Cross-layer Techniques for Optimizing Systems Utilizing Memories with Asymmetric Access Characteristics

Yong Li and Alex K. Jones
Department of ECE
University of Pittsburgh
Pittsburgh, USA
Email: {yol26, akjones}@pitt.edu

Abstract—Non-volatile memory technologies promise a variety of advantages for memory architectures of next generation computing systems. However, these capabilities come at the cost of some inefficiencies governing the operation of these memories. The most well understood is the asymmetry of access. In order to most effectively take advantage of the benefits of these memory technologies in terms of density and reduced static power in systems while mitigating access complexity an one-size fits all method is not sufficient for all types of applications. Instead, cross-layer techniques that include the compiler, operating system, and hardware layer can extract characteristics from the application that can be used to deliver the highest possible performance while minimizing power consumption for systems using these memories.

Keywords—Compiler; Retention; STT-RAM; Network-on-Chip; Cache; Buffer;

I. INTRODUCTION

CMOS technology scaling is increasingly affected by the high leakage power and thermal challenges due to reduced device dimensions and near-threshold voltage operation. Potential improvements include the use of high performance and low power alternative memory technologies. In particular, Spin-Torque Transfer Magnetic RAM (STT-RAM) has received considerable attention [1] and is regarded as one of the most promising future memory candidates. Substantial research efforts have been made to mitigate its drawbacks, most notably the asymmetric access characteristics. Common methodologies to address the memory asymmetry issue include using counter-based mechanisms to track the data accesses and appropriately migrate data blocks [2], pro-actively operating on predicated data elements [3] and sacrificing data retention time for balanced read/write penalties [4].

In this paper, we discuss cross-layer methods to optimize systems that use STT-RAM in various components within chip-multiprocessor (CMP) systems such as the shared caches and buffers within the network-on-chip (NoC). We discuss how high-level knowledge of the application behavior can be used to mitigate the access asymmetry of read and write behaviors. A cross-layer approach can leverage a high-level view of the data access behavior of an application and use this information to configure features of the system hardware a priori in order to increase the performance and power-efficiency. In contrast to a hardware-oriented approach that reacts to system behavior, a cross layer approach can often avoid misleading or temporary program behavior that can lead to inefficiencies and configuration thrashing.

In particular, we describe methodologies that leverage compiler analyses, which expose data access and communication patterns for multi-threaded programs. Further, we present mechanisms for how this data can be communicated from the system level into the runtime system. Finally, we demonstrate how including configurability into the architecture can use this information to optimize the use of STT-RAM for improved efficiency. We consider system configurability such as: hybrid SRAM/STT-RAM cache designs [2], hybrid SRAM/STT-RAM buffer designs [5] combined with fast, circuit-like paths [6] in NoCs, and use of reduced data retention times in STT-RAM to improve write performance [4].

We demonstrate the effectiveness of this approach through two case studies. In the first case-study we examine compiler guided writing within a hybrid STT-RAM with primarily standard STT-RAM that requires a long write delay combined with a small amount of STT-RAM optimized or faster writing due to a reduced retention time but, which requires refreshing. Our results show that the compiler-guided data distribution ensures that 89% of writes are handled in the faster writing memory, which only comprises 3% of the total cache capacity and yields a 5% performance improvement and near 10% power saving compared to a traditional SRAM design. In our second case-study we examine method to integrate STT-RAM buffers into a configurable NoC. In our studies, nearly 90% of the network traffic can take advantage of a limited set of compiler guided circuit-switch style fast-paths leading to a more than 5% reduction in network delay in spite of the increase buffer write time of STT-RAM while leveraging its low leakage benefit.

II. RELATED WORK

Due to the asymmetrically high delay/energy write characteristics of many emerging non-volatile memories, intensive research efforts have been made to mitigate these write
penalties at various levels of the system. For example, Guo et al. [7] use STT-RAM to re-design a number of non-write-intensive micro-architectural components and adopt a subbank write buffering policy with read-write bypassing to increase write throughput and hide the high write latency. Rasquinha et al. [8] address the high write energy of STT-RAM by adopting a new replacement policy that increases the residency time of dirty lines at the expense of higher miss rates.

Wu et al. [2] proposes a region-based hybrid cache architecture (RHCA) and level-based hybrid cache architecture (LHCA) with various non-volatile memories including STT-RAM and phase-change memory (PCM). Data is swapped between slow and fast regions based on data access behavior recorded with hardware counters. A similar concept was proposed for a hybrid STT-RAM/SRAM cache [9]. Jang et al. [5] propose a hybrid STT-RAM/SRAM input buffer design for NoCs to save static power while achieving high network throughput.

Recently, it has been proposed to reduce STT-RAM write delay/energy at the expense of a shorter data retention time [4], [10]. This requires the system ensure that data does not degrade.

These hybrid architectures coupled with the reduced retention STT-RAM provide opportunities for cross-layer optimizations where system configurability can exploit application specific characteristics to perform more efficiently than a generic approach. We describe two examples of this in the next two sections of the paper.

III. COMPILER-CACHE CROSS-LAYER OPTIMIZATION

Most applications have a relatively small working-set of frequently written (FW) data. FW data can be identified by the compiler. Through effective use of a small bank of STT-RAM with relaxed retention (STT-RR), the write penalties of STT-RAM can be mitigated.

A. Cache Design

Building on a standard set-associative cache, each set can contain primarily traditional STT-RAM with a small fraction STT-RR blocks serving as write-friendly (WF) memory and a significant portion of standard STT-RAM, as depicted in Figure 1. To leverage the static power benefit of STT-RAM and achieve the write performance comparable to SRAM, STT-RR must be predominantly used to service writes. However, as STT-RR requires a refresh mechanism, thus, frequently read (FR) data should be predominantly stored in standard retention STT-RAM (STT-SR). As access characteristics change during application execution, data must be appropriated migrated or swapped between STT-SR and STT-RR banks.

![Figure 1: Cache configuration utilizing hybrid STT-RAM.](image)

B. Compiler Analysis

Using compiler-based dispatch, which identifies the write reuse patterns and inserts instructions to guide the hardware to perform the migration/swap, dynamically, it is possible to avoid the pitfalls of a runtime-only system [11]. High temporal reuse serves as a indicator of FW data. In the next two sections we describe compiler methods to detect high temporal reuse in array and linked-style data structures.

1) Write Reuse Identification for Arrays: The temporal reuse information of affine array accesses can be analyzed by solving linear algebra equations [12]. Consider Figure 2 as an example. Given the array accesses $A[i + 2][j]$ and $B[2][i][2 * i + 1]$ in the nested loop shown in Figure 2(a), we first convert the subscript functions to the matrix expressions, as illustrated in Figure 2(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 ($k'$ and $k''$ represent two different index vectors in the iteration space). 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, which is smaller than its dimension.

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 Figure 2(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 is a
null vector and the rightmost element in the dropped raw
is nonzero. Using the same rule we can determine that
\( B[2][i][2*i+1] \) does not have spatial reuse.

2) 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.

Figure 2: Array accesses and the corresponding matrix
representations (a): array accesses (b): matrix representation

Figure 3: Code and control flow graph examples for write
reuse identification. (a): type definition code (b): write reuse
in the same basic block (c): write reuse across one basic
block and all its successors (d): write reuse broken by
function call (e): write reuse broken by one successor

Presuming standard compiler passes such as constant
propagation, expression folding and branch elimination are
applied on the CFG, the CFG is traversed to examine the
basic blocks. Memory writes (right-hand side of assignments)
are all written in the same block.

In a similar fashion, a group of accesses can indicate
reuse even if they span multiple contiguous basic blocks
assuming no function calls separate them in the CFG. They
are also span conditionals in the case that all corresponding
paths in the CFG exhibit similar frequency of access. Thus,
in (c) reuse is present because three contiguous locations
are written in all paths. However, in (d) the function call
separates the access and in (e) the one path does not store to
\( nd->prev \), each indicating no reuse. These rules help ensure
these analyzed writes occur in a single memory block. From
this perspective the writes are intensive/frequent.

C. Code Instrumentation
To communicate the information about the FW data to
the system, we use code instrumentation. When write reuse
has been identified, the compiler inserts a pre-dispatch
instruction into the code prior to the memory access to
notify the CPU to perform the migration or swap operation
to ensure the writes occur in WF memory.

IV. COMPILER-NETWORK CROSS-LAYER OPTIMIZATION

Multi-threaded applications often imply a particular traffic
pattern either from data partitioning or data flow between
threads. However, shared caches can often obfuscate this
traffic pattern [13], but the underlying data partitioning
and resulting traffic pattern can often be extracted by the
compiler [13] and used to configure a hybrid packet/circuit-
switched network [13], [14]. By employing compiler, cache,
and network cooperation in this fashion, the traditionally
SRAM buffers can be predominantly replaced with STT-
RAM in the NoC to reduce static power while maintaining
a high performance.

A. Hybrid Network Infrastructure

A configurable hybrid NoC using express virtual channels [15]
allows a circuit-like function to exist within a
packet-switch. Packet switched traffic requires significant
overhead at each switch point including stages like virtual
channel allocation, route computation, switch allocation
and switch traversal. Express channels are pre-configured
and allow express traffic to bypass the routing logic at each
switch and only add one cycle latency per hop for buffering.

Express channels/circuits can be established at runtime
using hardware counters that track the communication char-
acteristics of the network [14]. For a cross-layer approach
the compiler extracted communication pattern can be used to
schedule the available circuits efficiently without the runtime
overheads [13]. If the majority of the traffic uses circuits,
then virtual channel buffers for the packet-switch traffic
are less performance critical and can employ STT-RAM
with a slower writing time. Figure 4 provides an example
switch configuration for using STT-RAM-based NoC where
the shaded buffers support the packet-switched traffic and
employ STT-RAM. The unshaded buffers support circuit traffic and use WF memory for best performance.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{hybrid_router.pdf}
\caption{Hybrid router design using STT-RAM}
\end{figure}

\subsection{B. Compiled Communication Pattern}

Discovering the data partitioning and communication pattern of the application begins with array access region analysis. Within a loop nest for a single threaded program, a reference to an array A can be generally represented as $A[f(L)]$, where $f(L)$ is the subscript function defined on a set of loop indices $L = l_1, ..., l_m$. The span of $f(L)$ resulting from $l_k$ is the maximum distance traversed by varying only $l_k$ from its lower bound $l_k$ to its upper bound $u_k$ [16], [17]:

$$
\text{span}_{l_k} = |f(1, ..., l_{k-1}, u_k, l_{k+1}, ..., l_m) - f(1, ..., l_{k-1}, l_k, l_{k+1}, ..., l_m) |	ag{1}
$$

Similarly, the stride is defined as the minimum distance across memory by changing only $s_k$ by its step $s_k$ :

$$
\text{stride}_{l_k} = |f(1, ..., l_{k-1}, l_k + s_k, l_{k+1}, ..., l_m) - f(1, ..., l_{k-1}, l_k, l_{k+1}, ..., l_m) |	ag{2}
$$

Thus, an array access region can be described by the following form, where $O$ denotes the starting offset:

$$
R = \{A_{\text{span}_{l_1}, ..., \text{span}_{l_m}} + O\} \tag{3}
$$

The region concept can be extended to multi-threaded applications to represent data partitions and resulting communication [13]. The MMAP (Multi-threaded Memory Access Pattern) provides a data access pattern representation for multi-threaded applications. Given a phase of parallel code for $n$ threads, the MMAP for an array access is a list of $n$ regions with each region corresponding to one thread:

$$
M = \{R(0), ..., R(n-1)\} \tag{4}
$$

Each region in the MMAP can be calculated by replacing thread specific variables\footnote{a variable is thread specific if it has different values for different threads} with the actual values for the corresponding thread. An MMAP, $M$, is non-overlapping when $\forall R(x), R(y)$ in $M, R(x) \cap R(y) = \emptyset$. Otherwise, the MMAP is overlapping.

The access weight for region $R(x)$, denoted as $W(R(x))$, is defined as the number of accesses from thread $x$ on its region $R(x)$. The access weight can be calculated based on the information from the loops associated with the region. An estimate of the data partition can be determined by the compiler through MMAP analysis and the calculation of their relative access weights. For a program with $n$ threads numbered from 0 to $n-1$, a partition $P$ is a set of non-overlapping regions: $\{R(0), ..., R(n-1)\}$. Tile $x$ is the owner for region $R(x)$ in the partition.

When data owned by a particular tile is accessed by a remote tile this results in communication in the system. Let $R(y)$ represent the data region owned by tile $y$ in a partition. Then in the same fashion, if data is distributed to the tiles as dictated by the compiler, accesses by other tiles to data in $R(y)$ require communication. The volume of these remote accesses is defined as non-local access weight and provides the potential communication weight between two tiles $x$ and $y$ in a particular phase within the application execution. To determine a partition from an overlapping MMAP, each overlapping array element can be placed into the region that minimizes non-local access weight.

The compiler can communicate this partitioning information into the runtime system through the page-table. During data allocation the compiler-extracted partitioning provides each page (or potentially sub-pages) with an owner. During data access the owner is used to enforce placement within the cache. The communication pattern extracted by the compiler, based on the partitioning, can be used to guide the establishment of the circuits in the network. The circuit configurations can be communicated to the runtime system by adding network configuration instructions into the code [18] at appropriate points determined by the compiler.

\section{V. Evaluation}

In our case studies, we use a modified version of CACTI [19] and HSPICE simulations to model the latency, power and area of our memory designs, as shown in Table I. The write latencies for STT-RAM with different retention times are calculated based on the relationship between

\begin{table}[h]
\centering
\caption{SRAM and STT Parameters (45nm)}
\begin{tabular}{|c|c|c|}
\hline
Size & 128k SRAM & 512k STT \\
\hline
Area & 1.339mm\textsuperscript{2} & 1.362mm\textsuperscript{2} \\
Read Latency & 1.78ns & 1.82ns \\
Write Latency & 1.68ns & 8.725ns (STT-RAM) \\
Read Energy & 0.574nJ & 0.550nJ \\
Write Energy & 0.643nJ & 3.243nJ \\
Leakage Power & 0.185W & 0.013W \\
\hline
\end{tabular}
\end{table}
MTJ switching time and the switching current at the given retention levels [4]. The simulated architecture is described in Table II including a baseline and our hybrid STT-RAM design for comparison. We utilize multi-threaded workloads from the SPLASH-2 [20] and PARSEC [21] benchmark suites and Wind River Simics [22] as our simulation environment.

A. Compiler-Cache Cross-layer Results

Table III shows parameters for replacing the SRAM L2 cache with a hybrid STT-RAM cache, which provides additional capacity with the same die area as SRAM. Figure 5 shows that STT-RR, which comprises just over 3% of the cache capacity, serves 89% of the write requests. This indicates an efficient data distribution that dispatches most writes onto the WF memory (STT-RR). By utilizing various forms of STT-RAM, the cache saves considerable static power. The result is an reduced average memory access latency due to the increased cache capacity and reduced power consumption from the static power savings, as illustrated in Figure 6 and Figure 7, respectively. For the outlier, SWAPIONS, the memory latency is degraded compared to the SRAM design since it does not benefit from the extra capacity achieved by the STT-RAM cache and the longer latency from the writes that occur in STT-SR causes the performance degradation. On average, the hybrid STT-RAM cache design reduces the memory latency by 4.6% while achieving 10% total energy savings. If the refresh of STT-RR cannot be tolerated, standard SRAM can serve the role of WF memory without significant static power overhead.

B. Network Performance

Figure 8: Communication pattern for OCEAN

To demonstrate the effectiveness of the compiler extracting the communication pattern, Figure 8 compares the OCEAN benchmark where the left matrix is the compiler generated pattern and the right is the runtime generated pattern, each for 64 processors. Communication volume is indicated by the intensity of the color, black is negligible communication, and red, orange, yellow, and white represent increasingly heavy communication. The compiler generated pattern provides a good approximation of the runtime effect, which is less precise due to effects of classification at the page granularity.

Table IV: Hybrid network configuration

Figure 9: Amount of flits traveling on different networks
The network parameters are shown in Table IV. We utilize a medium retention time of 3.2s for packet switch buffers, which writes approximately 4X slower than 26.5us retention memory [4]. We assume packets in the network for longer than 3s would be dropped. Circuit switch traffic is never delayed making 26.5us retention adequate. Using the communication patterns to guide circuit establishment in the hybrid network helps satisfy the aims of efficient utilization of both the fast writing and the low power memory in our hybrid network design. As shown in Figure 9, nearly 90% of the network traffic traverses circuits. Consequently, Figure 10 reports an average of 5.3% reduction in network latency due to the use of hybrid network design, while achieving significant static power savings.

![Figure 10: Normalized network latency](image)

VI. Conclusion

In this paper, we describe cross-layer approaches to detect data access and communication patterns within the applications and use this information to efficiently configure hardware that uses STT-RAM. Our evaluation through two case studies demonstrates that caches and on-chip interconnects built with STT-RAM storage can effectively cooperate and utilize the cross-layer information to improve performance and reduce power consumption.

REFERENCES


