Software Techniques for Energy Efficient Memories

Pooja Roy
(M.S., University of Calcutta, 2010)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
December 2014

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis. This thesis has also not been submitted for any degree in any university previously.

(POOJA ROY)

Abstract

The recent times are known as the *dark silicon era. Dark* implies the percentage of the chip that cannot be switched-on at a given time to keep the power consumption in budget. As a consequence, researchers are innovating energy efficient systems. Memory subsystem consumes a major part of energy and so it is imperative to evolve them into energy-efficient memories. In the past few years, new memories such as resistive memories or non-volatile memories have emerged. They are inherently energy efficient and are promising candidates for the future memory devices. However, the application and program layer is not aware of the new memory and new architectural designs. Thus, the application layer is not specifically optimized for energy efficiency.

In this thesis, we propose compiler optimization and software testing methods to optimize programs for energy efficiency. Our techniques provide cross-layer support to fully utilize the advantages of the energy-efficient memories. In most of our works, we assume a resistive technology based hybrid memories as L1 data cache, L2, L3 and main memory level. In hybrid memory designs, data placement is critical as the resistive memories are sensitive to write operations. Therefore, it is common to place a smaller SRAM or DRAM alongside to filter the write accesses. However, caches are transparent to the application layer and so it is challenging to influence the data traffic to the caches at runtime. Our solution is a new virtual memory design (EnVM) that is aware of resistive technology based hybrid caches. EnVM is based on the memory access behaviour of a program and can control the data allocation to the caches. The merits of EnVM diminish at the main memory level, as the size of basic data unit differs from caches. Caches address cache line size data where as main memory addresses a page which is much larger. We propose a new operating system assisted page addressing mechanism that accounts for cache line size data even in the main memory level. Thus, we can magnify the effects of hybrid memory at the main memory level.

The next challenge is a characteristic of the energy-efficient memories that makes them prone to errors (bit-flips). This is not only true for the resistive memories, undervolted memories also exhibit such characteristics. Adapting error detection and correction mechanisms often offsets the gain in power consumption. We propose a framework that exploits the inherent error resiliency of some application to solve this issue. Instead of mitigating, it allows errors if the final output is within a given Quality of Service (QoS) range. Thus, it is possible to run such applications on the energy-efficient memories without having to provide error-correction support. In addition, the gain in energy efficiency is magnified. The above framework, based on a dynamic program testing accrues a large search space to find an optimal approximation configuration for a given program. The running time of the analysis and book-keeping overheads of such techniques scales linearly with increase in program size (lines of code). In out next work, we propose a static code analysis which deduces accuracy measures for program variables to achieve a given QoS. This compile-time framework complements the dynamic testing schemes and can improve their efficiency by reducing the search space.

In this thesis, we show that with proper support from the software stack, it is possible deploy energy efficient memories in the current memory hierarchy and achieve remarkable reduction in power consumption without compromising performance.

Acknowledgments

“You need the willingness to fail all the time. You have to generate many ideas and then you have to work very hard only to discover that they don’t work. And you keep doing that over and over until you find one that does work.” – John Backus

I thank my advisor Professor Weng Fai Wong, who placed his trust in me, and without whom this thesis would not be real. Prof. Wong has taught me all I know about research and the art of solving problems. I learnt from him the kind of rigor, focus and precision that is imperative in research. Not only he encouraged me to generate new ideas, to work hard on them till it comes to fruition, he is also the person I have always turned to regarding basics of compiler optimizations. I am especially thankful for his patience and his faith in me during the most difficult times of my research. I am always inspired by his integrity and sincerity. I hope to be a researcher and a professor of brilliance as his.

I thank Professor Tulika Mitra, for her constant support, valuable guidance and feedback. She has always been my inspiration since I joined the School of Computing. I thank Professors Siau Cheng Khoo and Wei Ngan Chin for their precious time and guidance. I thank Professors Debabrata Ghosh Dastidar and Nabendu Chaki, for their support throughout my undergraduate and graduate studies in India. I thank Dr. Rajarshi Ray and Dr. Chundong Wang for their support as seniors, Manmohan and Jianxing for being amazing colleagues.

I thank my friends in Singapore for making this city a home away from home. I am deeply thankful my wonderful roommates Damteii, Sreetama, Sreeja and Priti for taking care of me everyday. I thank my friends in Kolkata, especially Debajyoti, for their assurance and love in the times I needed the most. I thank all my seniors and friends of Soka Gakkai, especially Dr. M. Sudarshan, for their constant prayers and encouragements.

I thank all the staffs in Dean's office and the graduate department for helping me in administrative matters and for making it possible for me to attend conferences and present my work.

Finally, I thank my grandmother for she is my first friend and my first teacher, my uncle for his constant encouragements, my little cousins and my late aunt, who has a place next to my mother's in my life. I also thank all my close relatives for always making me feel pampered and loved. I thank Avik for his patience, love and for making my dreams his priority.

I thank my parents, who instilled in me the passion to study and provided me with all the faculties to pursue my dreams. Without their love and support, I would not have been anything near to what I am today. Lastly, I thank my mentor in life Dr. Daisaku Ikeda, whose words of encouragement kept me going through the roller coaster ride of my doctoral studies and to whom I dedicate my thesis.

To Sensei.

Contents

List of Figures

List of Tables

List of Algorithms

List of Publications

1. Pooja Roy, Manmohan Manoharan, Weng Fai Wong. *Fine Grain Management of Non-Volatile Hybrid Main Memories*, Manuscript in preparation.

2. Pooja Roy, Jianxing Wang, Weng Fai Wong. *PAC: Program Analysis for Approximation-aware Compilation*, Working Paper.

3. Pooja Roy, Manmohan Manoharan, Weng Fai Wong. *EnVM : Virtual Memory Design for New Memory Architectures*, In Proceedings of the 2014 International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES 2014), Article No. 12, New Delhi, India, October 12 - 17, 2014, ACM.

4. Pooja Roy, Rajarshi Ray, Chundong Wang, Weng Fai Wong. *ASAC: Automatic Sensitivity Analysis for Approximate Computing*, In Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES '14, pages 95 - 104, Edinburgh, UK, June 12 - 13, 2014, ACM.

5. Pooja Roy, Manmohan Manoharan, Weng Fai Wong. *Write Sensitive Variable Partitioning for Resistive Technology Caches*, 51^{st} Design Automation Conference (DAC), poster, San Francisco, USA, June 1 - 5, 2014.

Chapter 1

Introduction

The evolution of computer systems has reached a juncture where the percentage of chips that can be utilized, keeping the power consumption within a budget, is decreasing exponentially. This is commonly known as the *utilization wall* or the *power wall*. As memory devices are the primary consumers of power, it is imperative to evolve them into energy efficient memories. Architectural innovations have been explored and applied extensively to make the memory devices energy efficient. Dynamic voltage/frequency scaling (DVS/DVFS) based memories, non-volatile memories (NVMs, Flash), reconfigurable memories are some of the widely accepted examples. In this thesis, we attempt to explore software techniques to enable improved utilization of the energy efficient memories.

1.1 Energy Efficient Memories

There are broadly two kinds of energy efficient memories. First, memories that are built with low power consuming devices or materials. Non-volatile memories such as flash, NAND flash, magnetoresitive random access memory (MRAM), spin transfer torque random access memory (STT-RAM), phase change memory (PCM), racetrack or domain-wall memory (DWM) are some of the examples.

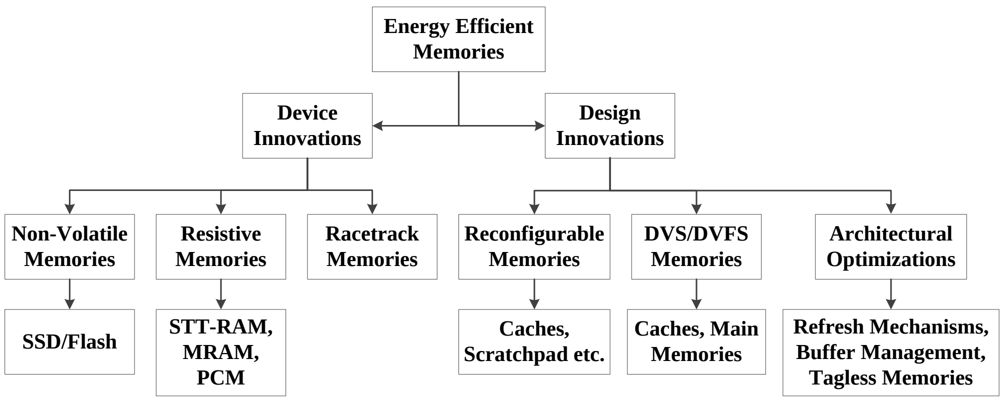


Figure 1-1: Broad classification of energy efficient memories

Second class energy efficient memories are the ones that are operated in an optimized fashion to reduce their power consumption. These are essentially architectural designs that apply to any type of memory device. However, such optimization techniques depend on the level of the memory device in the memory hierarchy. For example, refresh mechanisms for DRAM based main memories reduces the number of times a DRAM bank is periodically recharged and this is one of the earliest attempts to reduce power consumption. Operating memory devices at different voltage and frequency levels is another way of optimizing them for power, often known as DVS/DVFS based memories. Recently, reconfigurable caches, where the number of sets and ways can be dynamically controlled depending on some constraints are also being extensively researched for energy efficiency of the memories. Figure 1-1 illustrates the classification of the energy efficient memories that will aid in understanding the perspective of this thesis.

Limitations of Conventional Memories

In a discussion on energy efficient memories, it is important to describe the limitations of the conventional memory devices and architectures. First, let us examine the SRAM devices. SRAM is widely used to build processor caches. SRAM is fast, which makes it suitable to be placed very close to the performance critical pipeline. However, SRAM suffers a power penalty in terms of leakage current. As the technology node scales and capacity increases, the leakage current of SRAM becomes a more serious concern. Therefore, for higher capacity off-chip memories, DRAM is the usual choice. DRAMs are denser and cheaper compared to SRAMs. Though they do not exhibit leakage current component, the power ditch is the refresh energy. DRAM cells discharge with time and thus need to be refreshed to keep the data alive. This refresh mechanism constitutes the majority of the power consumption in DRAMs.

Multi-core systems demand larger memory on and off-chip to be able to provide higher compute power and functionality. On the other hand, low-power embedded devices such as smartphones and tablets, though do not demand huge compute capabilities, poses higher power constraints in terms of battery provision. In both scenarios, the demerits with respect to power consumption, makes it difficult to put more SRAM and DRAM to suffice the requirements and constraints. Therefore, the gradual shift from conventional memory designs and devices to energy efficient memories is inevitable.

Resistive Memory Devices

Resistive memory devices are essentially non-volatile memories that are capable of retaining data independent of the power supply. Therefore, they are free from leakage current or refreshes. Resistive memories such as MRAM, STT-RAM and PCM are well studied and considered for on-chip and off-chip memory levels. Specifically, STT-RAM is considered as a suitable device for processor caches. They are 4x denser than SRAM, which either provides bigger caches or reduces the silicon area budget of the chips. At the main memory level, PCM is considered to be the next alternative of DRAM providing faster and bigger off-chip memories. However, these memories have few drawbacks. First, the access latencies of load (read) and store (write) are asymmetric. The memory write access is usually 3x longer than memory read. Secondly, the write endurance of the resistive memories is much lower than their conventional counterparts. Write endurance is defined as the maximum number of write operations a memory cell can endure before failing permanently. Moreover, the write current is also higher and so, the resistive memories are also known as write-sensitive memories.

Therefore, if the resistive memories receive a large amount of write operations without any control, the lifetime of the entire chip will be reduced. The non-volatility of the resistive memories could be relaxed to gain lower access latency for memory read and write. The time period for which it can preserve memory content without a refresh is known as the *Retention Time.* However, beyond the retention time, these memories are susceptible to stochastic error in terms of single or multiple bit-flips. This characteristic is similar to that of soft errors in the conventional memory devices. Such errors are inherently a part of dynamic voltage and frequency scaled memories, which is described in the following section. We will refer to this issue as error susceptibility. Table 1.1

Features	**SRAM**	**DRAM**	**MRAM**	**PCM**	**STT-RAM**
Non-Volatile	No	No	Yes	Yes	Yes
Cell-Size(F^2)	5-120	6-10	16-40	6-12	6-20
Read Latency (ns)	1-100	30	3-20	20-50	2-20
Write Latency (ns)	1-100	15	3-20	50/120	2-20
Endurance	10^{16}	10^{16}	$>10^{15}$	10^{8}	$>10^{15}$
Write Power	Low	Low	High	Low	Low
Other Power Consumption	Leakage	Refresh	None	None	None

Table 1.1: Comparison of features of different memory technologies

shows a comprehensive comparison of all the memory technologies mentioned above.

DVS/DVFS Based Memory Designs

In a DVS or DVFS based memory, the voltage or frequency is dynamically changed to reduce power consumption. Decreasing the operating voltage of a memory is also known as *undervolting.* Together with reducing power consumption, undervolting also reduces reliability and renders the memory prone to errors. DVS/DVFS is a popular energy controlling mechanism at all levels of memory hierarchy. Beginning from instruction and data L1 caches, it can be applied to all cache levels and aptly to main memories too. Researchers have explored many novel architectures and policies to utilize DVS/DVFS based memories. However, the error handling and book-keeping involved in all such techniques, always negates the energy gain to an extent.

1.2 Motivation & Goal

In this thesis, we would explore the various possibilities of deploying energy efficient memories at various levels of the memory hierarchy. Specifically, we would propose compiler and software assisted techniques that unleash the full potential of these memories. We base our works on *hybrid memory* architectures. In hybrid memory systems, a resistive memory is supported by a conventional SRAM/DRAM memory with a smaller capacity to filter out write accesses. Summarizing the scope and attempt of this thesis in a comprehensive way -

- We assume an energy-efficient memory hierarchy consisting of resistive technology based hybrid memories at each level. Though these memories will exhibit similar properties, the implications are different when they are placed at different levels of the memory hierarchy.
- Specifically, we will focus on compilation and software techniques and how such methods can be applied to aid the energy-efficient memories.
- Finally, we would engage our efforts to deal with two specific challenges, namely, **write sensitivity** and **error susceptibility** of energy efficient memories.

Overall, we attempt to answer the following question -

How to optimize programs so that they can alleviate the weaknesses

of the energy-efficient memories in the underlying hardware architecture?

Software Support for Memory Hierarchy

Usually, it is a common practice to analyze and optimize program code based on the underlying hardware on which it is expected to be executed. Information on the program code is used to optimize and compile it so that it gains maximum in terms of performance and correctness at runtime.

For example, *Registers* are one of the very limited, yet important hardware resources. Registers play a key role in the performance, as they are situated closest to the processor. *Register Allocation*, therefore, is a very significant step in the compilation process that determines which variables could be allocated to registers and at what point of program execution should they be written back to the memory. As the numbers of registers are limited and in contrast, the numbers of variables in a program are much larger, it is a difficult task to sieve and allocate the variables to registers in an optimal fashion. Register allocation techniques are well-studied over decades and still it remains one of the most important research areas as it plays a significant role in the performance of systems.

However, as the power consumption has now become a threat to further evolution of computer systems, program codes are being analyzed and transformed in various new ways, so that it consumes less energy too, while it is executed. In pursuit of this, it is essential to examine if existing off-the-shelf program analysis techniques already provide such a support.

In this thesis, we are chiefly concerned with energy consumption of memory devices. When a program is analysed for its memory usage, generally the load and store instructions are of prime importance. In most of the conventional program analysis and optimization techniques, the memory accesses are considered to be symmetric i.e. a read access is equivalent to a write access in terms of latency and probe. In addition, correctness of the program output is regarded as the goal while optimizing programs for a particular underlying architecture.

While the above-mentioned assumptions are no longer valid for architectures using energy efficient memories, it is therefore, imperative to design new program analyses and optimizations to perceive the advantages of energy efficient memories.

1.3 Contributions

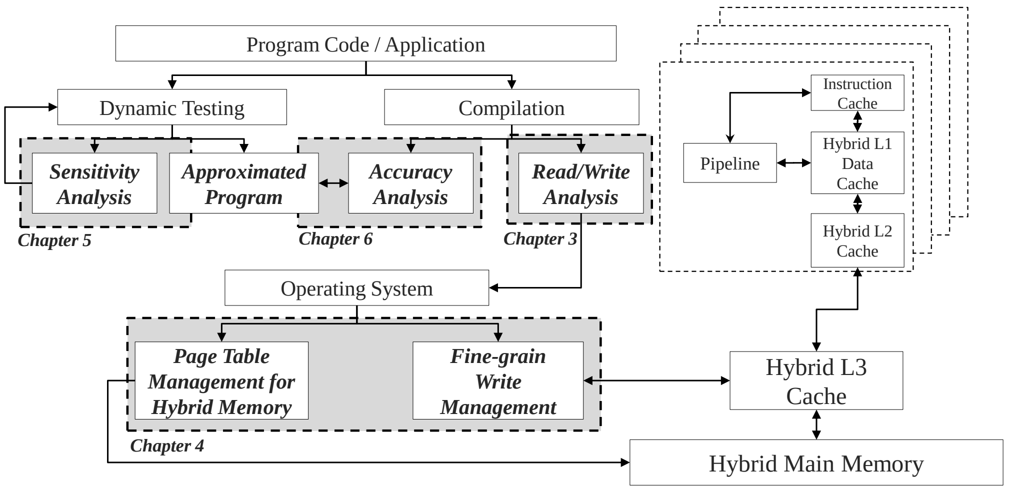


Figure 1-2: A comprehensive illustration of the scope of this thesis.

In this thesis, we would explore the various ways a program can be optimized for a completely energy efficient memory hierarchy. Figure 1-2 illustrates the possible influences of software and compiler techniques over memories at different levels of the memory hierarchy and a comprehensive illustration of the scope of this thesis. The *gray* boxes represent the works proposed in this thesis.

1.3.1 Write Sensitivity of Hybrid Memories

Optimizing Programs for Hybrid Caches

Caches are the most critical memories to the performance of a system. A resistive memory based cache hierarchy as the next generation of on-chip memories is well explored. However, as mentioned before, if caches are built with resistive memory technology, they will be sensitive to write operations. Compilation techniques that are aware of this write sensitivity and access latency asymmetry are able to support the resistive memories on behalf of the software stack. Differentiating between read and write operations would not only enhance performance, reduce power consumption, it will increase the lifetime of the chips also. Unfortunately, caches are transparent to the application layer. The only way to control the data allocation to the caches is to influence the physical address of memory objects. The physical address of memory objects are strongly mapped to the virtual addresses.

Therefore, we propose a new virtual memory design, *EnVM*, which is aware of resistive memory based hybrid caches. In particular, we assume a STT-RAM and SRAM based hybrid cache, deployed at any level of cache hierarchy. Virtual addresses are generated according to memory access behaviour of the program variables. Read and write intensive data are allocated separately in the virtual memory area, introducing a data locality based on the memory access behaviour. The new virtual memory layout is implicitly used to allocate data to STT-RAM 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. Chapter 3 elaborates more on this work.

Operating System Assisted Hybrid Main Memories

EnVM is capable of influencing data allocation to all the memories in the entire memory hierarchy from L1 caches to the main memory. As it is a virtual memory design, unique to a process, it is also applicable to multi-core and multi-tasking environments. EnVM is supported by a small hardware component which is coupled with the address translation unit. Thus, it closely monitors and intercepts cache fills and writebacks. Read and write intensive data are read and written back to the resistive and conventional SRAM/DRAM partitions respectively, in all levels of caches. However, the data exchange between the last level cache (LLC) and main memory is different in nature. The unit of data copied between the caches is the size of cache lines (say 64 bytes), generally same for different levels of caches. In case of LLC writebacks, there is a disparity between the sizes.

LLC usually maintains a cache line size writeback. On the contrary, the main memory maintains data in units of pages (say 4KB) which is much larger than the cache line size. Therefore, any read or write intensive data that is written back from the LLC under the influence of EnVM, has no guarantee to maintain the locality based on memory access intensity in the main memory too. As the page size is large, it is difficult to allocate all the read and write intensive data separately in the resistive and DRAM partitions. To achieve that, the virtual memory area should be aligned with page size containing same size chunks of read and write intensive data, which is very unlikely.

So, we propose a new operating system assisted, LLC writeback scheme to the hybrid main memory. In this technique, the main memory maintains sub-page level data and is able to differentiate between dirty and clean data at the cache line size granularity. The key mechanism is that the LLC always writes back to the DRAM partition and LLC fills are always served by the resistive memory partition. This interaction and mapping of sub-page level activity is entirely maintained by the operating system. More details on this work are included in Chapter 4.

1.3.2 Error Management of Hybrid Memories

Dynamically Testing Programs for Approximation

With the two techniques mentioned above, the entire software stack is aware of the underlying hybrid memory system. The applications and operating system assists the memory sub-systems to achieve energy efficiency and performance. Hence, the write sensitivity problem of the resistive memories is now acknowledged. Next, would focus on error susceptibility issue of these memories. Resistive memories are exposed to stochastic errors, which are commonplace for the DVS- DVFS based memories too, commonly known as soft-errors. Many researchers have proposed error detection and error correction techniques for reliability against soft-errors. This implicitly assumes a framework that ensures correctness of a program even at the cost of power consumption. In addition, such methods demand high book-keeping overheads. On the flip side, with popularity of high configuration embedded devices such as smartphones and tablets, power constraints in terms of battery usage have become a bottleneck.

Many applications that are usually run on these devices are resilient to errors to some extent. In other words, accuracy of some applications can be relaxed i.e. approximated, if there is a reduction in power consumption as a consequence. In our third work, we propose a framework to analyse a program to extract approximable data which, even if incurs errors, will not lead to catastrophic failure of the application and will produce output within an acceptable quality of service (QoS) band. We propose a dynamic testing framework based on statistical sensitivity analysis which characterizes program data into critical and approximable classes. The approximable data are allocated to the resistive or DVS/DVFS based memories and other data to SRAM/DRAM. The apt usage of energy efficient memories to hold approximated program data reduces the power consumption required to maintain the correctness or mitigate errors. Chapter 5 elaborates on this work in details.

Statically Analyzing Programs for Approximation

Dynamic testing frameworks involve computationally intensive algorithms and profiling of applications to characterize approximation spaces in a program. They are based on large search spaces with a goal to find a near-optimal approximation configuration for a given application. The ideal configuration is one that would minimize the energy consumption of the application during runtime with no QoS loss. However, it is a difficult problem and thus, the state-of-the-art solution involves programmer's expertise to manually annotate the applications for possible approximations. Our previous work attempts to alleviate programmer's effort and generates approximation spaces automatically with a penalty of a complex and compute intensive analysis.

In this work, our aim is to statically analyze a program to extract approximations in program variables based on the required correctness (QoS) of the output variable. As a compile time analysis has limited knowledge about program runtime, our ulterior goal is to merely reduce the huge search spaces the dynamic testing based methods incur, by heuristically determining possible approximations. Chapter 6 elaborates on this work in details.

1.4 Thesis Outline

This thesis continues with an extensive study on the related literature and state-of-the-art techniques in Chapter 2. We introduce our first proposal of a static analysis and code generation technique for the deployment of hybrid memories as processor caches in Chapter 3. Further, we propose a system-wide and operating system assisted framework to support hybrid memories at the main memory level in Chapter 4.

After the previous two proposals to solve the write sensitivity of the hybrid memories, in Chapter 5, we propose a solution to mitigate the error susceptibility issue of energy efficient memories. We continue by elaborating on the limitations of the proposed technique and thereby, proposing a complementary static analysis in Chapter 6. Finally, the thesis concludes in Chapter 7.

Chapter 2

Background & Related Works

In this chapter, we would elucidate on the existing literature and researches related to resistive memories and their usage to reduce the energy consumption of computer systems. First, we would start with a short description on the device level details of resistive memories followed by various schemes to deploy them in the current memory hierarchy.

2.1 Resistive Memories

Resistive memories are *memristor* [2] based non-volatile memories. Recent studies [3–7] show that they are promising as next generation alternatives to SRAM and DRAM. Resistive memories are inherently energy efficient and provide better performance than other non-volatile memories like NAND Flash etc [8, 9]. One variety of resistive memory, namely, STT-RAM (Spin Torque Transfer Random Access Memory) is a suitable candidate for processor caches and thus, can be an alternative to SRAM [3, 4, 10–12]. STT-RAMs are denser (4x) than SRAM and do not exhibit any leakage current, thus, highly energy efficient. With the increasing demand of many cores and network-on-chip architectures, denser and power efficient caches like STT-RAMs opens a way forward for Moore's scaling.

Other works [5–7, 13] suggest that a class of memories, namely PCM (Phase Change Memories) which are similar to the resistive memories and shares all the merits and demerits, are good candidates for main memory as an alternative to DRAM.

However, resistive memories disclose two main drawbacks which hinder them from being adapted in the memory hierarchy in a straight-forward fashion. First, *write sensitivity*, i.e. the read and write access latencies are different. A memory write requires longer (3x) than a memory read. In addition, write current is higher than read current. Thus, writes to resistive memory devices are expensive and critical to performance and lifetime. Second drawback is *error susceptibility* of the resistive memories. Smullen et al. reduces the write latency of the resistive memories by introducing a relaxed non-volatility design [14], which exposes the resistive memory cells to stochastic errors. The relaxed non-volatility entails these devices with a retention time - a time interval for which a memory cell can hold the content without being refreshed. Beyond the retention time, the memory cells are susceptible to errors.

2.2 Write Sensitivity of Hybrid Memories

Due to the above idiosyncrasies researchers have proposed a *Hybrid Memory* design which comprises a large partition of STT-RAM/PCM assisted by a small SRAM/DRAM partition to aid the write sensitivity of their counterparts, as shown in Figure 2-1.

Figure 2-1 illustrates a simple hybrid memory hierarchy with hybrid cache(s) and hybrid main memory. There are two main challenges -

- **Data Allocation** - A random data allocation to the two partitions of hybrid memory may result in unaccounted write operations in the resistive memory. Therefore, it is important to allocate data to the two partitions

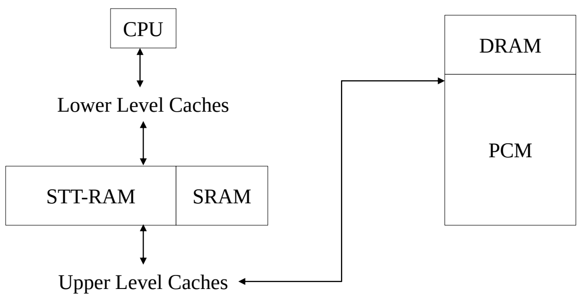


Figure 2-1: Simple hybrid memory hierarchy

wisely. Depending on which level of memory hierarchy the hybrid memory is placed in, the data allocation policy will have different implications.

- **Write Reduction** - In addition, the data allocation strategy should be such that the writes to the resistive memory are minimized. The write reduction is of prime importance as it impacts the performance, writes being 3x slower and also the lifetime of the chip, as the write endurance is lower.

2.2.1 Hybrid Caches

Towards the reduction of writes in hybrid caches comprising SRAM and STT-RAM, *data migration* techniques have been proposed where cache blocks are migrated to SRAM to absorb write accesses, and then moved back to the STT-RAM from where they can service read requests [3, 15]. However, such hardware managed schemes require significant energy overhead for the additional hardware units which can offset the energy gain. Moreover, the migration traffic is a serious concern. Zhou et al. [16] suggested a method to reduce writes by performing a read operation before the write operation. This checks if the write operation is redundant i.e. rewriting the same data. Such redundant writes are terminated and the total number of writes to STT-RAM is reduced. These works require both runtime and hardware support, and thus poses significant overhead. Most of the hybrid memory management techniques are hardware controlled. A few schemes, concentrating on compiler assisted and profiling techniques have been directed at embedded system where the applications are stable and known ahead of time [17–19].

Hybrid L1 Cache

Deployment of STT-RAM in the L1 cache is most challenging problem as L1 cache is closest to the processor and hence is time critical [20]. Li et al. [15] 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, they proposed a preferential cache allocation policy that places migration intensive blocks into SRAM to further reduce write accesses to STT-RAM [17].

Hybrid L2 & Last Level Cache (LLC)

Mao et al. [21] proposed a novel prefetching technique for STT-RAM based LLC to reduce write accesses due to aggressive prefetching. This method demands extensive hardware support. Chen et al. [19] presented 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 each cache line. 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.

2.2.2 Hybrid Main Memories

As main memories are further away from the processor and pipeline, advantages of using resistive memory (PCM) at the main memory level are enhanced. There are two types of architectures proposed for the hybrid memory as shown in 2-2. In the first type (2-2a), the DRAM is seen as a last level cache of the system. In order to do this, DRAM must be stacked on the CPU chip using 3D die stacking techniques. The second type (**??**) of hybrid memory has the DRAM occupying a separate address range in the physical address space of the processor. This is the architecture envisioned in our work. The main objective is to enhance the lifetime of PCM and improve the overall system write performance.

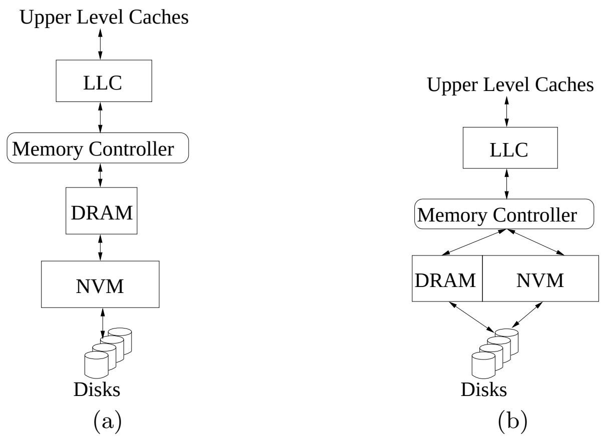


Figure 2-2: Different designs of hybrid main memory

For the first type, Qureishi et al. [7] suggested using DRAM as an LLC with a sophisticated cache controller. They also suggested a mechanism to improve the access latency of hybrid main memories that adjusts the scheduling of memory accesses using write pausing [22]. Architectures with DRAM as the LLC, requires on-chip tag stores implemented in SRAM. For very large DRAM caches, the overhead associated with storing the tag array is significant. Dong et al. [23] reduced the size of the tag stores by using very large cache line size in the DRAM cache. Though this reduces the tag store, fragmentation and increased traffic when fetching data from the PCM memory worsens memory bus contention. Loh et al. [24, 25] overcomes the issue of on-chip tag storage by storing both data and tag in the same DRAM row. The latency associated with a tag lookup from the DRAM is reduced through a parallel on chip lookup structure called Missmaps, and a technique called compound access scheduling where data and tag lookup is scheduled side by side in the same memory transaction. Zhou et al. [16] manages the DRAM cache with the aim of reducing writebacks to the PCM memory. This work also distributes writebacks among write queues evenly to spread the writes across PCM, popularly known as *wear levelling*.

Among other works that assume the second type of hybrid memory architecture with a disjoint address space and arranged linearly [5, 26–29], Dhiman et al. [5] proposed a technique based on counting the number of writes to individual PCM frames. Once the count reaches a threshold, the data is moved to a DRAM frame.

Zhang et al. [26] introduced a similar concept of recording the writebacks to individual frames of an on-chip DRAM memory. A multi-queue (MQ) algorithm is used to migrate write intensive pages from PCM to DRAM. Implementing on-chip tables to store writes to individual PCM frames is not scalable. The storage overhead associated with storing these tables may not always be realizable for large scale systems with terabytes of PCM memory.

Ramos et al. [27] used another kind of memory controller that implements a modified MQ algorithm to rank page frames. The pages are migrated to DRAM on the basis of the read and write references. The memory controller performs page migration between DRAM and PCM without support from the OS.

A purely OS-based hybrid page management technique implemented in the Linux kernel was explored by Park et al. [28]. The page fault handler is modified to allocate DRAM frames to writable memory regions of the process, while non-writable regions are allocated PCM frames. Shin et al. [29] made use of a kernel daemon to monitor the write activity to the pages by scanning through the page tables of all the processes running in the system. Energy is saved by powering down blocks of DRAM memory after migrating these pages back to PCM.

2.3 Error Susceptibility of Hybrid Memories

Smullen et al. [14] proposed a design of STT-RAM where the non-volatility property is relaxed and the write access is faster as a consequence. Relaxing the non-volatility of the resistive memories, introduces the concept of *Retention Time* - the maximum time period for which a memory cell can keep data alive without a refresh operation. Therefore, relaxed non-volatility designs of STT-RAM and PCM demands a regular refresh mechanism, without which, the memory cells are susceptible to stochastic errors beyond the retention time.

Various authors suggested different STT-RAM and PCM cell structures with different retention times [4, 14, 30], varying from 10ms to 24.5μs. Li et al. [31] proposed smart refresh scheme for such STT-RAM based caches, while other works relied on error detection and correction (EDC) to ensure data integrity [32, 33]. Refresh techniques are power hungry and have performance impact. On the other hand, EDC schemes have high book-keeping and space overheads.

In this thesis, we approach the error susceptibility issue of the resistive memories with a perspective to allow errors resulting in a graceful degradation of application and as a consequence reduce power consumption. This would not only apply to resistive memories, but would also aid other energy-efficient memories such as DVS/DVFS based devices that are error prone. Thus, the abatement of power consumption is twofold. First, using energy-efficient memories without any refresh or EDC mechanisms saves significant overhead. Second, by trading-off accuracy of application, it is possible to achieve better performance which reduces runtime and saves power. In recent literature, such an approach is popularly known as *Approximate Computing.*

2.4 Approximate Computing

There are many popular applications in commercial embedded devices that do not require a strict QoS as long as they meet an acceptable threshold [34]. Building on this idea, approximate computing has gained much attention. It allows programs to relax their accuracy in order to save on energy consumption, instead of focusing on mitigating soft-errors in programs [34–36].

2.4.1 Approximation in Programs

Recently there have been wide explorations on how to allow a disciplined approximation to relax the accuracy of a program and reduce energy consumption as a consequence [1, 37–41]. Approximation is achievable at different levels of abstraction such as code approximation, program approximation, approximate computer architectures and device level approximations.

Baek et al. [37] suggested the idea of this trade-off by proposing a loop and function approximation framework. In this, the programmers are expected to provide multiple versions of a function or a loop structure. The framework consists of a calibration that generates different QoS models and allows a graceful QoS loss during runtime to save energy. However, this solution places a demand on the programmer's expertise and involvement. With the popularity of open-source application development for embedded devices, it is generally not feasible to request multiple versions of a code to allow approximate computing. In addition, compiling (or re-compiling, in case of legacy software) a program with extra versions of functions and loops would result in code bloat and larger executables which is not suitable for tight budget and low power devices.

Sampson et al. [1] proposed a type-qualifier based programming paradigm, EnerJ, to facilitate approximation of program data. This ensures safety in terms of maintaining a distinction between approximate and precise computation of program data. Only with explicit programmer's endorsements, a conversion from precise to approximate or vice-versa is allowed. It provides an exclusive compiler to generate instructions for the underlying dynamic voltage scaling-based hardware called “Truffle” to switch between high and low power modes [39].

Carbin et al. [42] proposed a technique that classifies code regions into approximable and critical by a training method that uses fuzzed input data. Depending on the program path taken by different inputs, it is able to identify critical program regions and approximable regions. In other works, they propose program transformations and code generation techniques to allow approximate computation. The motivation in these works is to save computation power i.e. loop iterations, floating point operations etc. One such method is known as “loop perforation” where loop iterations are skipped in order to save computation which results in approximation in the output [43]. Misailovic et al. [44, 45] proposes probabilistic accuracy tests to allow for program level approximations. However, in most of the above mentioned works, one common drawback is the programmer’s involvement, and the lack of scalability.

Shafique et al. [46] proposes a technique to discover errors that are masked by program flow and operations on data. This indicates an inherent error resilience and approximation capability of a program. However, this is based on static code analysis and thus is not accurate as a whole program optimization framework. Error concerning only statically allocated data and compile-time inferable computation is exposed to this technique. Program data that are dynamically allocated or are influenced by runtime computations are hard to analyse.

2.4.2 Approximation in Hardware Devices

Approximation can be introduced in both the hardware and software stack. There are many other works investigating and designing architectural or device level approximation infrastructures [47–49]. Chippa et al. [50] presented a work on characterizing error resilience in applications based on approximate adders.

They also proposed "Impact", an approximate adder circuit that saves energy by approximating addition operations [51]. Many other designs for approximate adders have been proposed thereafter [52–56]. Liu et al. [40] proposed a DRAM refresh mechanism that protects critical data and approximates non-critical data to save refresh energy. Kahng et al. [52] propose an accuracy-configurable adder which can adaptively adjust during runtime based on the required accuracy. Memories that can control power supply at the bit level are widely explored [57–59]. For such memories, it is imperative to know how many bits of a variable are approximable. Finally, Nesenbergs et al. [60] proposed an approximate comparator that runs on low power and reduces the cost of comparisons which are otherwise highly expensive.

Chapter 3

Compilation Framework for Resistive Hybrid Caches

In this chapter, we introduce a compiler framework to attenuate the write impact on hybrid memories. Specifically, we would propose a new virtual memory layout which is aware of the hybrid memory architectures at the processor cache level. We propose analyses targeted towards both statically and dynamically allocated data and generate program executable with new virtual addresses according to their memory access behaviour. Such an approach is able to influence all memory devices in the memory hierarchy simultaneously.

3.1 Motivation

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 and therefore it is not aware of write sensitivity issue of hybrid memories.

From systems' perspective, allocating data without differentiating between read and write accesses, is detrimental to the hybrid memories. Unmonitored and excessive write operations can impede performance, and reduce the lifetime of the on-chip hybrid caches and hence of the processors [9, 11]. So, 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. Recent works have explored novel data allocation techniques towards efficient utilization of the hybrid caches. Chen et al. [19] 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 STT-RAM to SRAM to ameliorate the write endurance issue. Li et al. [17] 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 STT-RAM. 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 STT-RAM 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.

For example, in the algorithm by Chen et al. [19], 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 other algorithm [15] 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 weakening 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.

3.2 Our Proposal

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. 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 essential for maintaining performance and energy efficiency.

We propose a new virtual memory design *EnVM*, which is aware of hybrid 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 analysis that generates virtual addresses for statically allocated data used for 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 tendency i.e. read or write intensive. 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 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 the hybrid memory partitions i.e. SRAM and STT-RAM.

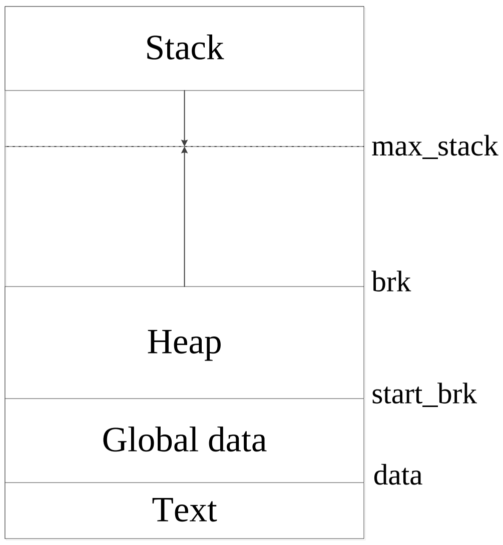
There are several advantages of EnVM. First, it is able to influence the data allocation across all levels of memory hierarchy without requiring specialized hardware at each level. This helps in easier adoption and scalability to deeper cache hierarchies. Secondly, EnVM provides a holistic design for 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 [61] and compare EnVM with two existing works on software assisted data allocation for hybrid caches [17, 19]. Details of evaluation and experimental results are presented in Section 3.6.

Summarizing the contributions of this work -

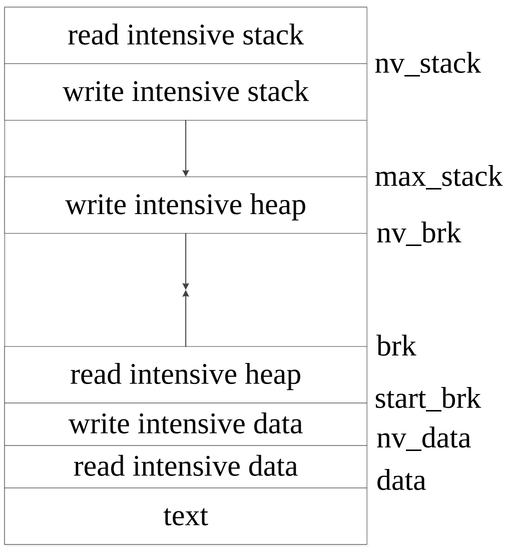
- We propose EnVM, the first virtual memory design that is aware of memory hierarchies built using the new memory technologies. EnVM provides an uniform data allocation mechanism to all the levels of memory hierarchy. This is an important step towards an *all hybrid memory* based memory hierarchy.
- 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 STT-RAM.
- We propose a new programmer's interface to be able to allocate read and write intensive heap memory exclusively during runtime with the help of optimized system libraries and the operating system.
- EnVM is the only virtual memory design that enables data allocation to hybrid caches built with SRAM and STT-RAMs. It utilizes existing hardware and advocates migration-less hybrid cache design.

3.3 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 shown in Figure 3-1a. EnVM contains fine-grained logical segments that are based on the memory access affinity of the memory objects as shown in Figure 3-1b. In other words, memory objects that exhibit read affinity are placed separately from those that show write affinity.



(a) Classic VM



(b) EnVM

Figure 3-1: Existing and proposed virtual memory design for hybrid memories.

In order to examine the nature of programs, we examined benchmarks from SPEC2006 [62] for the total proportion of variables showing read and/or write affinity in each application. The results presented in Figure 3-2, 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 90% variables (on average) shows affinity towards either read or write accesses. On an average, 59% of variables show read affinity over write and 31% show write affinity over read. Thus, it is viable to allocate these two classes to the STT-RAM and SRAM respectively. The premise of this work rests on this aspect.

In EnVM, the read and write intensive groups are separated by segment boundaries known to the operating system (OS). At runtime, the OS manages 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.

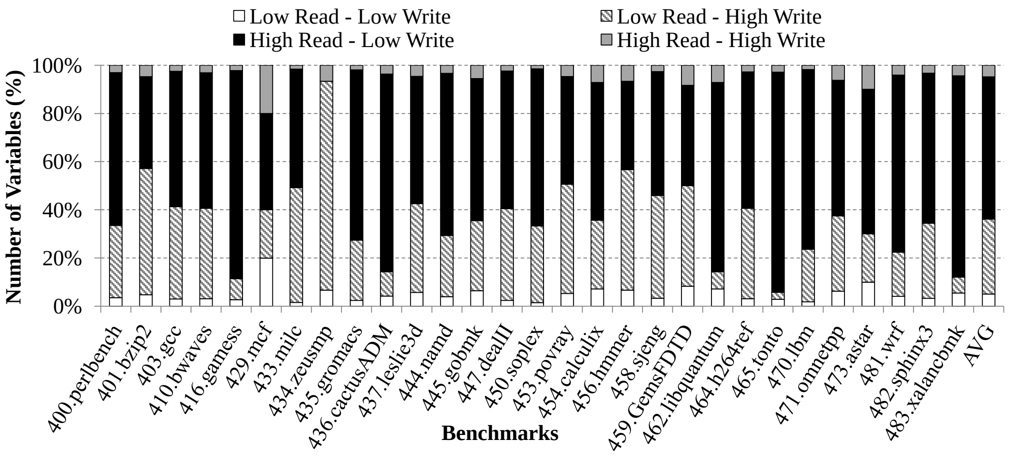


Figure 3-2: Percentage of variables in a program with certain memory access affinity.

3.3.1 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 3.3.1 (Abstract Domain). *The abstract domain of the analysis is a tuple containing an identifier for the variable, it's 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, $((V, \mathbb{N}, \mathbb{N}) \cup \{\top\}, \sqsubseteq_F)$, where $\top$ is the top element and $\sqsubseteq_F$ is the partial order defined as*

$$(X \sqsubseteq_F Y) \textit{ iff } (X(V) \sqsubseteq_{RW} Y(V)) \textit{ for each variable } V_i \in V \tag{3.1}$$

$$(V_i \sqsubseteq_{RW} V_j) \textit{ iff } (R_i \leq R_j) \wedge (W_i \leq W_j) \tag{3.2}$$

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 analyzing 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 phenomenon is described later with the discussion of *meet* operator. Each instruction i, in a basic block is passed through two *transfer functions*, $\boldsymbol{F}$ and $\boldsymbol{B}$, for forward and back edges, respectively.

Definition 3.3.2 (Transfer Function)**.** *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*

$$F_i(X) = Gen[i] \sqcup Probe_i(X) \tag{3.3}$$

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 -

$$\forall V' \in X, \ \textit{where } V' = (V, R, W)$$

$$\begin{aligned} R &= R + 1 \ \textit{if } \boldsymbol{i} \textit{ reads } V \\ W &= W + 1 \ \textit{if } \boldsymbol{i} \textit{ writes } V \end{aligned} \tag{3.4}$$

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 $\boldsymbol{i}$ succeeding instruction $\boldsymbol{j}$ through a back edge, all variables $V \in$ instructions between $\boldsymbol{j}$ and $\boldsymbol{i}$, $R = R + k$ and $W = W + k$, where k is a static loop bound [63]. This will have a similar effect as going through the loop instructions k times. When resolving branches and ϕ functions, we apply the *meet* operator $\sqcup$.

Definition 3.3.3 (Meet Operator)**.** *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*

$$(V_i, R_i, W_i) \sqcup (V_j, R_j, W_j) = \begin{cases} \top \text{ , } \mathit{if}(V_i = V_j) \wedge (V_i \not\sqsubseteq V_j) \\ (V_i, \max(R_i, R_j), \max(W_i, W_j)) \\ \qquad \text{, } \mathit{if}(V_i = V_j) \wedge (V_i \sqsubseteq V_j) \\ \{(V_i, R_i, W_i) \cup (V_j, R_j, W_j)\} \\ \qquad \text{, } \mathit{if}(V_i \neq V_j) \end{cases} \tag{3.5}$$

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. Elaborately, for the same variable, traversing different paths, we consider the path with maximum of the read and write counts accrued among all paths. However, if the partial order is not maintained, that means, the read and write counts attains maximum value in different paths, it is marked as inconclusible. 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. In later steps, we will describe how to allocate these variables to SRAM and STT-RAM. 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 a worklist based iterative algorithm using the dataflow equations described as follows -

Definition 3.3.4 (Dataflow Equations). *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 -*

$$RWA_{entry}(l) = \emptyset \text{ if } l \in init(S_\star) \tag{3.6}$$

$$RWA_{entry}(l) = \sqcup(RWA_{exit}(l') \cup B_i, \text{ if } (l, l') \in Flow(S_\star) \tag{3.7}$$

$$RWA_{exit}(l) = (RWA_{entry}(l) \cup F_i(RWA_{entry}(l)) \tag{3.8}$$

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. For our implementation, we have utilized the natural loop detection routine provided in the GCC compiler. 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 point 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.

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 - read intensive and write intensive variables using these estimated counts. The memory access behaviour of applications differs to a large extent. Applications that are computation intensive have different memory access 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:

Class 0 - Low read and low write **Class 1** - Low read and high write

Class 2 - High read and low write **Class 3** - High read and high write

Algorithm 3.1 Address Generation for Global and Stack Data (Partial)

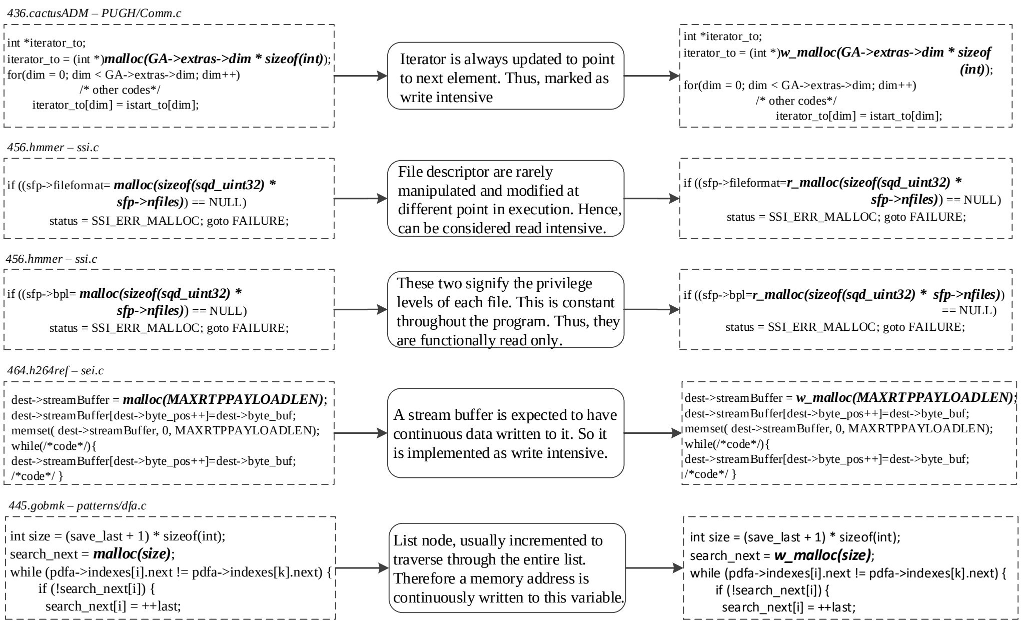
```
Require: source code of the program
Ensure: virtual addresses for statically allocated memory objects
 1: CFG ← Control Flow Graph of the program
 2: procedure ADD_GEN(CFG)
 3:     Initialize var_array ← ∅ /* var_array is a 2-d array with variables and their assigned classes */
 4:     Initialize analysis_outcome ← ∅ /* analysis_outcome contains all the variables with read and
    write counts */
 5:     for all function(F) in CFG do
 6:         analysis_outcome ← PASS_RW_ANALYSIS(F)
 7:     end for
 8:     var_array ← CLUSTERIZE(analysis_outcome)
 9:     current_global ← .text_section_end
10:     current_stack ← .stack_base
11:     for all variable(V) in var_array do
12:         if V is global data and CLASS(V) !=3 then
13:             allocate V to current_global
14:             realign current_global
15:             remove V from var_array
16:         else if V is stack data and CLASS(V) == 0 or 2 then
17:             allocate V to current_stack
18:             realign current_stack
19:             remove V from var_array
20:         end if
21:     end for
22:     .global_write ← current_global
23:     .stack_write ← current_stack
24:     for remaining variables(V) in var_array do
25:         if V is global data then
26:             current ← current_global
27:         else if V is stack data then
28:             current ← current_stack
29:         end if
30:         allocate V to current
31:         realign current
32:         remove V from var_array
33:     end for
34: end procedure
```

Algorithm 3.1 shows the address generation scheme for statically allocated data. At the outset, a 2-D array is initialized which will hold the variables and their assigned classes (line 3). This data structure will be the output of the algorithm. Next, using the control flow graph, the dataflow analysis is applied across functions for each basic block (lines 4-6). The outcome of the analysis is stored in another 2-D array comprising variable names and their corresponding read and write counts. This particular data structure is clusterized by the K-

Means algorithm (line 8) into 4 clusters. Afterwards, using the existing flags for each variable in the compilation framework, they are characterized as global data or stack data (line 12, 16). All global data belonging to classes 0, 1 and 2 are assigned addresses and placed in an elf section (lines 13-15). Class 3 global data is then placed above starting from a new elf section marker *.global_write* (lines 22, 25-26). Next, for stack data, classes 0 and 2 are placed together (lines 17-19) and classes 1 and 3 are placed together from a new elf section marker *.stack_write* (lines 23, 27-18). Global variables usually show high affinity towards either read or write operations. Number of global variables showing both high read and write counts are few. Therefore, for global data, we place Class 0, 2 and 3 variables contiguously and then Class 1 variables. For stack data, we place Class 0 and 2 variables contiguously and then Class 1 and 3 variables as there are a large number of variables showing high read and write affinity. The virtual addresses separating the sections are embedded in the final executable which are later recognized by the OS. This yields the read and write intensive virtual memory segments shown in Figure 3-1(b) earlier.

3.3.2 Dynamically Allocated Data

Dynamically allocated memory objects occupy a large region in the virtual address space of many processes, and are managed at runtime. Precise analysis of dynamically allocated memory at compile time is computationally hard [64]. Though, heap memory management is well studied for efficient garbage collection and detecting memory leaks [65, 66], analyzing 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


436.cactusADM – PUGH/Comm.c

```
int *iterator_to;
iterator_to = (int *)malloc(GA->extras->dim * sizeof(int));
for(dim = 0; dim < GA->extras->dim; dim++)
                /* other codes*/
        iterator_to[dim] = istart_to[dim];
```

Iterator is always updated to point to next element. Thus, marked as write intensive

```
int *iterator_to;
iterator_to = (int *)w_malloc(GA->extras->dim * sizeof
                                                  (int));
for(dim = 0; dim < GA->extras->dim; dim++)
                /* other codes*/
                    iterator_to[dim] = istart_to[dim];
```

456.hmmer – ssi.c

```
if ((sfp->fileformat= malloc(sizeof(sqd_uint32) *
                          sfp->nfiles)) == NULL)
        status = SSI_ERR_MALLOC; goto FAILURE;
```

File descriptor are rarely manipulated and modified at different point in execution. Hence, can be considered read intensive.

```
if ((sfp->fileformat=r_malloc(sizeof(sqd_uint32) *
                                sfp->nfiles)) == NULL)
        status = SSI_ERR_MALLOC; goto FAILURE;
```

456.hmmer – ssi.c

```
if ((sfp->bpl= malloc(sizeof(sqd_uint32) *
                    sfp->nfiles)) == NULL)
        status = SSI_ERR_MALLOC; goto FAILURE;
```

These two signify the privilege levels of each file. This is constant throughout the program. Thus, they are functionally read only.

```
if ((sfp->bpl=r_malloc(sizeof(sqd_uint32) *  sfp->nfiles))
                                                  == NULL)
        status = SSI_ERR_MALLOC; goto FAILURE;
```

464.h264ref – sei.c

```
dest->streamBuffer = malloc(MAXRTPPAYLOADLEN);
dest->streamBuffer[dest->byte_pos++]=dest->byte_buf;
memset( dest->streamBuffer, 0, MAXRTPPAYLOADLEN);
while(/*code*/){
dest->streamBuffer[dest->byte_pos++]=dest->byte_buf;
/*code*/ }
```

A stream buffer is expected to have continuous data written to it. So it is implemented as write intensive.

```
dest->streamBuffer = w_malloc(MAXRTPPAYLOADLEN);
dest->streamBuffer[dest->byte_pos++]=dest->byte_buf;
memset( dest->streamBuffer, 0, MAXRTPPAYLOADLEN);
while(/*code*/){
dest->streamBuffer[dest->byte_pos++]=dest->byte_buf;
/*code*/ }
```

445.gobmk – patterns/dfa.c

```
int size = (save_last + 1) * sizeof(int);
search_next = malloc(size);
while (pdfa->indexes[i].next != pdfa->indexes[k].next) {
      if (!search_next[i]) {
        search_next[i] = ++last;
```

List node, usually incremented to traverse through the entire list. Therefore a memory address is continuously written to this variable.

```
int size = (save_last + 1) * sizeof(int);
search_next = w_malloc(size);
while (pdfa->indexes[i].next != pdfa->indexes[k].next) {
      if (!search_next[i]) {
        search_next[i] = ++last;
```

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

that is hard to analyse.

Coburn et al. explores the possibilities and threats of heap memory management for persistent memory systems such as NVMs [67]. However, in their work, the read and write properties of the heap region are 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 work, we have done the former and annotated the source codes of our benchmarks with the new malloc function calls, as shown in Figure 3-3.

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 sizable 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 3-3 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 runs out of memory space to allocate, mainly due to a boundary limit, we allow the use of the other.

Algorithm 3.2 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
```

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 3.5) 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 3.2 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 STT-RAM.

3.4 Putting It All Together

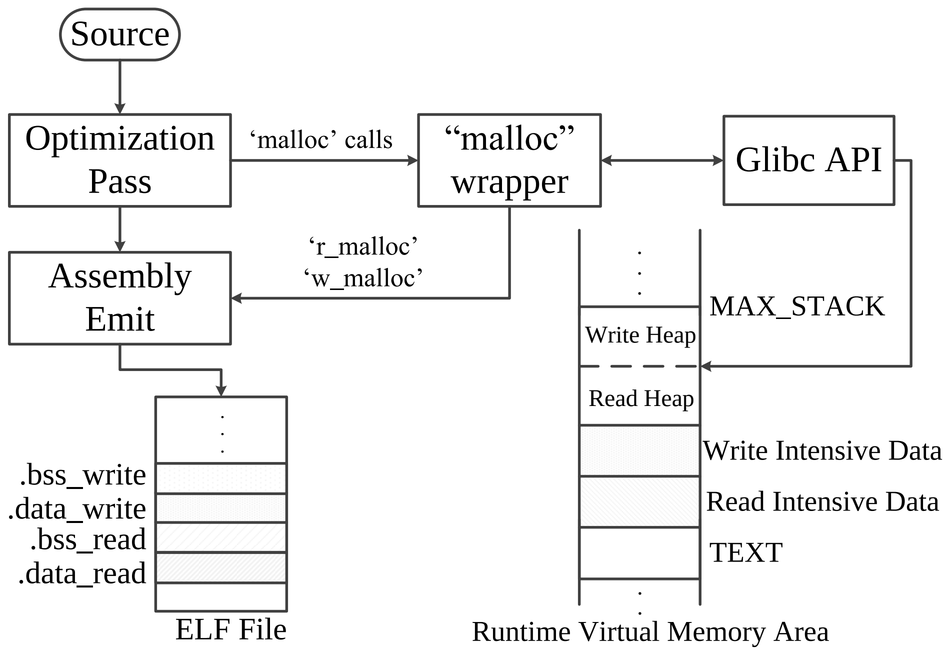


Figure 3-4: Overall framework of EnVM.

The framework to create EnVM is illustrated in Figure 3-4. During compilation, a program is analyzed 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.

3.5 Architectural Support

In this section, we would describe the architecture support required for our virtual memory design. For evaluation, we assume a popular hybrid cache model [61]. In addition we would discuss few aspects of virtual to physical address mapping, and other hardware implications of EnVM.

3.5.1 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 coupled with the address translation hardware unit. For the x86 architecture, the existing set of segment registers can be used for this purpose. During process creation and context switches, the operating system is responsible for loading the 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 3-5) 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.

3.5.2 Cache Properties

The delay associated with the boundary registers and address checking depends on the cache probe logic. We consider two kinds of caches here to analyze the delay - PIPT (physically tagged, physically indexed) and VIPT (virtually tagged,

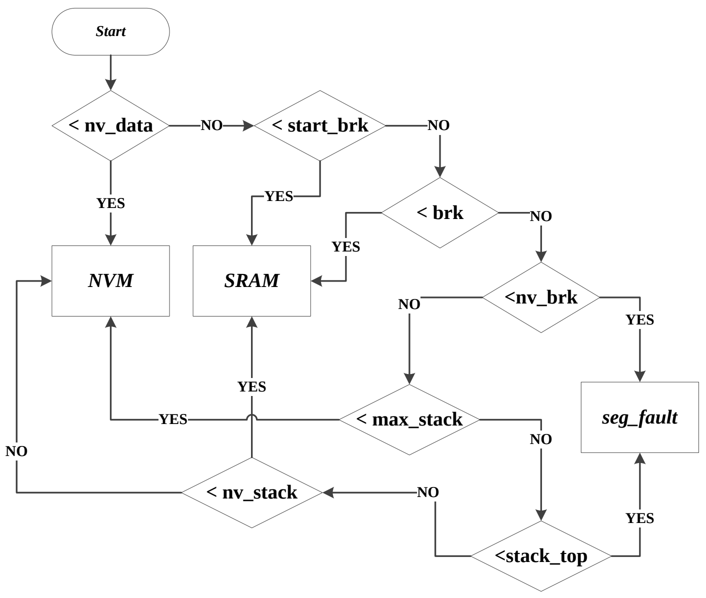


Figure 3-5: Cache Selection Logic.

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 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.

3.6 Evaluation

3.6.1 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 [62]. The results are based on the 'ref' input on all the benchmarks. Our back-end OS is Linux (kernel version 3.2.51). We implement the hybrid caches in MARSSx86 [68] cycle-accurate full system simulator. The complete configuration is given in Table 3.1. NVSim [69] was used to generate the latency and energy parameters for STT-RAM assuming a 32nm process technology. All the hybrid cache configurations roughly occupy the same silicon area as their pure SRAM counterpart [69]. We further assumed that the STT-RAM partition has error-correcting code (ECC) to mitigate stochastic bit-flip error [33] as the retention time for STT-RAM cells are myriad [4, 30].

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

Table 3.1: Simulation Configuration

3.6.2 Results

For evaluation, we implemented two hybrid cache designs at L1 [19] and L2 [17] referred to as *SW1* and *SW2*, respectively. We compare our method with another hardware based hybrid memory management scheme [3] referred to as *HW*.

Write Reduction

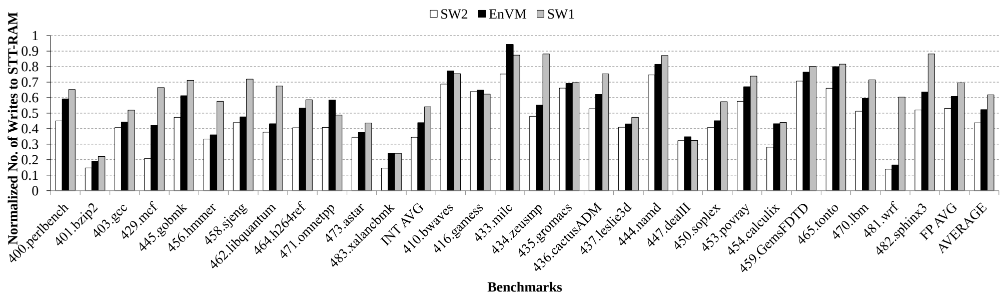


Figure 3-6: Total writes to STT-RAM in a hybrid cache design normalized to the total number of writes to a *pure STT-RAM* cache.

The primary objective of all schemes of hybrid memory management is to reduce the number of write operations to the STT-RAM caches by redirecting write intensive data to the SRAM counterpart. Figure 3-6 shows the number of write accesses to the STT-RAM partition, normalized to an architecture with pure STT-RAM and no assisting SRAM. The purpose of comparing with an architecture with pure STT-RAM is to illustrate the critical importance of the smaller SRAM partition.

After the pure STT-RAM scheme, the *HW* method incurs the maximum write accesses. In *HW*, data is primarily fetched into STT-RAM and migrated to SRAM only upon saturation of a 3-bit counter. Therefore, only selected data get migrated to SRAM and rest of the data receive write accesses being in the STT-RAM partition.

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. So, EnVM performs better than *SW1*.

Quantitatively, 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 similar write traffic to STT-RAM as compared to *SW2*, a profile-based technique.

Energy Consumption

Resistive memories exhibit 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. In addition, we compare the energy consumption of hybrid memory caches with a pure SRAM cache, to examine the merits of hybrid memory design, quantitatively.

The energy model is given by the sum of leakage energy, dynamic energy. We add the energy overheads due to various additional hardware units to the total energy consumption. In our scheme, we include the energy due to the usage of the boundary registers as mentioned earlier. For other schemes, the overhead energy includes migration of cache lines from one partition to the other. Formally, the energy model is as follows -

$$E_{total} = E_{leakage} + E_{dynamic} + E_{overhead} \tag{3.9}$$

$$E_{leakage} = P_{leakage} * t_{exec} \tag{3.10}$$

$$E_{leakage} = E_{write} * N_{writes} + E_{read} * N_{reads} \tag{3.11}$$

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 [70] to calculate the energy consumption by the boundary registers.

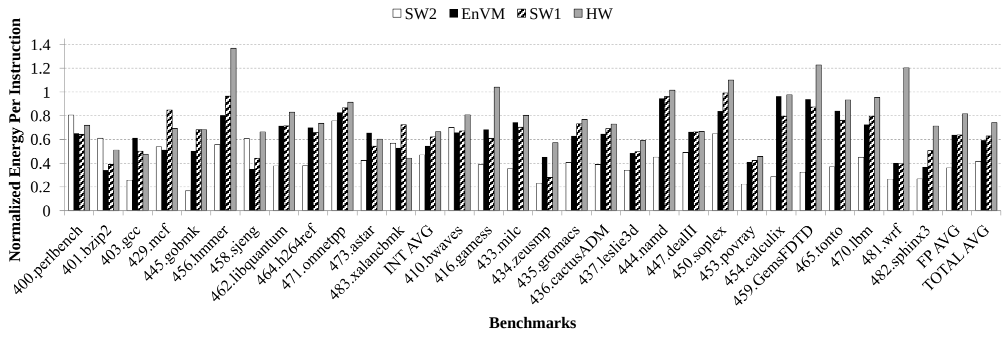


Figure 3-7: Energy per instruction normalized against *pure SRAM* cache.

Figure 3-7 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, respectively. For some C benchmarks, such as `400.perlbench`, `401.bzip2`, EnVM has 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 3-8 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 3-9 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

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.cactusADN**	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.libquantui	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.GemsFDTI**	0.0602	8E-05
473.astar	0.0483	0.0001	**465.tonto**	0.0401	6E-05
483.xalancbml	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 3-8: Energy (joules/instruction) consumed by the additional hardware units for HW and EnVM.

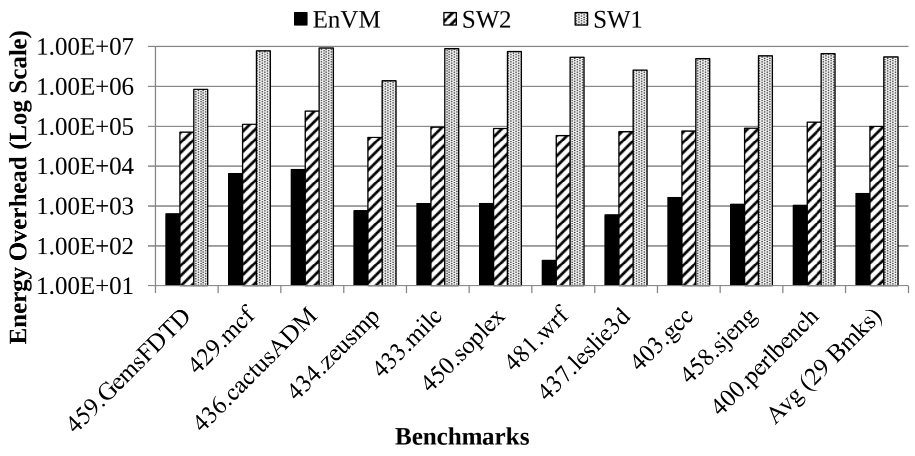


Figure 3-9: Total energy consumption by additional hardware components.

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. In our evaluation, we measured the energy overhead of the three techniques as shown in Figure 3-9.

Performance Impact

In Figure 3-10 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 (+1 cycle) required by the boundary address checking. We considered a VIPT cache, giving an upper bound on the additional delay. In PIPT cache design, the IPC is expected to be less affected by EnVM. The IPC is normalized to a baseline of 32K SRAM L1 cache. Quantitatively, EnVM achieves 26% better IPC than *HW* and comparable performance with *SW1* and *SW2*.

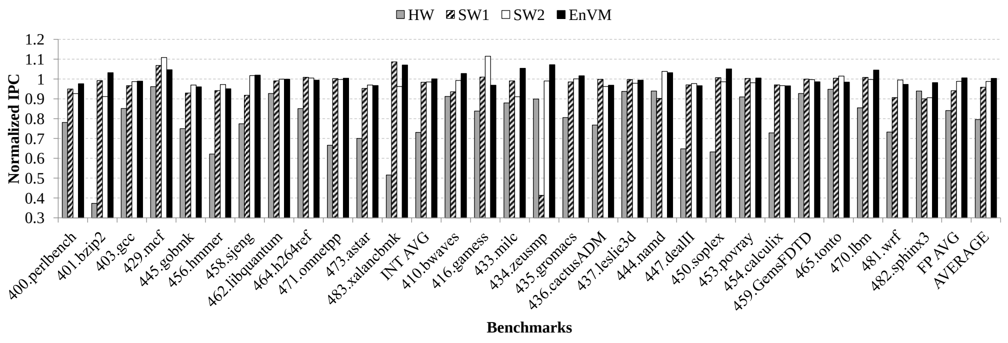


Figure 3-10: Instructions Per Cycle (IPC) normalized to pure SRAM based cache design.

As compared to pure SRAM architecture, hybrid memory managed by EnVM achieves similar performance. 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.

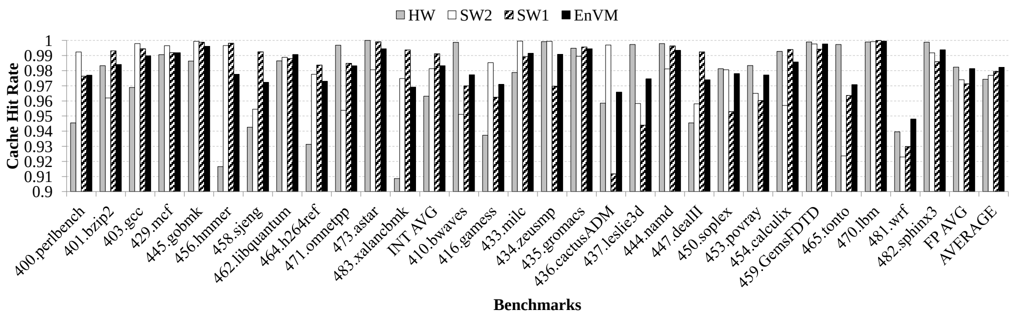
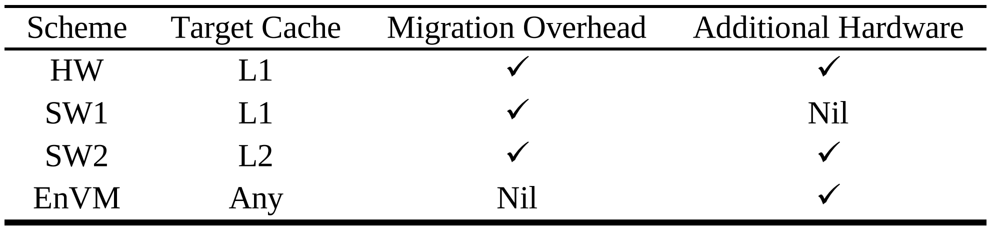


Figure 3-11: Cache hit rate for the hybrid L1 cache design.

To further quantify the impact of cache sizes on performance, we measured the cache hit rate (see Figure 3-11). 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.



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

Figure 3-12: Summary of state-of-the-art methods and EnVM.

Table 3-12 summarizes the features of 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* assume 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.

3.7 Chapter Summary

In this chapter, we have proposed EnVM, a virtual memory design optimized for STT-RAM 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 STT-RAM 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 STT-RAM 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.

Chapter 4

Operating System Assisted Resistive Hybrid Main Memory

In this chapter, we consider a hybrid main memory comprising PCM and DRAM. We would propose a fine-grain interaction scheme between last level cache (LLC) and the main memory which reduces writes to the PCM partition. In addition, we would propose a new operating system managed page reclamation algorithm that is aware of the underlying hybrid memory architecture. Such a two-pronged approach reduces writes to PCM efficiently and enhances the power-performance benefit.

4.1 Motivation

The focus of this chapter is management of hybrid main memory by leveraging on the disparity between the data granularities of last-level cache writebacks and main memory. Because of its small size, the DRAM partition must use the PCM partition as back up, demanding *migration* of pages between the two partitions.

Note, that this is a necessity independent of the actual management scheme of the hybrid main memory.

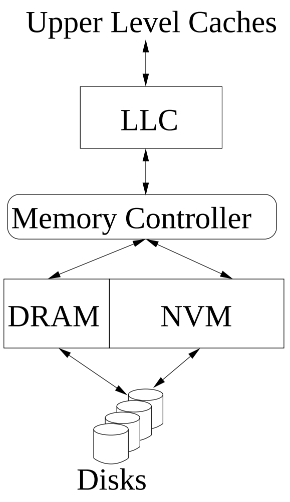


Figure 4-1: Different designs of hybrid main memory

Figure 4-1 is the assumed hybrid main memory design for our framework. Therefore, the challenge in managing such hybrid design is the data allocation for last level cache writebacks. In this chapter, we will deeply investigate the non-trivial relation between the last level cache writebacks and the data allocation between DRAM and PCM.

Disparity in Data Granularity

The last-level caches (LLC) and the main memory are different both in terms of structure as well as behaviour. LLC usually reads and writes data in units of cache lines. Data in LLC are uniquely identified by simple hashing of their physical addresses. Main memory is organized in *pages* that consist of many cache lines. For today's systems, a page typically contains 64 cache lines. While caches are completely hardware managed and transparent to software, main memory is managed by the operating system (OS) often with complex mapping schemes such as virtual paging. The OS maintains information for each page

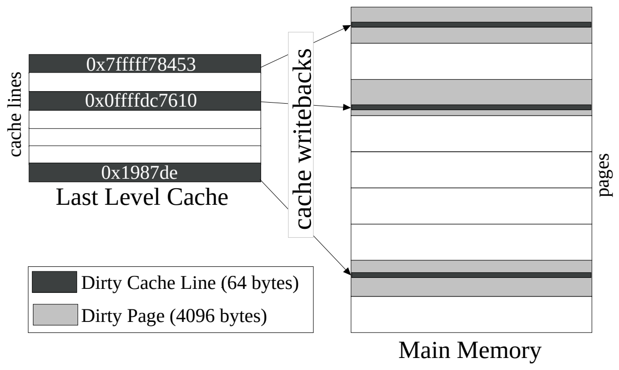


Figure 4-2: An example showing the extra amount of dirty data in main memory due to cache line size writebacks.

and its attributes in page tables. When a LLC writeback occurs, the data is written to its corresponding physical address within a page.

For example, as shown in Figure 4-2, let the cache line size be 64 bytes and page size be 4 kilobytes. This means a page can contain up to 64 cache line size worth of data. When a LLC writes back 3 cache lines located in 3 different pages, it marks all the three pages as dirty. Therefore, $64 * 3 = 192$ bytes of dirty data in LLC get translated to $4 * 3 * 1024 = 12288$ bytes of dirty data in the main memory. This indicates about 98.4% of data which are not modified, are marked as dirty too. In a hybrid memory scenario, due to the smaller size of the DRAM, pages are evicted frequently to PCM in order to maintain the write working set in the DRAM. As the operating system is unaware of the cache writebacks, the 98.4% of the data which is unmodified will also be written, generating a huge number of wasteful or redundant writes to the PCM partition.

However, we conducted an experiment to examine this phenomenon in reality. Figure 4-3 shows the amount of dirty cache lines per memory page of six memory intensive benchmarks from SPEC2006 and PARSEC over a period of 4 billion cycles. To obtain these plots, we computed the average number of dirty cache lines per page at intervals of 10 million cycles. We see that the average number of dirty lines per page never exceeded 10 for `GemsFDTD` and at the most 12 for the rest of the applications except `dedup` that reached an average of about 42. Figure 4-3 shows that for the most part, memory pages contain clean data. Therefore, copying entire pages during the migration process will result in many redundant writes to the PCM partition.

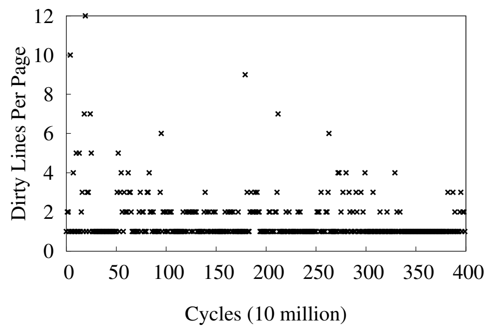
Scope of Improvements

The redundant writes, if eliminated, will improve the performance and power consumption of the hybrid memories in several ways. To examine the impact of this, we first need to elaborate on the critical issues of the hybrid memories.

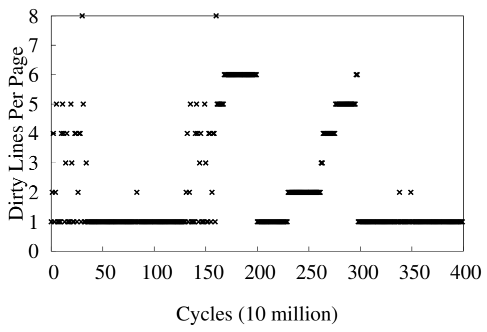
- **Unmonitored cache writebacks**. The problem originates when cache lines are written back to memory. We therefore need to monitor such writebacks using special hardware mechanisms so that dirty data in a page can be identified. The challenge is to devise an efficient way to maintain the fine-grain dirty data information, and to reduce of the overhead of the extra book-keeping that is required. Only with such information can a less wasteful migration scheme be implemented.

- **Page Allocation**. Ideally, all the cache writebacks must be directed to DRAM and cache fills must be serviced from the PCM partition. However, in practice, as a page is much bigger, it is possible to have write and read intensive cache lines mapped to the same page. Therefore, it would be productive to have some cache lines of a page reside in the DRAM partition, while having others of the same page reside in the PCM partition.

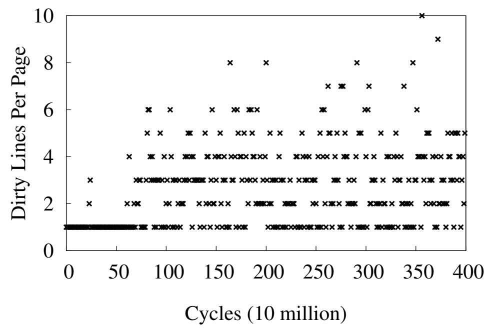
- **Migration traffic**. As the DRAM partition of a hybrid memory is smaller than PCM partition, it is necessary to evict DRAM pages from time to time. The evicted pages should be migrated to the PCM partition so as to reduce disk accesses for future accesses to them. Current reclamation schemes choose the least-recently-used page and write them to the PCM,



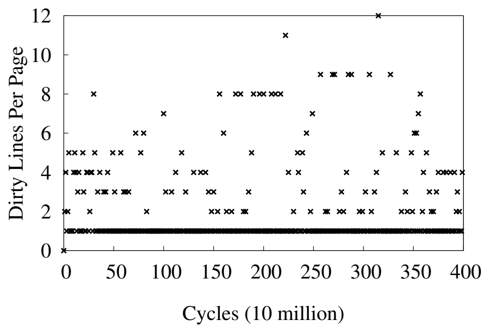
(a) milc



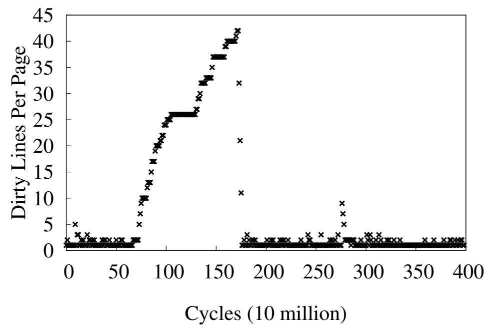
(b) mcf



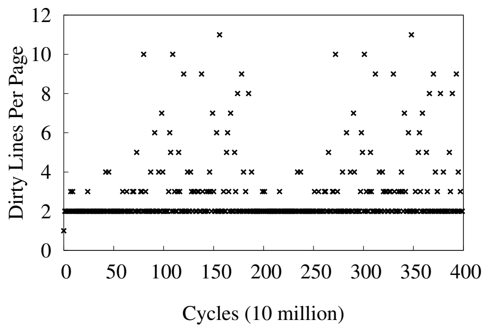
(c) gemsFDTD



(d) h264ref



(e) dedup



(f) facesim

Figure 4-3: Average number of dirty cache line per main memory page of six memory intensive applications

without considering the amount of dirty data within the page. A fine-grain page reclamation policy would choose pages that are least dirty to be migrated from DRAM to PCM. This would further reduce the number of write operations that the PCM partition actually receives.

4.2 Our Proposal

We propose a hardware-software framework consisting of Fine-Grain Writes (FGW) and Fine-Grain Page reclamation (FGP). The former is a hardware mechanism while the latter is a new operating system based page reclamation algorithm for PCM-based hybrid memories. Our framework solves the challenges mentioned in previous section. It not only reduces write accesses to the PCM partition, but also minimizes the data traffic between the two partitions.

FGW exploits the disparity between the data granularity of LLC and main memory and redirects dirty cache lines to the DRAM partition. Cache writebacks do not reach PCM partition and thus reduces the number of write accesses. FGP is a page reclamation policy that introduces a notion of *dirtiness* of DRAM pages and evicts only least dirty page from DRAM to maintain a free pool of pages in it. The evicted DRAM pages are stored in PCM and thus, PCM receives only the least dirty pages. Thereby, reducing writes to a large extent.

Conceptually, the smaller DRAM holds the write working set of the current set of processes in the system and PCM acts as a large storage, servicing all read requests. However, depending on the combination of DRAM size and write working set size, pages have to be evicted from DRAM to PCM occasionally to ensure free space in the former. Here, FGP plays a crucial part and ensures the write working set to remain in DRAM for as long as possible.

The key contributions in this work are as follows -

- We propose a *fine-grain write* (FGW) mechanism for hybrid memories which writes only dirty cache lines to main memory, thereby eliminating redundant writes.
- Further, to detect the amount of dirty data within a page we propose

an operating system assisted page table management. Our technique can measure the *dirtiness* of a page which is needed in the fine-grain page migration between DRAM and PCM.

- In order to further reduce the migration overhead between DRAM and PCM, we propose a novel operating system based *fine-grain page reclamation* algorithm (FGP) which makes use of the dirtiness information of each page to evict least dirty pages from DRAM.

4.3 Fine-Grain Writes

To redirect writes to DRAM and reads to PCM, we propose a new concept of *shadow pages*. A shadow page is a DRAM page that receives write operations on behalf of a corresponding page in the PCM partition. Every shadow page has a corresponding PCM page, but the reverse is not true since the PCM partition is much larger. When a write operation is performed on a PCM page that does not yet have a shadow page, a blank shadow page is allocated in DRAM and is associated with this PCM page. All LLC writebacks to this page goes to the shadow page. When the shadow page is evicted, the dirty data is merged into the corresponding PCM page. On the other hand, LLC fills are performed using both the shadow and PCM page.

4.3.1 Shadow Page Management

The mapping between the DRAM and PCM pages are maintained by the OS in a page table like structure, called the *shadow table*. Figure 4-4 shows how shadow table maintains the mapping of cache lines in the shadow pages and PCM pages. Each shadow table entry consists of the DRAM page frame number (DRAM PFN), together with a bitmap indicating the location of the corresponding cache lines in that page. We assume the page sizes of DRAM and PCM to be equal.

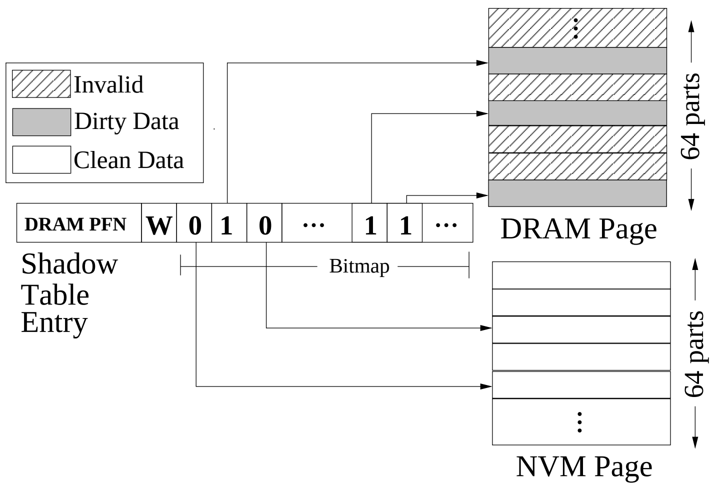


Figure 4-4: Shadow page and shadow table entry

So, data in PCM page A at an offset d would be situated in a shadow frame S at the same offset d.

Suppose each page contains 64 cache lines' worth of data. The bitmap for each page would then be 64 bits long. A '1' indicates that the data is present in the DRAM shadow page, and a '0' indicates that the data is present in PCM page. The shadow table is checked before any main memory access, and the bitmap decides which partition is to be probed. If it's a write access, it is redirected to the shadow page. For a read operation, if the data to be read was previously written and hence dirty, then the corresponding bit in the bitmap would already be set during a previous write and the read is serviced from the shadow page in the DRAM partition. However, if the data to be read is clean then there would be no preceding write access that cache line and the corresponding bit would be '0'. Hence, the request is serviced from the PCM page. Thus, the bitmap ensures that there are no stale reads and spurious writes occurring in the memory.

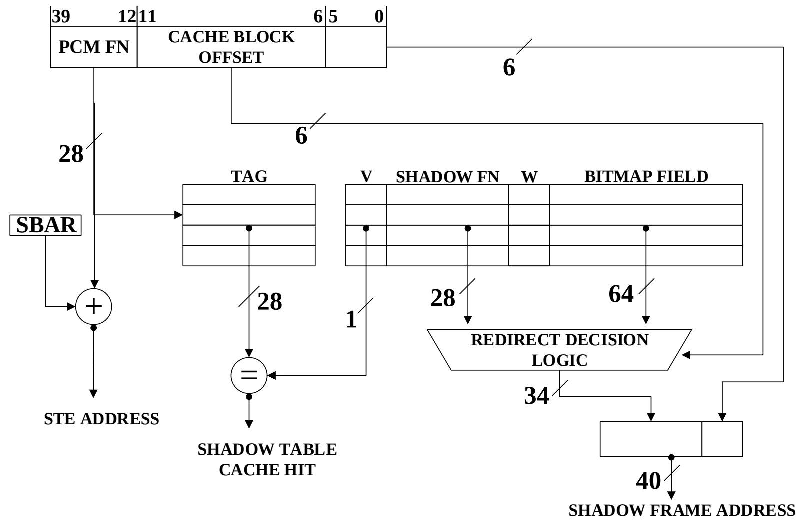


Figure 4-5: PCM to shadow page physical address translation.

4.3.2 Extended LLC

All memory accesses are performed by the LLC. Therefore, to redirect memory read and write accesses to the correct partition, the LLC must be aware of the fine-grain data management. In particular, it must use the shadow table entry to find the location of the data to be read or written. Figure 4-5 illustrates the modified architecture of the LLC. The *redirect decision logic* (RDL) is a new component that processes the bitmap. Bits `[11:6]` of the PCM address, namely *cache block offset* is used to address all cache lines of a PCM page uniquely. The RDL generates the shadow address by replacing bits `[39:12]` of the PCM address with the DRAM PFN provided by the shadow table entry. The bits `[11:0]` remains the same in PCM address and shadow address as the page sizes are assumed to be the same. The 28 most significant bits are used as tags to look up the shadow table. The SBAR register stores the base address of the shadow table and STE stores the physical address of the shadow table entry, both of which are located in DRAM.

4.3.3 Shadow Table Cache

The shadow table is indexed by the PCM page frame number and each shadow table entry consists of a physical address and a bitmap. Therefore, the size of the shadow table linearly increases with the total capacity of the PCM. The shadow table resides in the DRAM. For each memory access, the shadow table is checked before the actual data access. Such a mechanism increases the memory access delay by 2×. Using the same trick to speed up virtual address translation, namely the introduction of the TLB (translation look-aside buffer), we propose a TLB-like small cache, that we have named the *shadow table cache.* The shadow table cache contains the most recent shadow table entries, thereby speeding up LLC reads/writes from/to the memory. The shadow table cache in our framework contains 1024 entries. However, in our experiments, we will demonstrate the impact of the shadow table cache capacity on power and performance.

4.4 Fine-Grain Page Reclamation

The allocation of shadow pages is managed by the operating system. If a LLC writeback tries to write in a page that does not have a shadow page in the DRAM partition, (i.e., a miss of the shadow table) the operating system issues a *minor* page fault and disallows the write access to the PCM page. Following the minor page fault, it allocates a shadow page in the DRAM partition, and allows the write access to the shadow page instead. However, as the DRAM partition is smaller, eviction of the shadow pages from the DRAM partition is essential to maintain a steady pool of free pages. Evicted pages from DRAM are written back to the PCM partition. Since this introduces writes to the PCM partition, the page reclamation policy affects the amount of writes to the PCM partition. As the least recently *used* DRAM page is not necessarily the least written to DRAM page, if the eviction policy is conventional LRU (least recently used), there would

Algorithm 4.1 Write Aware Page Reclamation

```
Require:
 1: $shad\_alloc\_list$ : the circular list of allocated shadow pages,
 2: $shad\_free\_list$ : the list of free shadow pages,
 3: $N_{dirty}(s) \in \{1, \dots, 64\}$ : the number of dirty cache lines in $s \in shad\_alloc\_list$,
 4: $Range(s)$ : a function of $s \in shad\_alloc\_list$, defined as follows:
 5: $Range(s) = \begin{cases} Range\_0, \text{if } N_{dirty}(s) \in \{1, \dots 16\}, \\ Range\_1, \text{if } N_{dirty}(s) \in \{17, \dots 32\}, \\ Range\_2, \text{if } N_{dirty}(s) \in \{33, \dots 48\}, \\ Range\_3, \text{if } N_{dirty}(s) \in \{49, \dots 64\}, \end{cases}$
 6: $overlook(s)$ : number of iterations overlooked during reclamation for $s \in shad\_alloc\_list$,
 7: $F$ : Shadow page pointed by the clock hand in $shad\_alloc\_list$,
 8: $next(s)$ : Shadow page next to $s \in shad\_alloc\_list$,
 9: $threshold$ : Maximum number of iterations that can be overlooked, this sets this as 2 to match
    CLOCK algorithm.
10: Start
11:     procedure Allocate_Shadow_Page
12:         if $shad\_free\_list \notin \emptyset$ then
13:             return $s \in shad\_free\_list$
14:         else
15:             $s \leftarrow Reclaim\_Shadow\_Page$
16:             return $s$
17:         end if
18:     end procedure
19:     procedure Reclaim_Shadow_Page
20:         for $range = Range\_0$ to $Range\_3$ do
21:             $p \leftarrow F$
22:             if $Range(p) == range$ then
23:                 if $dirty\_bit(p) == 1$ then
24:                     $dirty\_bit(p) \leftarrow 0$
25:                     $overlook \leftarrow 0$
26:                 else
27:                     if $overlook(p) < threshold$ then
28:                         $overlook(p) \leftarrow overlook(p) + 1$
29:                     else
30:                         $F \leftarrow next(p)$
31:                         return $p$
32:                     end if
33:                 end if
34:             end if
35:             $p \leftarrow next(p)$
36:         end for
37:     end procedure
38: End
```

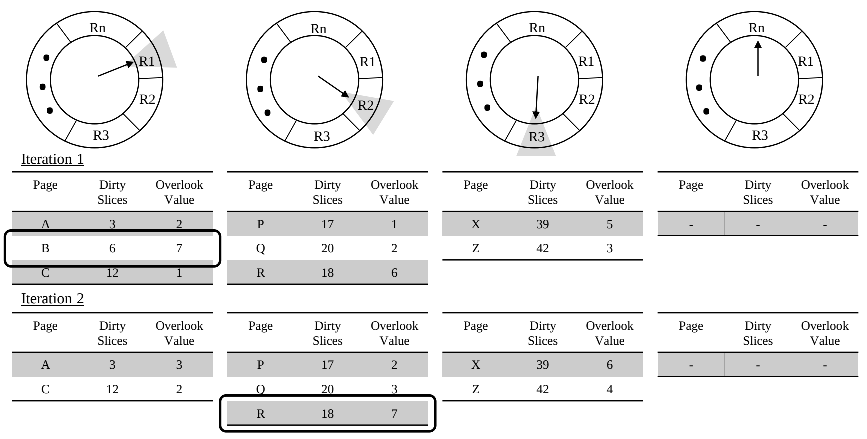
eventually be no control over the writes to the PCM partition. Evicting a least recently used but extensively modified page would translate to a large number of writes to the PCM partition.

We propose a new page reclamation algorithm that is aware of the *dirtiness* of a page. We define a *dirtiness* metric that is essentially the total number of dirty cache lines within a page. A page with more dirty lines is considered to be dirtier than a page containing lesser number of dirty lines. Selecting a shadow page with the least number of dirty lines as the eviction candidate is desirable not only because it reduces the number of writes to PCM, it also reduces the migration overhead due to the copying of data. At the same time, it is necessary to ensure that shadow pages that are currently part of the working set, do not get reclaimed. Frequent reclamation of write intensive shadow pages that are in the current working set increases the number of minor page faults and writes to its associated PCM page, resulting in thrashing.

Our page reclamation algorithm is based on the CLOCK algorithm [71] and tracks the recent writebacks to a shadow page. Algorithm 4.1 describes the pseudo-code of our proposed algorithm. The function $N\_dirty(s)$ (Line 3) stores the number of dirty lines in a shadow page s. The function *Range(s)* (Line 5) identifies the range of dirty lines in the shadow page s. Function *overlook(s)* (Line 6) counts the number of reclamation iterations for which the shadow page s was ignored. The maximum number of iterations for which a shadow page can be overlooked for reclamation is set as 2.

When the LLC requests a shadow page by generating a minor page fault, the procedure *Allocate_Shadow_Page* is called. The OS checks if there are free shadow pages available in the *shad_free_list* (Line 12). If a free shadow page is available, it is allocated to the request. Otherwise, the shadow page reclamation tries to reclaim a shadow page from the circular *shad_alloc_list*, by calling the function *Reclaim_Shadow_Page* (Line 19). To reclaim an allocated shadow page, *shad_alloc_list* is scanned, starting from the shadow pages containing dirty cache lines in the lowest range (*Range_0*) (Line 20). When a suitable candidate is found, the dirty bit is first checked. If it is set, then this candidate is not considered for reclamation for the current iteration.

The dirty bit of the candidate is reset and the overlook value for the shadow page is set to 0 (Lines $23 - 25$). If the dirty bit of the candidate has not been



Iteration 1

Page	Dirty Slices	Overlook Value
A	3	2
B	6	7
C	12	1

Page	Dirty Slices	Overlook Value
P	17	1
Q	20	2
R	18	6

Page	Dirty Slices	Overlook Value
X	39	5
Z	42	3

Page	Dirty Slices	Overlook Value
-	-	-

Iteration 2

Page	Dirty Slices	Overlook Value
A	3	3
C	12	2

Page	Dirty Slices	Overlook Value
P	17	2
Q	20	3
R	18	7

Page	Dirty Slices	Overlook Value
X	39	6
Z	42	4

Page	Dirty Slices	Overlook Value
-	-	-

Figure 4-6: Example of dirtiness aware page reclamation with an overlook value of 8.

set, then the number of overlooks the shadow page has enjoyed is checked. If it is equal 2, then this candidate is chosen as the victim. Otherwise, the overlook count for the candidate is incremented (Lines $27 - 31$). The scanning proceeds to the pages with the next range of dirty lines (*Range_1*), (*Range_2*) and so on.

Figure 4-6 illustrates an example of a typical iteration of the algorithm with an overlook value of 8, for ease of explanation. In the first iteration, page A has the least number of dirty slices, however, the overlook count is only 2. Therefore, the next page B is considered. As the overlook value for B is 7, B is chosen as the victim for this iteration and the overlook counts for rest of the pages are incremented by 1. In the next iteration, none of the pages in the first range have saturated the overlook counter. Thereby, the algorithm probes the next range of pages i.e. P, Q and R. In this group, even though both P and Q have lesser number of dirty cache lines, R has saturated its overlook counter. So, R is chosen as a victim in this iteration. The overlook counter ensures that least recently used pages are evicted while the order of the pages to be probed (no. of dirty slices and group of pages based on ranges) ensures that the least recently used and least written to page is evicted.

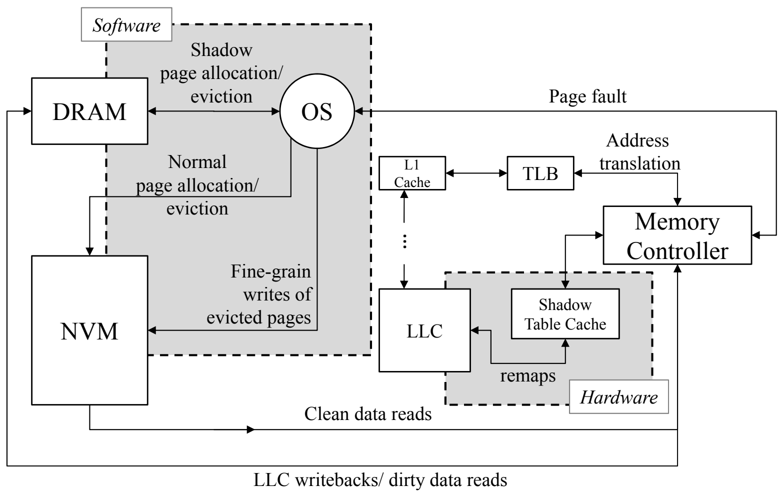

Software
Shadow page allocation/ eviction
DRAM
OS
Page fault
Address translation
Normal page allocation/ eviction
L1 Cache
TLB
Memory Controller
NVM
Fine-grain writes of evicted pages
LLC
Shadow Table Cache
remaps
Hardware
Clean data reads
LLC writebacks/ dirty data reads

Figure 4-7: Overview of our proposed framework

Putting It All Together

Figure 4-7 shows the overall structure and functionalities of our proposed framework. In our proposed hardware, every time the LLC controller performs a main memory access, the shadow table cache is probed. The shadow table cache is implemented like the translation-lookaside buffer. Just like a TLB miss, the operating systems is invoked on a shadow table cache miss. It will service the miss by a cache fill from the shadow table which resides in the kernel space in the DRAM partition.

In the configurations we have used in our experiments, the shadow tables are typically 2.4 Mbytes. On processing a shadow table cache miss, if the OS fails to locate an entry in the shadow table, then it will examine if the virtual page is present in main memory. If it is not, then a full-fledged page fault will be triggered. If the page is present, it results in a minor page fault. This means that the virtual page is already mapped into the PCM partition, but a shadow page in the DRAM partition has yet to be allocated. Should the access be a read, then it is straightforward to satisfy the read request from the PCM partition.

If the access is a write, then the OS will allocate a DRAM shadow page, and perform the necessary book-keeping. Such minor page faults do not have the similar heavy performance penalty as a conventional page fault.

In our experiments, we found that the total numbers of minor page faults are not large because once a shadow page is allocated, further writes to that page will not result in a minor page fault. In such cases, only the bitmap in the corresponding shadow table cache entry is updated by the FGW mechanism during every write. The two rectangles in Figure 4-7 represent the software (FGP) and the supporting hardware (FGW) components. While, FGW captures the finer granularity of reads and writes, FGP enables the fine grain data allocation/eviction for the DRAM and PCM. We believe that such a software-hardware is necessary for achieving our overall goals.

4.5 Evaluation Methodology

Experimental Setup

To evaluate our framework, we used a modified version of MARSSx86 [68], a full system simulator for x86 architectures to work together with DRAMSim2 [72] to model our framework. The processor simulated is a four-core, 3 GHz out-of-order processor with a three level cache hierarchy. In addition, we have implemented our page reclamation algorithm in a version 2.6 Linux kernel that runs on the simulator together with the workloads. The detailed configuration is given in Table 4.1. In order to eliminate effects of cold memory misses, we warmed up the memory hierarchy for the first 500 million simulated cycles before taking readings in the simulation of the next 4 billion *instructions* for each workload.

As workload, we used all the benchmarks of SPEC2006 [62] and PARSEC [73] benchmark suite. Table 4.2 shows the benchmarks and the size of their working sets. We grouped them into 11 workloads containing a mix of the benchmarks

Configuration	
Processor	4 Out-of-order cores, Freq - 3 GHz
L1 Cache	I-Cache and D-Cache, 32KB 4-cycle access latency, 64 byte line size
L2 Cache	2MB 4-way, 64 byte line size 12-cycle access latency
L3 Cache	Shared, 8MB, 64 byte line size 28-cycle access latency
Shadow Table Cache	1024 Entries, Area - 0.36 mm^2 9-cycle access latency Energy - 0.06 nJ/access
Main Memory	DRAM : varied size + PCM : 8GB DRAM Access Latency - 50 ns Energy - 0.1 nJ/bit PCM Read Latency - 50 ns, Energy - 0.2 nJ/bit PCM Write Latency - 100 ns Energy - 1 nJ/bit

Table 4.1: Simulation Configuration

as given in Table 4.3.

Among the 11 workloads, 4 are highly memory intensive (`hm1`, `hm2`, `hm3`, `hm4`), 4 have medium memory intensive working sets (`mm1`, `mm2`, `mm3`, `mm4`) and 3 are light in terms of memory usage (`lm1`, `lm2`, `lm3`). While the highly memory intensive workloads will reveal the savings in power consumption, the light memory workloads helps us to gain insight in the overheads involved.

For comparison, we used two state-of-the-art works on hybrid memory management. One is a purely software based management [74] and the other is a well established hardware solution for hybrid memories [7].

Comparison 1 - `clock-dwf`

Lee et al. [74] advocates an operating system driven page reclamation framework for DRAM-PCM hybrid memories.

Benchmark	**Working Set Size (MB)**	**Benchmark**	**Working Set Size (MB)**
swaptions	0.5	soplex	201
blackscholes	2	wrf	163.5
bodytrack	8	leslie3d	75.2
streamcluster	16	gcc	70
vips	16	sjeng	57
x264	16	perlbench	51
ferret	64	libquantum	33
fluidanimate	64	xalancbmk	27.8
freqmine	128	astar	26
canneal	256	bzip2	24.4
dedup	256	omnetpp	24
facesim	256	dealII	14.7
GemsFDTD	800	sphinx3	10.6
mcf	680	hmmer	8.2
bwaves	474	tonto	6.2
lbm	402	h264ref	2.9
zeusmp	270	povray	0.4
milc	230		

Table 4.2: SPEC2006 and PARSEC benchmarks and their working set sizes

Comparison 2 - `dram-cache`

Qureshi et al. [7] proposed a hardware assisted memory management scheme where DRAM is used as a cache to the bigger PCM partition.

Energy Models

One of the key contributions of our proposal is the reduction in energy consumed by the memories. We used CACTI [70] to model our proposed hardware component, namely the shadow table cache, and extracted the power consumption of DRAM from the DRAMSim simulator. For the PCM partition, we used widely accepted energy parameters [7, 74]. As the scope of this work is primarily data management for hybrid memories, we mainly compare the energies of DRAM

Workload	**Benchmarks**
hm1	gemsFDTD, milc, sjeng, gcc, astar
hm2	mcf, soplex, zeusmp, perlbench, xalancbmk
hm3	lbm, bwaves, wrf, leslie3d, libquantum
hm4	dedup, canneal, facesim, freqmine
mm1	gcc, bzip2, dealII, wrf
mm2	leslie3d, omnetpp, sphinx3, soplex
mm3	ferret, streamcluster, x264, freqmine
mm4	freqmine, fluidanimate, vips, ferret
lm1	sphinx3, bzip2, hmmer, h264ref
lm2	dealII, tonto, omnetpp, povray
lm3	streamcluster, swaptions, blackscholes, bodytrack

Table 4.3: Workloads

and PCM with that for `clock-dwf` and `dram-cache`.

Performance Impact

As we used a full system simulator, the delays associated with all the components are implemented as a part of the system. CACTI [70] was used to generate all the delay numbers associated with our additional hardware components. We also modelled and accounted for the delays due to the minor page faults and the page reclamation algorithm in the operating system.

4.6 Experimental Results

4.6.1 Write Reduction to PCM

Tables 4.4, 4.5 and 4.6 shows the total number of write accesses to the PCM partition for each of the methods. Our solution disallows any last level cache writeback to the PCM and thus, we achieve a remarkable reduction in writes.

Work-loads	**DRAM Reads** $\times 10^6$	%	**DRAM Writes** $\times 10^6$	%	**PCM Reads** $\times 10^6$	%	**PCM Writes** $\times 10^6$	%	**Total** $\times 10^6$
hm1	63.64	0.13	17.54	37.08	20.99	44.38	8.70	18.40	47.30
hm2	32.09	0.27	3.33	28.18	7.13	60.25	1.34	11.30	11.83
hm3	26.73	0.04	26.51	37.66	28.22	40.08	15.65	22.23	70.40
hm4	31.92	0.10	9.03	29.60	17.23	56.45	4.23	13.85	30.52
mm1	13.11	0.05	8.59	35.99	10.44	43.76	4.82	20.20	23.86
mm2	21.89	0.07	11.09	37.15	13.54	45.35	5.20	17.42	29.86
mm3	3.84	0.02	5.41	25.64	13.05	61.81	2.65	12.53	21.11
mm4	62.38	0.27	7.17	31.41	11.58	50.76	4.01	17.56	22.82
lm1	0.42	0.03	0.56	34.54	0.79	48.82	0.27	16.61	1.62
lm2	0.84	0.07	0.35	30.89	0.62	54.90	0.16	14.13	1.13
lm3	1.70	0.03	0.83	15.08	4.27	77.63	0.39	7.25	5.50
Avg	23.50	0.10	8.22	34.00	11.62	48.08	4.31	17.83	24.18

Table 4.4: Detailed memory access counts for `clock-dwf`

Work-loads	**DRAM Reads** $\times 10^6$	%	**DRAM Writes** $\times 10^6$	%	**PCM Reads** $\times 10^6$	%	**PCM Writes** $\times 10^6$	%	**Total** $\times 10^6$
hm1	57.49	0.12	17.63	37.11	21.15	44.53	8.66	18.23	47.50
hm2	50.74	0.21	8.08	34.15	10.91	46.13	4.61	19.51	23.65
hm3	18.98	0.04	19.94	37.79	21.71	41.13	11.10	21.04	52.77
hm4	28.49	0.10	8.38	29.31	16.14	56.47	4.04	14.12	28.58
mm1	34.79	0.34	3.68	35.66	4.96	48.00	1.65	16.00	10.33
mm2	8.83	0.06	5.43	35.62	6.93	45.45	2.88	18.87	15.25
mm3	5.84	0.02	5.74	21.77	17.96	68.14	2.66	10.07	26.37
mm4	37.27	0.21	5.55	31.07	9.18	51.40	3.09	17.32	17.87
lm1	0.77	0.05	0.56	33.98	0.84	51.52	0.24	14.45	1.64
lm2	0.29	0.02	0.37	30.40	0.67	55.62	0.17	13.95	1.21
lm3	4.77	0.16	0.34	11.69	2.43	83.00	0.15	5.15	2.93
Avg	22.57	0.11	6.88	33.19	10.26	49.50	3.57	17.21	20.73

Table 4.5: Detailed memory access counts for `dram-cache`

Work-loads	**DRAM Reads** $\times 10^6$	%	**DRAM Writes** $\times 10^6$	%	**PCM Reads** $\times 10^6$	%	**PCM Writes** $\times 10^0$	%	**Total** $\times 10^6$
hm1	3.18	8.23	17.24	0.45	18.20	0.47	64.00	0.17	38.61
hm2	5.13	24.42	8.48	0.40	7.41	0.35	64.00	0.30	21.02
hm3	3.47	8.19	20.29	0.48	18.61	0.44	128.00	0.30	42.37
hm4	2.59	10.28	8.65	0.34	13.95	0.55	128.00	0.51	25.19
mm1	1.73	15.95	4.37	0.40	4.74	0.44	64.00	0.59	10.84
mm2	3.15	25.16	5.46	0.44	3.91	0.31	64.00	0.51	12.52
mm3	4.71	21.25	4.70	0.21	12.76	0.58	448.00	2.02	22.17
mm4	1.49	9.94	5.64	0.38	7.85	0.52	64.00	0.43	14.98
lm1	0.11	7.82	0.56	0.41	0.71	0.51	128.00	9.32	1.37
lm2	0.02	1.37	0.35	0.25	1.01	0.73	64.00	4.63	1.38
lm3	2.13	42.94	0.65	0.13	2.18	0.44	704.00	14.17	4.97
Avg	2.52	14.21	6.94	0.39	8.26	0.47	174.50	0.98	17.73

Table 4.6: Detailed memory access counts for our framework

Compared to both `clock-dwf` and `dram-cache`, our framework reduces writes by four orders of magnitude. On an average, `clock-dwf` incurs 4.3 million and `dram-cache` incurs 3.6 million writes to PCM, where as our framework incurs 174.5 writes to PCM in a span of simulation of 4 billion instructions. Due to limitation in simulating disk accesses, for all the three methods, the PCM writes excludes the PCM fills from the disk.

4.6.2 Memory Utilization

Another interesting angle to evaluate the three techniques is to see how well they utilize the memory resources available. Tables 4.4, 4.5 and 4.6 show a detailed insight on the total number of reads and writes accrued by DRAM and PCM. Ideally, most of the reads should be serviced by the PCM partition while most of the writes should be serviced by the DRAM partition. In addition, in a quest to reduce writes to PCM, the larger PCM partition should not be under-utilized for that would be a waste. The tables 4.4, 4.5 and 4.6 show that under our framework there is still a substantial fraction of total reads going to the PCM partition with writes to PCM being nearly eliminated.

4.6.3 Energy Consumption

Figure 4-8 shows the total energy consumption by all the techniques. The energy model we used is as follows -

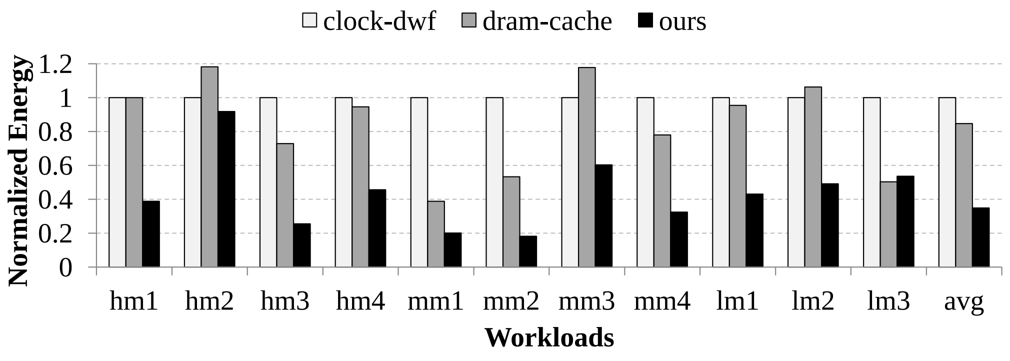
$$E_{\text{Total}} = E_{\text{Dynamic}} + E_{\text{Refresh}} + E_{\text{Overhead}} \tag{4.1}$$

where, E_{Dynamic} refers to the dynamic energy due to read and write accesses to the main memory. E_{Refresh} refers to the amount of energy spent in refreshing the DRAM.

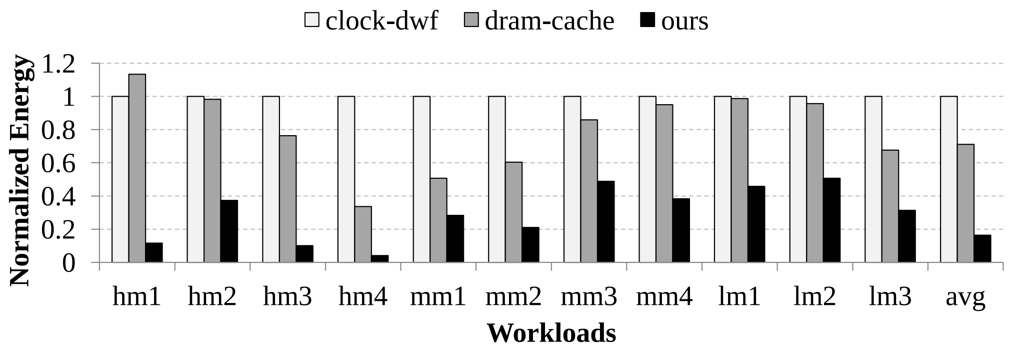
In addition, E_{Overhead} consists of the energy consumed by additional hardware or extra components for each of the techniques. For our framework, the energy consumption of the shadow table cache is added into E_{Overhead}. The energy for the shadow table cache accesses was obtained from CACTI and is given in Table 4.1. For `clock-dwf` and `dram-cache` the overheads are in terms of book-keeping for scanning the dram pages and additional migration to and fro from DRAM and PCM, respectively. Such overhead cases are covered within the E_{Overhead}.

Figure 4-8(a) shows the total energy consumption in a system consisting of an 8 GB PCM partition and a 1 GB DRAM partition. Our proposal reduces energy by 58.8% as compared to DRAM-cache and 65% as compared to `clock-dwf`. The DRAM partition consumes the most energy together with the write operations to the PCM partition. A smaller DRAM therefore should intuitively reduce the overall energy consumption.

However, a smaller DRAM partition also translates to increased write traffic to PCM. Therefore, to quantitatively verify this phenomenon, we tested a system with 8GB of PCM supported by a smaller 512MB DRAM shown in Figure 4-8(b). The energy consumption of `dram-cache` and `clock-dwf` increases due to the increased write traffic to the PCM partition. However, even in this case, our



(a) 1GB DRAM + 8GB PCM

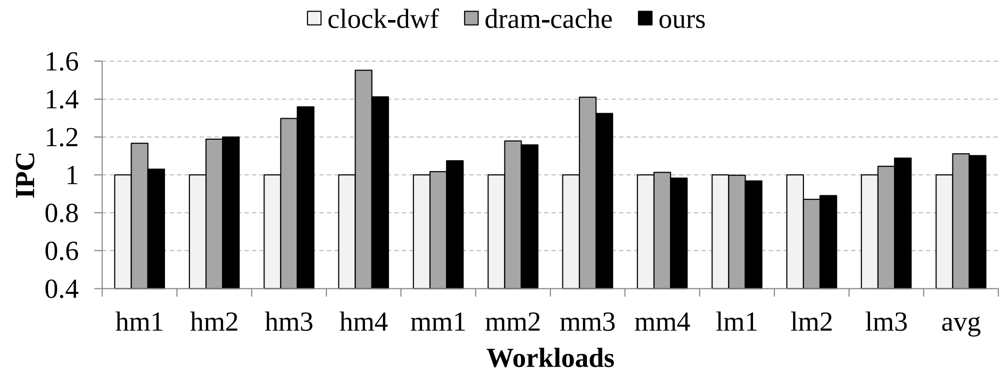


(b) 512MB DRAM + 8GB PCM

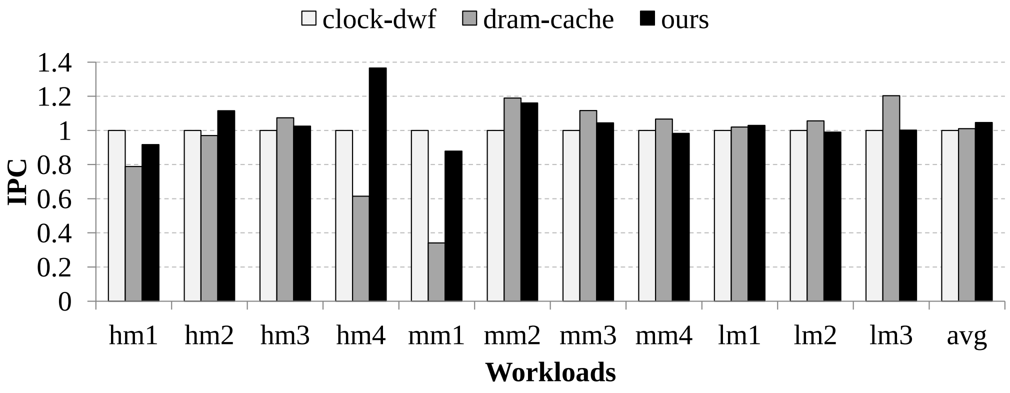
Figure 4-8: Dynamic energy of hybrid memory (DRAM+PCM) for two sizes of DRAM, normalized to energy consumption of `clock-dwf`.

framework shows energy reduction of 76.9% as compared to `dram-cache` and 83.6% as compared to `clock-dwf`.

This is due to the write aware page reclamation policy (FGW component) which prevents heavily written pages from being evicted and thus generates huge write traffics to PCM. Thus we achieve a higher energy gain as other two methods do not consider the amount of dirty data within a page while eviction. Especially, for `dram-cache`, the replacement policy is LRU and thus a smaller DRAM aggressively evicts pages that still belong to the write working set of the application.



(a) 1GB DRAM + 8GB PCM



(b) 512MB DRAM + 8GB PCM

Figure 4-9: Throughput in terms of instructions per cycle (IPC) for two sizes of DRAM, normalized to the IPC of `clock-dwf`.

4.6.4 Performance

We quantified performance in terms of the instructions per cycle (IPC). We compared the IPC with `dram-cache` and `clock-dwf` in Figure 4-9. Our framework has an IPC penalty of 0.7% on average as compared to `dram-cache`. However, when compared to `clock-dwf`, our framework shows an IPC improvement of 10%. For workloads, like `hm2`, that have large write working sets, `dram-cache` performs better as it uses DRAM to service LLC writebacks.

However, it also fails to control the writes to PCM, thereby increasing energy consumption as we have seen. Similarly, when the DRAM partition is small, the performance of `dram-cache` degrades drastically. As DRAM size decreases, more pages have to be migrated to the PCM partition due to more frequent evictions. This impacts the main memory access latencies, hence degrading performance. The same workload `hm2` generates larger writes to PCM and thus degrading IPC.

In our framework, if the write working set of an application is small enough to be accommodated by the DRAM partition, such as `mm1` and `lm3`, the overall performance is better than both `dram-cache` and `clock-dwf`. Under such a scenario, the application benefits from the fast DRAM writes as well as the larger PCM capacity.

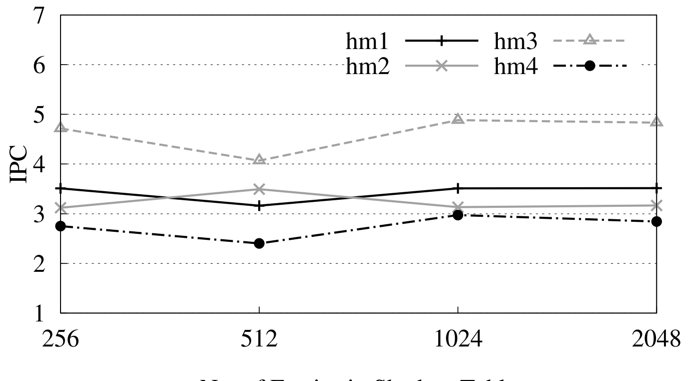
4.6.5 Shadow Table Cache

Our framework uses an additional shadow table cache to assist in the fine-grain data management. This cache contains the mapping between DRAM and PCM address together with the bitmap to indicate whether the cache line to be written back is present in the DRAM or PCM partition. We call the process of directing the access to either the DRAM or PCM partition a *remap*.

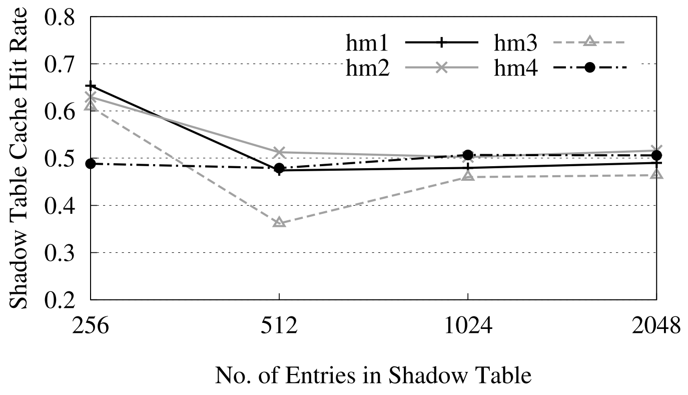
Figure 4-10a shows the impact of the shadow table cache of various sizes on performance. A larger shadow table cache would presumably result in better performance. However, our experiments show that the IPC stabilizes with a shadow table cache with 1024 entries. On further investigation, we found that this was due to a stable hit rate being attained. Figure 4-10b shows the hit rate of the shadow table cache for the varying sizes. The hit rate stabilizes at 1024 entries, and therefore for all our experiments, we used a shadow table cache with 1024 entries.

4.6.6 DRAM Sizes

The main motivation of a hybrid memory design is to use a smaller DRAM to filter out write accesses to a bigger PCM. For our experiments, we simulated an



(a) IPC for varied size of Shadow Table Cache

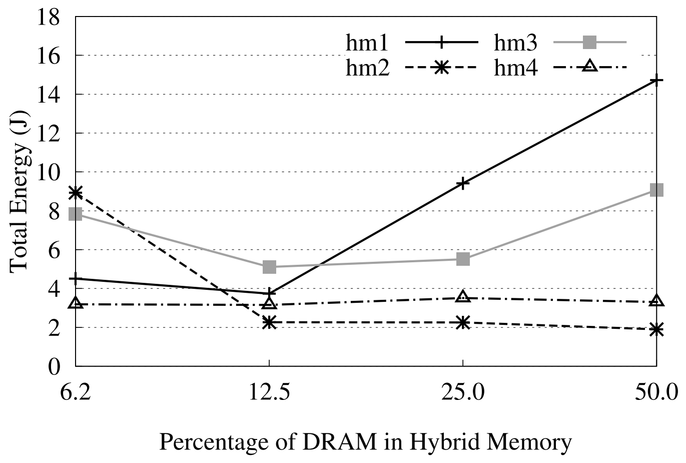


(b) Shadow Table Cache Hit Rate

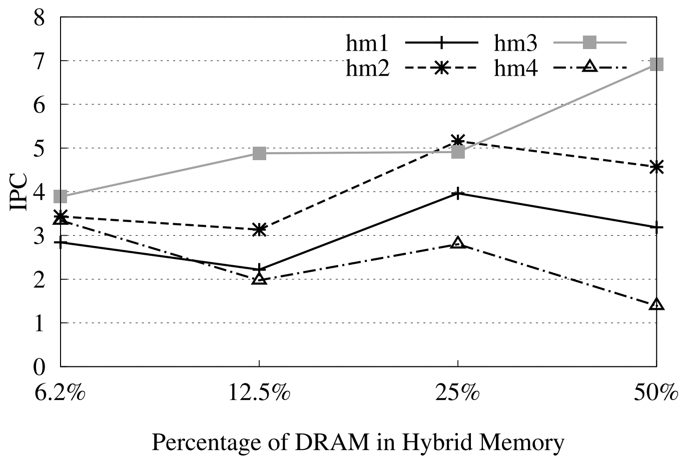
Figure 4-10: Study on Shadow Table Cache.

8 GB PCM partition. This is the maximum capacity our simulator can handle. Alongside this 8 GB PCM partition, we varied the size of the DRAM partition to examine its impact on power and performance.

Figure 4-11a presents the energy consumption of the main memory for varying sizes of DRAM for the high memory workloads. The percentage of DRAM in hybrid memory represents the size of DRAM as compared to the PCM. A very small DRAM translates to a lot of write accesses to the PCM partition as it is too small to hold the write working set of the application. Therefore, for 6.2%,



(a) Energy with Varied DRAM Sizes



(b) Performance Impact with Varied DRAM Sizes

Figure 4-11: Study on varied DRAM sizes.

i.e. 512 MB of DRAM with 8 GB of PCM, the energy consumption is high. For 12.5% and 25%, the energy consumption is the minimal as the DRAM partition is able to accommodate the write working set, and alleviating the writes to PCM. As expected, with an increased DRAM size, the energy consumption also increases. Figure 4-11b shows the impact varying the size of the DRAM partition has on performance.

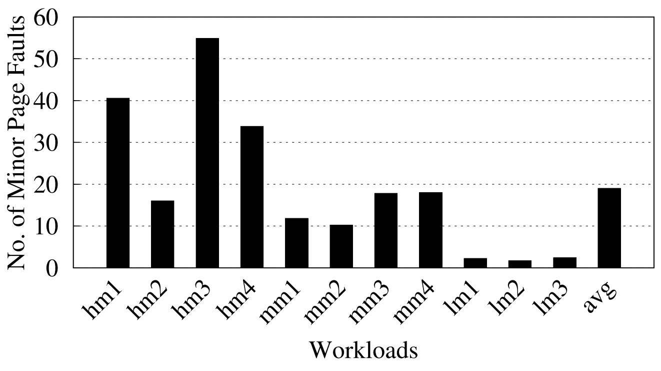


Figure 4-12: Total number of minor page faults.

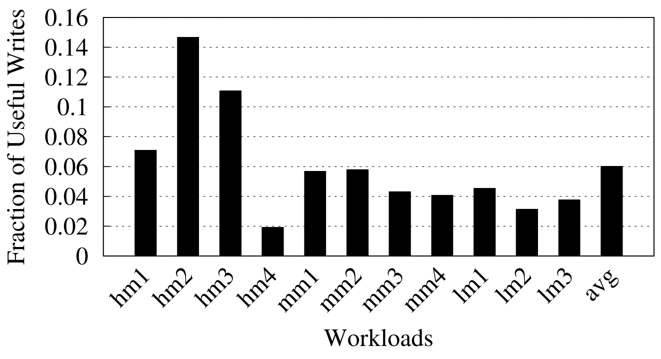


Figure 4-13: Amount of useful writes to PCM.

4.6.7 Page Reclamation

The cache writebacks to PCM generates a minor page fault for which the operating system allocated a DRAM shadow page to service writes. As the page does not need to be filled in from the disk, the latency of the minor page fault is significantly lower than that of a conventional page fault. Figure 4-12 shows the total number of such minor page faults triggered by the workloads during the execution of 4 billion instructions. Figure 4-13 shows the fraction of dirty cache lines within a page that was written to the PCM due to shadow page evictions. The graph shows that on average 94% of the page contained clean data and only 6% was dirty and our framework is successful in writing only this 6% of data to the PCM eliminating the rest.

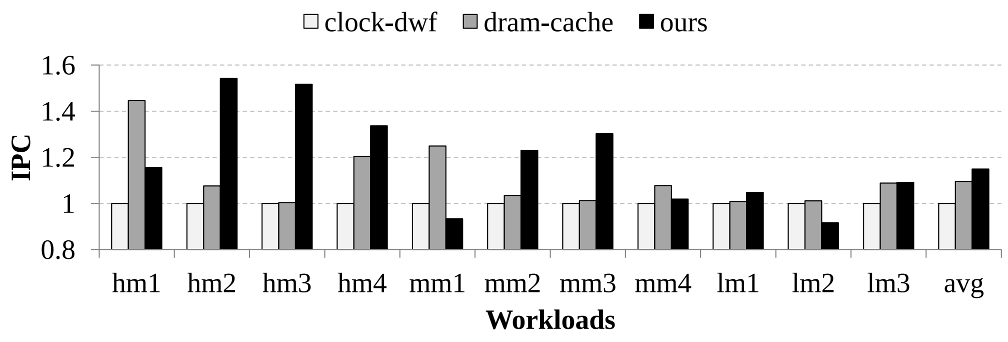


Figure 4-14: IPC performance when L2 is the LLC.

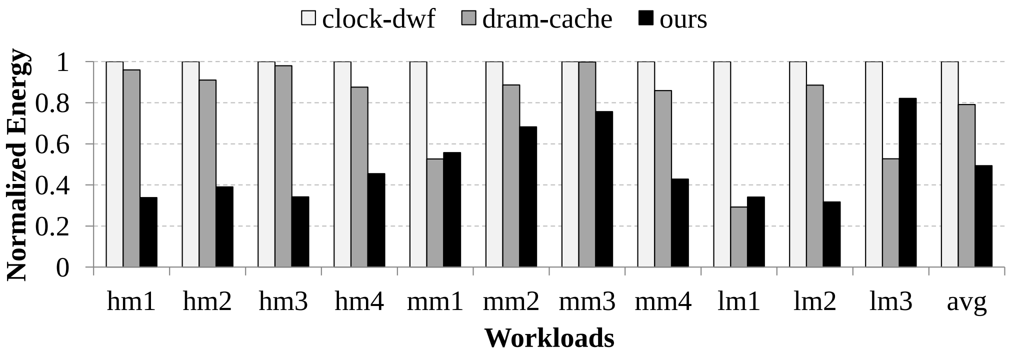


Figure 4-15: Normalized energy consumption when L2 is the LLC.

4.6.8 L2 as Last Level Cache

For all the above results, we assumed a three level cache hierarchy. We have also evaluated a processor configuration in which there are only two levels of cache hierarchy and the unified L2 cache is the last level cache. This translates to increased memory access to the main memory. In such scenarios, it could be that an optimal algorithm for hybrid memory with the L3 cache as last level cache might not be sufficient to control the writes to the PCM partition.

To check this, we repeated our experiments assuming the L2 cache as the last level cache, using `clock-dwf` and `dram-cache` for comparison. As `dram-cache` utilizes a smaller DRAM as a cache and PCM is accessed only for DRAM misses, having increased DRAM traffic significantly increases writes to the PCM. This not only impacts performance, also worsens energy consumption. Figure 4-14 shows the performance of such a system. Our framework outperforms both `clock-dwf` and `dram-cache` in terms of IPC showing an improvement over `clock-dwf` and `dram-cache` of 14% and 4.8%, respectively. The gain in energy efficiency over `clock-dwf` and `dram-cache` are 49% and 54%, respectively. Thus, our framework is effective for hybrid main memory management regardless of whether the LLC is L2 or L3. Having L2 as LLC indicates that the writes to the DRAM partition increases. However, our write aware page reclamation policy keeps the write intensive pages in DRAM and thus continuous LLC writebacks to these pages are filtered by DRAM.

4.7 Chapter Summary

In this chapter, we propose a fine-grain data management for hybrid main memories composed of a DRAM and a PCM partition. In our scheme, cache writebacks are monitored and redirected to the DRAM partition only, thereby reducing the amount of write operations the PCM partition is subjected to. Furthermore, during DRAM eviction, only dirty data in units of cache lines is written to the PCM partition, instead of entire pages that we found mostly contained clean data. This reduced the writes to the PCM partition drastically. We compared our framework with two state-of-the-art techniques `clock-dwf` and `dram-cache`. Our experiments show that our framework offers 58.8% and 65.5% reduction in energy consumption compared to `dram-cache` and `clock-dwf`. Our solution reduces writes to PCM by four orders of magnitude as compared to the above mentioned works. In addition, our solution exhibits a mere 0.7% degradation in performance in terms of instructions per cycle when compared to `dram-cache`, and a 10% improvement in IPC over `clock-dwf`. We believe that the hardware overhead of our proposal is modest, and the combined benefit from involving the operating systems in the process is significant gain from the state-of-the-art. We hope this will contribute to the acceleration of the adoption of the next generation non-volatile memories in actual systems.

Chapter 5

Error Management through Approximate Computing

This chapter focuses on the second crucial challenge faced by the energy efficient memories i.e. *Error Susceptibility.* We would propose approximate computing based approach which abates the overheads of refreshes or error detection and correction for the energy efficient memories. In particular, this chapter would elaborate on a specific problem of identifying which program parts are favourable to approximation, towards which we would propose a dynamic program testing scheme.

Approximate computing is a new programming paradigm allowing programs to trade-off accuracy in favour of lower power consumption. It is especially appealing to low-power embedded devices where energy efficiency is of serious concern. Further, there are many applications targeted to smartphones, tablets, etc. that are capable of tolerating inaccuracy while maintaining the desired quality of service (QoS).

Many recent works have shown this to be a promising trade-off for current and future embedded platforms [1, 37, 39, 40, 43, 75]. Programs contain specific parts that contribute to the correctness of the output and others that do not. A correct output usually lies within a Quality of Service range of the application. The parts of a program that do not affect the output beyond a tolerable extent are deemed *approximable*, while parts that are important are *non-approximable*. Depending on the application, these two parts can vary significantly.

The presence of approximable data in a program is the mainstay of the approximate computing paradigm. The non-approximable program data can be computed and stored in a high power mode, while the approximable regions in a low power mode [39]. Applications allowing such behaviour are called *error-tolerant*. Error-tolerance of applications running on devices prone to soft-errors is well studied [34–36]. However, in this new paradigm, instead of mitigating the errors, a controlled degradation of QoS due to the errors is allowed.

5.1 Motivation

Discovering distinct approximable and non-approximable parts of a program *automatically* is a difficult task. Sampson et al. recently proposed source code annotations and type-qualifiers for programmers to indicate whether a variable (data) is error resilient, in other words, approximable [1]. However, this implies rewriting or annotating source codes. This may be easy for small programs, but is difficult for complex programs and legacy software.

Other works [37, 46, 76, 77] have shown that program approximations can be achieved through algorithmic choices, runtime decision making frameworks, and on the architectural or device level. The provision of algorithmic choices too is the programmer's responsibility and the application is compiled using all the versions of a procedure. This is not only difficult to apply to large applications having large numbers of procedures, it also inflates size of executables. Such consequences impede the usage of these solutions for embedded devices.

Our motivation for this work is twofold. First, is to alleviate the existing burden placed on the programmer in facilitating approximate computing. We aim to provide an automatic analysis for a program and identify data that can be approximated. In scenarios where annotating programs without programmer's knowledge is considered unsafe, ASAC would serve as the suggestive framework for annotating bigger and more complex programs. Programmers can then fine-tune ASAC's analysis results to obtain the final partitioning.

In any case, it is obviously expensive, time consuming, and in some scenarios, infeasible to identify approximable and non-approximable data, and annotate the application completely manually. Moreover, for legacy software and other programs that have undergone significant changes over many versions, it may be difficult to understand the implications of approximated variables and their effect globally. Therefore, an automated analysis is indispensable for approximate computing in the large.

Our second motivation is to study the error-resilience of internal program data i.e. program variables, etc. Error-resilient program transformations have been well studied. However, all the existing works focus on approximating different components such as procedure approximation, input data approximation, control- flow based approximation etc. [42, 46]. Other works have studied the error-resilience of data in architectural components such as the arithmetic units, register files, etc. [77, 78]. Here, we are proposing a framework to analyse and approximate internal program data while maintaining an acceptable QoS according to the application.

5.2 Our Proposal

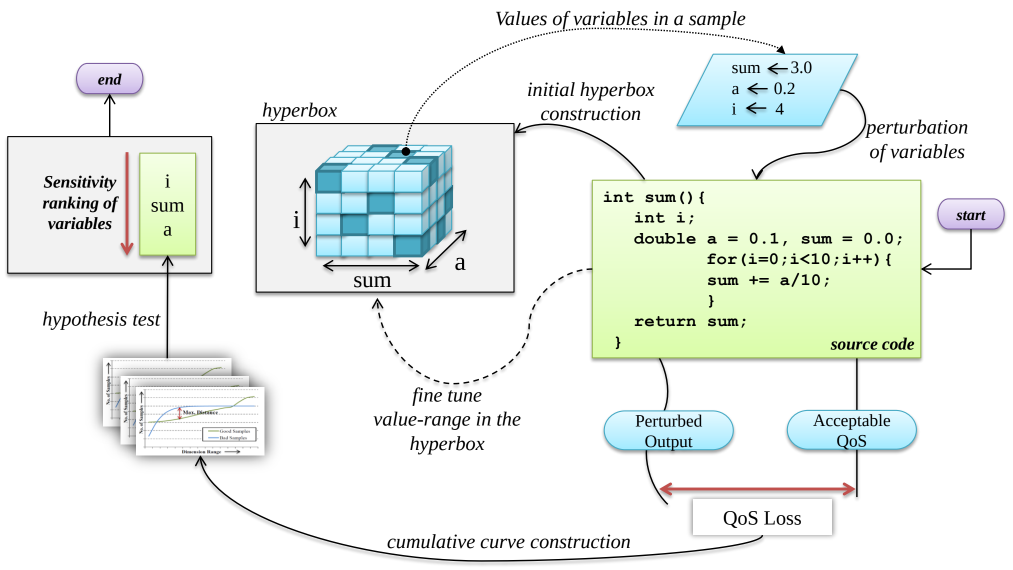
In this chapter, we propose "ASAC" - *Automatic Sensitivity analysis for Approximate Computing*, a framework to *automatically* discover approximable data from a program. The key idea is to systematically perturb the program variables and to observe its effect on program output. By quantifying the sensitivity of the output to the perturbations, we can discern program variables in terms of their contributions to the output. A variable that does not contribute to the correctness of the output or the functionality of the program beyond a certain extent is not considered as critical, and therefore can be approximated. Conversely, critical variables cannot be approximated and must be precise.

The main component of this framework is a specialized sensitivity analysis using statistical methods. Sensitivity analysis of parameters of mathematical models using statistical methods is known in literature [79]. Our contribution in this work is the use of statistical methods for sensitivity analysis of program data.

Figure 5-1 illustrates ASAC consisting of 3 main stages, namely *discovery*, *probe* and *testing*. In discovery stage, we extract the variables of a program along with the range of values that each can assume during the execution. The cartesian product of the variable range intervals defines an n-dimensional hyperbox. This hyperbox is the sample space for the statistical experiments performed by the sensitivity analysis module. Each dimension represents a variable and the corresponding edge of the hyperbox is the range of that variable. Therefore, the total number of dimensions in the hyperbox is determined by the number of variables in the program.

The hyperbox represents the value-space of the program during its execution. At the subsequent probe stage, we first divide the hyperbox into smaller hyperboxes of equal sizes. We select a subset of these smaller hyperboxes, the *samples*, and choose a number of points from among them. Each of these points are n-tuple coordinates containing the values of each variable at that point. These points are passed to the program and the values are forcefully assigned (perturbed) to corresponding variables during the execution by means of binary instrumentation.

Due to the intrusion, the program output can be expected to be deviated from the correct output. Our aim is to measure this incorrectness. According to


Values of variables in a sample
end
hyperbox
initial hyperbox construction
sum ← 3.0
a ← 0.2
i ← 4
perturbation of variables
Sensitivity ranking of variables
i
sum
a
i
sum
a
start
source code
hypothesis test
fine tune value-range in the hyperbox
Perturbed Output
Acceptable QoS
QoS Loss
cumulative curve construction

```
int sum(){
   int i;
   double a = 0.1, sum = 0.0;
          for(i=0;i<10;i++){
          sum += a/10;
          }
   return sum;
 }
```

Figure 5-1: Overview of "ASAC" framework. Each box represents a step and the arrows are the dataflow between them. There is an information flow from Sampler back to the Hyperbox Construction to facilitate further optimization in range analysis.

the difference between the QoS threshold of the application and the perturbed outputs, we mark each such sample as "good" (pass) or "bad" (fail).

Next, in the testing stage, a cumulative distribution curve is obtained by plotting the number of good or bad samples against the range of each dimension of hyperbox. The two curves undergo a hypothesis test that generates the maximum distance between them. A large distance between the curves means that the program output is very sensitive to the variable representing that dimension of the hyperbox. Conversely, a smaller distance implies the opposite. All the stages are described in details in the following sections.

ASAC is fully automatic and alleviates the programmer's involvement. With minor modification, it can also be applied to programs where the source code is not available. A direct application of this framework can be as a feedback system to a compiler, providing information about how the programs may be approximated. Moreover, ASAC can be used as a black-box tester to gain insight about the sensitivity of program output against program data. This would be valuable information for platforms susceptible to soft-errors, where instead of allowing the approximation, the sensitivity of the variables can be used as a metric to decide which data should be protected.

We evaluated our analysis against a 'gold' standard where a programmer has made type-qualifier based annotations to programs to facilitate approximation [1]. We achieve 86% accuracy in determining approximable data with respect to this manually annotated baseline (MAB). In addition, to show the scalability and generality of our analysis, we apply it to bigger and more complex programs from MiBench and SPEC2006 benchmark suites. Our contributions in this work are summarized as follows:

- The first automated software analysis that allows approximate computing based programming paradigm.
- A framework to discover program data that can be approximated without compromising the QoS of a given application.
- A black-box analysis that can test programs and order the variables in terms of their contribution to the correctness of the final output.

5.3 Automated Analysis

In this section, we will describe the three stages in detail. First, we explain two main concepts integral to the discovery stage - range analysis and hyperbox construction. Range analysis is well studied. It is commonly used to detect integer overflows, etc. However, here we apply range analysis to estimate the values that a variable can assume during program's execution.

Definition 5.3.1. *For each variable V_i in program under analysis, let value(V_i) be the value that V_i can assume during program execution. Then, range(V_i) =*

Algorithm 5.1 Range Analysis

```
Require:
 1: Program P, QoS Threshold Q
Ensure:
 2: $R[n]$, where n ← no. of variables in P

 3: Initialize $rangeOf(V_i) = \emptyset \ \forall \ V_i$ in $P$
 4: for each variable $V_i$ in $P$ do
 5:     if $rangeof(V_i) = \emptyset$ then
 6:         $var \leftarrow V_i$
 7:         $R_i[2] \leftarrow$ RANGE_ANALYSIS(var) /* standard widening & narrowing operator based */
 8:         if $R_i[0] \vee R_i[1] = \infty$ then
 9:             if DATATYPE(var) = int_32 then
10:                 $R_i[0] \leftarrow -32767$ /* standard data */
11:                 $R_i[1] \leftarrow 32767$ /* range for int type */
12:             else if DATATYPE(var) = float then
13:                 $R_i[0] \leftarrow 0$ /* dummy range */
14:                 $R_i[1] \leftarrow 1$ /* that will shrink over runs */
15:             else
16:                 /* handle all datatypes similarly */
17:             end if
18:         end if
19:     end if
20: end for
21: return $R_i[]$
```

Variables	Datatype	Initial Range	Tuned Range
LineSadBlk0	double	[1 , 1]	[0.0 , 780.0]
P_A	int	[2048 , 2048]	[128 , 128]
P_E	int	[-32768 , 32767]	[34 , 244]
D_dis1	double	[1 , 1]	[-15.0 , 177.0]

Table 5.1: Ranges of some variables in H.264

$[R_{i1}, R_{i2}]$, *where* $R_{i1} \leq value(V_i) \leq R_{i2}$. *If* $R_{i1} = \pm\infty$ *or* $R_{i2} = \pm\infty$, $range(V_i)$ *is given by the datatype of* V_i.

This value range is essential for the construction of the hyperbox. We employ widening and narrowing operators based dataflow analysis to calculate the value ranges of the variables [80]. Algorithm 5.1 gives a pseudo-code description of our range analysis. In cases where the analysis is unable to generate a finite value range, we fine-tune the range based on the data type of the variable (line 9-12). For floating-point variables, we assume a dummy starting range of zero (line 13-14). In order to extract the real value range, we have an information loop back (see Figure 5-1) from the sampler to hyperbox construction which

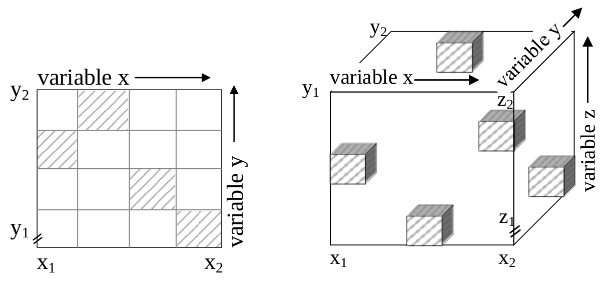


Figure 5-2: Example of 2 dimensional and 3 dimensional hyperboxes

makes it easy to get narrow and precise value-ranges from the profile runs of the program. Therefore, even if the dataflow analysis generates an infinite range for a variable, it is soon mitigated. This is shown in Table 5.1 with the examples of some of the variables in H.264. After calculating the ranges of the variables, we can construct the hyperbox.

Definition 5.3.2. *An n-dimensional hyperbox $\mathbb{H}$ is the cartesian product of the range intervals of each of the n variables.*
$\mathbb{H} = [R_{11}, R_{12}] \times [R_{21}, R_{22}] \times \ldots \times [R_{n1}, R_{n2}]$*, where* $[R_{i1}, R_{i2}]$ *is the range of variable* (V_i)*.*

Figure 5-2 shows a conceptual diagram of hyperboxes. Each dimension represents a variable and thus with n variables it will have $n - dimensions$. The starting and ending point of each dimension is R_1 and R_2 of each variable i.e. the range. As the value range of a variable can narrow or widen over runs, the hyperbox may also shrink and grow. The shaded areas are called *samples*. These are small hyperboxes obtained by discretizing the edges, and selecting only a subset from among them. Discretization provides a finite sampling space from the original hyperbox which has infinitely many sample points. The finite sample space can then be sampled using any statistical sampler.

In our framework, we have used the *Latin Hyperbox Sampling* (LHS) algorithm [81]. LHS ensures that the sampling is bias free and with a fairly well coverage of the sample space. As shown in Figure 5-2, the 2-D hyperbox is discretized into equal sized grids, and only one sample from each row and column are qualified to be in the subset. For n-dimension, LHS selects only a subset of the samples based on their positioning. The complexity of this method depends on two factors - n, number of variables, i.e. the dimension of the hyperbox, and the constant k i.e. the discretization parameter. Empirically, the number of samples to be selected from a hyperbox can be defined as follows -

$$\text{Number of samples} = (\prod_{n=0}^{(k-1)} (k-n))^{n-1}$$

Next, in the first step of the probe stage, we choose m uniformly random points from each sampled hyperboxes. We will present a study of the effects of the constants in a later section. Each of these points are an n-tuple coordinate, where n is the number of variables in the program. For example, a point m_i from a sample s_i has the coordinates $(m_{i_1}, m_{i_2}, ...m_{i_n})$, where m_{i_1} is the value of variable V_1 at the point m_i.

The points can be represented as a vector of real numbers. We use these vectors to introduce perturbation in the program execution by passing the values dynamically with an instrumentation tool. We call a program execution with the perturbed values a *probe run*. As the hyperbox was originally constructed by the value ranges of the variables, the perturbation for each variable lies within the range of values the variable is expected to assume during executions.

Definition 5.3.3. *Let P_i be a vector of the outputs of all the probe runs of sample S_i, f_{obj} be an objective function, and θ is a constant threshold. If $f_{obj}(P_i) \geq \theta$ then designate P_i to be a "good" sample, else mark it as a "bad" sample. We define the objective function as*

$$f_{obj} = (\sum_{j=0}^{j=k} \omega(P_i))/k$$

where $\omega(P_i)$ = 1 if $P_i \geq T_{qos}$, otherwise $\omega(P_i)$ = 0. T_{qos} is the QoS threshold for the application given by the user.

Algorithm 5.2 Hyperbox Construction & Sampling

```
Require:
 1: $Range[n][2]$, where n is no. of variables in program
 2: $k$, discretization factor
Ensure:
 3: $Vector[n][k]$, containing the values to be passed to program for perturbed run
 4: procedure HYPERCUBE(Range[n][2],n)
 5:     Initialize $H \leftarrow \emptyset$
 6:     Initialize $dim = n$
 7:     for $i = 0$ to $dim$ do
 8:         $H[i].leftdiagonal \leftarrow Range[i][0]$
 9:         $H[i].rightdiagonal \leftarrow Range[i][1]$
10:     end for
11: end procedure

12: procedure LATIN HYPERCUBE SAMPLING(H[n],dim,k)
13:     for $i = 0$ to $k$ do
14:         for $j = 0$ to $dim$ do
15:             $L = H[j].leftdiagonal$
16:             $U = H[j].rightdiagonal$
17:             $Interval\_Size = (U - L)/k$
18:             $Interval\_Val =$ CHOOSERANDOM(i,j)
19:             $LowLim = Interval\_Val * Interval\_Size$
20:             $T[0][0] \leftarrow L + LowLim$
21:             $T[0][1] \leftarrow L + LowLim + Interval\_Size$
22:             $Sample[i][j] \leftarrow Sample[i][j] \cap$ HYPERCUBE(T[1][2],j)
23:         end for
24:     end for
25:     return $Sample^T$
26: end procedure
```

Algorithm 5.2 illustrates the detailed steps involved in the construction of hyperbox and how points from among the samples are chosen. First, a hyperbox is built using the preliminary value ranges obtained from the range analysis (line 3-10). For each variable represented by a particular dimension (edge) of the hyperbox, the edge is discretized into k intervals (line 13-17).

Therefore, we have $dim * k$ number of smaller hyperboxes after this step, where dim is the total number of variables and k is the discretization constant. A subset of these smaller hyperboxes is chosen using LHS to have a fair coverage of the ranges (line 18-22). The samples represent the set of values to be passed to the program in the probe runs.

The perturbed outputs from all the probe runs are partitioned into two classes - "good" or "bad", based on the QoS threshold of the application. From all the samples marked as either good or bad (0 or 1), we construct a cumulative curve

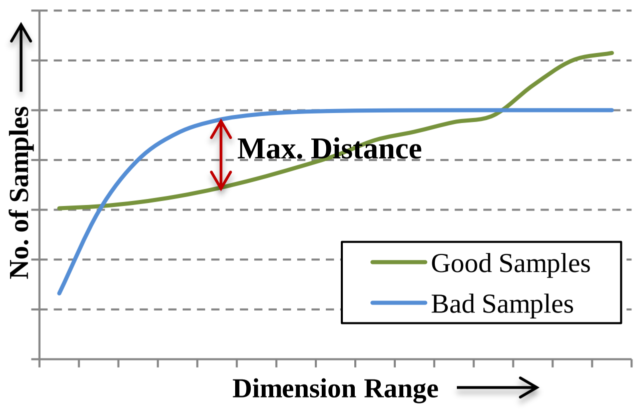


Figure 5-3: Example CDFs of "good" and "bad" samples based on the QoS and distance metric.

for each dimension of the hyperbox. The number of good samples is counted, and plotted against the range of that dimension. Similarly, a second curve is obtained by counting the samples marked as bad. These two curves are regarded as two cumulative distributions obtained from the perturbed program runs.

Intuitively, the distance between the two curves denotes the contribution of these variables towards the program output. We apply the *Kolmogorov-Smirnov* hypothesis test [79] to calculate the maximum distance between the two curves. This is called the d-statistics, and it translates to the sensitivity ranking: the higher the distance, the higher is the sensitivity of the output to this variable, and vice-versa. Formally -

Definition 5.3.4. *Let Sen_i denote the sensitivity score for a variable V_i. Let $f_{good}V_i$ and $f_{bad}V_i$ be the two cumulative distribution function (CDF) for variable V_i. Then, $Sen_i = max_x|f_{good}V_i(x) - f_{bad}V_i(x)|$, where x is a point in the value range of the variable V_i at which the CDFs are calculated.*

Figure 5-3 shows an example of the cumulative curves, and the maximum distance between them. A detailed step-by-step description of the generation

Algorithm 5.3 Sensitivity Ranking

```
Require:
 1: $Vector[n][k]$, containing the values to be passed to program for perturbed run
 2: $Q$, QoS Threshold
Ensure:
 3: $SenScores[n]$, sensitivity scores for variables
 4: procedure PROGRAM PROBE(Vector[n][k])
 5:     Initialize $Values[n] \leftarrow \emptyset$
 6:     Initialize $dim = n$
 7:     for $j = 0$ to $k$ do
 8:         for $i = 0$ to $dim$ do
 9:             $Values[i] \leftarrow Vector[i][j]$
10:         end for
11:         $Output[j] \leftarrow$ program executed with $Values[]$
12:         Update hypercube
13:     end for
14:     return Output[]
15: end procedure

16: procedure HYPOTHESIS TEST(Output[],Q)
17:     for $i = 0$ to $k$ do
18:         $err =$ GETERRORFUNCTION(Output[i])
19:         if $err \leq Q$ then
20:             $Good[i][] = Values[]$
21:         else
22:             $Bad[i][] = Values[]$
23:         end if
24:     end for
25:     for $i = 0$ to $dim$ do
26:         $j \leftarrow R_i[0]$
27:         while $j \neq R_i[1]$ do
28:             if $j \in Good[i][]$ then
29:                 $C_{good}[j]++$
30:             else if $j \in Bad[i][]$ then
31:                 $C_{bad}[j]++$
32:             end if
33:             $j+= Interval_Size$
34:         end while
35:         $SenScores[i] \leftarrow$ KS_TEST$(C_{good}, C_{bad})$
36:     end for
37: end procedure
```

of the sensitivity scores is given in Algorithm 5.3. First, the program probe step is detailed in lines 3-15. The procedure receives the vector of values from the hyperbox as input and runs the program by forcefully assigning these values to the variables. Each program run produces a result that is stored to be compared for QoS at a later stage. In this procedure, the hypercube is also updated with fine-tuned range of the variables. Next, in the hypothesis test procedure, an error is calculated from the obtained result and the original result of the program (line 18). This error is used to mark a sample as good or bad. Following this marking, considering all the samples from the hyperbox, a cumulative graph is plotted against each dimension (lines 27-33). Two curves are obtained for each dimension of the hyperbox and they are passed to the KS-Test for the distance metric (line 35).

5.4 Optimizations

5.4.1 Discretization Constant

Our proposed analysis has one tunable parameter, the discretization constant, k. This determines the size of the samples for each dimension in the hyperbox. In other words, all the value ranges of the variables are divided up using this constant k so as to reduce the value space (see Algorithm 5.2).

The completion time of the analysis is affected by this parameter. A larger value will cause the analysis to take a longer time to complete because the hyperbox is divided into smaller grids. However, the sensitivity scores obtained from the analysis is not affected by the value of k as shown in Table 5.2. Thus, we can conclude that the sensitivity of program output with respect to its variables is a characteristic of the program. For our evaluation, we tested with $k = 10, 50, 100, 200$.

Figure 5-4 shows the total time taken by ASAC to complete its analysis. As shown, when $k \leq 100$, ASAC takes longer time to rank the variables. Table 5.2 shows the percentage of total variables marked as approximate with two different k values. The percentages for $k = 5$ and $k = 200$ are same as that for $k = 10$ and $k = 100$, respectively. The difference between $k = 10$ and $k = 100$ is attributed to the fine tuning of the ranges of the variables. As the percentages are averaged over 20 runs, different program paths will result in different fine-tuning of the variable ranges. Nonetheless, the difference of percentage of variables marked as approximate shows no significant variation over the values of k as shown in

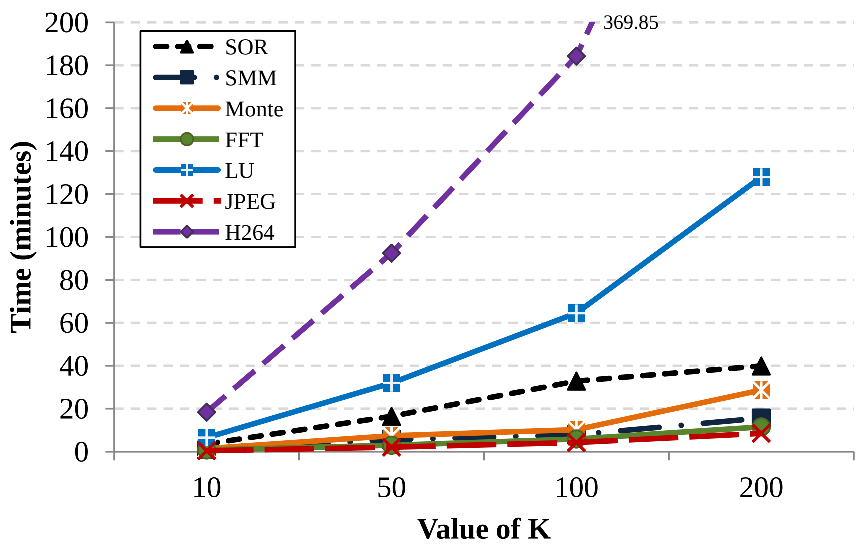


Figure 5-4: Total runtime (minutes) of ASAC with values of k while $m = 2$.

Table 5.2.

There is another constant m, which determines how many points will be chosen from within one sample to perturb the program. We observe an interesting trend in the relationship between k and m. As the samples are small in size with a high value of k, increasing the value of m, i.e. choosing many points within a narrow range, results in passing similar values for probing.

Therefore, the value of this constant m has no significant impact on the variable ranking when k is high. Nonetheless, a high value of both k and m will translate to higher running time for our analysis. When k is small, the value of m has an impact on the variables' ranking. A small value of k and m will result in sampling a few representatives from a large hyperbox causing poor coverage of the sample space. This behaviour is accentuated in bigger programs, such as JPEG and H.264. However, it is important to have perturbations with values of variables that are uniformly distributed over its range. Therefore, for our experiments we used $m = 5$ and $k = 100$.

Bench-marks	Total Decls	Data Approximable(%)			
		$k = 2$		$m = 5$	
		$m = 2$	$m = 10$	$k = 10$	$k = 100$
SOR	28	28	28	28	28
SMM	29	27	27	27	27
Monte	15	33	33	33	33
FFT	85	32	35	36	35
LU	150	6	9	9	10
JPEG	1174	6	10	11	11
H.264	11857	7	15	16	16

Table 5.2: Percentage of variables marked as approximable by ASAC with different values of k and m.

5.4.2 Perturbation Points

In the probe stage, we force variables to assume values chosen from the hyperbox. We use the dynamic instrumentation tool PIN [82] to inject the values at runtime. There are two important issues that we would like to discuss here. First, it is a challenge to identify program points where the variables are perturbed. For example, if a perturbation is introduced at a point where a variable is first *used* after being *defined*, then the effect on output will be different than if the perturbation is introduced at a later point.

In the former, the error might propagate and accumulate, resulting in a large deviation from correct output. On the other hand, the error might get masked by further arithmetic operations on the variables [46]. We introduce the perturbations at the first usage of a variable after it is defined. Nonetheless, it would be interesting to study the effects of the perturbation at other program points. The second challenge is in injecting error into loop structures. It is difficult to force values into loop variables because of its iterative nature.

Our aim is to perturb a variable to see the effect on the output. However, if the loop factor is high, then injecting the perturbation at every iteration becomes too aggressive. Instead we chose to perturb only a subset (25%) of the loop iterations. This technique is analogous to the concept of *loop-perforation* [43].

5.4.3 Instrumentation & Testing

ASAC involves ranking the variables using their identifiers, i.e. names, which are not easily accessible after the code generation, especially at runtime. Therefore, it is difficult to pass the perturbation values to the program during the probe runs. To force a sample value into a program, we implemented a compile-time pass that will inject additional code at the appropriate program point in the code to read the value to be forced into a variable from a file, and perform the write of that value into the variable.

We also found that for larger applications, it was easier to use the PIN tool to inject such values - provided they are not bound to registers - into variables using their virtual addresses. In the actual implementation, we used a combination of both. In the testing and evaluation of ASAC, we adapted the *bitflip error model* used by many prior works [42, 78, 83] to introduce errors into the application.

A bitflip error essentially means that one or more bits within a data toggles one or more times during execution of the program, inducing an error. We used the same two techniques described above except that in the testing and evaluation, instead of forcing a targeted variable to take a certain value, we choose a (uniformly) random bit among the 16 lower bits of its current, and toggle it. There are many other error models available in literature, we chose bitflip because it is fairly simple to understand and model. Nonetheless, more complicated error models could also be used.

5.5 Evaluation

We evaluated ASAC against a manually annotated baseline (MAB) that uses type-qualifiers [1]. The authors kindly provided us with benchmarks from SciMark2 [84] that had such annotations made. We also apply ASAC to two bench-

Application	Benchmark	Error Metric	LOC
SOR	SciMark2	Mean Square Error	36
SparseMatMult	SciMark2	Normalized difference	38
MonteCarlo	SciMark2	Normalized difference	59
FFT	SciMark2	Mean Square Error	168
LU	SciMark2	Mean Square Error	283
JPEG	MiBench	SNR	30781
H.264	SPEC2006	SNR	46190

Table 5.3: Description of all the benchmarks used for evaluation.

marks from SPEC2006 [62] and MiBench [85] to test its scalability.

To measure the QoS loss due to approximation, we defined the error metric for each application, shown in Table 5.3. For FFT, LU and SOR, we use the mean squared error between the correct answer and the approximated output to quantify the degradation. For applications like SparseMatMult and MonteCarlo, we measure the normalized difference i.e. 0 if the approximated output is equal to correct output and 1 if not. For JPEG and H.264, we use the signal-to-noise ratio (SNR). The error estimation module as well as the QoS threshold is deemed to be provided by the user for our analysis. This makes it easy and portable.

Comparison with Manually Annotated Baseline (MAB)

Table 5.4 shows the detailed comparison of ASAC with MAB. We shall examine the *precision, recall and accuracy* metrics of these experiments.

Precision measures how frequent a variable marked by ASAC to be approximable is also annotated as approximable in the MAB. Empirically it is -

$$\frac{tp}{tp+fp}$$

where ‘tp’ and ‘fp’ are the ‘true positive’ and ‘false positive’ in Table 5.4, respectively. The former are those variables found to be ‘approximable’ in both ASAC and MAB. The latter are variables that ASAC declared to be ‘approximable’ but were annotated as ‘non- approximable’ in MAB. ASAC achieved a precision

Bench-marks	**True Positive *(tp)***	**Flase Positive *(fp)***	**False Negative *(fn)***	**True Negative *(tn)***	**Precision**	**Recall**	**Accuracy**
SOR	5	0	1	2	0.83	1.00	0.88
SMM	1	0	1	6	0.50	1.00	0.88
Monte	2	0	1	2	0.67	1.00	0.80
FFT	15	2	2	12	0.88	0.88	0.87
LU	7	1	1	5	0.88	0.88	0.86
				Average	0.75	0.95	0.86

Table 5.4: Comparison of *ASAC* with "EnerJ" [1].

of 75%. The 25% loss in precision is due to the fact that our framework is more optimistic in marking variables as approximable.

Recall measures the robustness of our analysis. It is the complement of the percentage of variables our analysis mistakenly classifies a variable as non approximable while MAB has annotated it as approximable, defined as follows

$$\frac{tp}{tp+fn}$$

where 'fn' is 'false negative', variables that are marked as 'non-approximable' as ASAC but annotated as 'approximable' by MAB. These are the cases where our analysis fails to exploit approximable variables. Our analysis shows a high recall value of 95%.

Accuracy is a metric that combines precision and recall, and quantifies how much we can match the classification by MAB. It is defined as

$$\frac{tp+tn}{tp+tn+fp+fn}$$

where 'tn' are the 'true negatives', i.e., the variables that both ASAC and MAB agree are non-approximable. We achieve a high accuracy of 86%, using ASAC's fully automatic approach. The accuracy can be improved further by optimizations discussed in Section 5.4.

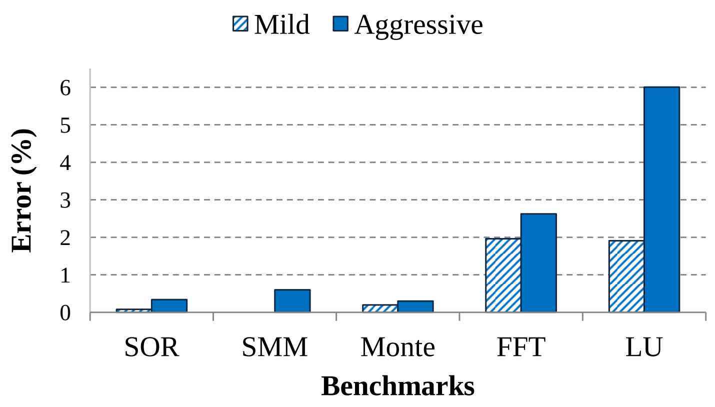


Figure 5-5: Percentage of error after approximating program data. The two bars are different error percentage after approximating either one-third or all the data that are classified as approximable by ASAC.

Error Measurement

In order to quantify the error due to approximation of program data, we evaluated different levels of error injection: two levels in JPEG, and three levels for H.264. First, in the `Mild` injection, errors are injected to only 50% of the variables marked as approximable. This half is chosen from the lower ranked variables (lower sensitivity scores) among those that are marked as approximable. Bitflip errors were injected into these variables during runtime.

Second, in the `Aggressive` injection, errors are injected to all the variables identified as approximable. For H.264 that has a large number of variables, we created one more level of inject - `Medium`. For this benchmark, we chose the lowest scored one-third as the `Mild` injection, 60% for `Medium` and 100% (all) for `Aggressive`. Figure 5-5 shows the error percentages for the SciMark2 applications under `Mild` and `Aggressive` error injection.

Figure 5-7 shows the result when `Mild` and `Aggressive` error injections were applied to the JPEG benchmark. We applied error injection to the encode and decode steps separately to show the effect of error accumulation. In the Figure 5-7(e) the errors are aggravated as it takes Figure 5-7(d) as its input which

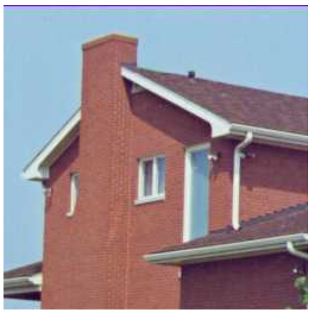
H.264	SNR_Y	SNR_U	SNR_V	BitRate
Correct	36.67	40.74	42.31	149.62
Mild	36.69	37.64	37.65	146.6
Medium	34.05	36.92	36.79	147.12
Aggressive	29.78	32.89	32.99	146.03

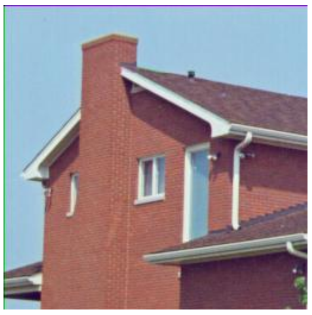
Table 5.5: H.264 Approximation Results

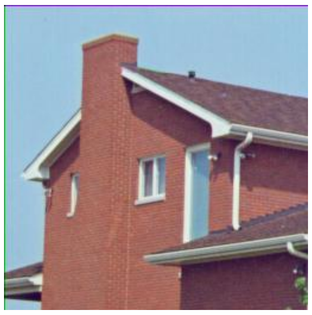
already contained the errors injected in encode step. Therefore, the `Aggressive` approximation for decode step is actually more severe than what it would have been if taken in isolation. Table 5.5 shows the approximation results for all the `Mild`, `Medium` and `Aggressive` applied to H.264.

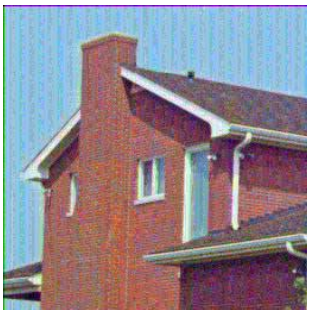
Further Studies on JPEG and H.264 - Sanity Check

As we do not have manual annotations for JPEG and H.264 benchmarks, we studied the effect of injecting errors into the variables that ASAC has marked as non- approximable. Essentially, there were two scenarios. First, when `Aggressive` error injection was applied to those variables deemed non-approximable (i.e., precise), the output of the JPEG benchmark was a corrupted image file, while the H.264 benchmark simply terminated pre-maturely with segmentation fault. This is because ASAC marks all pointers and memory addresses as non-approximable, hence an `Aggressive` error injection into memory addresses naturally resulted in crashes. Next, we tried to inject errors only into variables that ASAC has marked as non-approximable and are not memory addresses. Figure 5-7 shows the encode and decode outputs of JPEG. It clearly shows that ASAC is able to correctly mark not only approximable data, but also non-approximable data. For H.264, even a `Mild` error injection into non-pointer variables led to the application crashing.


(a) Original


(b) Encoding - Mild Approx.


(c) Decoding - Mild Approx.


(d) Encoding - Aggressive Approx.

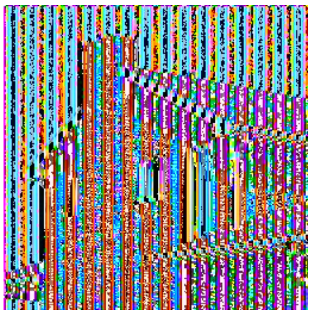
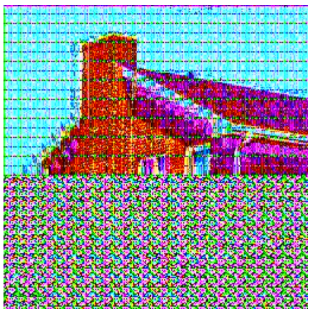

(e) Decoding - Aggressive Approx.

Figure 5-6: JPEG benchmark with various levels of approximations separately in *Encode* and *Decode* stages. Image (a) is the original image. Images (b) and (c) are result of introducing mild approximation (in 30% of the variables). Images (d) and (e) are result of introducing aggressive approximation (in all the variables that are approximable).


(a) Encoding - Mild Approx.

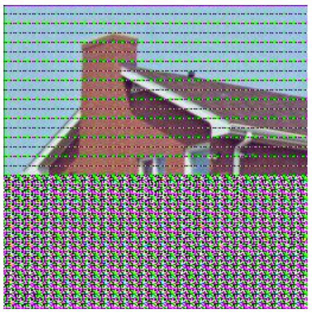

(b) Decoding - Mild Approx.

Figure 5-7: JPEG benchmark with errors in data that are marked as "Precise" by ASAC.

5.6 Chapter Summary

In this chapter, we present ASAC, a framework to automatically classify internal program data as approximable and non-approximable. We propose a novel sensitivity analysis that makes use of statistical sampling in performing a controlled perturbation based program testing. We are able to achieve 86% accuracy in identifying approximable data as compared to a manually annotated baseline. We also show that ASAC is scalable, and is able to analyze large applications such as JPEG and H.264.

Our experimental results show that using our annotations to approximate program data resulted in program outputs that are within the acceptable QoS thresholds. ASAC is easy to adapt in either a compilation or a software testing framework. In addition, it can be used to provide suggestive annotations for large-scale programs that are difficult to annotate manually. As a part of future work, ASAC can be extended to comprise more complex analysis and study sensitivity of program data across software versions. We expect ASAC to be a key contribution as the first automatic framework to classify program data in the field of approximate computing, which will grow as energy efficiency demands become more prevalent.

Chapter 6

Compilation Framework for Approximate Computing

6.1 Overview

Approximate or inexact computing trades-off accuracy of applications to save memory or computational resources, and is especially attractive for power constrained embedded devices. Low power approximate adders produce inexact sum of the inputs and introduce approximation in arithmetic operations [52–54, 56, 86]. Approximate memories operate at lower voltages saving substantial energy at the risk of possibly compromising the accuracy of the data stored [87–89]. Such approximate circuits and devices require collaboration from the software stack. Certain Instruction Set Architecture (ISA) extensions enable approximate hardware to switch between accurate and approximate computation during runtime [90]. However, identifying instructions or data of a program where approximation could be allowed without a loss of in the overall quality of service (QoS) of the application is a difficult task. The state-of-the-art methods rely on expressed type-classifiers and pragmas to indicate critical and approximable constructs in the source code [91, 92], thereby transferring the responsibility to the programmer. A few recent works involving profiling and iterative testing of applications are, unfortunately, computationally intensive [93, 94].

In this chapter, we introduce *PAC - Program Analysis for Approximation aware Compilation*, a compiler framework to analyse and identify appropriate parts of a program where approximation may be applied with an acceptable loss in QoS. PAC computes a *degree of accuracy* (DoA) for each *program component* that is required to attain the QoS of the program. A program component is a variable, operation, instruction, function call, basic block or a procedure. PAC outperforms the current state-of-the-art techniques in the following ways -

- PAC is a purely *static* framework and thus, does not require computationally expensive runs to extract approximable program constructs. For instance, ASAC [94], Chisel [93] and ApproxIt [95] are techniques that explore a search-space by running the application repetitively to achieve an acceptable approximation regime. A program analyzed with PAC, can significantly reduce the search space and overheads of such dynamic testing methods. Moreover, being a compile-time technique, PAC is easy to use and easily complements other techniques.

- PAC takes an application and its QoS requirements (translated as DoA of the outputs) as input and automatically computes the DoA of program components of the application. These in turn can be used to automatically (or semi-automatically, keeping the programmer in the loop) generate type-classifiers like `@approx`, `@endorse` [91] and annotations [92] to facilitate approximation.

- PAC assigns a quantifiable measure of accuracy i.e. the DoA metric, for each program component that indicate their contribution to the overall QoS of the application. Such non-binary classification of data and instructions is more useful than the state-of-the-art binary (approximable or accurate) classification. For example, on reconfigurable devices, it is more useful to know *how many bits* of data or an operation can be approximated, than to merely know that it can be approximated.

Novel Contributions in PAC

The key idea is to propagate the expected accuracy of the output (QoS) to the entire program. Based on the definition and usage of variables and interdependence of the instructions, it computes the DoA for all program components. We introduce the *Component Influence Graph (CIG)* that captures the relations between the various components. Using the CIG and dataflow equations, the analysis calculates the DoA. Lee et al. [96] claimed that program variables affecting the control flow such as conditional statements, must always be accurate and approximation can only be introduced in multimedia data. However, PAC comprises a novel program transformation technique which allows conditional statements to be approximated.

Target Architectures

PAC is useful to architectures that support approximate computing. Kahng et al. [52] proposes an accuracy-configurable adder which can adaptively adjust during runtime based on the required accuracy. As PAC provides the required accuracy of all addition or arithmetic operations, it is possible to exploit the adaptive nature of such adders. Thus, instead of only allowing an addition to be approximated, PAC can provide such adders with the DoA of that particular addition. Memories that can control power supply at the bit level are widely explored [57–59]. For such memories, it is imperative to know how many bits of a variable is approximable. We believe that such information can be derived from PAC's DoA.

Evaluation Summary

We compared PAC with the state-of-the-art techniques proposed in [91, 94, 96–99]. Compared to current state-of-the-art techniques of approximate computing, PAC achieves a high accuracy of 92% (compared to [91]) and 85% (compared to [94]). In addition, runtime of PAC is $\approx 10^3\times$ less than ASAC [94]. When compared to [98], PAC achieves an accuracy of 91% on average. A detailed presentation on evaluation results are in Section 6.3.

6.2 PAC Framework

The key idea of PAC is to propagate the accuracy (given as user defined QoS margins) required by the output to all the program components. Formally, we define DoA it as follows -

Definition 6.2.1. *Degree of Accuracy-* For a variable v, DoA(v) is the accuracy required to maintain the QoS margins of the application. If DoA(v) $= 1$, it indicates that all the bits belonging to variable v must be correct in order to remain within the given QoS margins. Conversely, DoA(v) $= 0$ means that none of the bits of variable v matters to the program output, such variables can be removed by dead code elimination. In practice, the accuracy is usually $0 <$DoA(v) ≤ 1.

The QoS of an application is required to be translated to the DoA of output variable(s). PAC assumes that output variable(s) and their DoA is available beforehand and the translation is done apriori. The DoAs are propagated using *influence relations* among the variables. The influence relations connect variables via the *def-use chains* (du-chain) such that an error in one variable impacts the other. A du-chain consists of the definition of a variable and all its uses. Formally we describe an *Influence Relation* as follows:

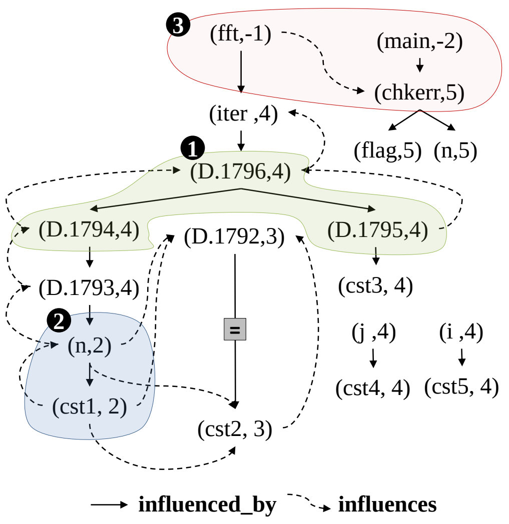
fft1

```
<bb 2>:
  if (n <= 1)
    goto <bb 3>;
  else
    goto <bb 4>;
<bb 3>:
  D.1792 = 999;
  goto <bb 35>;
<bb 4>:
  D.1793 = (double) n;
  D.1794 = log (D.1793);
  D.1795 = log (2.0e+0);
  D.1796 = D.1794 / D.1795;
  iter = (int) D.1796;
  j = 1;
  i = 0;
  return iter;
```

main

```
<bb 5>:
  flag = 0;
  chkerr = fft1 (n, flag);
```

(a) Example Code



(b) Corresponding CIG

Figure 6-1: A kernel and corresponding CIG from fft.c (MiBench)

Definition 6.2.2. *Influence Relation*- Two variables u and v share an *influence relation* iff an error in u may result an error in v, or vice versa. We define two types of influence relation - *influenced_by* and *influences*. Variable u is influenced_by v if an error in v introduces error in u. We also say variable v influences u.

6.2.1 Component Influence Graph (CIG)

The component influence graph captures the influence relations of all the program variables. Each node in CIG is a tuple consisting of a variable and a basic block identifier. There are two types of edges in a CIG representing the two types of influence relations mentioned above.

Figure 6-1 illustrates a sample kernel of `FFT` benchmark from MiBench [85] and its corresponding CIG. An 'influenced_by' edge in CIG, connecting two nodes, also contains information about the operator that relates the variables of the nodes. For example, in Figure 6-1b the edge [(D.1792,3),(cst2,3)] denotes the

Algorithm 6.1 CIG Construction

```
Require: Source code of program
Ensure: dug_head, a pointer to the first node of DUG
 1: for all function in CFG do
 2:     cfun←current function
 3:     for all basic block in cfun do
 4:         bb← current basic block; stmt$_i$ ← assignment statement i in bb
 5:         if stmt$_i$ is assignment then
 6:             lhs=assigned variable; rhs1=first operand; rhs2=second operand;
 7:             (lhs,bb)=CREATE_NODE(lhs);
 8:             if rhs1 is a constant then
 9:                 (lhs,bb)→child=FIND_NODE(cst,bb);
10:             else
11:                 (lhs,bb)→left=(rhs1,bb);(lhs,bb)→right=(rhs2,bb);
12:                 (rhs1,bb)→parent=(lhs,bb); (rhs2,bb)→parent=(lhs,bb);
13:             end if
14:         else if stmt$_i$ is conditional statement then
15:             lhs=first operand; rhs=second operand;
16:             for each edge E$_j$ from bb do
17:                 bb$_j$ = E$_j$ → dest → bb;
18:                 (lhs,bb)→parent= ∀var ∈ bb$_j$;
19:                 (rhs,bb)→parent= ∀var ∈ bb$_j$;
20:             end for
21:         else if stmt$_i$is call statement then
22:             call_return =first operand; callee=second operand(func_name);
23:             return=return value of callee;
24:             (return,callee)→parent=(call_return,cfun);
25:         end if
26:     end for
27: end for
```

operator '='. Any node together with its immediate child (or children) can be mapped to an instruction (eg. group 1 in Figure 6-1b). Moreover, a sub-graph of all nodes with the same basic block identifier captures the influence relation for the entire basic block (eg. group 2 in Figure 6-1b). Special nodes that are tuples consisting of a function name and a negative integer each represents a procedure. Such nodes are connected to the rest of the nodes in CIG via the return value and the parameter variables (eg. group 3 in Figure 6-1b). A node of CIG together with all its outgoing edges is equivalent to the variables' du-chains. Thus, the CIG is the union of du-chains of all variables of a program.

The CIG is constructed after the control-flow graph during compilation. The detailed explanation of CIG construction is given in Algorithm 6.1. For each assignment statement (line 5), a CIG node is created for the lhs of the assignment (line 7). It is assumed that assignment of a variable v in a basic block bb is an unique pair (v, bb), as in SSA form. Afterwards, n 'influenced_by' edges are created from this node to the existing nodes in the CIG representing n operands of the assignment statement (lines 8,10); $1 \leq n \leq 2$ due to SSA form. In addition, from the n operand nodes, one 'influences' edge is created, pointing back to the lhs node. For conditional statements (line 14), 'influences' edges are created from both operands of the condition to all the variables of the target basic blocks (lines 16-19). Note, that no 'influenced_by' edges are created as any event of error in the condition operands would result only in erroneous branching and not errors in other variables of the target basic blocks. Similarly, for function calls, edges are created between the parameters passed and the return value of the function (lines 22-24). These nodes are variables and function identifier (in negative integers) pairs instead of basic block (line 24).

6.2.2 Accuracy Equations

The CIG, together with a set of accuracy equations, is used to generate the DoA(v) for each variable v. As discussed before, PAC expects the user to provide the DoA of the output variable(s) using annotation. The accuracy equations are then applied to all other variables having an influence relation with the output variables. For example, if DoA(O) is the accuracy of a variable O, then the DoA of any variable V in an influence relation with O is derived from DoA(O) and other variables influencing O. From the runtime perspective, errors occurring in variables are non-trivially dependent events. CIG of a program can easily characterize this phenomenon in the following way.

Definition 6.2.3. *Error Independence* - Two variables u and v share an *Error Independence* relation if (a) u does not appear in the sub-graph $G \in$ CIG, where G consists of v with all its children, and (b) v does not appear in the sub-graph $H \in$ CIG, where H consists of u and all its children. Such variables are said to be error independent.

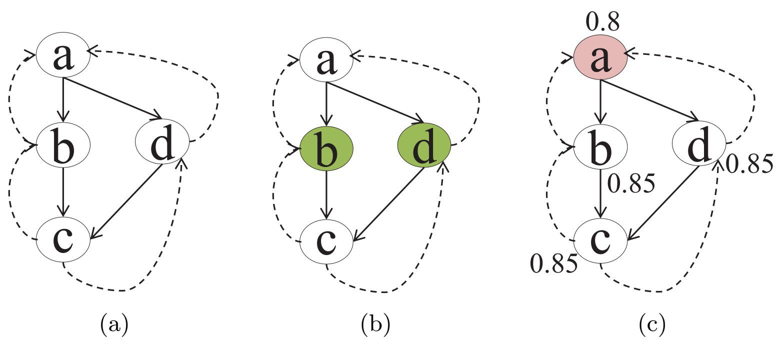


Figure 6-2: An example of a CIG showing the 'Error Independence' relations.

For instance, in Figure 6-2(a), an error in b would not result in an error in d. However, an error in c would likely result in an error in b. Note that in the CIG, a child node's basic block occurs earlier than its parent in program order. So, Figure 6-2(b) shows that variables b and d are error independent. However, a and b (or b and c) are not, as they appear in each others' sub-graph in the CIG. We broadly classify instructions into three forms - copy statements, operation statements and branching statements, and define the accuracy equations for each of them.

1. Copy Statements of form $A = B$.

For simple copy statements of this form, the DoAs are calculated as -

$$\text{DoA}(A) = \text{DoA}(B) \tag{6.1}$$

The propagation of DoAs is a backward dataflow analysis (Section 6.2.3). So, DoA(B) is equal to the value of DoA(A), which is already known. Thereafter, for copy statements where B is the left hand side expression, the value of DoA(B) will be used to derive the DoA of the variable on the right hand side. In such copy statements, it is said that A has a direct error dependence on B.

2. Operation Statements of form $A = B\ op\ C$.

These are standard assignment statements where the error dependencies between B and C are used to derive DoA(B) and DoA(C).

$$\mathrm{DoA}(A) = \mathrm{DoA}(B|C)\mathrm{DoA}(C) + \mathrm{DoA}(C|B)\mathrm{DoA}(B) \tag{6.2}$$

where DoA($B|C$) is the DoA of B given a DoA of C. *CASE I: B and C are error* independent. In other words, an error in B would not result in an error in C or vice versa. Then, DoA($B|C$) = DoA(B) and DoA($C|B$) = DoA(C). However, depending on the type of operator, the effect of the error is different. Assuming the source of error is unbiased, both B and C are equally likely to incur error. So,

$$\mathrm{DoA}(B) = \mathrm{DoA}(C) = \begin{cases} \sqrt{\mathrm{DoA}(A)}, \text{when op} \in \{+, -\} \\ \sqrt{\mathrm{DoA}(A)/2}, \text{when op} \in \{*, /\} \end{cases} \tag{6.3}$$

Taking the square root prevents the DoA of the operands from diminishing in a long *du* chain. Moreover, it preserves the notion of error accumulation. In other words, errors in both B and C, would result in higher deviation of A. Conversely, a given DoA of A (lhs), would imply that the DoA of the operands (rhs) must be higher. The square root also achieves normalization, i.e., $0 \leq DoA(A) \leq 1$ always.

CASE II: B and C are not error independent. In this case, B or C must exist in each other's subtree in the CIG. Therefore, there must exist a chain of influence relations between B and C, such that $B \rightarrow X_i \rightarrow C$, where $0 \leq i \leq n$ for n nodes in the subtree. Also, because the DoAs are propagated backward, the event of error in a variable occurring in a statement is not dependent on an error event occurring *later* in program order. If B is defined at a program point earlier than C, then

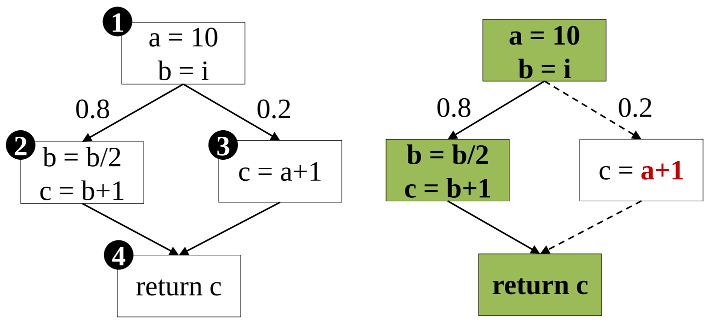


Figure 6-3: DoA propagation for branching statements in a CFG.

$$\text{DoA}(C|B) = \text{DoA}(B, X_i)\text{DoA}(X_i|X_{i+1})...\text{DoA}(X_{i+n-1}|C) \tag{6.4}$$

3. Branching Statements

Branching is a control flow decision. Every basic block containing a conditional or branching statement, has two successor basic blocks in the CFG. One of them is taken during execution, while the instructions in the path not taken remain unexecuted. For example, in Figure 6-3, if the path taken is $1 \to 2 \to 4$ then the instruction `c=a+1;` is never executed. This implies that the instruction `a=10;` or the variable a in basic block 1 can be safely approximated. The branching probability of the edges from a basic block to its successors depicts the likelihood of the path being taken during runtime. This information is easily obtained from the compiler (for example, using the *-fguess-branch-prob* flag for GCC). The edge with lesser probability (for example, $1 \to 3$), leads to the basic block containing instructions that are less likely to be executed and thus, are more amenable to approximation. Therefore, for all variables whose reachability is found to be in either of the successor basic blocks and not in both, the DoA is lowered using the branch probabilities.

Algorithm 6.2 elaborates on the method we apply, to handle branching statements. First, the probabilities of branch edges are obtained (lines 3-4). Reacha-

Algorithm 6.2 Branching Statements' Accuracy Propagation

```
Require: List of basic blocks with edge probabilities
Ensure: Updated DoAs of affected variables
 1: for all branching statements do
 2:     bb←current basic block
 3:     dest_major← target branch with higher edge probability $e$
 4:     dest_minor← other branch with probability $1 - e$
 5:     for ∀ variables v∈ bb do
 6:         if USED(v)∈ dest_minor∨ USED(v)∉ dest_major then
 7:             stmt← GET_USE(v);
 8:             lhs = GET_LHS(stmt);
 9:             DoA(lhs)=DoA(lhs)*$(1 - e)$;
10:         end if
11:     end for
12: end for
```

bility of the variables is calculated by applying a standard reachability analysis. For every variable that reaches only one of the destination of the current branching (line 6), the branch probabilities are multiplied with the DoA already obtained using the accuracy equations mentioned earlier (line 9). Multiplication results in lowering of the DoAs of the variables according to whether the branch is taken.

6.2.3 Analysis & Propagation

DoA propagation is modelled as a program analysis problem. The analysis is solved in an iterative manner where every iteration has two phases. Phase 1 is a backward flow analysis that considers the variables belonging to all statements except conditional statements. Phase 2 is a forward flow analysis for variables involved in conditional statements.

The flow of the analysis is represented in Equations 5 and 6. Phase 1 uses Equations 6.1 and 6.3 (Section 6.2.2). Phase 2 comprises of the technique described as 'form 3' in Section 6.2.2. The iterations of the analysis partially fill Equation 6.4 with the DoAs that are calculated in previous iterations. The analysis attains a maximum fixed point (MFP) solution when the assigned DoAs do not change between successive iterations. This safe termination is ensured by keep track of variables that have obtained a value other than `Init` in the lattice

Algorithm 6.3 PAC dataflow Analysis (Partial)

```
Require: Control Flow Graph
1: $BB_e \leftarrow$ Entry basic block;
2: equation[]$\leftarrow$set of unsolved accuracy equations;
3: for all basic blocks bb $\in$ CFG do
4:     if bb $\in BB_e$ then
5:         dfin(bb) = Init;
6:     else
7:         dfin(bb) = $\top$;
8:         worklist = variables v$\in$ bb;
9:     end if
10: end for
11: for all basic blocks bb $\in$ CFG do
12:     for $\forall$variables v$\in$ bb & v$\in$worklist do
13:         if matches form 1 or 2(I) then
14:             Calculate DoA(v) using equation 1 or 4;
15:             worklist -= v;
16:             FILL(equation,v);
17:         else
18:             if !SOLVE(equation[$bb_v$]) then
19:                 equation[$bb_v$]$\leftarrow$ partial equation 5;
20:             end if
21:         end if
22:     end for
23: end for
```

of the analysis. `Init` denotes the initial state of the variables, which is *Critical*, i.e., a DoA of 1. The analysis results in lowering of the DoAs. Variables with a DoA of 0, i.e., dead variable, will not be consider further in the analysis.

$$\text{Out}(B) = \begin{cases} \texttt{Init}, \text{ for } B = \texttt{Exit} \\ \prod_{P \in \text{Succ}(B)} F_{1,2}(\text{OUT}(P)), \ \forall \text{var} \in B \wedge \text{var} \notin \text{COND} \end{cases} \tag{6.5}$$

$$\text{IN}(B) = \begin{cases} \texttt{Init}, \text{ for } B = \texttt{Entry} \\ \prod_{P \in \text{Pred}(B)} F_3(\text{IN}(P)), \ \forall \text{var} \in B \wedge \text{COND} \end{cases} \tag{6.6}$$

Algorithm 6.3 elaborates on the steps of the analysis as implemented in our framework. It follows the generic steps of a worklist based dataflow analysis with slight modifications. At the outset, the basic block dataflow information is initialized with the $\top$ of the lattice (lines 4-7), i.e. all variables are assumed critical. All variables are added to a *worklist* as they have not been assigned any

DoA value at this step (line 8). Afterwards, traversing through the control flow graph (lines 11,12), each statement is matched against the forms discussed in Section 6.2.2 (line 13,17). If the corresponding accuracy equation can be solved, the variables are assigned the resulting DoA (line 14,18), and are removed from the worklist (line 15). Otherwise, from the program components found, equation 5 is partially filled (lines 16,19). When all the equation elements are available, the equation is solved and the variables are assigned with the DoA (line 18).

6.2.4 Approximating Comparisons

Comparison expressions are central to branching and loop termination and thus are considered as *critical* instructions [96]. From the perspective of approximate computing, variables in the comparison expression are often considered non-approximable.

In our framework, we propose a simple program transformation that allows comparisons too to be approximated without any change in the program behaviour. Apart from the known benefits of approximation, allowing inexact comparison allows for the use of approximate comparators [60], thereby potentially resulting in a better power-performance.

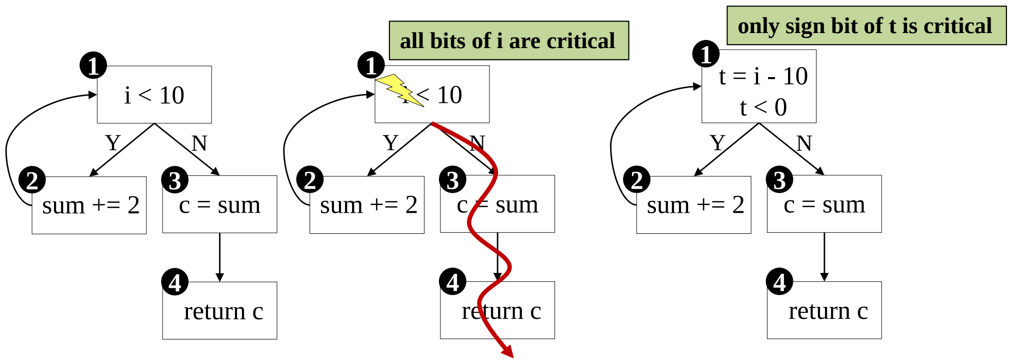


Figure 6-4: Transformation for approximate comparison.

Figure 6-4 shows a frequently occurring pattern in any program. Typically, a loop induction variable or other branching conditions comprise of relational operators, specifically $<, \leq, >, \geq$. For these operators, we propose a transformation technique that allows the particular comparison statement to be safely approximated.

As a penalty, for every comparison statement, a temporary variable is introduced. Such temporary variables do not pose a large overhead due to the SSA form. Formally we define the premise of the transformation as follows -

Definition 6.2.4. For any comparison statement of the form if(A op B){}, where op $\in \{<, \leq, >, \geq\}$, there exists a pair of statement:

$$\text{temp} = A - B;$$
$$\text{if (temp op 0)}\{\}$$

which is semantically equivalent and can replace the original comparison without any change of the program behaviour.

Using the above, all the comparison statements are replaced with the appropriate pair of new statements. For example, in Figure 6-4, the statement `i<10;`, where `i` is a loop induction variable causing `i<10;` to be executed only in full precision. However, with the transformation `t = i - 10; t < 0;`, only the sign-bit of `t` remains critical and rest of the bits can tolerate errors without any change of program's behaviour. Approximation is thus introduced in the control flow statement with a penalty of 1 additional computation.

6.3 Evaluation

We evaluated PAC in three ways. First, we compared it with the state-of-the-art methods for approximate computing. Next, we compared PAC with compile-time techniques designed for reliability against soft-errors. Such methods categorize program variables into critical and non-critical with the intention

EnerJ [91]					
Bench-marks	**True Posi-tive**	**False Posi-tive**	**True Nega-tive**	**False Nega-tive**	**Accu-racy**
SMM	4	0	4	0	**1**
MonteCarlo	2	0	4	0	**1**
LU	8	0	12	2	**0.9**
FFT	9	0	15	7	**0.77**
SOR	6	0	7	1	**0.92**
			Average		**0.92**

Table 6.1: Comparison with EnerJ to show PAC's accuracy.

of 'hardening' critical data against soft-errors. We will show with our experiments that data identified as non-critical by these techniques are not always approximable. Finally, we evaluated PAC by injecting errors in the applications and thereby measuring the resulting QoS and overhead.

6.3.1 Comparison with approximation techniques

We compared PAC with two state-of-the-art methods, namely EnerJ and ASAC. EnerJ [91] uses type-classifiers such as `@approx` to annotate program variables meant for approximation. ASAC [94] ranks variables in terms of the output's sensitivity towards them, and allows approximation for less sensitive program variables. Table 6.1 and Table 6.2 show how PAC performs as compared to both the techniques. Though PAC can produce DoA for each program component, for the comparison we only considered program variables.

Furthermore, in order to perform the comparison, we assumed that variables with DoA less than 0.5 are approximable and rest are not. This is a conservative assumption and can be fine-tuned according to the demand of the application. We present the standard metrics of true positive (PAC classifies approximate data correctly), false positive (PAC mistakenly classifies critical data as approximable), true negative (PAC correctly identifies critical variables) and false

ASAC [94]					
Benchmarks	**True Positive**	**False Positive**	**True Negative**	**False Negative**	**Accuracy**
SMM	3	0	3	2	**0.75**
MonteCarlo	2	0	2	2	**0.66**
LU	8	0	9	5	**0.77**
FFT	9	0	12	10	**0.68**
SOR	7	0	5	2	**0.85**
			Average		**0.74**

Table 6.2: Comparison with ASAC to show PAC's accuracy.

Benchmarks	**GCC -O3 (seconds)**	**PAC (seconds)**	**ASAC (seconds)**
SOR	0.147	0.168	1345.009
MonteCarlo	0.105	0.113	1929.476
SMM	0.104	0.127	1138.159
LU	0.164	0.186	1831.876
FFT_scimark2	0.135	0.219	1062.417
FFT_MiBench	0.56	0.83	53.069
adpcm	0.342	0.378	222.272
susan	1.2	1.45	30.014
JPEG	6.973	6.601	13.642 (only DCT kernel)

Table 6.3: Runtime of PAC as compared to standard `-O3` optimization flag in GCC and ASAC

negative (PAC marks a variable as critical where it can be approximated). PAC is a static method and hence it is conservative. In particular, false negatives are to be expected. However, false positives would be unsafe approximation of program variables that might lead to unacceptable QoS or unexpected termination of applications.

In our experiments, we used the Scimark2 [84] benchmarks, as `@approx` annotations are available only for this suite. We applied ASAC to the same benchmarks and present the results in Table 6.1 and Table 6.2. PAC achieved an accuracy of 92% when compared to EnerJ, and 74% when compared to ASAC, on average. ASAC is based on profiling of application and thus, has runtime information to analyse the sensitivity of the variables. However, for simple applications like `SOR` PAC is able to have an accuracy that is 85% that of ASAC. The key reason is that it has a simple CFG and the accuracy equations are mostly of forms 1 and 2(I).

The main advantage of PAC over ASAC is the runtime overhead of the analysis. Table 6.3 shows the different runtime of both methods. PAC is a compiler analysis pass, therefore, we also compare PAC's runtime with the compilation time of `-O3` of GCC. The runtime of ASAC depends on the total number of variables and the dynamic instruction count of the application. We tested comparatively small programs to measure the runtime of PAC and ASAC. Table 6.3 shows that ASAC is 3 orders of magnitude slower than PAC. As application becomes larger, the difference in runtime also increases. The standard `-O3` optimization in GCC, on the other hand, is 3% faster than PAC on an average (Table 6.3). In other words, PAC has minimum impact on compile time.

6.3.2 Comparison with software reliability techniques

To compare with state-of-the-art techniques for ensuring program level reliability, we use three applications, `adpcm`, `susan` and `jpeg` from MiBench [85] and three applications `464.h264ref`, `433.milc` and `482.sphinx3` from SPEC2006 benchmark suites (Table 6.4).

A. Bitwidth Analysis [Ste00]

Bitwidth analysis determines and reduces the number of bits required for program variables [97]. This is often used to minimize the memory budget in silicon compilation. Intuitively, if the bitwidth analyzed by these techniques is shorter than the width of the data type declared by programmer, the extra bits can be approximated safely. With this assumption, we compare PAC's analysis with a

Application	**Lines Of Code**	**Description**	**Error Metric**
adpcm	283	Adaptive differential pulse code modulation (variation of PCM)	SQNR(Signal to Quantization Noise Ratio)
susan	888	Image recognition (edge/corner detection)	Mean Pixel Difference
jpeg	10176	Image compression	SNR (Signal to Noise Ratio)
464.h264ref	18696	Video Compression	PSNR (Peak Signal to Noise Ratio)
433.milc	5401	Quantum Chromodynamics	Error per site (provided with benchmark)
482.sphinx3	7721	Speech recognition	Word error rate

Table 6.4: Description of the applications

state-of-the-art bidwidth analysis [97]. Table 6.5 shows the quantitative comparison. It illustrates 3 cases: CASE I where both PAC and bitwidth analysis marks a variable as approximable; CASE II where PAC characterizes a variable with varying bitwidth as approximable; and CASE III where PAC characterizes a variable with varying bitwidth as non-approximable. Lastly, Table 6.5 also shows the coverage of the two methods as a ratio of number of variables analysed by PAC to bitwidth analysis. PAC identifies 3× more variables, on average, that can be approximated, as the premise of approximate computing is to introduce as much as approximation possible to reduce energy consumption. This is due to the fact that PAC considers the interdependence of variables and also transforms conditional statements to more approximable equivalents. In addition, PAC has a better coverage of code, 40% more than bitwidth analysis. This can be attributed to PAC's interprocedural influence relations.

B. Program Dependency Graph (PDG) Scheme [Cong11]

The second scheme we compared PAC with is based on a weighted program dependence graph [98]. The authors proposed a technique to identify critical

Application	CASE I	CASE II	CASE III	Coverage
adpcm	28	68	174	1.13
susan	147	435	3064	1.254
jpeg	134	531	1552	1.54
464.h264ref	165	231	46082	1.82
433.milc	152	452	35250	1.1
482.sphinx3	45	276	7348	1.65
Average	111.83	332.16	15578.3	1.41

Table 6.5: Comparison with bitwidth analysis with no. of variables for all cases (above paragraph) and ratio of code coverage.

Application	True Positive	False Positive	True Negative	False Negative	Accuracy
adpcm	35	6	198	31	0.86
susan	498	31	3034	83	0.96
jpeg	620	45	1470	82	0.94
464.h264ref	312	84	44447	1635	0.96
433.milc	515	89	33268	1982	0.94
482.sphinx3	279	42	6027	1321	0.82
Average	376.5	49.5	14740.66	855.66	0.91

Table 6.6: Comparison with PDG based scheme with no. of matches identified by both methods and PAC's accuracy.

data based on the number of references to it in the whole program with the aim of protecting these data against soft errors. The technique classifies the data as *likely critical* (LC) or *likely not critical* (LNC). Table 6.6 shows the match between LC and LNC data with approximable and non-approximable data as characterized by PAC. The 'true positive' represents the number of variables with low (<0.5) DoA and also marked as LNC. Such variables can be safely approximated. 'False Positives' are variables that are marked as LC that, however, has a low DoA. This column suggests that approximability of program variables is not just the function of the total number of references to it. Later in Section 6.3.3, we will show that injecting errors into this class of variables also does not result in the loss of QoS. 'True negatives' are variables that both schemes agree on. Lastly, ‘false negatives’ are cases where PAC characterized the variable as non-approximable, but was marked as LNC. This shows that a variable which does not need extra protection from soft errors, may not tolerate errors aggressively due to deliberate approximation.

C. Multimedia Application Specific Data Partitioning [Lee06]

In this method, the authors suggested selective protection of data in multimedia applications [96]. Any variable affecting termination of the application is characterized as critical and multimedia data (input or output) is deemed non-critical. We compare with this scheme in terms of error percentage obtained by running the applications under a synthetic error injection framework described in Section 6.3.3. Figure 6-6 shows that the number of variables marked as approximable or non-critical by this scheme is 7% on average, which is much lesser than PAC’s 37.5% on average. Thus, we can conclude that PAC performs better in terms of identifying possible approximation in a program.

D. Instruction Vulnerability based characterization [Sha13]

The fourth technique is based on error masking and its effect on QoS of an application [99]. The scheme is based on the probability of masking of an error due to bitwise ‘AND’, shift or other similar operations. This technique, like the previous ones, suffers from poor coverage of source code and considers only specific cases. Figure 6-6 shows that it provides around 11% of coverage of a program. In applications where bitwise operators do not play a major role, this technique fails to identify possible approximations.

6.3.3 Impact of Errors

To evaluate the effectiveness of the data characterization, we present the quantitative QoS loss in terms of error percentage in Figure 6-5. We used a synthetic error injection framework, which injects a random bitflip into variables that are identified as approximable. During execution, the error injector, randomly selects one or more variables at a uniform interval and injects the bitflip. For each application, the error percentage is calculated based on the correct (provided) value of the metric mentioned in Table 6.4 and the output obtained upon error injection.

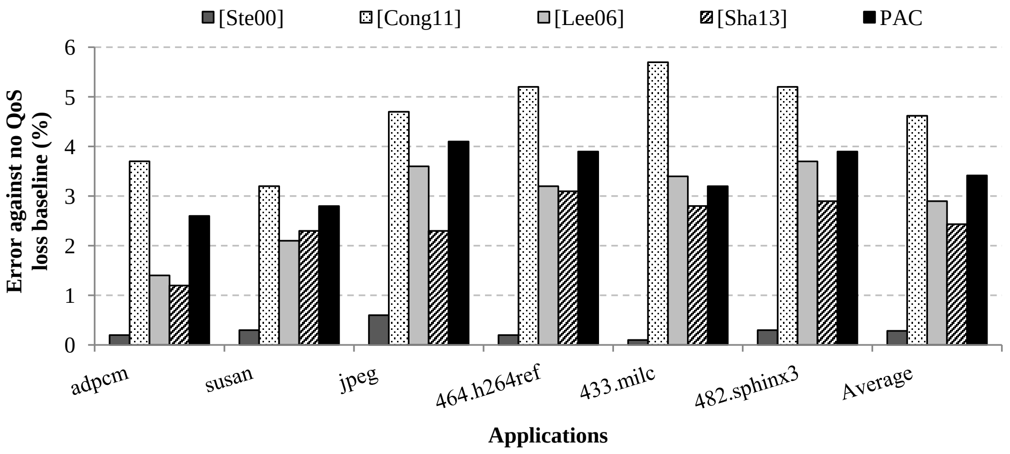


Figure 6-5: Error Percentage (error injected in approximable variables).

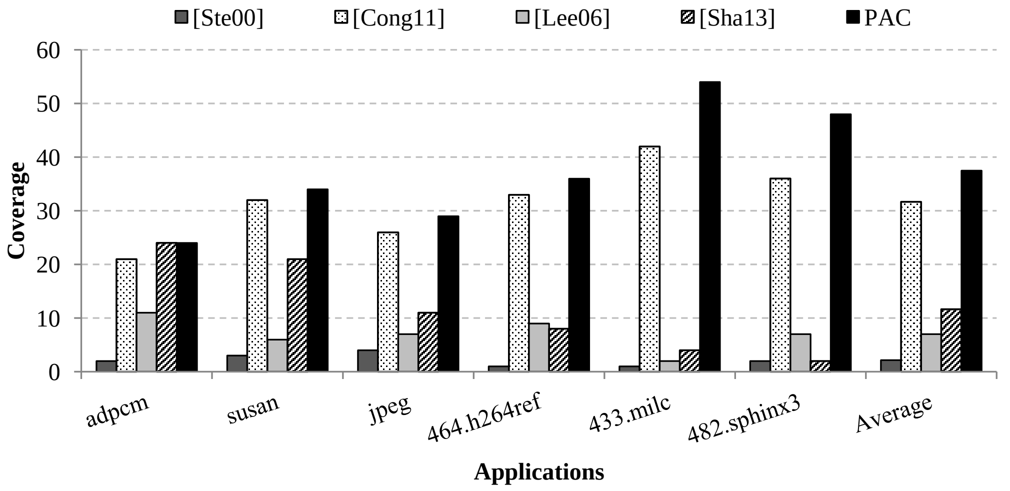


Figure 6-6: Impact of errors injection in approximable variables characterized by different methods.

Applications	**Total Conditional Statements**	**Transformed Conditional**	**Overhead (10^{-2})**
adpcm	157	143	3.2
susan	149	84	2.8
jpeg	1454	1239	1.14
464.h264ref	3128	2743	1.06
433.milc	626	581	0.86
482.sphinx3	1263	912	1.11
Average	1129.5	950.33	1.695

Table 6.7: Overhead of conditional transformation

On average, PAC accounts for 3.4% of QoS loss. Though, schemes Ste00, Lee06 and Sha13 perform better and show a QoS loss of 0.28, 2.9 and 2.4 % only, they do not provide a good coverage of approximation in the program. In other words, the total numbers of approximations allowed according to these schemes are much less than Scheme Cong11 and PAC. This phenomenon is presented in the graph of Figure 6-6. Poor coverage will leads to less opportunities to reduce energy or computational resources. So while these schemes are as scalable as PAC, they provide lower quality information.

6.3.4 Impact of Approximating Conditions

For each conditional statement that is transformed, one assignment statement is added to the code (Section 6.2.4). Table 6.7 present the number of transformed conditional statements and the overhead in terms of percentage of additional instructions over the total static instruction count of the application.

6.4 Chapter Summary

In this chapter, we present PAC, a program analysis for approximation aware compilation. PAC computes degrees of accuracy for each program component required to maintain the quality of service of an application. Other than having the user specifying the QoS requirement, PAC is a completely automatic static analysis. Compared to the manual annotation of EnerJ, PAC attains a 92% accuracy. When compared to ASAC, a compute intensive search procedure, PAC attains 74% accuracy in characterizing variables that can be approximated without any QoS loss. However, PAC is $10^3\times$ faster than ASAC. Compared to software reliability methods, PAC achieved better coverage while maintaining the QoS under error injected execution of the applications. In summary, PAC offers something unique to the state of the art. Of the current techniques that compute the same information as PAC, none can scale to the large program that PAC can handle. Compared to similarly scalable software techniques designed for other purposes that may possibly be used to derive DoA, PAC computes higher quality results. We believe that this makes PAC an attractive complementary analysis to enhance other approximation approaches.

Chapter 7

Conclusion

7.1 Thesis Summary

In this thesis, we explored four software solutions to enable an energy efficient memory hierarchy. Specifically, we assumed a resistive hybrid memory based memory hierarchy and the issues faced in their deployment as caches and main memory. Summarizing the contributions of this thesis -

- **Write Sensitivity** Towards mitigation of write sensitivity issues of resistive memories we proposed two solutions.

    1. Static Code Analysis - We proposed a code analysis where programs are analyzed at compile time and based on their memory access affinity, they are placed in the virtual memory area. During runtime, the virtual addresses influence their placement in the hybrid memory. Read intensive data are allocated to STT-RAM and write intensive data to SRAM. This technique reduces the energy consumption of L1 cache by up to 50% as compared to the state-of-the-art without noticeable degradation in performance.

2. Operating System Assisted Solution - For an architecture with hybrid main memory, we proposed a fine-grain write control mechanism to reduce write operations from the last level caches to the resistive memory partition in the main memory. In addition, we proposed a new page reclamation policy that evicts pages from DRAM partition based on the amount on dirty data in the page. This approach is able to reduce power consumption by up to 83% as compared to the state-of-the-art with a penalty of about 1% degradation in performance in terms of Instructions Per Cycle.

- **Error Susceptibility** To solve the error susceptibility of energy efficient memories, both resistive and DVS/DVFS based memories, we apply the concept of approximate computing and propose two solutions for it.

  1. Sensitivity Analysis - We proposed a dynamic testing scheme where the sensitivity of each variable is determined to quantify its contribution towards the Quality of Service of the application. A less sensitive variable is said to be approximable and as a consequence can be allocated to energy efficient memories. On the other hand, highly sensitive variables are critical and must be safely allocated to reliable SRAM/DRAM memories. Our framework identifies approximable variables in a program with an accuracy of 86% as compared to manual identification. Moreover, our solution is scalable and can be applied to larger programs where source codes are not available for manual classification.
  2. Accuracy-aware Static Analysis - The above mentioned scheme is accurate but demands high overhead in terms of running time and is computationally intensive. To mitigate these drawbacks, we proposed a static code analysis that characterizes program variables as approximable or critical. We also proposed a code transformation to enable

approximated conditional statements. Our framework, allows better approximation of programs and reduces the overheads of the dynamic testing based schemes. The analysis is able to identify approximations with an accuracy 74% compared to the state-of-the-art.

7.2 Future Research

We wish to achieve a complete energy efficient computing environment. We believe that, through cross-layer interaction, it is possible to utilize the abstractions exposed at the one level to better utilize the resources available at other levels. This thesis mainly concentrates on the memory subsystem of x86 architecture. One direction to extend this work is to observe and apply these techniques for other architectures such as Graphics Processing Units (GPU) or Network on Chips (NOC), where the data handling mechanisms are non-trivial.

Approximate computing is in nascent stage and a plethora of researches are possible in this field. Approximations for energy efficiency can be explored in all possible components of computer organisation and design. This thesis focuses of approximation for memories and independent of other aspects of underlying architecture. At the outset, in multi-core environment, approximation can be explored in task scheduling or memory bank partitioning. Similarly, approximation can be achieved at various levels of abstraction in the system stack.

Bibliography

[1] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 164–174. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993518

[2] L. Chua, "Memristor-the missing circuit element," *Circuit Theory, IEEE Transactions on*, vol. 18, no. 5, pp. 507–519, Sep 1971.

[3] 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*, Aug 2011, pp. 79 –84.

[4] 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*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 243–252. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228406

[5] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: a hybrid PRAM and DRAM main memory system," in *Proceedings of the 46th Annual Design*

Automation Conference, ser. DAC '09. New York, NY, USA: ACM, 2009, pp. 664–469. [Online]. Available: http://doi.acm.org/10.1145/1629911.1630086

[6] A. Hay, K. Strauss, T. Sherwood, G. H. Loh, and D. Burger, "Preventing PCM banks from seizing too much power," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44 '11. New York, NY, USA: ACM, 2011, pp. 186–195. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155642

[7] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 14–23.

[8] I. H. Doh, Y. J. Kim, J. S. Park, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Towards greener data centers with storage class memory: minimizing idle power waste through coarse-grain management in fine-grain scale," in *Proceedings of the 7th ACM international conference on Computing frontiers*, ser. CF '10. New York, NY, USA: ACM, 2010, pp. 309–318.

[9] 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*, ser. CODES+ISSS '11. New York, NY, USA: ACM, 2011, pp. 325–334.

[10] 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*, ser. ISLPED '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 73–78.

[11] 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*, Mar 2010, pp. 136 –141.

[12] J. Li, C. Xue, and Y. Xu, "Stt-ram based energy-efficiency hybrid cache for cmps," in *VLSI and System-on-Chip (VLSI-SoC), 2011 IEEE/IFIP 19th International Conference on*, Oct 2011, pp. 31–36.

[13] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 24–33. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555760

[14] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient stt-ram caches," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 50–61. [Online]. Available: http://dl.acm.org/citation.cfm?id=2014698.2014895

[15] 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*, ser. ISLPED '12. New York, NY, USA: ACM, 2012, pp. 351–356.

[16] 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*, ser. ICCAD '09. New York, NY, USA: ACM, 2009, pp. 264–268.

[17] 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*, ser. LCTES '12. New York, NY, USA: ACM, 2012, pp. 109–118.

[18] 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*, Jan 2011, pp. 129 –134.

[19] 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*, ser. ISLPED '12. New York, NY, USA: ACM, 2012, pp. 237–242. [Online]. Available: http://doi.acm.org/10.1145/2333660.2333717

[20] J. Wang, Y. Tim, W.-F. Wong, Z.-L. Ong, Z. Sun, and H. Li, "A coherent hybrid sram and stt-ram l1 cache architecture for shared memory multicores," in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, Jan 2014, pp. 610–615.

[21] Q. Li, M. Zhao, C. J. Xue, and Y. He, "Compiler-assisted preferred caching for embedded systems with stt-ram based hybrid cache," *SIGPLAN Not.*, vol. 47, no. 5, pp. 109–118, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2345141.2248434

[22] M. Qureshi, M. Franceschini, and L. Lastras-Montano, "Improving read performance of phase change memories via write cancellation and write pausing," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, jan. 2010, pp. 1 –11.

[23] X. Dong, N. P. Jouppi, and Y. Xie, "PCRAMsim: system-level performance, energy, and area modeling for phase-change ram," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, ser. ICCAD '09. New York, NY, USA: ACM, 2009, pp. 269–275. [Online]. Available: http://doi.acm.org/10.1145/1687399.1687449

[24] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 2011, pp. 454–464.

[25] G. Loh and M. D. Hill, "Supporting very large dram caches with compound-access scheduling and missmap," *Micro, IEEE*, vol. 32, no. 3, pp. 70–78, 2012.

[26] W. Zhang and T. Li, "Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures," in *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, sept. 2009, pp. 101 –112.

[27] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 85–95. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995911

[28] Y. Park, S. K. Park, and K. H. Park, "Linux kernel support to exploit phase change memory," in *Proceedings of the Linux Symposium*, 2010, pp. 217–224.

[29] D.-J. Shin, S. K. Park, S. M. Kim, and K. H. Park, "Adaptive page grouping for energy efficiency in hybrid PRAM-DRAM main memory," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*,

ser. RACS '12. New York, NY, USA: ACM, 2012, pp. 395–402. [Online]. Available: http://doi.acm.org/10.1145/2401603.2401689

[30] 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*, ser. MICRO-44 '11. New York, NY, USA: ACM, 2011, pp. 329–338.

[31] J. Li, L. Shi, Q. Li, C. J. Xue, Y. Chen, Y. Xu, and W. Wang, "Low-energy volatile stt-ram cache design using cache-coherence-enabled adaptive refresh," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 19, no. 1, pp. 5:1–5:23, Dec. 2013. [Online]. Available: http://doi.acm.org/10.1145/2534393

[32] B. Del Bel, J. Kim, C. H. Kim, and S. S. Sapatnekar, "Improving stt-mram density through multibit error correction," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '14. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014, pp. 182:1–182:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616606.2616830

[33] H. Naeimi, C. Augustine, A. Raychowdhury, S.-L. Lu, and J. Tschanz, "Sttram scaling and retention failure," *Intel Technology Journal*, vol. 17, no. 1, pp. 54–75, 2013.

[34] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian, "Mitigating soft error failures for multimedia applications by selective data protection," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '06. New York, NY, USA: ACM, 2006, pp. 411–420. [Online]. Available: http://doi.acm.org/10.1145/1176760.1176810

[35] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, “Compiler-assisted soft error detection under performance and energy constraints in embedded systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 8, no. 4, pp. 27:1–27:30, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1550987.1550990

[36] A. Shrivastava, J. Lee, and R. Jeyapaul, “Cache vulnerability equations for protecting data in embedded processor caches from soft errors,” in *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES ’10. New York, NY, USA: ACM, 2010, pp. 143–152. [Online]. Available: http://doi.acm.org/10.1145/1755888.1755910

[37] W. Baek and T. M. Chilimbi, “Green: A framework for supporting energy-conscious programming using controlled approximation,” in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’10. New York, NY, USA: ACM, 2010, pp. 198–209. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806620

[38] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard, “Randomized accuracy-aware program transformations for efficient approximate computations,” in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’12. New York, NY, USA: ACM, 2012, pp. 441–454. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103710

[39] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Architecture support for disciplined approximate programming,” in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII.

New York, NY, USA: ACM, 2012, pp. 301–312. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151008

[40] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 213–224. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950391

[41] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540712

[42] M. Carbin and M. C. Rinard, "Automatically identifying critical input regions and code in applications," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831713

[43] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 124–134. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025133

[44] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing sequential programs with statistical accuracy tests," *ACM Trans. Embed. Comput.*

Syst., vol. 12, no. 2s, pp. 88:1–88:26, May 2013. [Online]. Available: http://doi.acm.org/10.1145/2465787.2465790

[45] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard, "Randomized accuracy-aware program transformations for efficient approximate computations," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 441–454. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103710

[46] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel, "Exploiting program-level masking and error propagation for constrained reliability optimization," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 17:1–17:9. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488755

[47] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540710

[48] V. Chippa, A. Raghunathan, K. Roy, and S. Chakradhar, "Dynamic effort scaling: Managing the quality-efficiency tradeoff," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 603–608. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024863

[49] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar, "Scalable effort hardware design: Exploiting

algorithmic resilience for energy efficiency," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 555–560. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837411

[50] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 113:1–113:9. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488873

[51] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: Imprecise adders for low-power approximate computing," in *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ser. ISLPED '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 409–414. [Online]. Available: http://dl.acm.org/citation.cfm?id=2016802.2016898

[52] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 820–825. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228509

[53] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 48–54. [Online]. Available: http://dl.acm.org/citation.cfm?id=2561828.2561838

[54] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, "Aslan: Synthesis of approximate sequential circuits," in *Proceedings of the*

Conference on Design, Automation & Test in Europe, ser. DATE '14. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014, pp. 364:1–364:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616606.2617119

[55] Z. M. Kedem, V. J. Mooney, K. K. Muntimadugu, and K. V. Palem, "An approach to energy-error tradeoffs in approximate ripple carry adders," in *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ser. ISLPED '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 211–216. [Online]. Available: http://dl.acm.org/citation.cfm?id=2016802.2016853

[56] Y. Kim, Y. Zhang, and P. Li, "An energy efficient approximate adder with carry skip for error resilient neuromorphic vlsi systems," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 130–137. [Online]. Available: http://dl.acm.org/citation.cfm?id=2561828.2561854

[57] J. Kong and S. W. Chung, "Exploiting narrow-width values for process variation-tolerant 3-d microprocessors," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1197–1206. [Online]. Available: http://doi.acm.org.libproxy1.nus.edu.sg/10.1145/2228360.2228581

[58] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 148–157. [Online]. Available: http://dl.acm.org.libproxy1.nus.edu.sg/citation.cfm?id=545215.545232

[59] M. M. Islam and P. Stenstrom, "Characterization and exploitation of narrow-width loads: The narrow-width cache approach," in *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '10. New York, NY, USA: ACM, 2010, pp. 227–236. [Online]. Available: http://doi.acm.org.libproxy1.nus.edu.sg/10.1145/1878921.1878955

[60] M. Nesenbergs and V. O. Mowery, "Logic synthesis of some high-speed digital comparators," ser. Bell System Technical Journal'13.

[61] 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*, Feb 2009, pp. 239–249.

[62] "SPEC CPU2006," in *SPEC CPU2006*, ser. http://www.spec.org/cpu2006/.

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

[64] N. Rinetzky, G. Ramalingam, M. Sagiv, and E. Yahav, "On the complexity of partially-flow-sensitive alias analysis," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 3, pp. 13:1–13:28, May 2008. [Online]. Available: http://doi.acm.org.libproxy1.nus.edu.sg/10.1145/1353445.1353447

[65] U. P. Khedker, A. Sanyal, and A. Karkare, "Heap reference analysis using access graphs," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, Nov. 2007. [Online]. Available: http://doi.acm.org.libproxy1.nus.edu.sg/10.1145/1290520.1290521

[66] G. Novark, E. D. Berger, and B. G. Zorn, “Efficiently and precisely locating memory leaks and bloat,” *SIGPLAN Not.*, vol. 44, no. 6, pp. 397–407, Jun. 2009. [Online]. Available: http://doi.acm.org.libproxy1.nus.edu.sg/10.1145/1543135.1542521

[67] 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.*, vol. 47, no. 4, pp. 105–118, Mar. 2011. [Online]. Available: http://doi.acm.org.libproxy1.nus.edu.sg/10.1145/2248487.1950380

[68] 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.

[69] 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*, vol. 31, no. 7, pp. 994 –1007, Jul 2012.

[70] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, “Architecting efficient interconnects for large caches with cacti 6.0,” *Micro, IEEE*, vol. 28, no. 1, pp. 69–79, Jan.-Feb.

[71] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[72] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, “DRAMSim2: A cycle accurate memory system simulator,” *Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, jan.-june 2011.

[73] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: characterization and architectural implications,” in *Proceedings of*

the 17th international conference on Parallel architectures and compilation techniques, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[74] S. Lee, H. Bahn, and S. H. Noh, "Clock-dwf: A write-history-aware page replacement algorithm for hybrid pcm and dram memory architectures," *IEEE Transactions on Computers*, vol. 99, no. PrePrints, p. 1, 2013.

[75] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 199–212. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950390

[76] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, "Language and compiler support for auto-tuning variable-accuracy algorithms," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 85–96. [Online]. Available: http://dl.acm.org/citation.cfm?id=2190025.2190056

[77] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar, "Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 555–560. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837411

[78] J. Lee and A. Shrivastava, "Static analysis to mitigate soft errors in register files," in *Proceedings of the Conference on Design, Automation and Test in*

Europe, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 1367–1372. [Online]. Available: http://dl.acm.org/citation.cfm?id=1874620.1874949

[79] S. Palaniappan, B. Gyori, B. Liu, D. Hsu, and P. Thiagarajan, "Statistical model checking based calibration and analysis of bio-pathway models," in *Computational Methods in Systems Biology*, ser. Lecture Notes in Computer Science, A. Gupta and T. Henzinger, Eds. Springer Berlin Heidelberg, 2013, vol. 8130, pp. 120–134. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40708-6-10

[80] F. M. Quintao Pereira, R. E. Rodrigues, and V. H. Sperle Campos, "A fast and low-overhead technique to secure programs against integer overflows," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/CGO.2013.6494996

[81] M. D. McKay, R. J. Beckman, and W. J. Conover, "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code," *Technometrics*, vol. 42, no. 1, pp. 55–61, Feb. 2000. [Online]. Available: http://dx.doi.org/10.2307/1271432

[82] T. Naughton, W. Bland, G. Vallee, C. Engelmann, and S. L. Scott, "Fault injection framework for system resilience evaluation: Fake faults for finding future failures," in *Proceedings of the 2009 Workshop on Resiliency in High Performance*, ser. Resilience '09. New York, NY, USA: ACM, 2009, pp. 23–28. [Online]. Available: http://doi.acm.org/10.1145/1552526.1552530

[83] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," in *Proceedings of the*

33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 453–462. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254118

[84] R. Pozo and B. Miller, "Scimark 2.0," ser. www.math.nist.gov/scimark2/.

[85] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," ser. WWC '01, 2001.

[86] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: Imprecise adders for low-power approximate computing," in *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ser. ISLPED '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 409–414. [Online]. Available: http://dl.acm.org/citation.cfm?id=2016802.2016898

[87] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540712

[88] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 213–224. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950391

[89] V. Chippa, A. Raghunathan, K. Roy, and S. Chakradhar, "Dynamic effort scaling: Managing the quality-efficiency tradeoff," in *Proceedings*

of the 48th Design Automation Conference, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 603–608. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024863

[90] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 301–312. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151008

[91] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 164–174. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993518

[92] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 33–52. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509546

[93] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 309–328. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660231

[94] P. Roy, R. Ray, C. Wang, and W. F. Wong, "Asac: Automatic sensitivity analysis for approximate computing," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '14. New York, NY, USA: ACM, 2014, pp. 95–104. [Online]. Available: http://doi.acm.org/10.1145/2597809.2597812

[95] Q. Zhang, F. Yuan, R. Ye, and Q. Xu, "Approxit: An approximate computing framework for iterative methods," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 97:1–97:6. [Online]. Available: http://doi.acm.org.libproxy1.nus.edu.sg/10.1145/2593069.2593092

[96] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian, "Mitigating soft error failures for multimedia applications by selective data protection," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '06. New York, NY, USA: ACM, 2006, pp. 411–420. [Online]. Available: http://doi.acm.org/10.1145/1176760.1176810

[97] M. Stephenson, J. Babb, and S. Amarasinghe, "Bidwidth analysis with application to silicon compilation," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00. New York, NY, USA: ACM, 2000, pp. 108–120. [Online]. Available: http://doi.acm.org.libproxy1.nus.edu.sg/10.1145/349299.349317

[98] J. Cong and K. Gururaj, "Assuring application-level correctness against soft errors," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 150–157. [Online]. Available: http://dl.acm.org.libproxy1.nus.edu.sg/citation.cfm?id=2132325.2132360

[99] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel, "Exploiting program-level masking and error propagation for constrained reliability optimization," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 17:1–17:9. [Online]. Available: http://doi.acm.org.libproxy1.nus.edu.sg/10.1145/2463209.2488755