Combating Write Penalties Using Software Dispatch for On-chip MRAM Integration

Yong Li, Student Member, IEEE, Yaojun Zhang, Student Member, IEEE Yiran Chen, Member, IEEE and Alex K. Jones, Senior Member, IEEE

Abstract—Recent advances in the emerging memory technology magnetic RAM (MRAM) enrich the opportunities to build high density and low power embedded systems. One common way of utilizing MRAM is integrating it with conventional memories and distribute data to the appropriate type of memory to mitigate the high write penalty of MRAM. In this paper, we propose a software-based approach to identify data access characteristics and guide hardware to perform efficient data distribution. We use our technique to build an on-chip MRAM-SRAM hybrid cache and demonstrate a 86.8% reduction in leakage power, 9.8% reduction in total power and 5% memory performance improvement, compared to a traditional SRAM-only cache.

Index Terms—Magnetic RAM, Compiler, Cache

I. INTRODUCTION

MAGNETIC RAM (MRAM) is an emerging memory technology that offers drastically low leakage, relatively fast read access and high density compared to conventional memories such as SRAM and DRAM. Incorporating MRAM into embedded applications brings the potential benefits of reduced standby power and increased system density. These attractive benefits however, are often traded for its expensive write penalty, which leads to degraded performance and power consumption, especially for running workloads that are write intensive.

A common approach to mitigate the high write penalty issue is utilizing both MRAM and conventional memory to build hybrid memory systems. Prior research efforts [3], [7], [9] have been made to explore this approach. In particular, Guo et al. [3] attempt to re-design common on-chip components (e.g., register files, memory controllers, floating-point units, caches, etc.) using either conventional memory or MRAM, depending on the read/write characteristics of the designed components. Chen and Wu et al. [7], [9] explore on-chip hybrid memory caches and develop hardware mechanisms to distribute data accesses to appropriate types of memories. They keep track of a short history of write accesses using a counter and swap a data block from MRAM to conventional memory if the counter indicates the data block to be write intensive. These purely hardware-base approaches however, are easily misled by unpredictable runtime data access behavior. Significant mis-predictions can incur an expensive penalty of serving accesses in the wrong type of memory (e.g., write intensive accesses occurring in MRAM) as well as large swapping overhead. In contrast, a software mechanism has the advantages of taking actions preemptively to hide the swapping latency from the critical path and detecting data access patterns more accurately compared to simple run-time approaches.

In this paper, we design a hardware component named merge-swap unit to support efficient data swap between MRAM and SRAM in a MRAM-SRAM hybrid memory cache. To maximize the utilization of different types of memories and reduce the swapping overhead, we further develop a compiler-based mechanism to discover intensive writes and guide the hardware component to perform the data swap. We demonstrate that the compiler-assisted hybrid cache achieves over 5% memory performance improvement and 9.8% power savings compared to a SRAM-only cache.

II. HYBRID MRAM-SRAM CACHES

In this section we introduce our experimental system with hybrid MRAM-SRAM caches and hardware support for data swap between different types of memories.

A. Memory Modeling and Architecture Organization

We use HSPICE and a modified version of CACTI [6] to model the latency and power of SRAM and MRAM with comparable on-chip area, as shown in Table I.

<table>
<thead>
<tr>
<th>TABLE I: SRAM and MRAM Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size</strong></td>
</tr>
<tr>
<td><strong>Leakage Power</strong></td>
</tr>
<tr>
<td><strong>Read Latency</strong></td>
</tr>
<tr>
<td><strong>Write Latency</strong></td>
</tr>
<tr>
<td><strong>Read Energy</strong></td>
</tr>
<tr>
<td><strong>Write Energy</strong></td>
</tr>
<tr>
<td><strong>Area</strong></td>
</tr>
</tbody>
</table>

Base on the memory modeling we study the hybrid caches in a tiled many-core system consisting of 16 cores laid out as a 4 × 4 mesh with a 3-cycle per hop latency. It has a two-level cache organization in which the L1 cache is private to each core and the L2 cache is physically distributed but logically shared among all cores. We leverage the 3D integration technology, particularly the through silicon via (TSV) [5], to stack two device layers (i.e., SRAM and MRAM layer) vertically. The top layer is the MRAM L2 banks and the bottom layer is the CMOS logic including the 16 cores, 1L1s, switch/routing logics, coherence directories and SRAM...
fraction of the L2 cache. 1 SRAM line and 31 MRAM lines form one 32-way associativity (one cache set). Each switch on the CMOS layer connects one core to its neighbors as well as a through silicon bus (TSB), which leads to a corresponding MRAM bank. The architectural organization and parameters are presented in Fig. 1 and Table II, respectively.

### TABLE II: Architecture configurations

<table>
<thead>
<tr>
<th>Processor</th>
<th>16 SPARC cores, 2.6 GHz, 4W/core, in order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>64-bit Solaris 10</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>10KB/core, 4-way associative, 64B block size, 1-cycle hit latency, write-through</td>
</tr>
<tr>
<td>L1 Coherence</td>
<td>MESI protocol, in cache directory</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>128KB/bank</td>
</tr>
<tr>
<td>L2 Block Size</td>
<td>64B</td>
</tr>
<tr>
<td>L2 Associativity</td>
<td>32</td>
</tr>
<tr>
<td>L2 Read Latency</td>
<td>4 cycles</td>
</tr>
<tr>
<td>L2 Write Latency</td>
<td>18 cycles</td>
</tr>
<tr>
<td>L2 Energy</td>
<td>refer to Table I</td>
</tr>
<tr>
<td>Network</td>
<td>4×4 Mesh, packet switching, 3 cycles per hop</td>
</tr>
<tr>
<td>Main Memory</td>
<td>4GB, 150-cycle latency</td>
</tr>
</tbody>
</table>

![Fig. 1: 3-D Architecture With hybrid MRAM-SRAM Caches](image)

#### B. Write Dispatch Using Merge-Swap Unit

For systems in which SRAM and MRAM coexist, it is critical to assign or distribute write operations to the appropriate type of memories. We call this assignment or distribution process dispatch. A straightforward dispatch method is to migrate a line, upon a write request, to the SRAM line in the same associativity. This will however result in an eviction of the current SRAM line thus entirely destroy the incoming write localities. We design the merge-swap unit, a hardware mechanism that allows fast data swap without destroying the data write locality.

Fig. 2 shows the diagram for the merge-swap unit. Upon a write request, the address is decoded into tag, set index (SI) and offset address (OA). The tag and set index are used to uniquely locate a line in the L2 cache. If the write request hits on a MRAM line, that line is read out to the merge buffer, where it is merged with the write data based on the offset address. The line mask is used to avoid overwrite of the requested data by the MRAM cache line. In parallel, the LRU (Least Recently Used) time stamp of the SRAM line within the same associativity is examined. If the SRAM line is not the LRU line, it is migrated to overwrite the MRAM line which has been merged with the write data in the merge buffer. Otherwise it is evicted. Finally, the merge buffer dumps the line into the SRAM so that the subsequent write requests of this line will be served on SRAM. The entire merge-swap operation is managed by a dispatch control logic that controls the operation by keeping track of write access patterns.

![Fig. 2: Merge-swap unit](image)

A merge-swap decision should be careful made due to the involved overhead during swapping. Compared to a direct write into MRAM which requires 3.243nJ of energy and 18 cycles of latency, the merge-swap process first reads out the MRAM line before the SRAM line can be migrated, resulting in an extra MRAM read latency (4 cycles) on the critical path. With respect to power consumption, the merge-swap operation, in the most common case of the SRAM line being migrated, requires extra power for a MRAM read, a SRAM read and a SRAM write and thus the overhead is approximately 0.574nJ + 0.643nJ + 0.550nJ = 1.767nJ. Therefore, the merge-swap operation should only be performed on a data block if that block exhibits intensive writes after being swapped onto SRAM such that the swapping overhead can be recouped by the gains.

### III. Software Optimizations

This section presents the proposed compiler approach to identify write reuse for arrays and linked data structures to improve the efficiency of the hardware merge-swap operation.

#### A. Write Reuse Identification for Arrays

To make swapping decisions for a memory block containing continuous bytes, we detect the temporal and spatial write reuses from array accesses in the source program.

The temporal reuse information of affine array accesses can be analyzed by solving linear algebra equations [8]. Consider Fig. 3 as an example. Given the array accesses $A[i + 2][j]$ and $B[2][i][2 * i + 1]$ in the nested loop shown in Fig. 3(a), we first convert the subscript functions to the matrix expressions, as illustrated in Fig. 3(b). The array access now can be represented as $C * k + O$, where $C$ is the coefficient matrix, $k$ is the index vector and $O$ denotes the offset vector. Determining whether the array access has temporal write reuse now is equivalent to deriving the condition under which the equation $C * k' + O = C * k'' + O$ has solutions. In linear algebra theory, the necessary and sufficient condition under which the above equation has solutions is that $C$ is not full ranked. In our example, the coefficient matrix of $A[i + 2][j]$ has a rank of 2,
indicating no temporal reuse. $B[2][i][2 * i + 1]$ has temporal reuse since the rank of its coefficient matrix is 1.

An array write access exhibits spatial write reuse when the innermost enclosing loop index varies only the last coordinate of that array. To discover spatial write reuse, we use a truncated coefficient matrix by dropping the last row of the original coefficient matrix, as illustrated in Fig. 3(b). If the rightmost column in the truncated coefficient matrix (the coefficients that correspond to the innermost loop index) is a null vector and the rightmost element in the dropped row is nonzero, it is assured that the innermost loop only varies the last coordinate of the corresponding array.

In the above example, $A[i+2][j]$ exhibits spatial reuse since the rightmost column in the truncated matrix (the coefficient corresponding to the innermost loop index) is a null vector and the rightmost element in the dropped row is nonzero. Using the same rule we can determine that $B[2][i][2 * i + 1]$ does not have spatial reuse since the innermost loop index $j$ does not vary in the last coordinate of array $B$.

B. Write Reuse Identification for Linked Data Structures

To analyze the write reuse pattern for linked data structures such as linked lists and trees, a CFG (control flow graph) of the program is constructed. A CFG $G = (V, E, r)$ is a directed graph, with nodes $V$, edges $E$, and an entry node $r$. Each node $v$ in $V$ is a basic block, which consists of a sequence of statements that have exact one entry point and exit point. The CFG is traversed while the following rules are examined to determine whether a sequence of memory writes (right-hand side of assignments) exhibit memory reuse:

- The analyzed memory writes are common pointer based dereferences. That is, these memory writes only differ in their offsets from a common base pointer.
- There are at least three memory writes whose offsets are within the address range of a cache block size (i.e., 64 bytes).
- There are no function calls between the analyzed memory writes.
- The memory writes are either in the same basic block or in a set of direct successor basic blocks that meet the above three criteria. If there are conditionals, the second criterion must be satisfied in all branches.

These rules guarantee the analyzed writes are nearby in the address space thus occur in a single memory block, from the perspective of which the writes are intensive/frequent.

Fig. 4 provides various cases to explain the write reuse identification for linked structures. As defined in Fig. 4(a), the pointer $nd$ is declared to point to a data structure with the type $node_t$. Since the data members $x$, $next$ and $prev$ have integer/pointer type and are adjacent fields in the same data structure, they are consecutive in the address space and thus would typically reside in the same memory block such as a cache line. In the case of Fig. 4(b), the three memory writes in the same basic block (i.e., $nd->x=5$, $nd->next=A$ and $nd->prev=B$) have the common base pointer $nd$ and there is no interleaved function calls. Thus, Fig. 4(b) exhibits write reuse. The program in Fig. 4(c) also has write reuse since both successors of the basic block $nd->x=5$ lead to write reuse. Fig. 4(d) does not exhibit write reuse due to the presence of the function call $foo()$. In Fig. 4(e), one of the direct successors $nd->next=A$ only has one common pointer based memory write and thus will not be marked as having write reuse.

C. Pre-dispatch Instrumentation

Once the write reuse has been determined, the compiler inserts a pre-dispatch instruction into the code prior to the memory access to notify the CPU to perform the merge-swap operation. The pre-dispatch instruction can be implemented using the extra bits in the instruction opcode of a particular ISA. For example in the SPARC, the $prefetch$ instruction provides dedicated field $fcn$ to implement variants of prefetch instruction. The $fcn$ value from 16 to 31 is currently reserved and can be used to implement pre-dispatch instruction. In the ARM architecture, there are similar reserved bits that can be used to implement this instruction.

IV. PERFORMANCE AND POWER EVALUATION

To demonstrate the superiority of our proposed approach, this section compares the performance and power consumption of the software-optimized pre-dispatch scheme (SPD) with the pure SRAM cache as well as the hardware dispatch approach to migrate data upon two successive writes (MSW). To compare the memory latency and power impact of SRAM and MRAM L2 caches, we use Simics [4] as our simulation environment and implement the machines described above. We choose multi-threaded workloads from SPLASH-2 [1] and PARSEC [2] benchmark suits.
The off-chip miss rate shown in Fig. 5 demonstrates the advantage of low off-chip miss rate achieved by employing MRAM for on-chip storage. With 4 times as large as SRAM in L2 cache capacity, MSW and SPD reduce the off-chip misses by 38.9% and 40.0%, respectively. The reduction in expensive off-chip misses results in an average of 5% faster memory accesses for SPD, as shown in Fig. 6. MSW exhibits negligible improvement in memory performance despite the reduction in off-chip misses due to the inefficient dispatch of write requests to SRAM.

The static and dynamic power breakdown normalized to the total power consumption of the SRAM cache. Due to the leakage problem inherited by CMOS devices, the SRAM-only cache consumes a non-negligible amount of static power while MSW and SPD dramatically reduce the leakage power. However, MSW consumes an average of 18.4% more dynamic power than the SRAM-only design due to the high energy overhead incurred by writes on MRAM. In contrast, the dynamic power consumed by SPD is close to that consumed by SRAM since most of the writes have been dispatched efficiently to SRAM, resulting in very a small portion of writes served by MRAM. Overall, MSW results in an 0.3% more power consumption than SRAM-only in spite of its 86.6% leakage power savings. SPD has a similar leakage power savings of 86.8% as well as a total power savings of 9.8%.

V. CONCLUSION AND FUTURE WORK

In this paper, we presented a compiler-based approach to detect write reuse patterns within an application and use this information to guide the hardware and distribute data access to appropriate memory resources in hybrid memory caches. Our evaluation demonstrates using our technique the memory access performance is improved by 5% and memory power consumption is reduced by 9.8% over the SRAM-only cache organization. We consider several future directions of this work including OS-assisted dispatch techniques and more architectural designs that leverage applications’ data access characteristics.

ACKNOWLEDGMENT

This work is supported by NSF award CCF-0702452 and CSR-1116171.

REFERENCES