EnVM : Virtual Memory Design for New Memory Architectures

Pooja Roy, Manmohan Manoharan, Weng Fai Wong

School of Computing, National University of Singapore
{poojaroy,manmohan,wongwf}@comp.nus.edu.sg

Abstract

Virtual memory is optimized for SRAM-based memory devices in which memory accesses are symmetric, i.e., the latency of read and write accesses are similar. Unfortunately, with the emergence of newer non-volatile memory (NVM) technologies that are denser and more energy efficient, this assumption is no longer valid. For example, STT-RAMs are known to have high write latencies and limited write endurance which the virtual memory is unaware of. A popular architecture is a hybrid cache that uses both SRAM and NVM. There are a number of proposals for such architectures at nearly all the levels of the cache. However, these proposals are often self-contained with monitoring and management schemes implemented with special hardware at the level where the cache is deployed. With moves to use NVM at several levels of the memory hierarchy, such solutions may lead to duplication and higher overheads. Worse, because the management algorithms implemented can be different at different levels of memory, it may lead to negative interference between them resulting in impaired efficiency.

In this paper, we propose a virtual memory design, *EnVM*, that takes into consideration the idiosyncrasies of NVM-based hybrid caches. The new virtual memory layout is implicitly used to allocate data to NVM and SRAM at any level of the memory hierarchy and is not dependant on the particular arrangements of the two partitions. The proposed design successfully filters out write operations and allocates them to SRAM. Moreover, it can be applied to any existing fine-grained data allocation technique to enhance the efficiency of these memories.

1. Introduction

The concept of virtual memory is the key to managing multiple processes efficiently with the limits of the physical memory of a system. Virtual memory allows programs to execute with memory footprints that are larger than the available physical memory. However, the classic virtual memory is designed with the assumption that the underlying cache hierarchy is built using fast SRAM. Recently, non-volatile memories (NVM), say, STT-RAM technology, are becoming viable alternatives of SRAM for cache memories. These technologies allow caches to be denser and more energy efficient. However, these NVM technologies show characteristics that are different from that of SRAM. For example, their access latencies are asymmetric. In particular, writes are significantly slower than reads [1, 18]. Furthermore, they face the issue of write endurance, i.e. the number of write operations a NVM cell can endure before failing is lesser than that for SRAM.

From systems' perspective, allocating data without differentiating between read and write accesses is therefore detrimental to NVM memories. Unmonitored and excessive write operations can impede performance, and, in addition, reduce the lifetime of the on-chip caches and hence the processors [8, 22]. It is essential to judiciously manage memory accesses based on their access patterns and access types in order to achieve a balance between energy efficiency and performance. Researchers favoured the use of hybrid caches comprising of a smaller SRAM and a larger NVM partition as a promising design [2, 6, 11, 18]. The smaller SRAM filters out write operations, protecting the write sensitive NVM partition. Recent works have explored novel data allocation techniques to manage the hybrid caches efficiently. Chen et.al. [2] proposes a hardware-software co-optimized framework to allocate data to hybrid caches. Their compile-time analysis produces hints for each instruction that influences data placement in the partitions. The hardware support ensures that write intensive data is migrated from NVM to SRAM to ameliorate the write endurance issue. Li et.al. [11] proposed a new stack layout to optimize data allocation to the hybrid caches. They present a specialized address generation policy that reduces data migration between the two partitions, while, at the same time, reducing write operations to NVM. Their technique can be applied to global data too. However, all the techniques are specialized for a particular cache level and architecture. Most of the methods have hardware overheads. These partial approaches will result in even higher overheads when NVM based hybrid caches are adapted at all levels. Worse, it would lead to mutual interference between the different cache levels, subsequently resulting in impaired efficiency.

Our Proposal In this paper, we propose a new virtual memory design *EnVM*, that is aware of NVM based caches. The revised virtual memory design is able to influence data allocation across all the levels of memory hierarchy seamlessly. EnVM consists of a static code analysis that generates virtual addresses for statically allocated data and facilitates virtual memory layout of the global data and stack. The static analysis is able to discern the memory access affinity for each data and generates virtual address accordingly. The key idea is to enhance locality of data based on their memory access affinity i.e. read affinity and write affinity. For dynamically allocated memory i.e. heap area in the virtual address space, EnVM makes use of modified system libraries. Our new dynamic memory allocator interface is exposed to the programmer and provides

ESWEEK '14, October 12–17, 2014, New Delhi, India.

http://dx.doi.org/10.1145/2656106.2656121

the programmer with distinctive functions for read intensive and write intensive data structures. Virtual address generation for heap accesses is performed at runtime by the operating system. Our modified kernel supports the system libraries to manage the new heap area of virtual address space. The virtual to physical address translation is intersected by a group of conventional segment registers to facilitate data allocation to SRAM and NVM partitions.

Contribution There are several advantages of EnVM. First, it is able to influence the data allocation across all levels of memory hierarchy without requiring specialized hardware at each cache level. This helps in easier adoption and scalability to deeper cache hierarchies. Secondly, EnVM provides a holistic design with support for both statically allocated and dynamically allocated data, spanning the entire virtual memory address space. Finally, our experiments show that EnVM eliminates the need for data migration as the write operations are optimized and filtered out to SRAM. Although, cache management can be further optimized by some form of migration, EnVM serves as the base virtual memory for the new memory technologies. We implemented EnVM using the GCC compiler and GNU `malloc` library. In order to quantify the gain, we implemented a hybrid cache model described in [18] and compare EnVM with two existing works on software assisted data allocation for hybrid caches [2, 11]. Details of evaluation and experimental results are presented in Section 5. In summary, the contributions of this paper are as follows -

- We propose EnVM, the first virtual memory design that is aware of memory hierarchies built using the new memory technologies. EnVM provides an uniform data allocation mechanism to all the levels in the memory hierarchy. This is an important step towards an *all NVM* based memory hierarchy.
- EnVM provides a novel static code analysis that can identify and allocate data with read and write affinity separately in the virtual address space. It enables data allocation accordingly and reduces write operations to NVM.
- We provide a new programmer's interface to be able to allocate read and write intensive heap memory exclusively during runtime. The system libraries and the operating system are optimized for this new heap memory region.
- EnVM is the only virtual memory design that enables data allocation to hybrid caches built with SRAM and NVMs. It utilizes existing hardware and advocates migration-less cache design.

2. Related Works & Background

To further motivate our proposal, we will elaborate on state of the art techniques of hybrid cache management. In addition, we will provide a brief background on existing virtual memory management schemes.

2.1 Compiler Assisted Hybrid Caches

NVM-based hybrid caches are being studied in depth in recent years. Researchers proposed many solutions [5, 20, 23] that are either hardware or software controlled. In this paper, we are particularly interested in compiler assisted management of the hybrid memories and hence we shall illustrate the state-of-the-art of these methods.

Li et.al. [10] introduced one of the first compiler assisted approaches for managing hybrid caches. They assumed a hybrid L1 cache architecture that allows for migration of data from STT-RAM to SRAM to reduce write operations. They presented a novel stack data placement and proposed an arrangement of memory blocks in such a way that reduces migrations because copying data from one cache to another is an expensive operation. Further in [11], they proposed a preferential cache allocation policy that places migration intensive blocks into SRAM to further reduce write accesses to STT-RAM. Chen et.al. [2] presents a hardware and software co-optimized framework to aid STT-RAM based hybrid L2 caches. They proposed a memory-reuse distance based program analysis that allocates write intensive data in SRAM and read intensive data in STT-RAM. This analysis is supported by a runtime data migration technique using hardware counters for cache lines. Though their framework improved performance and also showed energy efficiency, they are based on the profiling of application. Profiling based methods suffer the well-known shortcomings in usability and scalability. Moreover, their memory-reuse distance based algorithm is applicable to L2 caches only.

The two works mentioned above targeted L1 and L2 caches, respectively. When we consider the use of NVM based memories at all levels of memory hierarchy, these techniques to manage a single level of cache in isolation may interfere with each other if used together. For example, in the algorithm in [2], memory blocks with a large memory reuse distance are assumed to incur write operations to L2 due to L1 capacity miss. Based on such heuristics, every memory block is provided with hints to be considered while placing the cache block in L2 SRAM partition or STT-RAM partition. Suppose we also have a hybrid L1 that uses the algorithm in [10] which places read and write intensive blocks in different localities if they are in the stack region. The data locality, then, becomes a function of the type of memory access and not temporal relationships. In such a setup of L1 and L2 caches, a large memory reuse distance for a L1 cache block does not necessarily result in capacity misses. Therefore the assumption for algorithm for L2 cache management is weaken significantly. These two cache management techniques for L1 and L2 will fail to cooperate with each other, and may in fact be detrimental to one another.

All the works proposed in literature target a specific level in the cache hierarchy. Many are profile based program analysis with hardware support to manage the cache blocks in accordance with the program behaviour obtained. Such hardware supports and program analyses are not scalable to the entire memory hierarchy. Though Li et.al. did take advantage of the virtual memory to influence data allocation across the memory hierarchy in [11], they only propose stack and global data arrangement based on a static code analysis customized only for L1 data caches as mentioned before. There is, therefore, a need for a holistic framework that manages the virtual memory area of a process to aid hybrid memories at any level of the memory hierarchy. The cache hierarchy is generally accessed using physical addresses that are computed from virtual addresses using specialized hardware. Virtual memory layout, therefore, influences optimized cache management. As the underlying memory technology changes, a shift in virtual memory design is inevitable for maintaining performance and energy efficiency. Before proposing our virtual memory design EnVM, we will briefly present a summary of existing virtual memory management.

2.2 Virtual Memory and Supporting Architecture

The virtual memory address space of a process is usually separated in sections. These sections are logically contiguous segments of virtual address that shows common characteristics. For example, the `text section` contains the code of a program and the `data section` contains all global variables and memory objects. In a Linux operating system, information about virtual address space of a process is usually embedded in the executable using a special format known as the Executable and Linkable Format (ELF). During the creation of a process or context switches, the operating system kernel loads the virtual memory address space of that process with all the related information from the ELF binary file. Every virtual

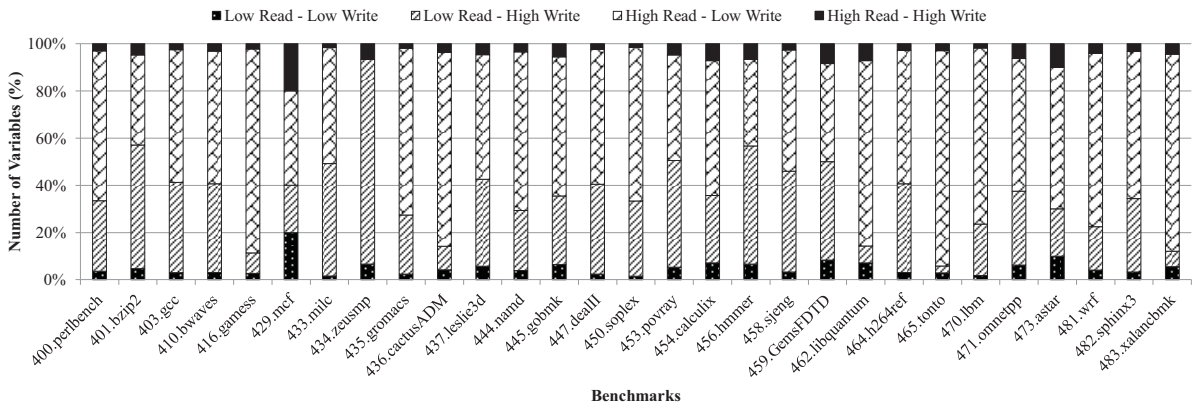


Figure 1: Percentage of variables

address gets translated to a physical address. This physical address is then used to access memory objects in the physical memories i.e. caches and main memories. A typical layout of the virtual memory address space is shown in Figure 2(a). Our proposed technique utilizes the virtual memory space of processes to provide a uniform scheme to manage hybrid caches and main memory.

3. EnVM

In this section, we will describe EnVM and its functionality in detail. We will first describe EnVM's new memory layout followed by the data management techniques for both statically allocated and dynamically allocated data.

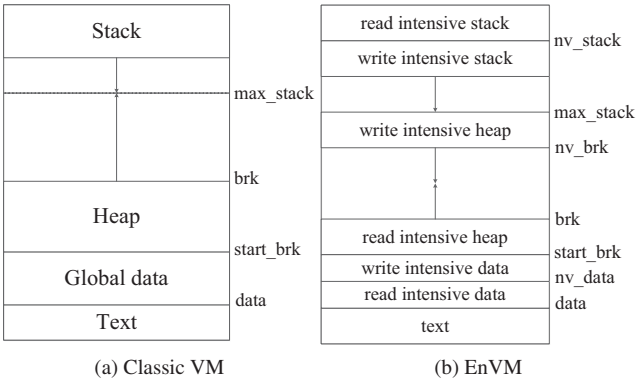
Traditionally, the virtual memory space is divided in logical segments as described in Section 2. EnVM contains fine-grained logical segments that are based on the memory access affinity of the memory objects. In other words, memory objects that exhibits read affinity are placed separately from those that shows write affinity. Figure 1 shows that memory objects that shows affinity to both read and write operations are less in proportion. In most of the benchmarks, only 5% of the variables show a high read and write affinity, where as 92% variables (on average) shows affinity towards either read or write accesses.

In EnVM, the read and write intensive groups are separated by segment boundaries known to the operating system (OS). At runtime, the OS will manage the data allocation to underlying hybrid memories using the segment boundaries. This is analogous to managing text segment and non-text segment for instruction caches and data caches separately. It is worth pointing out that the EnVM layout would work on existing systems with no modification.

3.1 Statically Allocated Data

Runtime behaviour of statically allocated data is possible to analyze at compile-time. To arrange the global and stack data in EnVM, we propose a new static code analysis for placing variables according to their memory access affinity. The analysis we present here estimates the number of reads and writes of each program variable. Unlike profiling techniques, it path insensitive, and therefore does not focus only on the frequently executed program path(s). The analysis is a dataflow analysis(DFA) problem. The DFA is applied as an interprocedural analysis on the control flow graph of the program.

Definition *The abstract domain of the analysis is a tuple containing an identifier for the variable, its read and write count represented as* (V, R, W)*, where* $V \in$ *set of all variables in the program,* R *and* $W \in \mathbb{N}$*. The domain forms a lattice,* $((V, \mathbb{N}, \mathbb{N}) \cup \{\top\}, \sqsubseteq_F)$*,*



(a) Classic VM (b) EnVM

Figure 2: Existing and Proposed Virtual Memory Design

where $\top$ *is the top element and* $\sqsubseteq_F$ *is the partial order defined as*

$$(X \sqsubseteq_F Y) \text{ iff } (X(V) \sqsubseteq_{RW} Y(V)) \text{ for each variable } V_i \in V \quad (1)$$

$$(V_i \sqsubseteq_{RW} V_j) \text{ iff } (R_i \leq R_j) \wedge (W_i \leq W_j) \quad (2)$$

where R_i, W_i, R_j *and* W_j *denotes the read and write counts for variables* V_i *and* V_j *respectively.*

The partial order defined above is significant for the termination of the dataflow analysis. It also plays an important role in analysing branches and joins in the control flow graph. The partial ordering rule says that two variables are partially ordered if and only if both the read and write counts are in natural order. For example, if the read count of one variable is higher than that of the other but opposite for the write count, then the analysis cannot determine any partial order between the two variables.

As the DFA we propose is counting based, the partial order between different variables do not influence the outcome of the analysis. However, the partial ordering between instances of the same variable is important during branch joins. This phenomena is described later with the discussion of *meet* operator. Each instruction i, in a basic block is passed through define two *transfer functions*, $\boldsymbol{F}$ and $\boldsymbol{B}$, for forward and back edges, respectively.

Definition *At each program point, the set of tuples* (V, R, W)*, denoted as* X*, and the transfer function for the current instruction* $\boldsymbol{i}$*, is defined as*

$$F_i(X) = Gen[i] \sqcup Probe_i(X) \quad (3)$$

The function Gen[i] discovers a variable from the instruction i, and the function Probe$_i(X)$ examines all the elements of the set X and updates it according to the rule below -

$$\begin{gathered}\forall V' \in X,\ \textit{where } V' = (V, R, W) \\ R = R + 1 \textit{ if } \boldsymbol{i} \textit{ reads } V \\ W = W + 1 \textit{ if } \boldsymbol{i} \textit{ writes } V\end{gathered} \tag{4}$$

The key idea is to examine whether an instruction i has a read operation on variable V, then read counter is incremented, and if i has a write operation on V, the write counter is incremented. For all back edges in the CFG, most likely a loop edge back to the start of the loop, we have a transfer function $B_i(X)$. For an instruction ***i*** succeeding instruction ***j*** through a back edge, all variables $V \in$ instructions between ***j*** and ***i***, $R = R + k$ and $W = W + k$, where k is a static loop bound [21]. This will have a similar effect as going through the loop instructions k times. When resolving branches and phi functions, we apply the *meet* operator $\sqcup$.

Definition *The meet operator $\sqcup$ is applied when two basic blocks have a common successor basic block. The OUT information from the two parent basic blocks are unified using the meet operator to form the IN information of the successor. It is defined as*

$$(V_i, R_i, W_i) \sqcup (V_j, R_j, W_j) = \begin{cases} \top \textit{, if} (V_i = V_j) \wedge (V_i \not\sqsubseteq V_j) \\ (V_i, \max(R_i, R_j), \max(W_i, W_j)) \\ \quad \textit{, if} (V_i = V_j) \wedge (V_i \sqsubseteq V_j) \\ \{(V_i, R_i, W_i) \cup (V_j, R_j, W_j)\} \\ \quad \textit{, if} (V_i \neq V_j) \end{cases} \tag{5}$$

The above rule says that when different instances of a variable along different paths are not in partial order then it is assigned the $\top$ element. For example, if a variable has a read count that is more than the write count for one path, but it is the other way around for another path, then we assign $\top$ to the variable. This means that it was not possible to conclude whether this variable has more reads or writes. For all other variables, we take the maximum of read and write counts over all the paths. This gives us an estimate of the upper bound. The dataflow problem is solved using iterative algorithm.

Definition *For each basic block l, we have two dataflow equations RWA$_{entry}(l)$ and RWA$_{exit}(l)$. These represent the set of tuples before and after processing a basic block. For our analysis, we define the dataflow equations as follows -*

$$RWA_{entry}(l) = \emptyset \textit{ if } l \in init(S_\star) \tag{6}$$

$$RWA_{entry}(l) = \sqcup(RWA_{exit}(l') \cup B_i,\ \textit{if } (l, l') \in Flow(S_\star) \tag{7}$$

$$RWA_{exit}(l) = (RWA_{entry}(l) \cup F_i(RWA_{entry}(l)) \tag{8}$$

where $init(S_\star)$ denotes the set of initial labels i.e. the starting basic blocks, $Flow(S_\star)$ denotes the flow of the program and (l, l') is a valid edge in the control flow graph.

As mentioned before, B_i is the transfer function applied while traversing a backward edge. If there is no back edge to the entry of the basic block then $B_i = \emptyset$.

Indirect memory accesses Apart from static variables, there are a large number of variables in a program which are accessed indirectly through pointers. Our analysis extends to the pointers to static variables through the help of *"may"* aliases of each variable. Points-to information gathered from the alias sets helps to associate variables to their probable source of access and the type (read or write). All pointer variables that points to statically allocated data are treated as independent data objects and can be classified differently than the points-to object. However, with each load and store that accesses a variable through the pointer, the read and write counts of that variable is updated. This satisfies two cases - firstly, where the pointer variable itself is updated or read, which is a common practice in pointer arithmetic and secondly, the data that the pointer points to is updated or read after dereferencing.

Algorithm 1 Partial Algorithm for Address Generation for Global and Stack Data

Require: source code of the program
Ensure: virtual addresses for statically allocated memory objects
CFG ← Control Flow Graph of the program
procedure ADD_GEN(CFG)
    Initialize var_array ← ∅ /* var_array is a 2-d array with variables and their assigned classes */
    Initialize $analysis\_outcome$ ← ∅ /* $analysis\_outcome$ contains all the variables with read and write counts */
    for all function(F) in CFG **do**
        $analysis\_outcome$ ← PASS_RW_ANALYSIS(F)
    end for
    var_array ← CLUSTERIZE($analysis\_outcome$)
    $current\_global$ ← $.text\_section\_end$
    $current\_stack$ ← $.stack\_base$
    for all variable(V) in var_array **do**
        if V is global data and CLASS(V) !=3 **then**
            allocate V to $current\_global$
            realign $current\_global$
            remove V from var_array
        else if V is stack data and CLASS(V) == 1 or 2 **then**
            allocate V to $current\_stack$
            realign $current\_stack$
            remove V from var_array
        end if
    end for
    $.global\_write$ ← $current\_global$
    $.stack\_write$ ← $current\_stack$
    for remaining variables(V) in var_array **do**
        if V is global data **then**
            $current$ ← $current\_global$
        else if V is stack data **then**
            $current$ ← $current\_stack$
        end if
        allocate V to $current$
        realign $current$
        remove V from var_array
    end for
end procedure

Address Generation The analysis provides an estimation of read and write counts for each program variable. Our aim is to partition the variables into two groups, i.e., read intensive and write intensive variables using this estimated counts. The memory access behaviour of applications differs to a large extent. Some applications are computation intensive, where memory accesses have different pattern than that of an I/O intensive application. Therefore, to enhance scalability of EnVM, we rely on an unsupervised machine learning technique to partition the variables. Although, a threshold based partitioning is simpler, it is inefficient as the threshold requires to be tuned for different applications separately. EnVM leverages on the K-Means clustering algorithm to partition the variables. The read and write information gathered are the feature inputs i.e. observations to the clustering algorithm. The program variables are partitioned into 4 classes, namely, write intensive ; non-write intensive; read intensive; and non-read intensive. The initial seed points are set to be the maximum and the minimum read and write counts obtained from the analysis. The four extreme values as seed points will move the clusters towards the read and write extremities. We obtain the following four classes by applying clustering over the read count and write arrays:

1. Class 0 - Low read and low write

2. Class 1 - Low read and high write
3. Class 2 - High read and low write
4. Class 3 - High read and high write

Algorithm 1 shows the address generation scheme for statically allocated data. Global variables usually show high affinity towards either read or write operations. Number of global variables showing high read and write counts are few. Therefore, for global data, we place Class 0,2,3 variables contiguously and then Class 1 variables. The virtual address separating these two sections are embedded in the final executable. For stack data, we place Class 0 and 2 variables contiguously and then Class 1 and 3 variables. The virtual addresses separating the two sections are also embedded in the final executable. This yields the read and write intensive virtual memory segments shown in Figure 2(b).

3.2 Dynamically Allocated Data

The previous section describes the memory layout for static data i.e. global and stack data. Next, we will describe the management of heap region in EnVM. Dynamically allocated memory objects occupy a large region in the virtual address space of many processes, and is managed at runtime. Precise analysis of dynamically allocated memory at compile time is computationally hard [16]. Though, heap memory management is well studied for efficient garbage collection and detecting memory leaks [9, 14], analysing dynamically allocated memory for read and write patterns is especially difficult at compile time due to their unbounded sizes and abstract types. For example, if a static memory object is marked as read intensive, a pointer to the static variable can be analyzed by de-referencing it symbolically at compile time. However, for dynamically allocated memory regions, the de-referencing of the pointers creates an unbounded space that is hard to analyse.

Coburn et.al. explores the possibilities and threats of heap memory management for persistent memory systems such as NVMs [3]. However, in their work, the read and write properties of the heap region is unexplored. For EnVM, an estimate on the read and write counts of any memory object is sufficient for the layout and address generation. However, an inappropriate data allocation would be detrimental to performance and lifetime of the chips. Therefore, for heap region, we rely on programmers' interface to provide distinction between read and write intensive heap accesses. EnVM provides new library functions, namely, `r_malloc()` and `w_malloc()` that would allocate from two heaps - one for read and another for write intensive dynamic objects. To incorporate these malloc calls, either the source can be annotated by the programmer or heuristic estimates may be applied. In this paper, we have done the former and annotated the source codes of our benchmarks with the new malloc function calls, as shown in Figure 3.

Just like the standard `malloc()`, the two new functions - `r_malloc()` and `w_malloc()` are tied to the system calls `sbrk()` and `brk()`. The allocation and deallocation from the two heaps are independently managed. During initialization, both `r_malloc()` and `w_malloc()` functions will each request for a sizeable memory chunk, usually spanning multiple pages, from the kernel. They subsequently maintain *bins* to cater to the malloc requests. Depending on the call, `r_malloc()` or `w_malloc()`, the requests are served from the respective chunks. Figure 3 shows an example of a code implemented with the two malloc calls. As the interaction of the malloc library and the kernel is usually through the page requests, there will not be any additional fragmentation (or holes) in the virtual memory area due to the split heap. In case when one of the heaps run out of memory space to allocate, mainly due to a boundary limit, we allow the use of the other.

For management of the two heaps at runtime, EnVM requires operating system support. The two heaps are bounded by markers `start_brk`,`brk`, `nv_brk` and `max_stack`, where both `start_brk`, `brk` and `max_stack` are conventional markers. `start_brk` and `max_stack` denotes the start and permissible end of heap area. `brk` is the virtual address marking the end of allocated memory. We introduce a new marker `nv_brk` to denote the end of allocated read intensive heap memory. The operating system is responsible for loading a boundary register (see Section 4) with the boundary addresses so that the cache fills and write-backs to the two partitions are managed accordingly. For evaluation, we modified only `malloc()` function calls. Programs that use other ways to dynamic memory allocation and deallocation, for example `new()`, are only evaluated based on the static analysis. However, we see no difficulty in extending this to other dynamic memory allocation functions.

Algorithm 2 Dual Heap Management

Require: modified malloc library support
Ensure: runtime dual heap management
1: kernel variables $read\_malloc$, $start\_brk$ and nv_brk set by operating system
2: malloc() sets $read\_malloc \leftarrow 0$
3: nv_malloc() sets $read\_malloc \leftarrow 1$
4: **while** 1 **do**
5: **for** all brk() system calls **do**
6: **if** $read\_malloc$ **then**
7: $dummy \leftarrow start\_brk$; $start\_brk \leftarrow nv\_brk$
8: service system call and allocate memory space
9: update $nv\_brk \leftarrow start\_brk$
10: restore $start\_brk \leftarrow dummy$
11: **else**
12: service system call and allocate memory space
13: **end if**
14: **end for**
15: **end while**

Algorithm 2 describes the overall runtime functionality of the dual heap management. With a `malloc()` system call, the library sets a kernel variable to denote the heap type i.e. read or write intensive heap (lines 2-3). Once the context is switched to the kernel, it checks whether the `malloc()` is for the read or write intensive heap (line 6), and will then sets the address in the variable `brk` accordingly (lines 7-10). The kernel proceeds to allocate memory to the requested heap (line 8 or 12). The variable `brk` is then restored to the default i.e. write intensive heap (line 10). The default heap allocation is serviced from write intensive heap to avoid unmonitored write accesses, for example security threats, to NVM.

3.3 Putting It All Together

The framework to create EnVM is illustrated in Figure 4. During compilation, a program is analysed for read and write intensive memory variables. The outcome of the analysis dictates the virtual address generation of these variables. As in the case of conventional virtual memory layout, static memory objects are placed in the virtual address space and the executable is generated. For dynamic memory objects, we provide a dual-heap management module that is assisted by the operating system. Customized system calls are used as a wrapper function to enable the dual heap structure. During runtime, the operating system allocates dynamic memory objects from distinctive read and write intensive heaps. Thus, in our proposed new virtual memory design, EnVM, memory objects arranged in the order of their memory access affinity.

4. Architectural Support

In this section, we shall describe the architecture support required for our virtual memory design. For evaluation, we assume a hybrid

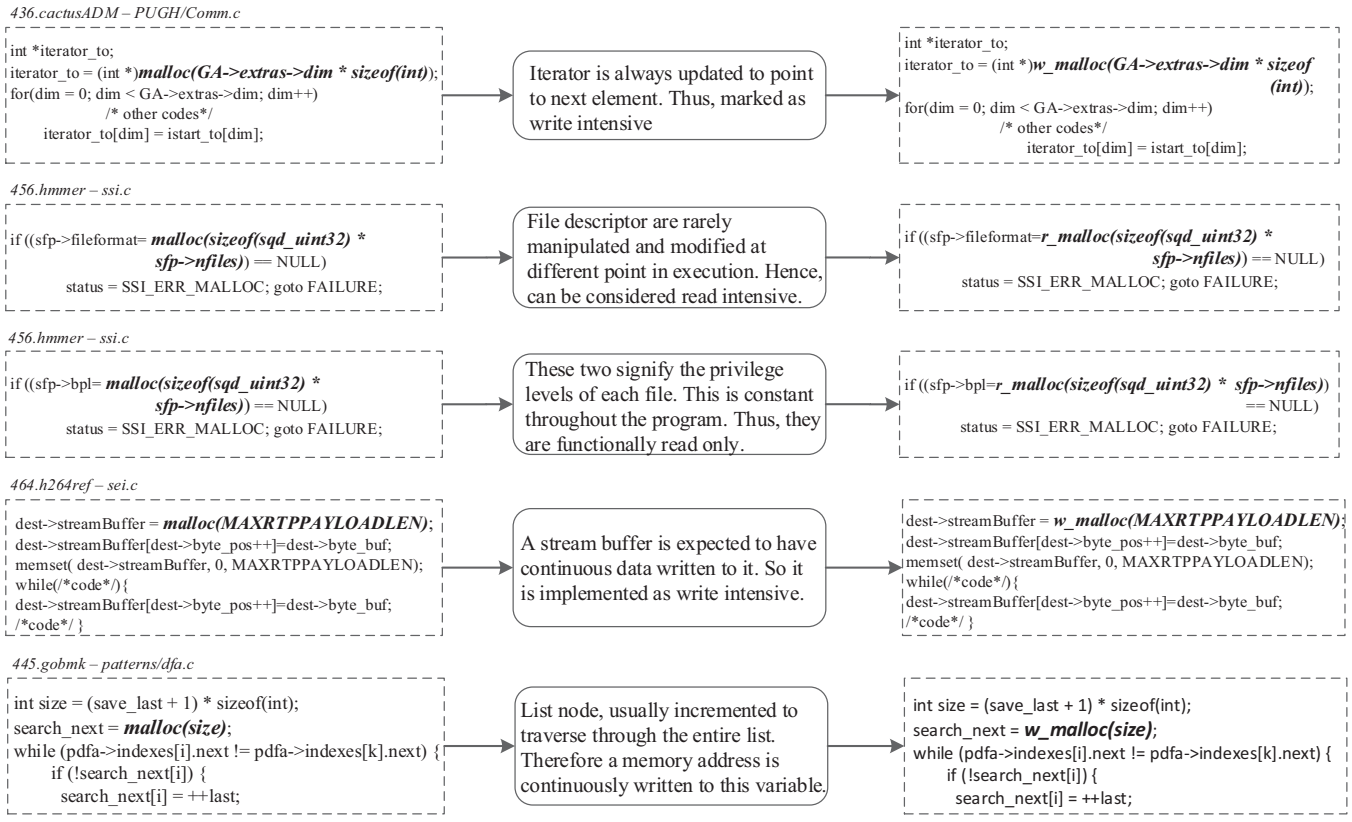


Figure 3: Example of modified code in the benchmarks with new malloc calls

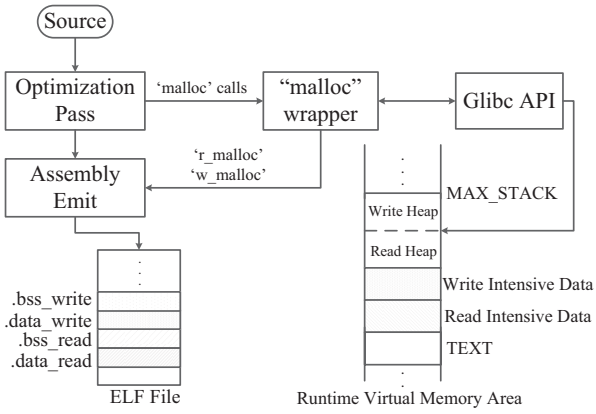


Figure 4: Framework

cache model presented in [18]. In addition we will discuss few aspects of virtual to physical address mapping, and other hardware implications of EnVM.

4.1 Boundary Registers

The layout of EnVM is used to influence the data allocation for caches across various levels. This is made possible by a set of boundary registers at the address translation hardware unit. For the x86 architecture, the existing segment registers can be used for this purpose. During process creation and context switches, the operating system is responsible for loading these boundary registers with the boundary addresses. For our evaluation, we propose six such boundary registers holding the addresses `nv_data`, `start_brk`, `brk`, `nv_brk`, `max_stack` and `nv_stack`. During the virtual to physical address translation, a simple hardware logic shown in Figure 5) enables the correct cache partition to be probed. However, the boundary registers are consulted for cache selection only for write operations to caches, i.e. either a cache fill from lower memory or a write-back from higher level. For read operations, the entire cache is probed without checking the boundary register. This optimization reduces any performance degradation due to the boundary address checking. Moreover, for indirect memory accesses, checking the entire cache prevents incorrect reads and extra cache fills.

4.2 Cache Properties

The delay associated with the boundary registers and address checking is dependant on the cache probe logic. We consider two kinds of caches here to analyse the delay - PIPT (physically tagged, physically indexed) and VIPT (virtually tagged, physically indexed). In PIPT caches, the TLB (translation lookaside buffer) is responsible for a complete virtual to physical address mapping. The TLB look-up is a blocking operation for PIPT caches and thus, the boundary registers are checked in parallel. Therefore, we do not consider any additional delay in PIPT caches. However, in VIPT caches, the TLB and tag array of the caches are looked up in parallel. In this case, the boundary register checking becomes a blocking operation. We assume that this delay is one clock cycle. For our evaluation framework, we assumed VIPT caches, adding

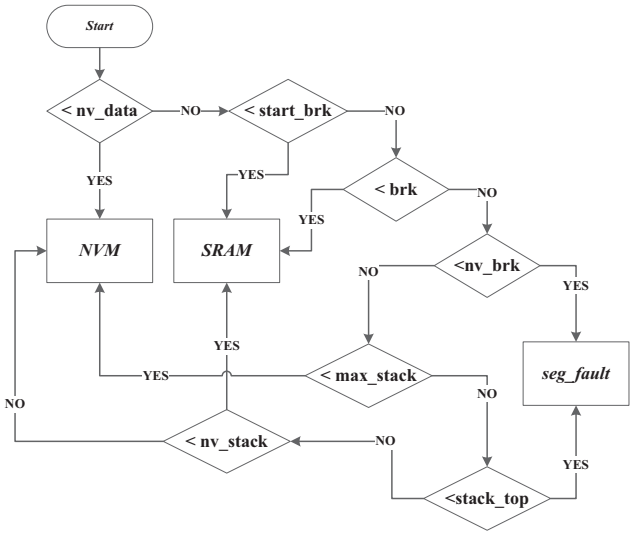


Figure 5: Cache Selection Logic

1 cycle delay for the boundary register checking. The delay overhead is minimal as the registers are checked only for write accesses. Moreover, if hybrid caches are adapted at L2 or L3 levels, the delay overhead is masked by L1 hit rate.

5. Evaluation

5.1 Tools & Benchmark

The dataflow analysis is implemented in GCC-4.7.1 as an optimization pass. We provide modified glibc-2.5 interface for the new dual-heap *malloc* function calls. For our experiments, we used the entire SPEC2006 benchmark suite [17]. The results are based on the 'ref' input on all the benchmarks. Our backend operating system is Linux (kernel version 3.2.51). We implement the hybrid caches in MARSSx86 [15] cycle-accurate full system simulator. The complete configuration is given in Table 1. As an instance of resistive memory technology, we have chosen to use the parameters of *spin transfer torque RAM* (STT-RAM). NVSim [4] was used to generate the latency and energy parameters for STT-RAM assuming a 32nm process technology. All the hybrid cache configurations roughly occupies the same silicon area as their pure SRAM counterpart [4]. We further assumed that the STT-RAM partition has error-correcting code (ECC) to mitigate stochastic bit-flip error as was proposed in [13] as the retention time for STT-RAM cells are myriad [7, 19].

5.2 Results

For evaluation, we implemented two hybrid cache designs at L1 [2] and L2 [11] referred to as *SW1* and *SW2*, respectively. We compare our method with another hardware based hybrid memory management scheme [6] referred to as *HW*. The primary objective of all the schemes and proposals is to reduce the number of write operations to the NVM caches by redirecting write intensive data to the SRAM counterpart. Figure 6 shows the number of write accesses to the STT-RAM partition. *HW* method incurs the maximum write accesses as data is primarily fetched into STT-RAM and migrated to SRAM only upon saturation of a 3-bit counter. *SW2* is a profile-based technique that is co-optimized by hardware and software having *a priori* information about memory accesses, and thus shows least number of writes to the STT-RAM. Unlike *SW1* which proposes stack data placement scheme, EnVM manages the entire virtual memory of a process and thus places all data accordingly, to the two partitions. EnVM reduces the total number of write accesses to STT-RAM by 47.6% as compared to *HW* and 15% as compared to *SW1*. In addition, for some benchmarks such as `403.gcc` and `456.hmmer`, EnVM achieves a comparable write traffic to STT-RAM as compared to *SW2*, a profile-based technique. NVMs use high write current that affects the total energy consumption. As all the schemes propose STT-RAM based hybrid caches, we will compare the energy consumption by the data arrays of the caches. The energy model is given by the sum of leakage energy, dynamic energy and overheads due to various additional hardware units.

Simulator Configuration	
Processor : Unicore, 3 GHz, Commit Width - 4	
Memory - Hybrid L1 Design	
L1 I-Cache (SRAM)	64KB, 8-way, 64B Line, 3 cycles
L1 D-Cache (Hybrid)	SRAM : 4KB, 4-way 3 cycles, STTRAM : 64KB 4-way Read 3 cycles, Write 10 cycles
L2 (SRAM)	2MB, 8-way, 15 cycles, 64B Line
Memory - Hybrid L2 Design	
L1 I-Cache (SRAM)	64KB, 8-way, 3 cycles, 64B Line
L1 D-Cache (SRAM)	32KB, 8-way 3 cycles, 64B Line
L2 (Hybrid)	SRAM : 1MB, 4-way 3 cycles, STTRAM : 2MB 8-way Read 11 cycles, Write 30 cycles
L3 (SRAM)	4MB, 8-way, 35 cycles, 64B Line

Table 1: Simulation Configuration

$$E_{total} = E_{leakage} + E_{dynamic} + E_{overhead} \quad (9)$$

$$E_{leakage} = P_{leakage} * t_{exec} \quad (10)$$

$$E_{leakage} = E_{write} * N_{writes} + E_{read} * N_{reads} \quad (11)$$

where, $E_{leakage}$ is *leakage energy* (in joules), $P_{leakage}$ is the *leakage power* (in Watts) and t_{exec} is the total execution time (in seconds) (of each benchmark). $E_{dynamic}$ is the total dynamic energy (in joules), E_{write} and E_{read} are the dynamic write energy and dynamic read energy, respectively. The energy required to allocate a cache block upon each miss is already accounted for in the total number of writes as N_{writes} and cache reads as N_{reads}. $E_{overhead}$ is the energy consumed by the additional boundary registers to manage EnVM. We used CACTI 5.3 [12] to calculate the energy consumption by the boundary registers. Figure 7 shows the total energy per instruction for each of the methods. Just as is the case for write reduction, EnVM is more energy efficient than *SW1* and *HW* showing an average of 21% and 6% reduction in energy consumption, respectively. For some C benchmarks, such as `400.perlbench`, `401.bzip2`, EnVM showed a lower energy consumption than even *SW2* with a maximum reduction of 50% for `458.sjeng`. The energy efficiency of EnVM is a result of including all memory objects, especially heap data, in its management. Figure 8 shows the energy overhead due to additional hardware units ($E_{overhead}$) of EnVM as compared to *HW* which is below 3%. In *HW*, there are two sets of 3-bit and 5-bit saturating counters per cache line and set respectively, accounting for the energy and space overhead. Figure 9 further shows the energy overhead of *SW1*, *SW2* as compared to EnVM. While there is no additional hardware component for *SW1*, it assumes a migration based L1 cache architecture. *SW2* too assumes a migration based L2 cache architecture. Migrating cache lines at L1 and L2 levels requires hardware to copy the cache lines, and incurs additional cache reads and writes. Though, EnVM requires a set of boundary registers, it can be used on a migration-less cache architecture at any level.

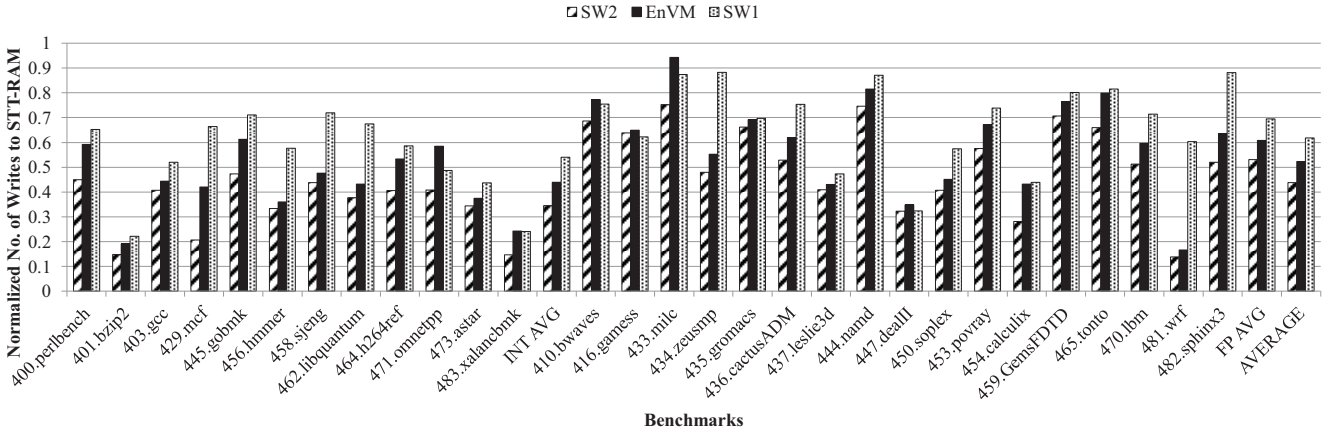


Figure 6: Total number of writes to STT-RAM in a hybrid cache design normalized to the total number of writes by *HW*.

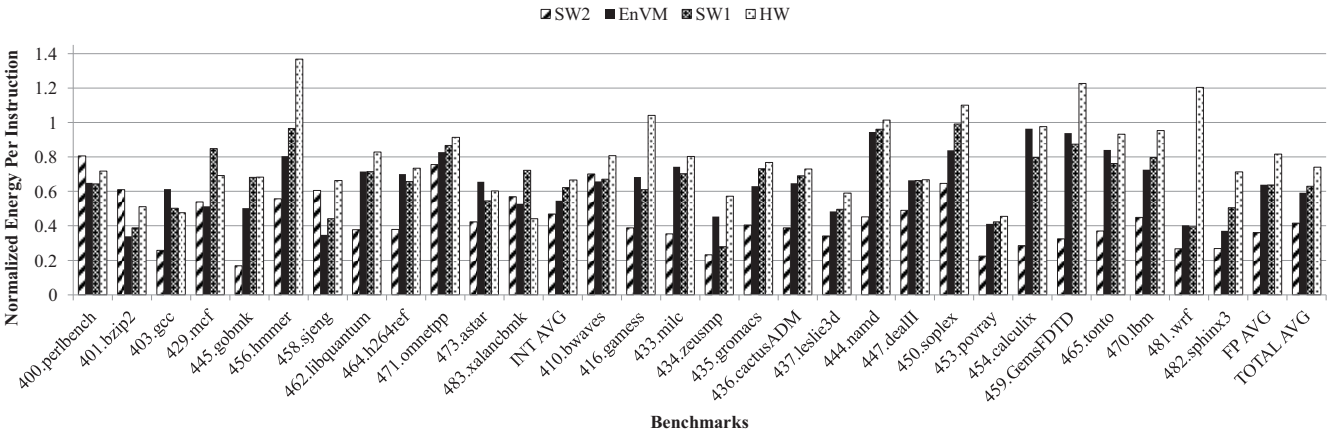


Figure 7: Energy per instruction normalized against *pure SRAM* cache.

Benchmarks	HW	EnVM	Benchmarks	HW	EnVM
400.perlbench	0.0346	1E-04	**434.zeusmp**	0.0339	0.0001
401.bzip2	0.0284	0.0002	**435.gromacs**	0.0411	0.0001
403.gcc	0.067	0.0002	**436.cactusADM**	0.0454	0.0001
429.mcf	0.055	0.0002	**437.leslie3d**	0.0454	1E-04
445.gobmk	0.0702	0.0003	**444.namd**	0.0256	7E-05
456.hmmer	0.0254	6E-05	**447.dealII**	0.0367	9E-05
458.sjeng	0.0554	0.0003	**450.soplex**	0.0323	1E-04
462.libquantum	0.0275	7E-05	**453.povray**	0.0443	0.0002
464.h264ref	0.0507	0.0001	**454.calculix**	0.066	0.0001
471.omnetpp	0.0349	8E-05	**459.GemsFDTD**	0.0602	8E-05
473.astar	0.0483	0.0001	**465.tonto**	0.0401	6E-05
483.xalancbmk	0.0304	0.0001	**470.lbm**	0.0274	0.0001
410.bwaves	0.0151	4E-05	**481.wrf**	0.0468	9E-05
416.gamess	0.042	9E-05	**482.sphinx3**	0.0212	0.0001
433.milc	0.0632	0.0002	**AVERAGE**	0.0413	0.0001

Figure 8: Energy(joules/instruction) consumed by the additional hardware units for HW and EnVM.

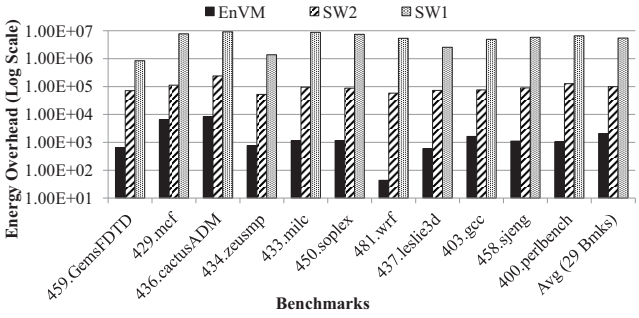


Figure 9: Total energy consumption by additional hardware components.

In our evaluation, we measured the energy overhead of the three techniques as shown in Figure 9. In Figure 10 we show that the performance of the system (an out-of-order x86 processor in our case), remains unperturbed with the introduction of EnVM based migration-less STT-RAM based hybrid cache at L1. We measured the IPC (Instructions per cycle), taking into accounts all additional

delays required by the boundary address checking. The IPC is normalized to a baseline of 32K SRAM L1 cache and 2MB L2 cache.

While the high write latency of STT-RAM and other resistive memories may erode overall performance, as they are denser, much bigger caches can be accommodated in the same die area. This increase in cache sizes compensates for the performance deterioration due to the higher write latency. To further quantify the impact of cache sizes on performance, we measured the cache hit rate (see Figure 11). The cache hit rate is measured only for L1 cache as it is most critical to the overall performance of a system. Though *SW2* assumes a hybrid L2 cache, we have reported the hit rate of L1 when *SW2* is applied. Table 12 summarizes the features of

Scheme	Target Cache	Migration Overhead	Additional Hardware
HW	L1	✓	✓
SW1	L1	✓	Nil
SW2	L2	✓	✓
EnVM	Any	Nil	✓

Figure 12: Summary of state-of-the-art methods and EnVM.

the state-of-the-art schemes and EnVM. *HW* scheme is optimized for L1 caches requiring hardware counters and assumes a migration based cache design. *SW1* and *SW2* assumes migration based caches for L1 and L2 respectively. Though *SW1* is a pure software based technique, it only optimizes stack data and is not scalable to other memory regions. *SW2* is hardware and software co-optimized scheme requiring hardware counters and buffers. EnVM is applicable to any level of caches and is not dependant on migration based design. While it does require hardware support, the hardware cost is amortized over the entire memory hierarchy as it is not exclusive to any particular level. Thus, we believe that EnVM is more scalable.

6. Conclusion

In this paper, we have proposed EnVM, a virtual memory design optimized for NVM based memory hierarchy. Enhancing the state-of-the-art, EnVM manages the entire virtual memory area of a process including code, static data, stack and dynamic data. It provides an uniform and holistic management of NVM based memory hierarchies, unlike current techniques that optimizes for specific levels of the memory hierarchy. As a part of EnVM, we propose a new static code analysis that distinguishes read-intensive from write-intensive variables. We also propose a new dual heap scheme that enables distinct memory regions for read and write intensive dynamically allocated variables at runtime.

EnVM is capable of managing any design of hybrid caches comprising SRAM and NVM partitions. Furthermore, it assumes a migration-less hybrid cache architecture and thus is not dependant on the effectiveness of migration techniques. EnVM serves as a base virtual memory for any further optimizations on architectural design and is thus orthogonal to state-of-the-art hardware managed schemes for hybrid caches. Furthermore, EnVM is backward compatible to the conventional SRAM/DRAM based memory systems.

References

[1] Y. Chen, W.-F. Wong, H. Li, and C.-K. Koh. Processor caches with multi-level spin-transfer torque ram cells. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, ISLPED '11, pages 73–78, Piscataway, NJ, USA, 2011. IEEE Press.

[2] Y.-T. Chen, J. Cong, H. Huang, C. Liu, R. Prabhakar, and G. Reinman. Static and dynamic co-optimizations for blocks mapping in hybrid caches. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 237–242, New York, NY, USA, 2012. ACM.

[3] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *SIGPLAN Not.*, 47(4):105–118, Mar. 2011.

[4] X. Dong, C. Xu, Y. Xie, and N. Jouppi. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(7):994 –1007, Jul 2012.

[5] Y. Huang, T. Liu, and C. Xue. Register allocation for write activity minimization on non-volatile main memory. In *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pages 129 –134, Jan 2011.

[6] A. Jadidi, M. Arjomand, and H. Sarbazi-Azad. High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement. In *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, pages 79 –84, Aug 2011.

[7] A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das. Cache revive: Architecting volatile stt-ram caches for enhanced performance in cmps. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 243–252, New York, NY, USA, 2012. ACM.

[8] Y. Joo, D. Niu, X. Dong, G. Sun, N. Chang, and Y. Xie. Energy- and endurance-aware design of phase change memory caches. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 136 –141, Mar 2010.

[9] U. P. Khedker, A. Sanyal, and A. Karkare. Heap reference analysis using access graphs. *ACM Trans. Program. Lang. Syst.*, 30(1), Nov. 2007.

[10] 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 *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, ISLPED '12, pages 351–356, New York, NY, USA, 2012. ACM.

[11] Q. Li, M. Zhao, C. J. Xue, and Y. He. Compiler-assisted preferred caching for embedded systems with STT-RAM based hybrid cache. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 109–118, New York, NY, USA, 2012. ACM.

[12] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Architecting efficient interconnects for large caches with cacti 6.0. *Micro, IEEE*, 28(1):69–79, Jan.-Feb.

[13] H. Naeimi, C. Augustine, A. Raychowdhury, S.-L. Lu, and J. Tschanz. Sttram scaling and retention failure. *Intel Technology Journal*, 17(1):54–75, 2013.

[14] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. *SIGPLAN Not.*, 44(6):397–407, June 2009.

[15] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference 2011 (DAC'11)*, 2011.

[16] N. Rinetzky, G. Ramalingam, M. Sagiv, and E. Yahav. On the complexity of partially-flow-sensitive alias analysis. *ACM Trans. Program. Lang. Syst.*, 30(3):13:1–13:28, May 2008.

[17] SPEC. Spec cpu2006. In *http://www.spec.org/cpu2006/*, 2006.

[18] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen. A novel architecture of the 3d stacked mram l2 cache for cmps. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 239–249, Feb 2009.

[19] Z. Sun, X. Bi, H. H. Li, W.-F. Wong, Z.-L. Ong, X. Zhu, and W. Wu. Multi retention level STT-RAM cache designs with a dynamic refresh scheme. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 329–338, New York, NY, USA, 2011. ACM.

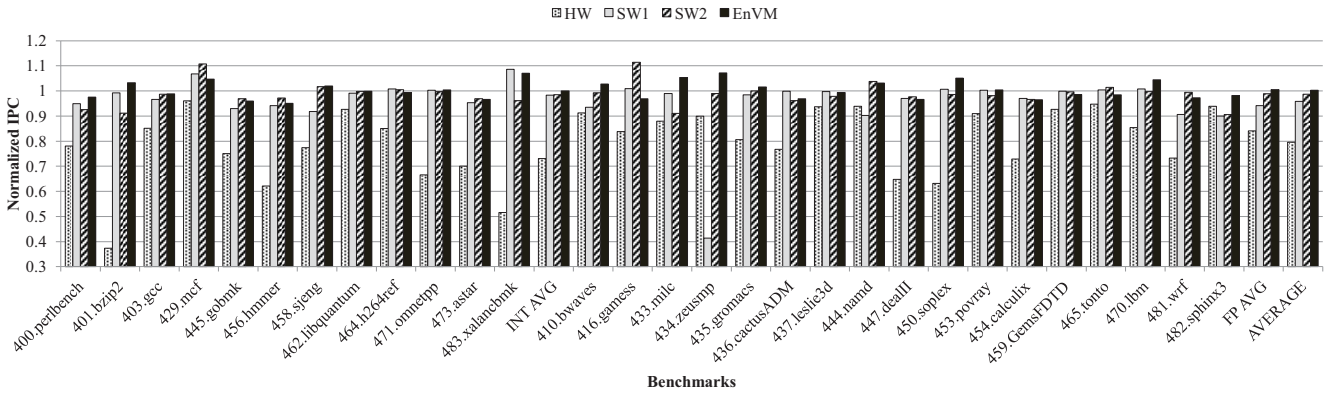


Figure 10: Instructions Per Cycle (IPC) normalized to conventional SRAM based cache design.

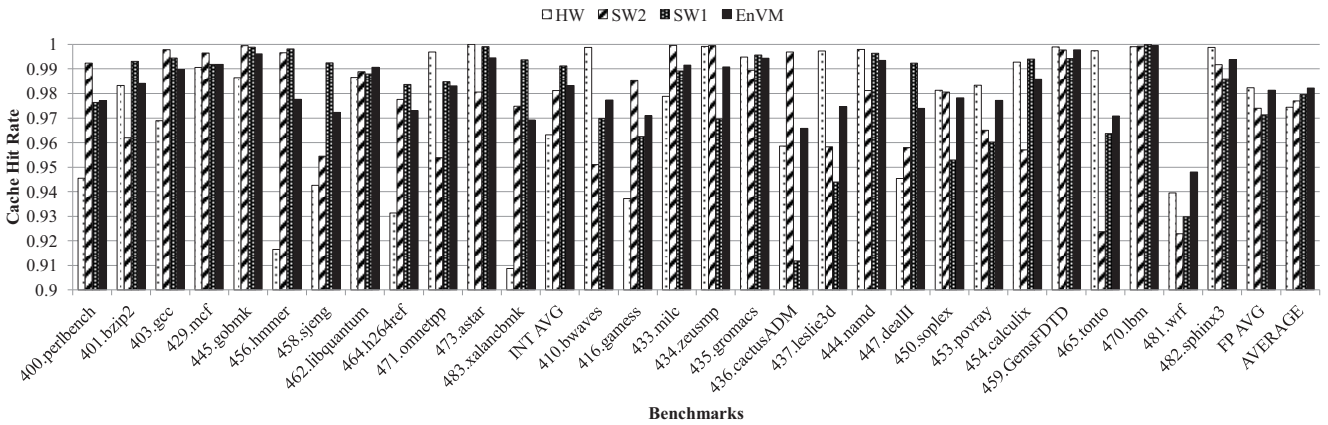


Figure 11: Cache hit rate for the hybrid L1 cache design.

[20] 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 *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 737–742, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[21] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pages 1–11, New York, NY, USA, 1994. ACM.

[22] C. J. Xue, Y. Zhang, Y. Chen, G. Sun, J. J. Yang, and H. Li. Emerging non-volatile memories: opportunities and challenges. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '11, pages 325–334, New York, NY, USA, 2011. ACM.

[23] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. Energy reduction for STT-RAM using early write termination. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 264–268, New York, NY, USA, 2009. ACM.