Static and Dynamic Co-Optimizations for Blocks Mapping in Hybrid Caches

Yu-Ting Chen, Jason Cong, Hui Huang, Chunyue Liu, Raghu Prabhakar, and Glenn Reinman
Computer Science Department, University of California, Los Angeles
Los Angeles, CA 90095, USA
{ytchen, cong, huihuang, liucy, raghu, reinman}@cs.ucla.edu

ABSTRACT
In this paper, a combined static and dynamic scheme is proposed to optimize the block placement for endurance and energy-efficiency in a hybrid SRAM and STT-RAM cache. With the proposed scheme, STT-RAM endurance is maximized while performance is maintained. We use the compiler to provide static hints to guide initial data placement, and use the hardware to correct the hints based on the run-time cache behavior. Experimental results show that the combined scheme improves the endurance by 23.9x and 5.9x compared to pure static and pure dynamic optimizations respectively. Furthermore, the system energy can be reduced by 17% compared to pure dynamic optimization through minimizing STT-RAM writes.

Categories and Subject Descriptors
B.3.2 [**MEMORY STRUCTURES**]: Design Styles – *Cache memories*.

General Terms
Algorithm, Design.

Keywords
L2 cache, Hybrid cache, STT-RAM, Endurance, Energy.

1. INTRODUCTION
The traditional SRAM caches suffer from huge leakage power, which dominates the total energy consumption in the on-chip memory system. To alleviate this problem, emerging non-volatile memory technologies such as phase-change RAM (PRAM) and spin-torque transfer magnetoresistive RAM (STT-RAM) are used as alternative on-chip memory with the advantages of low leakage and high density. However, non-volatile memories suffer from the challenges of limited endurance, higher write latency and energy. Compared to PRAM, STT-RAM has significantly higher endurance (10^9 versus 10^{12} write cycles) and shorter write latency [1] and is much more promising in the last-level cache design [2] [3][4][5][6][7]. Moreover, due to the intensive writes of caches, hybrid caches consisting of both SRAM and STT-RAM are investigated [2][3][4][7], where the SRAM can accommodate write-intensive data and the STT-RAM can accommodate other data with its dense capacity.

ISLPED'12, July 30–August 1, 2012, Redondo Beach, California, USA.

The STT-RAM endurance is an important issue to be considered in the last-level cache design. Although ITRS predicts the write cycles of STT-RAM will be 10^{15} at 2024 [1], the best available write cycles of STT-RAM are 4 x 10^{12} at present [5]. Supposed we execute *segmentation* [8], a medical imaging application, on a 4GHz CPU with 32 KB L1 cache, 2MB STT-RAM L2 cache continuously, the lifetime of a STT-RAM cache can last only 2.17 years without any optimizations applied. The endurance problem becomes even worse in the multi-level cell (MLC) STT-RAM technology [5]. Block placement optimizations is very important to shrink the large endurance gap between STT-RAM and SRAM in a hybrid cache. Recent work considers either static or dynamic schemes to optimize the block placement to reduce the average write frequency to STT-RAM cells, while maintaining the overall performance by making use of higher density of STT-RAM. Some of the proposed approaches targeted at PRAM, and those ideas can also be applied to STT-RAM with the same objective.

The first category of the prior work uses static schemes. In [9] the authors introduce data migration and re-computation to reduce the write frequency on PRAM main memory. In [10], the partitioning of the application working set into SRAM and PRAM can reduce 79% of the writes to PRAM.

The second category of the prior work uses dynamic schemes. Recent work in [5] uses periodically set-remapping to distribute the writes among sets in a STT-RAM cache. Another set of work migrates the write-intensive cache blocks to other cache lines in the same/different cache set or in the SRAM in order to reduce the average write frequency of the STT-RAM cache lines [4].

However, there exist intrinsic limitations in both approaches, which cannot be resolved independently – The static optimization decisions are made at compile time without run-time information, thus compiler may generate misleading hints to the hardware. On the other hand, pure dynamic optimization use blocked-based counter structures to learn the memory reference patterns on-the-fly. However, the dynamic scheme lacks a global view of the whole program and has no knowledge to future access pattern.

In this paper, we propose a combined approach in which the static and dynamic optimizations can compensate to each other. The compiler tries to provide data placement hints to hardware to reduce STT-RAM write frequency, while the hardware is designed to be able to correct compiler hints based on runtime cache behavior. Experimental results show that the combined scheme improves the endurance by 23.9x and 5.9x compared to pure static and pure dynamic optimizations, respectively, while maintaining similar performance. Furthermore, the system energy can be reduced by 17% compared to pure dynamic optimization since STT-RAM writes are reduced through initial placement from the proposed compiler flow.

2. PROBLEM FORMULATION

In this work, we assume that the L2 cache is a hybrid cache architecture with 4-way SRAM and 12-way STT-RAM, which is similar to the setting described in [4][7]. The block-level initial placement and dynamic migration is allowed to place the data blocks in either SRAM or STT-RAM. The initial placement is given by the compiler hints and the runtime cache pressure while the dynamic migration is designed with hardware mechanisms. Our assumptions in detail are described in Section 4.1 and Section 4.3 for better illustration of our co-optimization strategy.

The objective of this work is to improve the endurance of the hybrid cache and reduce system energy while maintaining performance through the combined scheme. Another meaningful objective is to co-optimize performance and energy while under the endurance constraint. A storage-efficient way to monitoring the endurance of each cache block is required under the second scenario. The discussion of this formulation is not included in this work but may be worthwhile for further investigation.

3. MOTIVATIONAL EXAMPLES

In this section, we use real-life examples to illustrate how pure static optimization and dynamic optimization may produce sub-optimal block placement decisions in a hybrid cache design.

The deficiency of pure static optimization comes from the fact that the runtime write frequencies of L2 cache (which is the hybrid last-level cache in our evaluated system) blocks are input-dependent, which cannot be fully obtained offline. First, the input data may change the control flow of the program, and this will change the write frequency of the data that are affected by the control flow variation. Second, since part of the writes to the L2 cache come from the write-back operations from the L1 caches, the compiler cannot accurately capture the L2 cache write behavior with the existence of the L1 caches. For example, given the LRU replacement policy, the data which is written fewer times in the code may be frequently evicted by the L1 cache and behaves much more write-intensive in the L2 cache than other data which are written more frequently. Note that life time of the STT-RAM mainly depends on the peak write count of all the cells [4][5]. Even static optimizations can reduce the total STT-RAM writes compared to dynamic optimizations via global optimization, the potential mis-predictions can still severely degrade the STT-RAM lifetime, since there is no dynamic scheme to migrate mis-predicted write-intensive blocks into SRAM. This causes a large peak write count to this cell. As an example, Table 1 shows the STT-RAM cell write count distribution of the *segmentation* application [8] for both of the pure static [10] and the pure dynamic [4] scheme. The peak write count of static optimization is significantly larger than that of the dynamic one.

Table 1: STT-RAM write count distribution

#writes	0-100	100-200	200-300	300-400	400-1000	1000-5000	>=5000	max
static	12262	6	5	0	0	10	5	5470
dynamic	12207	38	16	27	0	0	0	395

The deficiency of dynamic migration comes from the fact that it lacks the future memory access information and highly relies on the application to exhibit a bipolar L2 write frequency patterns – the L2 cache blocks are either rarely written or intensively written. Then the write-intensive blocks can swap their places with the rarely written blocks through dynamic migration. However, based on our observation, not all the applications have such characteristics, especially in the three medical imaging applications [8] used in our study. As shown in Figure 1, most of the blocks are uniformly written 2-3 times. Under this circumstance, if the migration threshold is set to be higher than 3, then there will be little migration, both the SRAM and STT-RAM will be evenly written based on the LRU scheme, which may impair the endurance. However, even if the migration threshold is set to be 2 or 3, the migration for most blocks does not save any write, since most blocks behave similarly. A correct approach is to place all these streaming accessed blocks into the SRAM, which can be obtained via static optimization in the compiler. After that, the expensive writes on STT-RAM can be significantly reduced and thus dynamic energy can reduced.

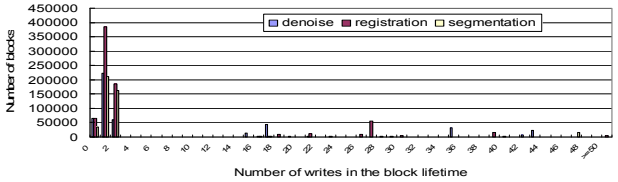

Figure 1: Write frequency distribution of the L2 cache blocks

To overcome these limitations while taking the advantage of both static and dynamic schemes, we use a combined strategy: the compiler tries to guide the hardware in order to rapidly achieve the desired placement, while the hardware corrects the compiler hints based on the run-time cache behavior. To the best of our knowledge, we are the first one taking such a hybrid approach.

4. THE COMBINED APPROACH

4.1 Compiler Support

In this work, we develop an automatic compilation flow to generate data placement hints for each memory reference. Here, we assume LRU replacement policy is used and L2 is an inclusive cache with the same block size of L1, which is widely used in modern processors because of easy coherence implementation.

Similar to [10], our compiler tends to place write-intensive references into SRAM and non-write-intensive data into STT-RAM. Based on our inclusive cache assumption, the write accesses on L2 STT-RAM cells occur at only two situations: (1) L1 dirty evictions due to L1 cache replacement and (2) L2 cache replacement. However, the work in [10] assumes there is no cache in the memory system, thus does not consider the effect of higher-level (L1) cache on the memory access behavior. Figure 2 shows an example code and its corresponding memory access behavior. We can find that both arrays *A* and *B* are written twice. However, since array A is more frequently accessed and can be kept in the L1 cache (we assume LRU replacement policy is used here), neither of the two writes falls into the L2 cache. On the other hand, since array *B* is evicted from the L1 cache before its next access, one write-back operation will be issued into L2.

loop 1: A[i] = …; B[i] = …; (*write array A and B in L1*)
loop 2: … = A[i] …; (*read array A in L1*)
loop 3: … = C[i] …; (*array B is evicted into L2*)
loop 4: A[i] = …; B[i] = B[i]…; (*write array A and B in L1*)

Figure 2: One sample code and its memory access behavior

To capture this effect, we use the concept of *memory reuse distance* (*MRD*) [11], which equals the total size of unique data elements accessed between two references to *X*. A larger memory reuse distance of *X* implies that *X* will not be accessed in the near future, and thus *X* is more likely to be evicted from the L1 cache.

Definition 1: For an write operation w of memory instruction X, assuming the future access sequence of X is w, r_1, r_2, ..., r_n, w', etc (r and w corresponds to read and write operation). w is called an *L1-writeback write* if one of the following conditions is satisfied: (1) there exists $MRD(r_i, r_{i+1}) > dist_{L1}$ (i = 1, ..., n-1) (2) $MRD(r_n, w') > dist_{L1}$ (3) $MRD(w, r_1) > dist_{L1}$. ($dist_{L1}$ is the average reuse distance to keep X in L1 cache)

From Definition 1 we can see, if the memory reuse distance between two accesses into a dirty data X is larger than a threshold value $dist_{L1}$, the compiler will treat the first write (w) to X as a *L1-writeback*, since X will be written back into L2. The other set of L2 write accesses comes from L2 misses and data are written into L2 from main memory. Here we use $dist_{L2}$ to indicate the average reuse distance to keep X in L2 cache. For two adjacent accesses to X, if the memory reuse distance between them is larger than $dist_{L2}$, the compiler will treat the second access as a L2 miss, which will introduce one L2 write operation.

In our flow, we provide a 2-bit compiler hint for each memory instruction to guide its data placement in L2 cache. For each access to reference X, we count the total number of future L2 writes to X including both L1-writeback writes and L2 misses. If there are frequent L2 writes, our compiler will generate hint "01" for X. On the other hand, hint "00" is generated to indicate that X will not be written frequently. For those accesses that the compiler cannot analyze accurately (e.g., due to unknown loop bound), hint "1x" is generated and the data placement is controlled by hardware. Note that a memory instruction in a regular loop is accessed multiple times with repeated access patterns [12], therefore we can apply the same hint to all the accesses to it. It should be noted that the compiler just tries its best to predict the write frequency. The value of $dist_{L1}$ and $dist_{L2}$ can be obtained from profiling on representative input or set to a fixed value by default. However, it is not feasible to profile all input sets. In this work, a conservative approach is used to set $dist_{L1}$ to L1 set associativity. This can ensure that data X will not be evicted out from L1 between two accesses with reuse distance less than $dist_{L1}$. Since the compiler cannot make an optimal decision without knowing the runtime cache behavior, these generated hints may not be followed in the hardware. We will discuss our combined scheme in Section 4.3.1.

4.2 Compiler-Hardware Interface

In our implementation, the compiler passes the hints to the hardware through setting two bits in the 32-bit instruction code. We assume that there are two extra bits in each memory instruction that the compiler can use to assist the run-time cache block replacement. Existing architectures already use these kinds of extra bits in the instruction, such as the *prefetch and evict-next* instruction in the Alpha 21264. We believe that, in most architectures, the increasing speed gap between memory and processor will justify the inclusion of additional bits in the instruction code to facilitate the reduction of this gap. Once a memory reference instruction is executed, if this is a L1 cache hit, these 2 bits will be discarded. If this is a L1 miss, then the bits will be passed to the L2 cache controller, if this is a L2 hit, then these 2 bits will be discarded; if this is a L2 miss, these 2 bits will be used as hints for the initial placement of the new block.

4.3 Hardware Support

In this work, we use a 1MB 16-way hybrid cache including a 4-way SRAM data array and a 12-way STT-RAM data array similar to the configurations used in [2][4][7]. The asymmetric configuration is chosen since smaller SRAM contributes less leakage while the bigger STT-RAM provides the advantage of higher density. We use separate replacement units on SRAM and STT-RAM in order to perform block replacement in these two arrays independently. We also introduce a global replacement unit for them in order to perform a global replacement among them if required. All of these replacement units use LRU policy.

4.3.1 Capacity-Pressure and Compile Hints based Initial Placement

The initial block placement decision is made based on both of the compiler hint and also the SRAM/STT-RAM capacity pressure monitored in the hardware.

Before discussing the decision making process, we first show our capacity pressure assessing hardware. To assess the SRAM/STT-RAM capacity pressure, we introduce two additional hardware structures: *missing tags* (MTs) and *MT counters*. The proposed structures are similar to the missing tags [12] and victim tags [14]. MTs and MT counters are integrated with the tag array design, as shown in Figure 3. In addition to the original 4-way SRAM tag array and the 12-way STT-RAM tag array, a 4-way SRAM MTs and a 4-way STT-RAM MTs are introduced. Moreover, for each cache set, there are a SRAM MT counter and a STT-RAM MT counter, and these counters indicate the capacity pressure of the SRAM and STT-RAM portions in that cache set, respectively. Note that the sizes of MTs are the same for both SRAM and STT-RAM arrays to provide similar pressure monitoring criterion.

We use the SRAM MT to illustrate the MT and MT counter functionality, and the STT-RAM MT and MT counter work in the same way. When a cache miss occurs in the SRAM, the tag of the victim block will overwrite the LRU tag in the same set in SRAM MTs and be marked as most recently reused tag. If there is a miss in the SRAM array and there is a hit in the SRAM MTs, this indicates that a potential hit will occur if the requested block were placed in STT-RAM. Then the SRAM MT counter in the corresponding cache set is incremented by one.

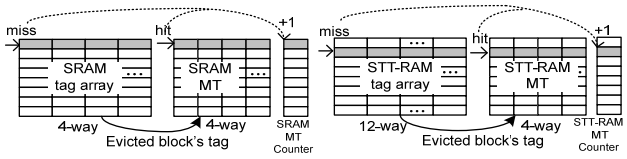

Figure 3: SRAM and STT-RAM missing tag and counter

We use an interval-based assessing approach, i.e., the value of the MT counters in the current interval will be used to guide the initial placement in the next interval. Considering the less-frequent access to the L2 cache, the interval length cannot be too short, but it can also be too long in terms of timely assessment. In this work, we set it to be 10K cycles. At the end of each 10K cycles, the value of all the MT counters will be evaluated to fill the *capacity pressure table* (CPT). The number of entries of CPT equals to the number of sets in the cache and each entry contains two bits: the SRAM capacity pressure and the STT-RAM capacity pressure of that cache set. If the value of the SRAM (STT-RAM) MT counter of a cache set is greater than a threshold, then the SRAM (STT-RAM) bit for that set in the CPT is set to 1 (*high*), otherwise set to 0 (*low*). Then all the MT counters are reset to 0. In the next interval, the CPT is accessed together with the compiler hints to decide the initial block placement.

Given the capacity pressure from the CPT and also the compiler hints, the L2 cache controller makes the initial placement as

shown in Table 2. If a block is going to be placed in SRAM (STT-RAM), then the LRU replacement unit of SRAM (STT-RAM) will be triggered to evict the victim in that cache set of SRAM (STT-RAM). If a block is going to be placed globally, the global LRU replacement unit is triggered to evict the LRU block of all the SRAM and STT-RAM cache lines in that cache set.

Table 2: Initial placement decision based on compiler hints and SRAM/STT-RAM capacity pressure

Capacity pressure		Compiler hint		
SRAM	STT-RAM	infreq write	freq write	unknown
High	**Low**	STT-RAM	STT-RAM	STT-RAM
Low	**High**	SRAM	SRAM	SRAM
High	**High**	STT-RAM	SRAM	Global
Low	**Low**	SRAM	SRAM	SRAM

4.3.2 Write-Frequency based Dynamic Migration

As pointed out in Section 4.1, the compiler hints are not absolutely accurate due to the input variation and the L1 cache impact. Moreover, according to Section 4.3.1, when capacity pressure unbalance occurs, blocks may be initially placed in the less-intensive used portion of the hybrid cache, instead of based on the write-frequency of the block itself, as shown in Table 2. Thus it is possible that a block is incorrectly initially placed.

We use dynamic migration to correct the initial placement by migrating the actually write-intensive STT-RAM data blocks to SRAM. We use the dynamic migration scheme similar to [4], which is briefly described as follows. Each L2 cache block is associated with a saturate 2-bit *write counter* to indicate the number of writes during its on-chip lifetime. If the write counter of a STT-RAM block saturates (three writes), the migration unit will check the write counters of the SRAM blocks in the same cache set. If there is any counter that is less than 3, then the corresponding SRAM block is swapped with that STT-RAM block. After that, all the write counters in this cache set are reset to 0. If the counters of all the SRAM blocks in a set are saturated, no migration will be performed. Therefore, the possibility that another write-intensive block could be swapped from the SRAM back to STT-RAM is avoided.

In sum, in our combined approach, if the compiler provides correct hints, the hardware can use them to rapidly achieve correct block placement. If compiler makes mis-predictions, the hardware corrects the compiler hints as shown in Table 3. Note that all the hardware corrections are automatically triggered by our introduced hardware counters.

Compiler mis-predictions: (a) Mis-predicts some write-intensive blocks as non-write-intensive. (b) Generates larger percent of non-write-intensive blocks that it actually is. (c) Generates larger percent of write-intensive blocks that it actually is.

Hardware corrections: (i) Distributes blocks to STT-RAM. (ii) Distributes blocks to SRAM. (iii) Migrates write-intensive blocks from STT-RAM to SRAM.

Table 3: Hardware corrections to the compiler mistakes

Compiler mis-predictions			Hardware corrections		
a	**b**	**c**	**i**	**ii**	**iii**
X					X
	X			X	
		X	X		X
X		X	X		X
X	X			X	X

5. EVALUATION METHODOLOGY

5.1 Compilation and Simulation Infrastructure

The compiler support for hint generation is implemented based on LLVM compiler infrastructure [15]. Omega library [16] is used in this flow to perform memory dependency analysis. Given a source program written in C/C++, we parse it into LLVM IR using LLVM's frontend. All standard optimizations in O3 are applied. Our hint-generating flow is invoked as a pass on the optimized LLVM intermediate representation (IR) code and will automatically generate data placement hints for each load/store instruction. We also modify LLVM backend to emit hint-included load/store instructions in the final assembly code. A potential issue of this LLVM frontend analysis is that some load/store instructions cannot be captured in IR level. For example, the loads/stores in pre-compiled library functions cannot be analyzed under this framework. Moreover, the loads/stores from operating system cannot be analyzed during compile time. Therefore, a hardware support mentioned in Section 4.3 is required to provide better optimization.

Table 4: Simics/GEMS simulator configurations

Core	Sun UltraSPARC-III Cu processor core
L1 Instruction/ Data Cache	32KB, 2-way set-associative, 64-byte block, 2-cycle access latency, pseudo-LRU
L2 Cache (Hybrid cache)	1MB, 16-way set-associative (4-way SRAM, 12-way STT-RAM), 64-byte block, access latency: 10-cycle for SRAM, 11-cycle (read) and 30-cycle (write) for STT-RAM
Main Memory	4GB, 320-cycle access latency

We extend the full-system cycle-accurate Simics [17] and GEMS [18] simulation platform to model the proposed hardware support. The system configurations of SIMICS/GEMS are shown in Table 4. We obtain the energy data of the SRAM array and MTs/MT counters through Cacti 6.5 [19] with 32nm process technology at 330K. The energy data of the STT-RAM array are obtained from NVSim [20]. Table 5 shows the energy model we use in our evaluation. Note that the low leakage cells (*itrs-lstp*) are used in SRAM data array and tag array. For peripheral circuitry, we use high performance cells (*itrs-hp*) to optimize performance and area. Note that we also try to implement the peripheral circuitry with low leakage cells for further leakage minimization. However, we observed that considerable area overhead may arise since the width of an *itrs-lstp* transistor should be large enough to provide the enough current for STT-RAM write operation.

Table 5: Energy/power data of the evaluated hybrid cache

	Read energy	**Write energy**	**Leakage power**
SRAM (4-way)	0.0603nJ	0.0603nJ	15.017mW
STT-RAM (12-way)	0.231nJ	1.306nJ	11.173mW
MTs (8-way)	0.0020nJ	0.0020nJ	2.805mW

5.2 Benchmarks

Our testbenchs consist of eight benchmark applications, which have been carefully chosen to represent memory intensive algorithms in the fields of data processing, massive communication, scientific computation and medical applications. The benchmark applications include three memory-intensive applications from SPEC2006 [21] (*bzip2, mcf* and *lbm*) and five applications from the medical imaging domain [8].

5.3 Reference Schemes

To demonstrate the effectiveness of our combined scheme (***combined***), we compare to two representative prior approaches:

Pure static optimization (*static*): The hardware will strictly follow the compiler-generated block placement hint. The compiler hints are generated based on the approach proposed in [10], and we further take the effect of L1 cache into consideration using the techniques discussed in Section 4.1.

Pure dynamic optimization (*dynamic*): We use the dynamic migration scheme proposed in [4]. Our dynamic migration scheme in Section 4.3 uses this scheme with the same migration threshold as 3. There is no compiler hint in this scheme.

Note that the energy overhead of MTs and MT counters is only applied on the *combined* scheme.

6. RESULTS

6.1 Endurance

In this work, we assume that the maximum write cycles of a STT-RAM cell is 4 x 10^{12} [5]. We assume that a workload continuously runs on the system. To model the endurance in a more sophisticated way, one can provide a loading factor, which is the percentage of the overall runtime occupied by the workload. The lifetime is measured from the start of the simulation until the first STT-RAM line becomes defective, which is similar to the estimation methodology proposed in [4] and [5].

Figure 4 demonstrates the lifetime which is normalized to the *static* scheme. The *static* scheme typically performs the worst among the three schemes (up to 1.2x~148x worse than the *combined* scheme). This is because that once a compiler mis-predicts a write-intensive block as a non-write-intensive one and places it into the STT-RAM, this block will be intensively written and there is no dynamic migration to mitigate it. The lifetime of the STT-RAM mainly depends on the peak write count of the cells. The exceptions are *fft*, *lbm* and *denoise* where the program only have negligible input-variation, so that the *static* scheme can have longer lifetime than the other two schemes. Note that the *static* scheme can only reduce the total writes instead of the peak write count among all blocks, as shown in Figure 5. Therefore, the *static* scheme is the worst in terms of endurance but it can save STT-RAM write energy, which is discussed in Section 6.2.

With the dynamic migration to average the writes to STT-RAM blocks, the *dynamic* scheme achieves up to 14x improvement of lifetime compared to the *static* scheme. However, the reduction of the peak write count of STT-RAM is accompanied with the cost of much more total STT-RAM writes, since it lacks global information to reduce the total STT-RAM writes. Figure 5 shows that *dynamic* scheme has 1.6x~36.6x more STT-RAM writes than the *static* scheme. In cases of *fft* and *lbm*, the data blocks are all uniformly written less than 3 times on either SRAM or STTRAM. Therefore, there is little migration in the *dynamic* scheme and it has a life time which is only 4%~13% of that of the *static* scheme.

The *combined* scheme has a 1.6x~14.7x lifetime compared to that of the *dynamic* scheme. By following the correct compiler hints, the *combined* scheme rapidly achieves the optimal block placement without additional migrations, especially in the cases where most of the blocks are uniformly written less than two or three times, as shown in the motivational examples in Section 2. This can save both the peak write count and also the total writes of the STT-RAM. Although the *combined* scheme has 0.8x~4.1x more total STT-RAM writes than *static*, it achieves 1.2x~148x lifetime due to averaging the writes to the STT-RAM cells (except *fft* where *static* has a 1.8x longer lifetime than *combined*).

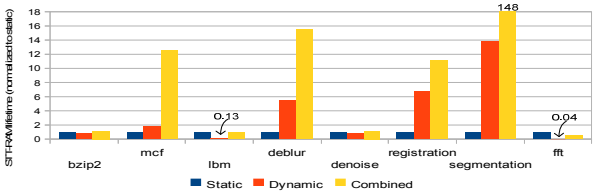

Figure 4: Comparison results of STT-RAM lifetime

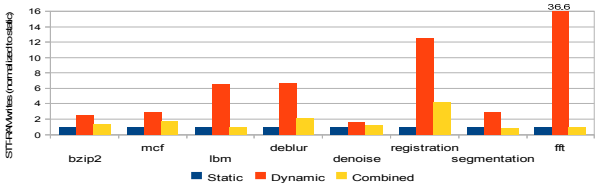

Figure 5: Comparison results of total STT-RAM writes

6.2 Energy

Figure 6 shows the distribution of hybrid cache (L2 cache) energy that is normalized to the *static* scheme. The leakage consumption of three schemes is similar. This is because leakage is proportional to program runtime and the runtime (as shown in Figure 7) of the three schemes is similar. Therefore, the key factor that influences the system energy is the L2 STT-RAM dynamic energy. The *static* scheme has the least energy, because the reduced STT-RAM writes (as shown in Figure 5) bring in considerable dynamic energy savings. Without the hints of initial placement, a large number of writes arises in the *dynamic* scheme, leading to 9%~80% energy overhead (38% overhead on average) compared to the *static* scheme.

The *combined* scheme achieves similar energy consumption to that of the *static* scheme (7%~20% energy overhead, 11% overhead on average) and outperforms the *dynamic* scheme (2%~39% energy reduction, 17% reduction on average). Note that the energy overhead of the *combined* scheme comes from both the leakage of the introduced MTs and the extra dynamic STT-RAM writes energy compared to the *static* scheme.

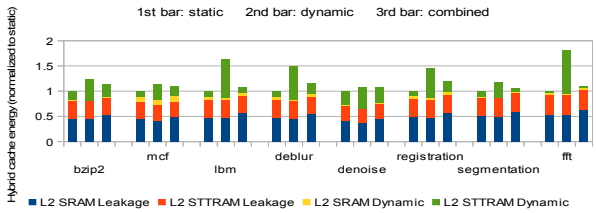

Figure 6: Comparison results of hybrid cache energy

6.3 Performance

Performance is measured by the runtime of a workload (in terms of number of clock cycles obtained from our simulation infrastructure). Figure 7 shows the comparison results of runtime that are normalized to the *static* scheme.

Since the total cache size for the three schemes are the same, the runtime does not varied significantly. The differences among the three schemes come from how efficiently they make use of the aggregate capacity of both SRAM and STT-RAM to reduce the

cache misses. The *dynamic* scheme typically performs the best due to the equivalent initial placement to SRAM and STT-RAM, which best utilizes the STT-RAM capacity. As mentioned in Section 4.3.1, compiler may generate larger write-intensive data on SRAM due to the input variation, thus impose high capacity pressure to the SRAM and result in high cache misses (as shown in Figure 8). Therefore, the *static* scheme performs -1%~9% worse than the *dynamic* scheme (with a -1% ~30% increase in the L2 cache misses). The only exception is *fft* where *static* outperforms *dynamic* due to accurate compiler hints.

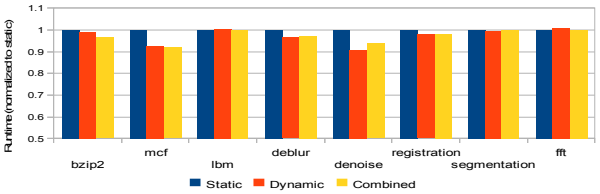

Figure 7: Comparison results of runtime

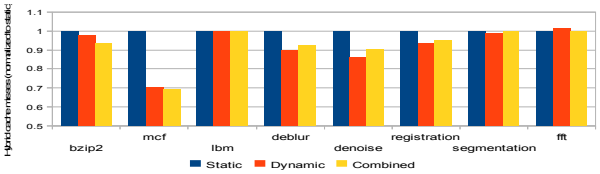

Figure 8: Comparison results of hybrid cache misses

In the *combined* scheme, the hardware can automatically correct the compiler mis-predictions as discussed in Section 4.3. Therefore, it achieves similar runtime to that of the *dynamic* scheme (within a -5% ~ 5% variation). These analyses are summarized in Table 6.

Table 6: Comparison summary of the experimental results

	static	***dynamic***	***combined***
Endurance	worst	fair	best
Performance	fair	best	best
Energy	best	worst	~best

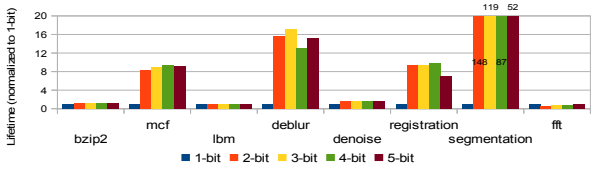

Figure 9: STT-RAM lifetime over different bit widths

6.4 Different Bit Widths of Write Counters

We perform the sensitivity analysis on different bit widths of the write saturation counters in our proposed *combined* scheme. The write counters are used for dynamic migration to improve the endurance. We justify that 2-bit counters are adequate enough for write counters. Figure 9 shows that the lifetime can be significantly enhanced in *mcf*, *deblur*, *registration*, and *segmentation* when 2-bit counters are applied. The 1-bit counters are inefficient since the SRAM blocks in the same set may easily saturate and thus prevent the migration of write-intensive STT-RAM blocks into SRAM ones. For the rest of workloads, the lifetime is insensitive to the bit width. According to our experimental results, the bit widths of write counters are insensitive to both energy and runtime among all workloads (less than 1% difference). In terms of energy, the only exception is *mcf*, where most of write intensive blocks cannot be migrated into SRAM when 1-bit counters are used. Therefore, the STT-RAM energy increases by 15% in the 1-bit counters case compared to the others (2- to 5-bit). For performance, it is insensitive to the widths of write counters since performance is maintained through cache capacity pressure monitoring, as described in Section 4.3.1.

7. CONCLUSIONS

In this paper, a combined static and dynamic scheme is proposed to optimize the block placement in a hybrid SRAM and STT-RAM cache, so that endurance and energy are co-maximized. The compiler tries to guide the hardware to rapidly achieve the desired placement, while the hardware corrects the compiler hints based on the runtime cache behavior. Experimental results show that the combined scheme improves the endurance by 23.9x and 5.9x compared to pure static and pure dynamic schemes, respectively, while maintaining similar performance. Meanwhile, the system energy can be reduced by 17% compared to the pure dynamic scheme.

8. ACKNOWLEDGEMENTS

This work is partially supported by the SRC Contract 2009-TJ-1984, and the Center for Domain Specific Computing (NSF Expedition in Computing Award CCF-0926127).

9. REFERENCES

[1] *International Technology Roadmap for Semiconductors. http://www.itrs.net/*: Semiconductor Industries Association, 2011.

[2] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," In *HPCA*, 2009, pp. 239-249.

[3] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," In *ISCA*, 2009, pp. 34-45.

[4] A. Jadidi, M. Arjomand, and H. Sarbazi-Azad, "High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement," In *ISPLED*, 2011, pp. 79-84.

[5] Y. Chen, W. Wong, H. Li, and C. Koh, "Processor caches built using multi-level spin-transfer torque RAM cells," In *ISLPED*, 2011, pp. 73-78.

[6] J. Li, C. J. Xue, Y. Xu, "STT-RAM based energy-efficient hybrid cache for CMPs," In *VLSI-SoC*, 2011, pp. 31-36.

[7] 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 *DATE*, 2012.

[8] A. Bui, K.Cheng, J. Cong, L. Vese, Y. Wang, B. Yuan, and Y. Zou, "Platform Characterization for Domain-Specific Computing," In ASPDAC, 2012.

[9] J. Hu, C. J. Xue, W.-C. Tseng, Y. He, M. Qiu and E. H.-M. Sha, "Reducing write activities on non-volatile memories in embedded CMPs via data migration and recomputation," In *DAC*, 2010, pp.350-355.

[10] T. Liu, Y. Chao, C. J. Xue, and M. Li, "Power-aware partitioning for DSPs with hybrid PRAM and DRAM main memory," In *DAC*, 2011, pp. 405-410.

[11] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," In *PLDI*, 2003, pp. 245-257.

[12] J. Cong, H. Huang, C. Liu and Y. Zou, "A reuse-aware prefetching algorithm for scratchpad memory," In *DAC* 2011, pp. 960-965.

[13] M. Zhang and K. Asanovic, "Fine-grain CAM-tag cache resizing using miss tags," In *ISLPED*, 2002, pp. 130-135.

[14] J. Cong, K. Gururaj, H. Huang, C. Liu, G. Reinman, Y. Zou, "An energy-efficient adaptive hybrid cache," In *ISLPED*, 2011, pp. 67-72.

[15] LLVM compiler. http://llvm.org/

[16] Omega library. http://www.cs.umd.edu/projects/omega/

[17] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," In *IEEE Computer*, vol. 35, pp. 50-58, 2002.

[18] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," In *Computer Architecture News*, pp. 92-99, 2005.

[19] HP Cacti, http://quid.hpl.hp.com:9081/cacti/

[20] C. Xu, X. Dong, N.P. Jouppi, and Y. Xie, "Design implication of Memristor-Based RRAM Cross-Point Structures," In DATE, 2011

[21] SPEC CPU2006, http://www.spec.org/cpu2006/