution time. As a side effect, it also lowers the number of possible branch mis-predictions by eliminating the branch altogether.

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

1) The load instruction introduced by our optimization may be redundant when there is already another 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 some Rodinia benchmarks that this situation is actually rather frequent (see *kmeans, backprop, heartwall* and *srad* applications in Table I).
2) 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. This sometimes happens in benchmarked applications, such as *b+tree, bfs* and *pathfinder* (see Table I). It 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.

C. 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 [21]. 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.

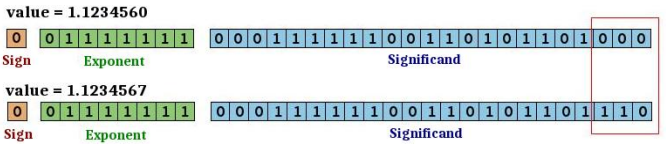

Fig. 5. Binary representation of floating-point numbers as defined by IEEE 754 – 32-bit floats. The mantissa (or significant) is 24-bit long. The two values differ only by their last 3 bits.

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 2 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 Fig. 5 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 Fig. 6: 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:
```

Fig. 6. Checking for Relaxed Silent-Stores

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 Fig. 7, 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 III).

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 by changed

[1]Modern branch predictors, e.g., TAGE [20], can detect complex patterns.

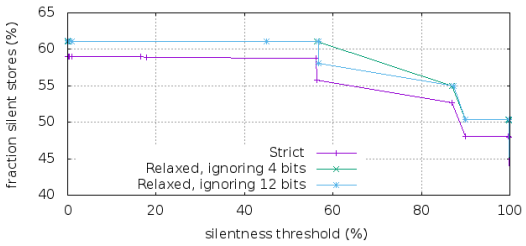

Fig. 7. Strict vs. relaxed silent-stores – Rodinia/*myocyte*

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 III
FRACTION OF SILENT STORES (STRICT AND RELAXED) AT SELECTED SILENTNESS THRESHOLDS – RODINIA/*myocyte*)

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 paper.

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

IV. EXPLOITING VARIABLE DATA RETENTION TIME

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 [3] for reliability concern. However, different designs can target shorter retention times as illustrated in Fig. 9 extracted from literature [22]. Typically, a 32 KB L1 cache with a retention time of 3.24 s leads to twice-less expensive write operations than an L1 cache with 4.27 years retention time, i.e., shorter retention time comes with a gain in access

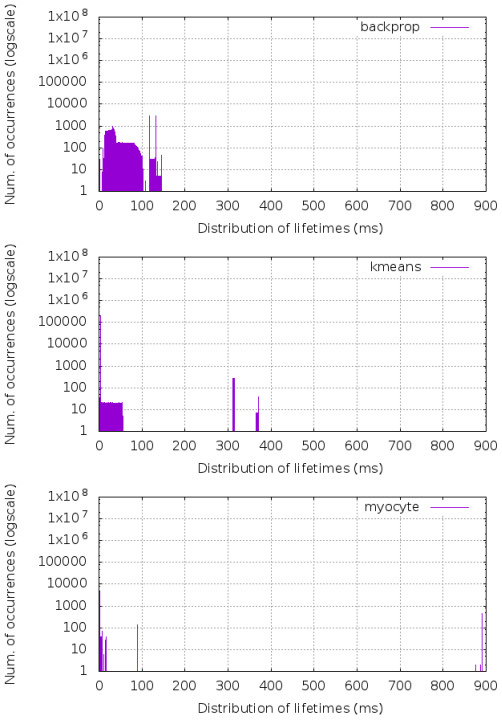

Fig. 8. Data lifetime distributions for *backprop, kmeans* and *myocyte*.

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 8 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.

Through Fig. 8, 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[2], configured for the hierarchy shown in Fig. 9 (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

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

	32KB (L1)						256KB (L2)	4MB (L2 or L3)					
	SRAM	lo1	lo2	lo3	md	hi	md1	SRAM	lo	md1	md2	md3	hi
Cell size (F^2)	125	20.7	27.3	40.3	22	23	22	125	20.7	22	15.9	14.4	23
MTJ sw time (ns)	/	2	1.5	1	5	10	5	/	2	5	10	20	10
Retention Time	/		26.5μs		3.24s	4.27yr	3.24s	/	26.5μs		3.24s		4.27yr
Read Lat (ns)	1.113	0.778	0.843	0.951	0.792	0.802	2.118	4.273	2.065	2.118	1.852	1.779	2.158
Read Lat (cycles)	3	2	2	2	2	2	5	9	5	5	4	4	5
Write Lat (ns)	1.082	2.359	1.912	1.500	5.370	10.378	6.415	3.603	3.373	6.415	11.203	21.144	11.447
Write Lat (cycles)	3	5	4	4	11	21	13	8	7	13	23	43	23
Read Dyn. Eng (nJ)	0.075	0.031	0.035	0.043	0.032	0.083	0.083	0.197	0.081	0.083	0.070	0.067	0.085
Write Dyn. Eng (nJ)	0.059	0.174	0.187	0.198	0.466	0.958	0.932	0.119	0.347	0.932	1.264	2.103	1.916
Leakage pow (mW)	57.7	1.73	1.98	2.41	1.78	1.82	14.24	4107	96.1	104	69.1	61.2	110

Fig. 9. Examples of cache memory configurations comparing SRAM and STT-RAM [22].

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.

V. Conclusion and perspectives

This paper presented some opportunities to fill the gap between known compile-time analysis and optimization techniques, and NVM integration in upcoming energy-efficient computer architectures. NVMs have a quasi-zero static power consumption thanks to their negligible leakage current. However, they have high read/write latency and power consumption compared to technologies such as SRAM. We advocated silent store elimination technique and worst-case execution time analysis as a solution. We showed how redundant write elimination and data lifetime analysis 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.

Future work includes a full validation of identified opportunities within an experimental full-system architecture simulation environment.

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.

References

[1] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1996, pp. 23–23.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, Dec. 1994.

[3] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. 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.

[4] 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," pp. 1–2, 04 2014.

[5] H. Noguchi, K. Ikegami, K. Kushida, K. Abe, S. Itai, S. Takaya, N. Shimomura, J. Ito, A. Kawasumi, H. Hara, and S. 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," pp. 1–3, 02 2015.

[6] Q. Li, L. Shi, J. Li, C. J. Xue, and Y. He, "Code motion for migration minimization in STT-RAM based hybrid cache," in *Computer Society Annual Symposium on VLSI*, 2012.

[7] Q. Li, J. Li, L. Shi, C. J. Xue, and Y. 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.

[8] J. Hu, C. J. Xue, Q. Zhuge, W. Tseng, and E. H. Sha, "Data allocation optimization for hybrid scratch pad memory with SRAM and nonvolatile memory," *Trans. VLSI Syst.*, 2013.

[9] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy reduction for STT-RAM using early write termination," in *International Conference on Computer-Aided Design*, ser. ICCAD, 2009.

[10] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient STT-RAM caches," in *Int. Symp. on High Performance Computer Architecture*, 2011.

[11] K. M. Lepak and M. H. Lipasti, "On the value locality of store instructions," in *Int. Symp. on Computer Architecture (ISCA)*, 2000.

[12] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, 2008.

[13] K. M. Lepak, G. B. Bell, and M. H. Lipasti, "Silent stores and store value locality," *Transactions on Computers*, vol. 50, no. 11, 2001.

[14] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *International Symposium on Workload Characterization (IISWC'10)*, 2010.

[15] 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.

[16] X. Wu, J. Li, L. Zhang, E. Speight, and Y. Xie, "Power and performance of read-write aware hybrid caches with non-volatile memories," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2009.

[17] J. Li, C. J. Xue, and Y. Xu, "STT-RAM based energy-efficiency hybrid cache for CMPs," in *Int. Conf. on VLSI and SoC (VLSI-SoC'11)*, 2011.

[18] W.-K. Cheng, Y.-H. Ciou, and P.-Y. Shen, "Architecture and data migration methodology for L1 cache design with hybrid SRAM and volatile STT-RAM configuration," *Microprocessors and Microsystems*, vol. 42, 2016.

[19] Qingan Li et al., "MGC: Multiple graph-coloring for non-volatile memory based hybrid scratchpad memory," *Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, 2012.

[20] A. Seznec and P. Michaud, "A case for (partially) TAgged GEometric history length branch prediction," *Journal of Instruction Level Parallelism*, vol. 8, 2006.

[21] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2016.

[22] Sun, Zhenyu et al., "Multi retention level STT-RAM cache designs with a dynamic refresh scheme," in *International Symposium on Microarchitecture*, ser. MICRO-44. ACM, 2011, pp. 329–338.