

UC Irvine

UC Irvine Electronic Theses and Dissertations

Title

Software Assists to On-chip Memory Hierarchy of Manycore Embedded Systems

Permalink

<https://escholarship.org/uc/item/8bx2m2hw>

Author

Namaki Shoushtari, Abdolmajid

Publication Date

2018

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at <https://creativecommons.org/licenses/by/4.0/>

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA,
IRVINE

Software Assists to On-chip Memory Hierarchy of Manycore Embedded Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Majid Namaki Shoushtari

Dissertation Committee:
Professor Nikil Dutt, Chair
Professor Alex Nicolau
Professor Elaheh Bozorgzadeh

2018

Portion of Chapter 2 © 2017 IEEE
Portion of Chapter 2 © 2018 ACM
Portion of Chapter 3 © 2015 IEEE
Portion of Chapter 3 © 2017 IEEE
All other materials © 2018 Majid Namaki Shoushtari

DEDICATION

In memory of my dear aunt Shahnaz, and my beloved grandmothers.

TABLE OF CONTENTS

	Page
LIST OF FIGURES	vii
LIST OF TABLES	xi
ACKNOWLEDGMENTS	xii
CURRICULUM VITAE	xiii
ABSTRACT OF THE DISSERTATION	xvi
1 Introduction	1
1.1 Technology Implications	1
1.1.1 Emerging Manycore Architectures	1
1.1.2 Memory Subsystem of Manycores	3
1.2 Workload Implications	5
1.2.1 Data-intensive Workloads and Variation in their Memory Requirements	5
1.2.2 Error-Resilient Workloads	6
1.3 Thesis Contributions and Organization	7
2 Software-Programmable On-chip Memory Hierarchy	10
2.1 Introduction	10
2.2 Prior Work on SPMs in Multicores and Manycores	14
2.3 Target System Architecture Template	16
2.3.1 SPM Programming Model	17
2.3.2 Architecture Model	20
2.3.3 Execution Model	21
2.3.4 Coherence Issues	22
2.3.5 OS/Runtime Support	23
2.4 Motivational Example	24
2.5 ShaVe-ICE Details	27
2.5.1 SPM Allocator	28
2.5.2 Allocation Policies	28
2.5.2.1 Non-preemptive Allocation Heuristics	29
2.5.2.1.1 Local Allocator(LA)	29
2.5.2.1.2 Random Remote Allocator (RRA)	30

2.5.2.1.3	Closest Neighborhood Allocator (CNA)	30
2.5.2.2	Preemptive Allocation Heuristics	31
2.5.2.2.1	Closest Neighborhood with Guaranteed Local Share Allocator (CNGLSA)	31
2.5.3	Hardware Support	33
2.5.3.1	Distributed Memory Management Units	33
2.5.3.2	Network Protocol	34
2.5.4	Data Movements in ShaVe-ICE	35
2.5.4.1	Allocations/Deallocations	35
2.5.4.2	Accesses	35
2.5.4.3	Data Migration	36
2.6	SPM Sharing Evaluations	37
2.6.1	Experimental Setup	37
2.6.2	Remote vs. Off-chip Access Latencies	37
2.6.3	Memory Microbenchmark	39
2.6.4	Performance Comparisons: ShaVe-ICE vs. Software Cache	41
2.6.5	Performance Comparisons: ShaVe-ICE Policies	42
2.6.5.1	Scenario 1: Core Underutilization	44
2.6.5.2	Scenario 2: Variation in Memory Working Set Size	46
2.6.5.3	Scenario 3: Reducing Network Traffic	47
2.6.5.4	Scenario 4: Guaranteeing SPM Share for Locally-pinned Thread	48
2.6.6	Energy Comparisons: ShaVe-ICE Policies	50
2.6.7	Experiments with Real Workload Mixes	51
2.7	Discussion on Overheads	54
2.8	Cache+SPM and Shared Data Support Evaluations	56
2.8.1	Experiment 1: Coherence Overhead Due to False Sharing	57
2.8.2	Experiment 2: Coherence Overhead Due to Shared Data	58
2.8.3	Experiment 3: Dynamic Partitioning of Local Memory	59
2.8.4	Experiment 4: Sharing SPMs Between Cores	60
3	Approximate On-chip Data Storage	62
3.1	Introduction	62
3.2	Prior Work on Memory Approximation	64
3.2.1	Taxonomy of Prior Research on Approximate Memory Management	64
3.2.2	Overview of Prior Research and Practices	69
3.3	Partially-Forgetful Memories	80
3.4	Relaxed Cache	82
3.4.1	Introduction	82
3.4.2	Motivation	84
3.4.3	Hardware Support	86
3.4.3.1	Tuning Relaxed Ways Based on Architectural Knobs	87
3.4.3.2	Defect Map Generation and Storage	88
3.4.3.3	Non-Criticality Table	90
3.4.3.4	Making Cache Controller Aware of Block's Tag	91
3.4.4	Software Support	91

3.4.4.1	Programmer-Driven Application Modifications	91
3.4.4.1.1	Data Criticality Declaration	91
3.4.4.1.2	Cache Configuration	92
3.4.4.2	Runtime System	95
3.4.5	Evaluations	95
3.4.5.1	Experimental Setup	95
3.4.5.2	Benchmarks	97
3.4.5.3	Experimental Results	97
3.4.5.3.1	Leakage Energy Savings	97
3.4.5.3.2	Fidelity Analysis	98
3.4.5.3.3	Performance Analysis	99
3.5	QuARK Cache	101
3.5.1	Introduction	101
3.5.2	STT-MRAM Reliability/Energy Trade-off Knobs	103
3.5.2.1	STT-MRAM Basics	103
3.5.2.2	Reliability-Energy Knobs	104
3.5.3	The QuARK Approach	105
3.5.3.1	Software Support	106
3.5.3.2	Hardware Support	107
3.5.3.2.1	QuARK Cache Approximation Table	108
3.5.3.2.2	QuARK Cache Controller	109
3.5.3.2.3	Support for Cache Fillings and Write-backs	109
3.5.4	Evaluations	111
3.5.4.1	Experimental Setup	111
3.5.4.2	Benchmarks	112
3.5.4.3	Experimental Results	113
3.6	Write-Skip SPM	116
3.6.1	Introduction	116
3.6.2	The Write-Skip Approach	117
3.6.2.1	Read-Before-Write	117
3.6.2.2	Approximate Equality	117
3.6.3	Evaluations	118
3.6.3.1	Approximate Value Locality	118
3.6.3.2	Output Fidelities	120
3.6.3.3	Energy Consumption in On-chip Memory	121
3.7	Controlled Memory Approximation	123
3.7.1	Introduction	123
3.7.2	Problem Modeling	125
3.7.2.1	Application Class	125
3.7.2.2	Monitoring Quality at Runtime	126
3.7.2.3	Memory Approximation Knob(s)	126
3.7.3	Quality Control with Feedback Control Theory	127
3.7.4	Case Study: Video Edge Detection	129
3.7.4.1	Application Description and Error Metric	129
3.7.4.2	Memory Approximation Knob	129

3.7.5	System Identification	129
3.7.5.1	Controller	131
3.7.5.2	Fault Injection Mechanism	132
3.7.5.3	Input Dependency	133
3.7.5.4	QoS Tracking	133
3.7.5.5	Comparison	133
4	Conclusions and Future Directions	137
4.1	Technical Contributions	137
4.2	Future Directions	138
	Bibliography	140

LIST OF FIGURES

	Page
1.1 High-level overview of the proposed software assisted memory hierarchy. . .	8
2.1 Number of memory accesses measured periodically for the execution of the wikisort and fft benchmarks: utilization of memory resources changes within a thread and between threads.	12
2.2 Software-assisted memory hierarchy vs. hardware-managed memory hierarchy. Relative size of gray components shows their contribution in managing memory hierarchy.	13
2.3 Number of CPU cycles that a thread has to be stalled for an SPM API to be handled by the OS/HW. SPM page size is assumed to be 64 bytes.	19
2.4 A code snippet using SPM APIs.	19
2.5 A 4X4 example of target manycore architecture with software-assisted memory hierarchy.	20
2.6 SPM manager.	23
2.7 ShaVe-ICE allocator is invoked when a thread T_i mapped on core C_j calls any of the SPM APIs. It virtualizes and ultimately shares the entire distributed SPM space between all the concurrently running threads. Dedicated hardware assist enables remote allocations and remote memory accesses.	25
2.8 An example showing how sharing the entire SPM space improves the overall performance of a 2X2 system with 3 thread running on it. The left-hand side column shows the state of SPM allocations when allocations are limited to the local SPM only; and right-hand side column shows the same scenario when remote allocations are also possible in addition to local allocations. (a)&(a'): only thread T_1 is running, (b)&(b'): thread T_2 starts execution after 20K cycles while T_1 is still running, (c)&(c'): thread T_3 starts execution after 25K cycles while both T_1 and T_2 are still running. Average and maximum execution times among all three threads are improved by about 2X and 2.9X respectively.	26
2.9 Hop-distance-based search for empty SPM space in Closest Neighborhood allocation policy.	31
2.10 SATT entry	33
2.11 Average access latency for various hop distances.	38

2.12	Overall flow of mem-ubench execution: each phase (prologue, body, epilogue) is either memory or compute intensive to a configurable degree. Operations are performed on three data arrays of configurable size with both regular (strided) and randomized access patterns. The number of executions of each phase as well as the entire 3-phase loop are configurable.	40
2.13	ShaVe-ICE vs. software cache – Case A: thread’s working-set is smaller than software cache and SPM, Case B: thread’s working-set is larger than software cache and SPM. Lower values are better.	42
2.14	Miss rate of various mem-ubench configurations for different available SPM capacities. Miss rate does not have a linear relationship with SPM capacity since not all the data is of the same importance.	43
2.15	Scenario 1: Average execution time among all threads in an 8x8 system - RRA normalized to LA. Core underutilization provides the opportunity to RRA to allocate more SPM space for the threads that have large working set size. . .	45
2.16	Scenario 2: Average execution time among all threads in an 8x8 system - RRA normalized to LA: Threads with small working set size provide the opportunity to RRA to utilize the SPM space unused by the locally-pinned thread in order to allocate more SPM space for the threads that have large working set size.	46
2.17	Scenario 3: Average execution time among all threads in an 8x8 system - CNA normalized to RRA: CNA allocates remote pages as close as possible to the owner core resulting in reduced network traffic, faster access latency and ultimately better overall performance compared to RRA.	48
2.18	Scenario 4: Difference in the local hit ratio - CNGLSA-CNA and average execution time among all threads in an 8x8 system - CNGLSA normalized to CNA: CNGLSA returns the remotely allocated SPM space to the locally-pinned thread should that thread needs it. In some cases, this results in improved local hit ratio and decreased average memory access latency compared to CNA.	49
2.19	Dynamic energy in memory subsystem and network: CNA normalized to LA.	50
2.20	Dynamic energy in memory subsystem and network: CNGLSA normalized to CNA.	50
2.21	Workload mixes with various core utilizations in a 4X4 platform.	52
2.22	Execution time comparison for various workload mixes as shown in Figure 2.21: CNGLSA normalized to LA. Lower values are better.	53
2.23	Runtime overhead of SPM allocator for different combinations of allocation size and page size: local policy (simplest policy).	55
2.24	Runtime overhead of SPM allocator for different combinations of allocation size and page size: closest neighborhood with local reservation policy (most complex policy).	56
2.25	Comparing software-assisted memory hierarchy with cache-based hierarchy implementing MESI protocol when false data sharing exists.	57
2.26	Comparing software-assisted memory hierarchy with cache-based hierarchy implementing MESI protocol when true data sharing exists.	58
2.27	Comparison of execution time with different configurations of a fixed size local memory: cache only, SPM and cache, SPM only. Lower is better.	59

2.28	Comparison of energy consumption with different configurations of a fixed size local memory: cache only, SPM and cache, SPM only. Lower is better.	59
2.29	Comparison of execution time and energy consumption when remote SPMs are available for allocation. Legends are shown as: (local cache, local SPM, remote SPM at hop distance 1). Lower is better.	60
3.1	Exploring performance-energy-fidelity space for Image Smoothing benchmark by adjusting Relaxed Cache controlling knobs (leakage energies and execution times are normalized to a baseline that Uses 700 mV for SRAM array supply voltage).	84
3.2	Trade-off between cache capacity, VDD, AFB and bit-cell leakage power (cache block size=64-byte, technology=45nm).	85
3.3	High-level diagram showing HW/SW components of Relaxed Cache and their interactions.	86
3.4	A Sample 4-way Cache with VDD=580mV and AFB=4.	88
3.5	Encoding and decoding defect map info in Relaxed Cache	89
3.6	Abstracting Relaxed Cache knobs up to metrics familiar to a software programmer (i.e., performance, fidelity, and energy consumption). Note that (VDD = High, AFB = Mild) and (VDD = High, AFB = Aggressive) combinations are sub-optimal, hence not applicable.	93
3.7	A sample code showing programmer’s data criticality declarations and cache configurations for Relaxed Cache.	94
3.8	SRAM BER for 45nm using models and data from [149]	96
3.9	Leakage energy savings for a 4-Way L1 cache with 3 relaxed ways (energy savings are normalized to a baseline cache that uses 700mV).	98
3.10	Fidelity results for (a) Scale and (b) Image-Smoothing benchmarks.	99
3.11	Fidelity results for Edge-Detection benchmark (VDD=480mV).	100
3.12	FPS-PSNR trade-offs with and without Relaxed Cache scheme (AFB=4).	101
3.13	STT-MRAM knobs for reliability-energy trade-off in 1MB cache. (a) Write Pulse Current Reduction (WPCR), and (b) Read Pulse Current Reduction (RPCR).	105
3.14	A pseudo-code example showing how QuARK Cache APIs can be used in a face detection application.	107
3.15	Integrating QuARK Cache into the architecture. Required changes are highlighted in gray.	108
3.16	Distribution of overall and approximation read and write accesses in L2 cache for the selected benchmarks.	113
3.17	QuARK Cache evaluation results: (a) Distribution of accesses in mixed-reliability workloads, (b) Energy savings (normalized to fully-protected STT-MRAM L2 cache), (c) Average relative error for blackscholes benchmark, (d) PSNR for Scale and Image Smoothing benchmarks, and (e) Mean pixel difference for Corner Detection, Edge Detection and Sobel benchmarks.	116
3.18	Percentage of approximately-equal writes in image-smoothing.	119
3.19	Percentage of approximately-equal writes in sobel.	119
3.20	Percentage of approximately-equal writes in k-means.	119

3.21	Output fidelity for image-smoothing.	120
3.22	Output fidelity for sobel.	120
3.23	Output fidelity for k-means.	121
3.24	Energy consumption in on-chip memory for image-smoothing normalized to the baseline where none of the write operations are skipped.	122
3.25	Energy consumption in on-chip memory for sobel normalized to the baseline where none of the write operations are skipped.	122
3.26	Energy consumption in on-chip memory for k-means normalized to the baseline where none of the write operations are skipped.	122
3.27	Open-loop knob settings (prior works) vs. closed-loop quality control (this work).	123
3.28	Closed loop approach (this work) for tuning memory approximation knob(s).	128
3.29	Predicted Model vs. Measured Output.	131
3.30	Variation of the quality of the edge detection in various video scenes when the bit error rate is constant.	134
3.31	Quality tracking results. Red curve shows the acceptable error and the blue curve shows the error achieved by the controller.	135
3.32	Comparing PI controller with manual step-wise re-calibration similar to [16].	136
3.33	Canny edge detection applied to different images with various write bit error rates resulting in different quality metrics.	136

LIST OF TABLES

	Page
1.1 ITRS mobile devices trend [71]	3
2.1 ShaVe-ICE APIs	17
2.2 Details of threads in the motivational example of Figure 2.8	25
2.3 SPM-related network messages in ShaVe-ICE	34
2.4 List of benchmarks for ShaVe-ICE experiments	51
2.5 List of microbenchmarks	56
3.1 Prior research In approximate memory management classified based on abstraction levels involved	65
3.2 Prior research In approximate memory management classified based on approximation objective	66
3.3 Prior research In approximate memory management classified based on the memory component	67
3.4 Prior research In approximate memory management classified based on the memory technology	67
3.5 Prior research In approximate memory management classified based on the approximation strategy	68
3.6 A sample criticality table for Relaxed Cache	90
3.7 gem5 settings for Relaxed Cache experiments	96
3.8 Relation between PSNR and perceptual quality in image processing domain	97
3.9 QuARK Cache APIs	106
3.10 gem5 settings for QuARK Cache experiments	111
3.11 Accuracy-energy transducer map for 1MB QuARK Cache-enabled STT-MRAM cache. Energy consumptions are reported for a 64-byte cache line.	112
3.12 List of approximate applications for QuARK Cache experiments.	112
3.13 List of workload mixes for QuARK Cache experiments.	113
3.14 List of approximate applications for Write-Skip experiments	118

ACKNOWLEDGMENTS

I would like to express my utmost gratitude to my adviser, Professor Nikil Dutt. His mentorship has helped me grow as a person and learn the necessary social skills to grow as a successful individual in the society. I thank him for having faith in me, through my initial struggles, and standing by me in my times of need.

I would like to thank the rest of my dissertation committee members Professor Alex Nicolau and Professor Eli Bozorgzadeh for their time, support and invaluable advice.

I thank my friends and colleagues at UCI, who made a great impact on my life. In particular, I thank Dr Hossein Tajik, Dr Abbas Banaiyan, Bryan Donyanavard, Hamid Nejatollahi, Sajjad Taheri, Dr Jurngyu Park, Dr Amir Rahmani, Tiago Muck, Roger Hsieh, Kasra Moazzami, Dr Luis (Danny) Bathen, Santanu Sarma, HIRAK KASHYAP, Amir Mahdi Hosseini Monazzah, Maral Amir, Zhi Chen and Dr Nga Dang for their friendship, feedback, guidance, and collaborations.

I thank Dr Abbas Rahimi for mentoring me at the initial stages of my PhD.

I must thank my dear friends Amirali Ghofrani, Farshad Yazdi, Morteza Kayyalha, Aida Ebrahimi, Mehrdad Biglarbegan, and Somayeh Sadeghi for their support and the good memories we have created together.

I would like to thank UC Irvine graduate division, NSF Variability Expedition (Grant Number CCF-1029783) and the school of ICS for providing funding opportunities during my PhD program.

I am also very grateful to Professor Alex Nicolau who provided me with many teaching opportunities and mentored me to be a better instructor.

I thank Melanie Sanders, Holly Byrnes, Kris Bolcer, Grace Wu and Melanie Kilian for making ICS and CECS such enjoyable places to work and for all their help and advice on so many different subjects.

I also thank ACM and IEEE for permissions to include parts of chapters 2 and 3 of my dissertation, which were originally published in IEEE Embedded System Letters, ACM Transaction on Embedded Computing Systems, ACM/IEEE International Symposium on Low Power Electronics and Design.

I am deeply thankful to my family for their love, continued support, and sacrifices; especially my brother Omid for setting high bars for educational and intellectual achievement.

CURRICULUM VITAE

Majid Namaki Shoushtari

EDUCATION

Doctor of Philosophy in Computer Science University Of California, Irvine	2018 <i>Irvine, CA</i>
Master of Science in Computer Engineering University Of Tehran	2012 <i>Tehran, Iran</i>
Bachelor of Science in Computer Engineering University Of Tehran	2009 <i>Tehran, Iran</i>

RESEARCH EXPERIENCE

Graduate Student Researcher University of California, Irvine	2012–2017 <i>Irvine, California</i>
Graduate Research Assistant University Of Tehran	2009–2012 <i>Tehran, Iran</i>

TEACHING EXPERIENCE

Teaching Assistant University of California, Irvine	2014–2018 <i>Irvine, CA</i>
Teaching Assistant University of Tehran	2009–2012 <i>Tehran, Iran</i>

WORK EXPERIENCE

Software Engineering Intern NVIDIA Corp.	June 2016 – Sept. 2016 <i>Santa Clara, CA</i>
Software Engineering Intern Rambus Inc.	June 2015 – Sept. 2015 <i>Sunnyvale, CA</i>
Engineering Intern Kiatel	June 2008 – Sept. 2008 <i>Karaj, Iran</i>

REFEREED JOURNAL PUBLICATIONS

- Exploiting Partially-Forgetful Memories for Approximate Computing** 2015
IEEE Embedded Systems Letters
- Automatic Management of Software Programmable Memories in Many-core Architectures** 2016
IET Computers & Digital Techniques
- SAM: Software-Assisted Memory Hierarchy for Scalable Manycore Embedded Systems** 2017
IEEE Embedded Systems Letters
- ShaVe-ICE: Sharing Distributed Virtualized SPMs In Many-Core Embedded Systems** 2018
ACM Transactions on Embedded Computing Systems

REFEREED CONFERENCE PUBLICATIONS

- ARGO: Aging-aware GPGPU Register File Allocation** 2013
International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)
- Multi-layer Memory Resiliency** 2014
Design Automation Conference (DAC)
- Cross-layer Virtual/physical Sensing and Actuation for Resilient Heterogeneous Many-core SoCs** 2016
Asia and South Pacific Design Automation Conference (ASP-DAC)
- QuARK: Quality-configurable Approximate STT-MRAM Cache by Fine-grained Tuning of Reliability-Energy Knobs** 2017
International Symposium on Low Power Electronics and Design (ISLPED)

TECHNICAL REPORTS

A Survey of Techniques for Approximate Memory Management **Sept. 2017**

UC Irvine, Center for Embedded and Cyber-physical Systems, CECS TR 17-03

SPM-vSharE: Memory Management in SPM-based Many-core Embedded Systems **Nov. 2016**

UC Irvine, Center for Embedded and Cyber-physical Systems, CECS TR 16-08

Relaxing Manufacturing Guard-bands in Memories for Energy Saving **Aug. 2014**

UC Irvine, Center for Embedded and Cyber-physical Systems, CECS TR 14-04

SOFTWARE

gem5-spm <https://github.com/duttresearchgroup/gem5-spm>
SPM support for gem5 architectural simulator.

pin-memapprox <https://github.com/mjshoushtari/pin-memapprox>
A PIN-based fault injector tool to simulate annotated approximate programs.

ABSTRACT OF THE DISSERTATION

Software Assists to On-chip Memory Hierarchy of Manycore Embedded Systems

By

Majid Namaki Shoushtari

Doctor of Philosophy in Computer Science

University of California, Irvine, 2018

Professor Nikil Dutt, Chair

The growing computing demands of emerging application domains such as Recognition/Mining/Synthesis (RMS), visual computing, wearable devices and the Internet of Things (IoT) has driven the move towards manycore architectures to better manage tradeoffs among performance, energy efficiency, and reliability.

The memory hierarchy of manycore architectures has a major impact on their overall performance, energy efficiency and reliability. We identify three major problems that make traditional memory hierarchies unattractive for manycore architectures and their data-intensive workloads: (1) they are power hungry and not a good fit for manycores in face of dark silicon, (2) they are not adaptable to the workload's requirements and memory behavior, and (3) they are not scalable due to coherence overheads.

This thesis argues that many of these inefficiencies are the result of software-agnostic hardware-managed memory hierarchies. Application semantics and behavior captured in software can be exploited to more efficiently manage the memory hierarchy. This thesis exploits some of this information and proposes a number of techniques to mitigate the aforementioned inefficiencies in two broad contexts: (1) explicit management of hybrid cache-SPM memory hierarchies, and (2) exploiting approximate computing for energy efficiency.

We first present the required hardware and software support for a software-assisted memory hierarchy that is composed of distributed memories which can be partitioned between caches and software-programmable memories (SPMs) at runtime. This memory hierarchy supports local and remote allocations and data movements between SPM and cache and also between two physical SPMs. The distributed SPM space is shared between a mix of threads where each thread explicitly requests SPM space throughout its execution. The runtime component of this hierarchy shares the entire distributed SPM space between contending threads based on an allocation policy. Unlike traditional memory hierarchies, we incorporate no coherence logic in this hierarchy. The program explicitly allocates the shared data on the distributed SPM space. For all threads of that program, the accesses to shared data are forwarded to the same physical copy.

Next, we augment caches and SPMs in this hierarchy with approximation support in order to improve the energy efficiency of the memory subsystem when running approximate programs. We present approximation techniques for major building blocks of our hybrid cache-SPM memory hierarchy. We introduce Relaxed Cache as an approximate private L1 SRAM cache where the quality, capacity, and energy consumption of this cache are controlled through two architectural knobs (i.e., voltage and the number of acceptable faulty bits per cache block). We then present QuARK Cache, an approximate shared L2 STT-MRAM cache. The read and write current amplitude provide two knobs to make a tradeoff between the accuracy of memory operations and the dynamic energy consumption. We then introduce Write-Skip, a technique that skips write operations in STT-MRAM data SPMs if the previous value and the new value are approximately equal. Finally, we discuss a quality-configurable memory approximation strategy using formal control theory that adjusts the level of approximation at runtime depending on the desired quality for the program’s output.

We implemented all software and hardware components of the proposed software-assisted memory hierarchy in the gem5 architectural simulator. Our simulations on a mix of RMS and

microbenchmarks show that our proposed techniques achieve better performance, energy, and scalability for manycore systems over traditional hardware-managed memory hierarchies.

Chapter 1

Introduction

1.1 Technology Implications

1.1.1 Emerging Manycore Architectures

Up to early 2000s, the principal approach to improve the performance of processors was to simply scale down the manufacturing technology, add more transistor and increase the clock frequency. This was being fueled by the Moore's law and Dennard Scaling. This was the trend for from 1980s to 2000s until the processors started hitting physical limitations in terms of their clock frequency and how effectively they could be cooled and still maintain accuracy.

Essentially, three primary factors led to the design of multicore processor:

1. The memory wall: the increasing gap between processor and memory speeds. This, in effect, pushes for cache sizes to be larger in order to mask the latency of memory. This helps only to the extent that memory bandwidth is not the bottleneck in performance.
2. The instruction-level parallelism (ILP) wall: the increasing difficulty of finding enough

parallelism in a single instruction stream to keep a high-performance single-core processor busy.

3. The power wall: the trend of consuming exponentially increasing power with each factorial increase of operating frequency. The power wall poses manufacturing, system design and deployment problems that have not been justified in the face of the diminished gains in performance due to the memory wall and ILP wall.

Multicores usually use a bus-based communication infrastructures, which may not scale beyond dozens of cores. As opposed to multicores, a manycore architecture consists of a network-like interconnect, and hundreds or even thousands of processing cores with their own local memories (SRAM/non-volatile caches and scratchpad memories). At the heart of these systems is the network-like communication infrastructure, which has been designed to scale well beyond the dozens of cores. Multicore processors, are usually designed to efficiently run both parallel and serial code, and therefore place more emphasis on high single thread performance (e.g., devoting more silicon to out of order execution, deeper pipelines, more superscalar execution units, and larger, more general caches), and shared memory. Manycore processors are distinct from multicore processors in that: they are optimized from the outset for a higher degree of explicit parallelism, and for higher throughput (or lower power consumption) at the expense of latency and lower single thread performance.

The manycore revolution is also driven by the demand from new softwares such as media rich applications (e.g., streaming content from the cloud), resulting in complex software stacks that require new ways to improve system performance to cope with the increasingly complex software stacks.

Examples of such manycore architectures are Intel SCC [100], Kalray MPPA-256 [44], Tilera TILE64 [27], Adapteva Epiphany [4], Kalray MPPA2-256 [75], IBM Blue Gene/Q [65].

Although manycore architectures have not been deployed yet for widespread use, one clear

trend exists in the mobile computing domain. In recent years, mobile devices, notably smartphones, have shown significant expansion of computing capabilities. ITRS [71] predicts that the number of cores for Application Processors (AP) will moderately increase by 4x in the time horizon. On the other hand, the number of cores in Graphics Processing Units (GPU) will increase by 50x in this time horizon. This increase in GPU processing capacity is necessary to keep up with the increasingly growing number of megapixel offered by the displays. It is expected that the traffic between AP and memory will have to correspondingly increase more than 2x.

Table 1.1: ITRS mobile devices trend [71]

Year	2015	2017	2019	2021	2023	2025
Number of AP cores	4	9	18	18	28	36
Number of GPU cores	6	19	49	69	141	247
Number of GPU cores Max frequency of any Component in System (GHz)	2.7	2.9	3.2	3.4	3.7	4
Number of Mega pixels in Display	2.1	2.1	3.7	8.8	8.8	33.2
Number of Mega pixels in Display Bandwidth between AP and Main memory (Gb/s)	25.6	34.8	52.6	57.3	61.9	61.9

Manycore technology has been viewed as a way to improve performance at the processor level, but its profound implications on the energy efficiency and reliability of future embedded systems with 100s or 1000s of cores has not been studied in depth. Facing the challenges brought by dark silicon [50], it is more important than ever to deploy energy-efficient mechanisms to continue supporting the growing high performance requirements of future embedded systems.

1.1.2 Memory Subsystem of Manycores

The advent of manycore computing platforms exacerbates the classical processor-memory performance bottleneck. As we scale to manycore systems, it becomes increasingly challenging to scale the traditional cache-based memory hierarchies [125]. One important reason is because the overhead of coherence logic increases rapidly with the number of cores. As we scale the

number of cores on a cache coherent system, “cost” in “time and memory” grows to a point beyond which the additional cores are not useful in a single parallel program. This is called Coherency Wall. Some processors have already tried to alleviate this problem by removing hardware cache coherence from processors either partially or completely, e.g. Intel SCC [100], Kalray MPPA-256 [44]. In these architectures, the coherence – whenever needed by the application/system – must be implemented in software. However in these systems, caching – without coherence – is still implemented in hardware. The fact that hardware caching in manycore architectures becomes power-hungry due to the complexity of caching logic is another challenge hardware implemented caches are facing in scaling to manycore architectures.

An alternative mechanism is to deploy software caching mechanisms for smart data management, using the raw memories in the processor. Here the data movement between the close-to-processor memory and the main memory has to be done explicitly in software, typically done through the use of Direct Memory Access (DMA) instructions. We refer to the raw memories in such processors as Software Programmable Memories (SPM). IBM Cell [69], Tiler TILE64 [27], Adapteva Epiphany [4] use SPMs in their on-chip memory hierarchy.

SPMs offer many advantages over caches. When application designers have deep understanding of the data requirements of their applications – especially in embedded systems – the use of SPMs allows developers to exploit application semantics effectively to achieve efficient execution. SPMs offer many other advantages over caches. The first is power efficiency by eliminating the hardware overhead of traditional caching. The second is predictability, a critical factor for real-time systems. Third, there is potential for performance improvement by orchestrating the management of data transfers explicitly in software.

The need for more energy efficient memories and denser memory space to accommodate for emerging data-intensive applications in the embedded domain has led designers to design Non-Volatile Memories (NVMs) as alternatives to SRAM for on-chip memories. Typically,

NVMs (e.g., Spin-Transfer Torque (STT) Memories, Phase-Change Memory (PCM)) offer high densities, low leakage power, comparable read latencies and dynamic read power with respect to traditional embedded memories (SRAM/eDRAM). One major drawback across NVMs is the expensive write operation (high latencies and dynamic energy per access) and wearout constraints overtime.

One of the potential benefits of using SPMs is the ability to explicitly manage data accesses for thermal and wearout constraints, particularly for NVMs.

1.2 Workload Implications

1.2.1 Data-intensive Workloads and Variation in their Memory Requirements

One of the most critical challenges for today's and future data-intensive and big-data problems (ranging from economics and business activities to public administration, from national security to many scientific research areas) is data storage and analysis. The primary goal is to increase the understanding of processes by extracting highly useful values hidden in the huge volumes of data. The increase of the data size has already surpassed the capabilities of today's computation architectures which suffer from the limited bandwidth, due to communication and memory-access bottlenecks.

Data-centric nature of several emerging media-rich applications in the embedded domain creates demand for denser memories. Memories are likely to dominate energy as well as reliability concerns [112] for computing systems.

While memory resources are becoming more vital to embedded computing platforms, because of the nature of the emerging applications running on them, their importance could vary

over time and at a time between concurrently running applications. Tajik et al. [143, 142] have shown the variation in memory requirements between concurrently running threads and within a thread during its course of execution.

1.2.2 Error-Resilient Workloads

Inherent application resilience is the property of an application to produce acceptable outputs despite some of its underlying computations being incorrect or approximate. It is prevalent in a broad spectrum of applications such as digital signal processing, image, audio, and video processing, graphics, wireless communications, web search, and data analytics. Emerging application domains such as Recognition, Mining and Synthesis (RMS) [49], which are expected to drive future computing platforms, also exhibit this property in abundance. The inherent resilience of these applications can be attributed to several factors: (1) significant redundancy is present in large, real-world data sets that they process, (2) they employ computation patterns (such as statistical aggregation and iterative refinement) that intrinsically attenuate or correct errors due to approximations, and (3) a range of outputs are equivalent (i.e., no unique golden output exists), or small deviations in the output cannot be perceived by users.

The distributed memory subsystem is one of the fundamental performance and energy bottlenecks in emerging manycore systems and is likely to dominate energy as well as reliability concerns for those systems. This error resilience can be exploited to build more efficient computing systems, more specifically to improve the energy efficiency of the memory subsystem in emerging manycore architectures.

1.3 Thesis Contributions and Organization

Any strategy for memory management of manycores should address a number of challenges: (1) adaptation to the workload with varying memory requirements (in terms of working-set size, access pattern, reliability of accesses) (2) energy efficiency, and (3) scalable coherence management.

This thesis takes the stand that many of these challenges cannot be addressed by today's software-agnostic hardware-managed memory hierarchies. We argue that these challenges could be overcome by using more sophisticated memory hierarchy management techniques that receive some form of software-assist (e.g., information about a program's semantics, memory access patterns, memory phases, etc). Figure 1.1 shows an overview of our contributions in this thesis towards achieving this software assisted memory hierarchy.

Chapter 2 presents a software-assisted memory hierarchy for manycore embedded systems along with the details of its hardware and software support. This hierarchy is composed of distributed memories that can be partitioned between caches and SPMs at runtime. The distributed SPM space is shared between a mix of unknown threads where each thread explicitly requests SPM space for its most accessed data objects throughout its execution. The runtime component of this hierarchy shares the entire SPM space between contending threads based on their requirements and a choice of allocation policy. It also decides about how local memories are partitioned between SPM and cache. Unlike traditional memory hierarchies, we incorporate no coherence logic in this hierarchy. The program explicitly allocates the shared data on the distributed SPM space. For all threads of that program, the accesses to shared data are forwarded to the same physical copy. This memory hierarchy supports local and remote allocations and data movements between SPM and cache and also between two physical SPMs.

Chapter 3 augments caches and SPMs in this hierarchy with approximation support in

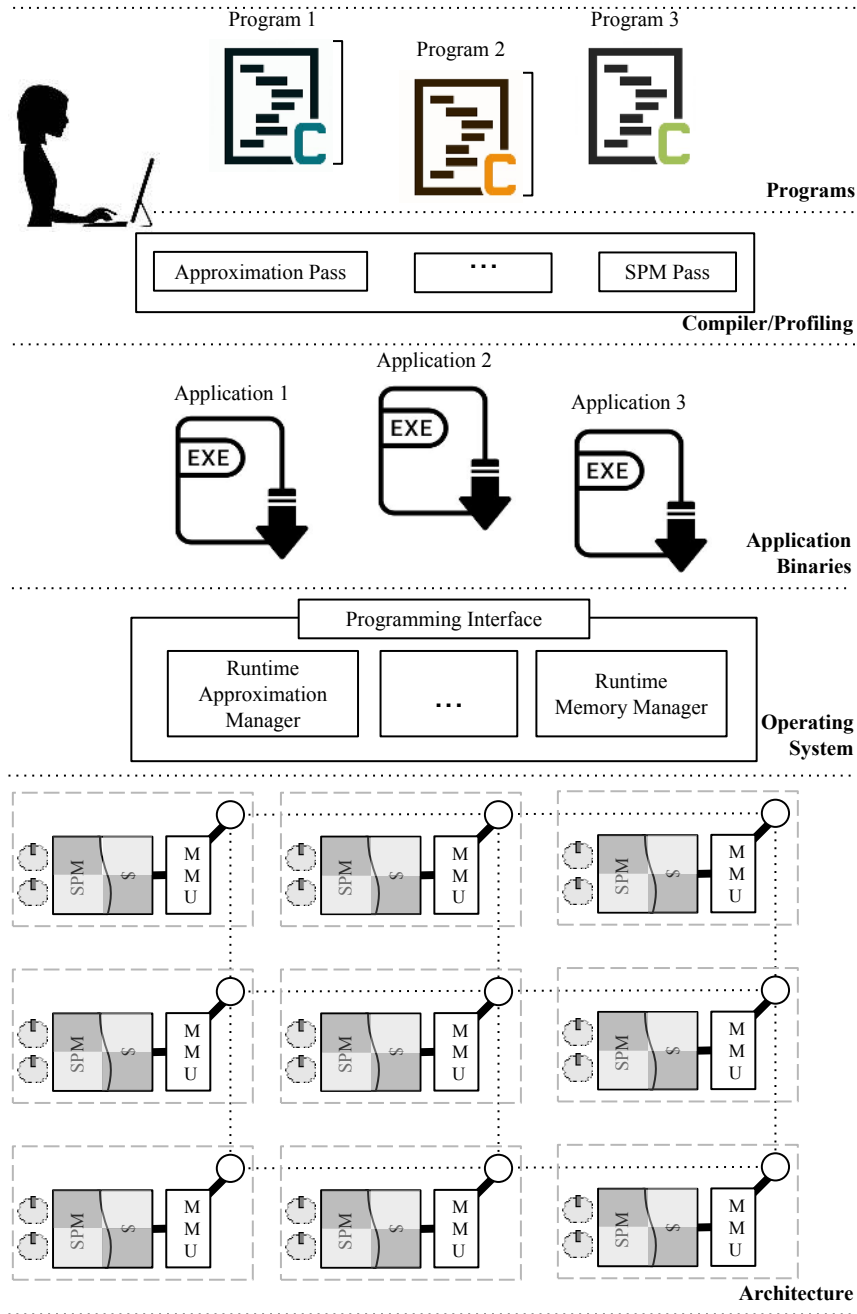


Figure 1.1: High-level overview of the proposed software assisted memory hierarchy.

order to improve the energy efficiency of the memory subsystem. We present approximation techniques for major components of the memory hierarchy introduced in Chapter 2. We introduce Relaxed Cache as an approximate private L1 SRAM cache where the quality, capacity, and energy consumption of this cache are controlled through two architectural

knobs. We then present QuARK Cache, an approximate shared L2 STT-MRAM cache. The read and write current amplitude provide two knobs to to make a tradeoff between the accuracy of memory operations and the dynamic energy consumption. We then introduce Write-skip for STT-MRAM data SPMs. Finally, we discuss a strategy to control the level of memory approximation at runtime depending on the desired output quality.

In Chapter 4 we conclude this thesis and address future directions for this research.

Chapter 2

Software-Programmable On-chip Memory Hierarchy

2.1 Introduction

Future embedded systems are expected to use 10s to 100s of simple processing cores, forming Manycore Embedded System (MES) platforms that hold the promise of increasing performance through parallel execution.

However, these systems cannot rely on traditional approaches for system integration. Most notably: (1) The bus-based communication architecture is not a scalable solution for these systems – adopting Network-on-Chip (NoC) communication allows the system to reach a high level of parallelism and scalability. (2) The traditional cache-based memory hierarchy imposes a huge inefficiency in power consumption, due to the complexity of caching logic, and also in performance, due to the network traffic generated by coherence protocols.

The traditional hardware-managed memory hierarchy needs to be revisited to: (1) address

shortcomings (e.g., coherence overhead) of current techniques when cores are replicated beyond certain numbers, and (2) adapt the memory subsystem to a diverse workload, with variable memory requirements, running on a MES.

Traditionally, cache-coherent memory hierarchies have been the default choice for the memory subsystem of embedded systems, mainly because they hide the memory hierarchy from software and don't require any software intervention. Unfortunately, the overhead of purely hardware-managed memory hierarchy grows very fast as the number of cores increases. This overhead manifests itself in the exacerbated power consumption of the memory subsystem and the cache coherence logic and also the coherence traffic in the NoC. On the other hand, hardware is generally oblivious to the variable requirements of individual threads in a workload. Tuning allocation of memory resources to threads in order to improve the overall performance of the entire workload proves challenging for purely hardware-managed hierarchies.

Software-Programmable Memories (SPM, also known as scratchpad memories) are a promising alternative to hardware-managed caches. For equivalent data capacity, SPMs are around 30% smaller, slightly faster, yet consume about 30% less power than cache [22]. Additionally, no coherence management is required due to their software-programmable nature. However, SPMs require explicit software management. Researchers have previously proposed methods to ease this burden [115, 95, 137, 20, 78, 17].

Most previous efforts have assumed a single thread running on a core with a private SPM in isolation of other threads running concurrently or entering the system at a later time, or they assumed the workload mix is known ahead of time. The shortcomings are twofold: (1) They do not utilize memory resources that are local to idle cores – these idle memory resources present an opportunity for more efficient use of on-chip memory to boost the overall system efficiency. (2) The variation in the memory intensity level as well as working set size between concurrently running threads in emerging diverse workloads is neglected – this

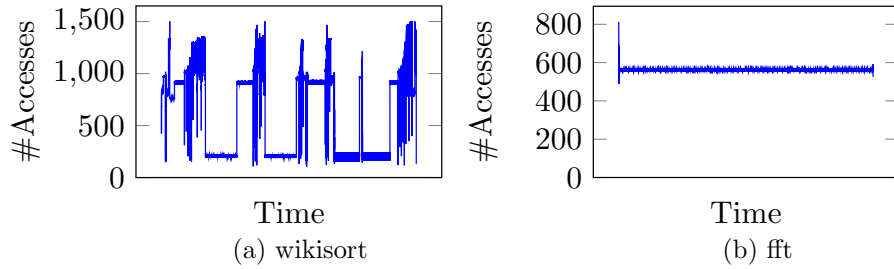


Figure 2.1: Number of memory accesses measured periodically for the execution of the wikisort and fft benchmarks: utilization of memory resources changes within a thread and between threads.

variation causes on-chip memory resources to be more valuable for some threads over others. The utilization of data memory can vary not only between threads in a workload, but also within a single thread over the course of its execution. Figure 2.1 illustrates the variation in temporal memory access patterns both between and within two different benchmarks.

Here we outline an alternative hierarchy organization for MES that employs a *Software-Assisted* Memory (SAM) hierarchy composed of both cache and SPM guided by static analysis as well as a holistic runtime support. SAM alleviates the aforementioned drawbacks of cache-only hierarchies. Figure 2.2 compares the SAM hierarchy with a purely hardware-managed hierarchy. SAM hierarchy requires less hardware management and relies on the software components of a system (i.e., application, compiler and operating system) to manage memory resources allowing it to be more adaptable to workload’s needs and consume less power.

We first, propose *ShaVe-ICE*, system software and hardware support for management of distributed SPMs in manycore embedded systems. ShaVe-ICE enables the system to efficiently *share* the SPM resources distributed across the chip. Sharing the SPM resources improves the *average* access latency for all the concurrently running threads, and hence improves the overall performance of the system. By *sharing* the physically distributed SPM space between all the running threads through virtualization, memory resources can be utilized more efficiently.

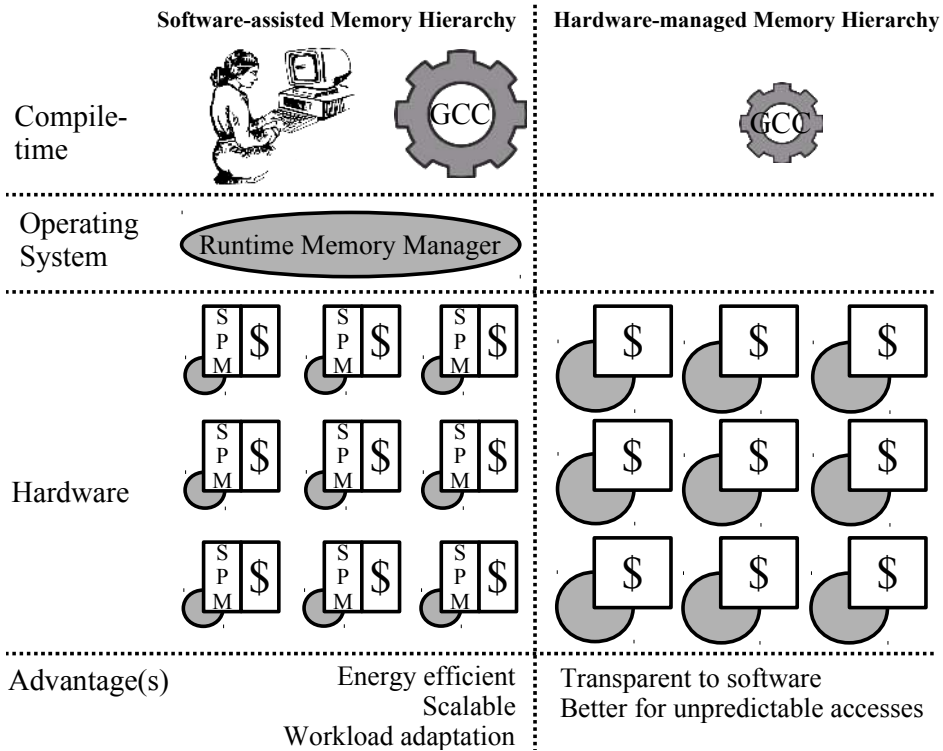


Figure 2.2: Software-assisted memory hierarchy vs. hardware-managed memory hierarchy. Relative size of gray components shows their contribution in managing memory hierarchy.

Unlike previous approaches, in this work we allow all threads to *opportunistically* exploit the entire physical memory space from all distributed SPMs for unpredictable workloads. ShaVe-ICE supports dynamic SPM sharing for workloads in which threads may enter and exit at any time and contend for on-chip memory resources.

It is worth mentioning that ShaVe-ICE does not focus on analyzing the access pattern of data objects in order to find the most profitable data objects to bring on-chip. That is an orthogonal problem which researchers have looked into extensively [43]. Instead, we focus on sharing a distributed SPM space between a mix of unknown threads where each thread explicitly requests SPM space for its most accessed data objects throughout its execution.

We define the necessary software/hardware components for sharing distributed SPMs under contention from unpredictable workloads, and build the foundation towards SPM-based manycore embedded systems. To the best of our knowledge, ShaVe-ICE is the first scheme

for sharing distributed SPMs in manycore embedded systems when an unpredictable mix of workloads is executing.

We then, extend ShaVe-ICE to support a hybrid composed of both distributed SPMs and caches and show how shared data is managed in such a hierarchy. Through sample microbenchmarks, we present the opportunities made possible by this hierarchy to reduce coherence protocol overhead, lower the energy consumed in memory subsystem, and adapt the memory configuration to workload’s requirements to reduce execution time.

2.2 Prior Work on SPMs in Multicores and Manycores

Extensive research exists on SPM management¹. Initially, researchers focused on static approaches to manage the SPM for a single thread running on a single-core architecture. The static SPM management techniques identify the part of overall data set that maximally improves the runtime performance of applications (e.g. most frequently used data) upon placement in SPM [116, 135, 15, 113, 147, 136]. However, data placement is fixed throughout the application’s execution meaning that the locations of data can not be changed during execution. Alternatively, dynamic approaches allow the movement of data at runtime, enabling the system to swap in the more frequently accessed data and swap out less-used data over time and achieve greater performance optimization [47, 89, 76].

Other works have considered a scenario in which multiple tasks are simultaneously running on a single-core architecture and sharing a single physical SPM. [57] proposes a compile-time analysis approach to support concurrent execution of tasks sharing the same SPM resource and assumes all working sets are known at compile time. [55] provides a high-level programming interface for SPM and DMA which can be used by the programmer for heap management. At runtime, a dynamic memory manager responds to memory space requests and maps data

¹A more comprehensive survey can be found in [133].

to the physical SPM as long as there is space.

More recently, the advent of integrating multiple processor cores on a single chip has resulted in a shift of focus for SPM management in the direction of multicore processors. In the realm of multicore SPM management, [78, 20, 95, 18, 19] all propose various compiler-based dynamic data management techniques for a memory hierarchy that incorporates SPM transparently without requiring explicit memory management.

Additionally, approaches have been proposed for managing SPM data in multicore systems with multiple tasks sharing SPM space. [137, 40] both propose compile time static analysis techniques for SPM allocation for a fixed set of tasks. [137] specifies an ILP formulation that integrates task scheduling with SPM partitioning and allocation at compile time to produce both a schedule and static data allocation that optimize performance by profiling the set of tasks. [40] generates SPM allocation of arrays using static analysis at compile time for a fixed set of tasks executing on a MPSoC sharing distributed SPMs in order to both improve performance and minimize energy consumption in embedded systems. Similarly, [98] defines an OpenMP extension and compiler optimization to allocate parts of data arrays to distributed SPM for parallel programs executing on an MPSoC. They improve performance by locating data near the processing elements that access it most frequently. [138] proposes algorithms that use profiling information to produce SPM mappings in order to minimize the worst case response time of a multitasking workload sharing SPM. These approaches all define SPM data allocation and mapping prior to execution, and, whether static or dynamic, therefore cannot handle diverse and unpredictable workloads.

Runtime adaptive approaches typically incorporate compile-time information with runtime observations to make allocation decisions at runtime. [39] profiles a fixed set of multimedia applications with varying inputs and passes this information to a runtime routine. The runtime routine monitors the application behavior and attempts to match it to one of the known profiles, and maps data to SPM accordingly to reduce energy consumption. [45] defines

a framework that allows programmers to guide runtime decisions for allocating heap data to SPM in order to reduce energy consumption. [26, 25, 23, 24] require programmer guidance for allocation, but also specify a virtualization layer that abstracts the explicit SPM management from the programmer and supports over-subscription. All of these adaptive approaches make allocation decisions at runtime for multicore architectures with multiple tasks sharing and contending for SPM space.

Recent efforts have considered a hybrid memory architecture where every core has a private SPM as well as a private cache. Their focus was to distinguish between data better suited for cache versus SPM. [7] and [81] both propose hybrid memory hierarchies (i.e. caches + SPM) that support globally addressable and coherent address spaces. Alvarez et al. [7] addressed the issue of keeping these two separate SPM and Cache memories within a core coherent. Chakraborty et al. [36] proposed a configurable hybrid local memory that can be dynamically partitioned between a cache and a SPM based on the varying requirements of a thread, with the idea that sometimes the varying workload may benefit from different SPM or cache capacities.

In ShaVe-ICE, we enable unpredictable mix of threads contending for SPM space to transparently allocate and access SPMs physically distributed throughout a chip. Through virtualization, our solution supports unlimited SPM allocation sizes, and makes global allocation decisions to benefit the performance of the overall workload. The access and allocation mechanism and policies are scalable for manycore systems.

2.3 Target System Architecture Template

We target an architecture in which a number of threads are running concurrently on a manycore system explicitly requesting SPM allocations/deallocations. Here we describe the

SPM programming model, architecture model and the execution model of such an architecture.

2.3.1 SPM Programming Model

With SPM-based memory hierarchy, the programmer and/or the compiler have explicit control on data movements. This enables them to prefetch a frequently used data object earlier than its actual access. SPM-based systems usually rely on explicit allocation/deallocation requests made by threads utilizing specified Application Programming Interfaces (APIs). A set of APIs, registered with the kernel, will be used by the programmer and/or the compiler to program SPMs. The API calls we define are listed in Table 2.1:

`SPM_ALLOC()` and `SPM_DEALLOC()` APIs can be used explicitly in the source code to request moving parts of a thread’s virtual address space between on-chip SPMs and the main memory through Direct Memory Access (DMA) transfers.

`SPM_ALLOC()` receives the base virtual address (`BaseVA`) and the size in bytes (`Size`) as arguments along with a flag (`AllocationMode`) which determines the allocation mode. There are two possibilities for the `AllocationMode` flag: `COPY` and `UNINITIALIZE`. When the flag is set to `COPY`, the allocated data is copied from main memory to SPM, whereas if the flag

Table 2.1: ShaVe-ICE APIs

Method	Parameters	Type	Note
SPM_ALLOC	BaseVA	uint64	Base virtual address of memory region
	Size	uint	Size of allocation
	AllocationMode	uint	Type of allocation
	isSharedData	bool	Flag to determine if this is shared data
	Metadata	uint	Other encoded hints for runtime manager
SPM_DEALLOC	BaseVA	uint64	Base virtual address of memory region
	Size	uint	Size of deallocation
	DeallocationMode	uint	Type of deallocation
	isSharedData	bool	Flag to determine if this is shared data
	Metadata	uint	Other encoded hints for runtime manager

is set to `UNINITIALIZE`, the allocation simply reserves SPM space without initializing the allocated memory. `UNINITIALIZED` allocations eliminate unnecessary DMA transfers when the data object values are not initialized in memory yet, so the current values do not need to be copied from main memory. `isSharedData` declares if the data belongs to one thread or it is shared data.

Similarly, `SPM_DEALLOC()` receives the base virtual address (`BaseVA`) and the size in bytes as arguments along with a flag (`DeallocationMode`) which determines the deallocation mode. There are two possibilities for the `DeallocationMode` flag: when this flag is set to `WRITE_BACK`, the data is written back to main memory from SPM; if the flag is set to `DISCARD`, the deallocation happens without updating the values in main memory. `DISCARD` flag is useful when the data is not needed anymore, preventing unnecessary memory write-back. A common case is to discard a piece of heap data in SPM that is going to be freed immediately, therefore does not need to be written back to main memory.

Figure 2.3 shows the number of cycles that allocation and deallocations of different sizes take with different modes. While the overhead of allocation and deallocation with `COPY` and `WRITE_BACK` grows substantially as the size of allocation grows, `UNINITIALIZE` and `DISCARD` keep this overhead low and near constant. Note that allocations with `COPY` could take more cycles than deallocations with `WRITE_BACK` because of the fact that some of the allocations have to wait additional cycles for data migration.

These method calls can be inserted manually by a domain expert programmer or automatically by a compiler pass or a combination of both. Finding best candidates for SPM mapping is an orthogonal issue that previous researchers have looked into and is not the focus of this work. Currently, in a preprocessing step, we analyze the source code automatically to identify heap allocation and deallocation sites and insert `SPM_ALLOC` and `SPM_DEALLOC` for each pair, accordingly. We also add `SPM_ALLOC` and `SPM_DEALLOC` to handle each function's stack. Anytime a function is called, a `SPM_ALLOC` is responsible for bringing that function's

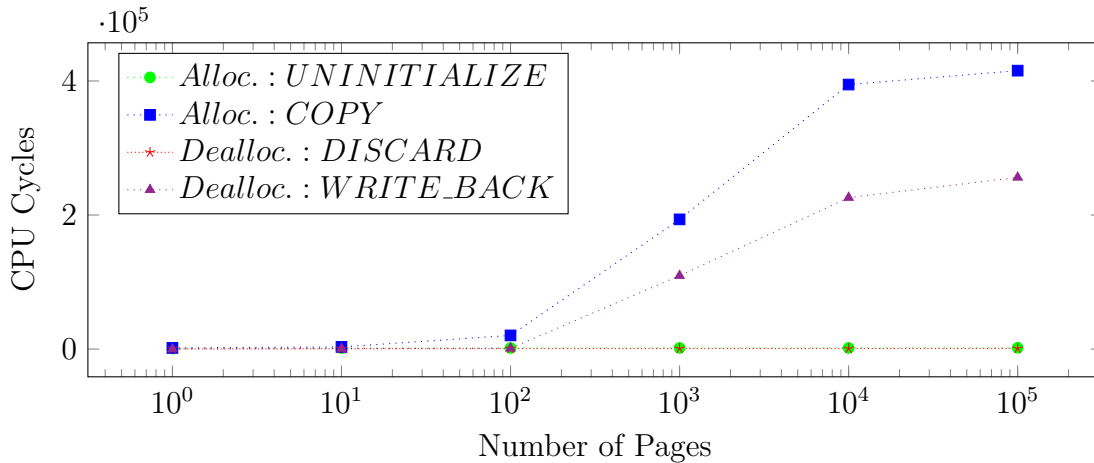


Figure 2.3: Number of CPU cycles that a thread has to be stalled for an SPM API to be handled by the OS/HW. SPM page size is assumed to be 64 bytes.

stack on-chip and right before returning from that function, a `SPM_DEALLOC` call will remove that from SPM. These SPM annotations are then pruned by profiling the program’s memory accesses and removing annotations for rarely accessed data objects.

The code snippet in Figure 2.4 shows how these APIs can be used within the source code of an example thread.

Calling these APIs transfers control to the operating system’s SPM allocator. The ShaVe-ICE allocator decides whether or not to grant the request and broadcasts messages throughout the platform accordingly. These API calls are blocking, meaning that control does not return

```

...
int *buffer = (int*) malloc (BUF_LENGTH*sizeof(int));
SPM_ALLOC (buffer, BUF_LENGTH*sizeof(int), UNINITIALIZED);
...
for (i = 0; i<BUF_LENGTH; i++) {
    buffer [i] = buffer [i] * 2;
}
...
SPM_DEALLOC (buffer, BUF_LENGTH*sizeof(int), WRITE_BACK);
...
free(buffer);
...

```

Figure 2.4: A code snippet using SPM APIs.

to the thread until the data transfers are complete.

2.3.2 Architecture Model

We target manycore embedded systems in which cores are connected via a mesh-like NoC as shown in Figure 2.5.

Each core consists of a simple in-order Central Processing Unit (CPU), a local memory (LM) that can be partitioned between L1 data cache and SPM, a Memory Management Unit (MMU), a Direct Memory Access (DMA) engine, a L1 instruction cache, an optional L2 cache for both data and instructions, and a network interface to manage NoC traffic for inter-core communication. The on-chip data memory is distributed evenly among all cores throughout the system.

In this architecture, the SPM address space is disjoint from the main memory address space, creating another layer in the memory hierarchy. All or a subset of distributed SPM space is accessible to each core. To enable that, each core’s MMU is extended to include an SPM Address Translation Table (SATT). SATT stores all of the virtual page to physical SPM address mappings for the pages that belong to that core.

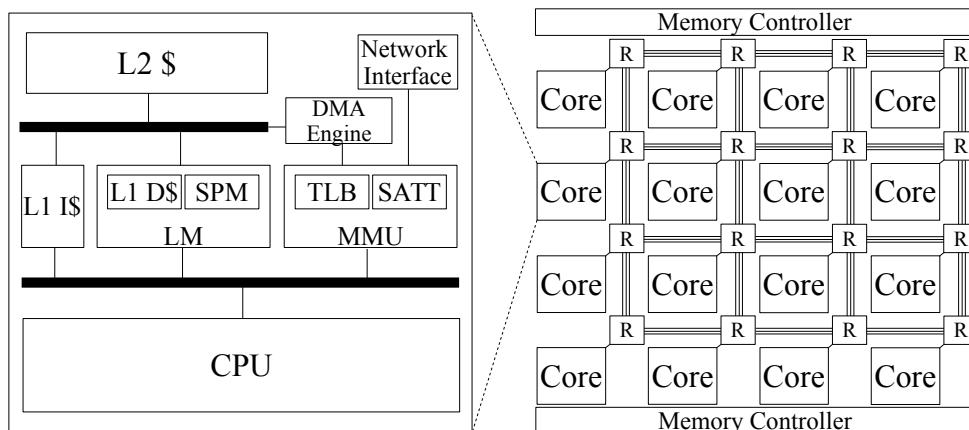


Figure 2.5: A 4X4 example of target manycore architecture with software-assisted memory hierarchy.

To avoid the problems of cache-coherent architectures, there is no cache coherence logic in this architecture and therefore caches are used to store private data only. On the other hand, the entire on-chip SPM space is accessible from any core. Any data that needs to be shared between multiple threads have to be explicitly copied to the SPM space by the software.

The local memory of each core is initially partitioned into two equally-sized cache and SPM. [36] has shown how having a hybrid local memory could be advantageous compared to a cache only hierarchy in terms of execution time and/or energy consumption. NVIDIA has used this approach in their Fermi GPGPUs [3]. We assume there are hardware mechanisms to change this balance at runtime. These hardware mechanisms have to be triggered by the runtime SPM manager, which we discuss later in this section.

2.3.3 Execution Model

Many modern embedded platforms support the concurrent execution of multiple independent applications on shared resources where applications start and stop at any time. We assume a number of processes, possibly each one spawning a number of threads, executing on this platform. Because of the abundance of cores, each thread is mapped to a core and stays there until completion so there is no context switching on a core.

In manycore systems, coherency is a major bottleneck for scalability. If a piece of data is duplicated on different cores, the consistency of those copies should be guaranteed by a coherence protocol. If any thread updates one of these copies, all other copies should be invalidated which is a very costly operation [157].

To avoid coherence issues, we only keep one copy of data on-chip and all threads access the same physical copy.

Note that, the support for mutual exclusion is still there. Every thread still needs to acquire

a lock in order to enter its critical section. The only difference is that, unlike cache coherence protocols that try to keep multiple copies coherent, here we only keep one copy.

Consider a multithreaded application with virtual address space shared between threads. Once a thread requests a part of the virtual address space to be mapped on-chip, accesses from all other threads to that region of virtual address space will be forwarded to the single copy of that data in the on-chip SPM.

For multithreaded applications, we assume that the programmer explicitly creates threads, for example using POSIX library, and synchronizations are done explicitly in software. Using SPM APIs, the programmer informs the runtime system about the data objects that are going to be shared between multiple worker threads before spawning them.

2.3.4 Coherence Issues

Since SATT is accessed in parallel with Translation Lookaside Buffer (TLB), for every memory request, there is never any confusion whether the data should be accessed from SPM or cache. Therefore, unlike [6], there is no coherence issue between SPM and cache within each core. To avoid coherence issues when a data is shared between multiple threads, we only keep one copy of each data with all threads accessing the same physical copy. This requires the SATT mappings for shared data to be replicated on all of the cores on which the worker threads are mapped to so that all accesses to shared data are forwarded to the unique copy on SPM. During a `clone` or `fork` system call, SATT mappings for shared data are copied as part of the process execution state along with other necessary data. Worker threads can allocate their own private data on SPM as well, but the assumption is that all shared data objects are allocated on SPM by the boss thread before worker threads are dispatched. Future work could relax this restriction by allowing cores to exchange SATT mappings.

2.3.5 OS/Runtime Support

The SPM manager, as part of the operating system's memory manager, is the core software component of an architecture with software-programmable memories. Any SPM API call from any thread invokes the operating system and this call will be forwarded to the SPM manager to manage the request. Consequently, a resulting research problem is to define intelligent policies to efficiently manage memory resources based on the current needs of the workload.

Figure 2.6 shows the organization of the SPM manager. SPM Manager receives the API calls and optionally some memory related hardware counters and makes decisions about SPM data allocations / deallocations / migrations and also partitioning LM into a cache and SPM for each core. These actions are effected by sending messages to MMUs distributed across the platform.

Repartitioning the local memory of each core requires the SPM manager to detect memory phase changes. Tajik et al. [142] has shown memory phases can be detected at runtime using simple hardware counters. Of course there is an open problem for determining the appropriate size of cache and SPM for each phase at runtime to be addressed in future work.

Static analysis during compilation can be used to classify data for cache/SPM mappings and could help the SPM manager at runtime decide how the local memories should be partitioned

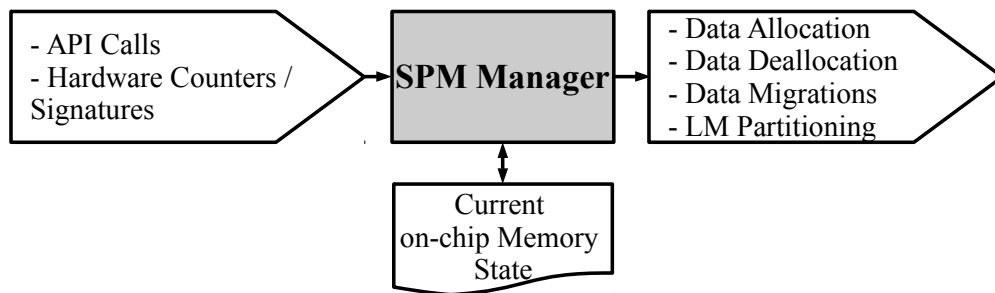


Figure 2.6: SPM manager.

[35].

2.4 Motivational Example

In many scenarios, the amount of data requested by a thread to be allocated on-chip is larger than the capacity of the SPM local to the core that the thread is running on. Therefore, some portion of the data must remain off-chip. In such cases, borrowing physical SPM space from neighboring SPMs could boost the overall performance.

Figure 2.7 shows an example of a platform with four cores, when three threads (T1, T2, and T3) with various memory working set sizes are running on cores C1, C4, and C2 respectively. Previous approaches would restrict each thread to use only the physical SPM local to the core it executes on. However, by sharing the SPM space, T2’s data can be placed not only on C4’s SPM but also on C3’s SPM, which is an idle core, and on C2’s SPM, where T3 has a small working set size and does not need its entire local SPM. At the same time, T1 can also benefit from sharing by using part of C3’s SPM that is not used by T2. The end result is the improvement in the *overall* performance of these three threads.

Figure 2.8 shows a more detailed motivating scenario using a synthetic example where three threads T1, T2, and T3 are sequentially entering a system with four cores (C1, C2, C3, and C4), where every core has a local data SPM. The left-hand side panel shows the state of the system when the local only allocator is used, while the right-hand side panel shows the same state when a local/remote allocator is used. In both panels, time is progressing from left to right. Details of each thread are shown in Table 2.2.

We consider hit latency to be 1 cycle for local hits, 5 cycles for remote hits to account for the network latency, and miss latency to be 100 cycles to account for off-chip memory accesses.

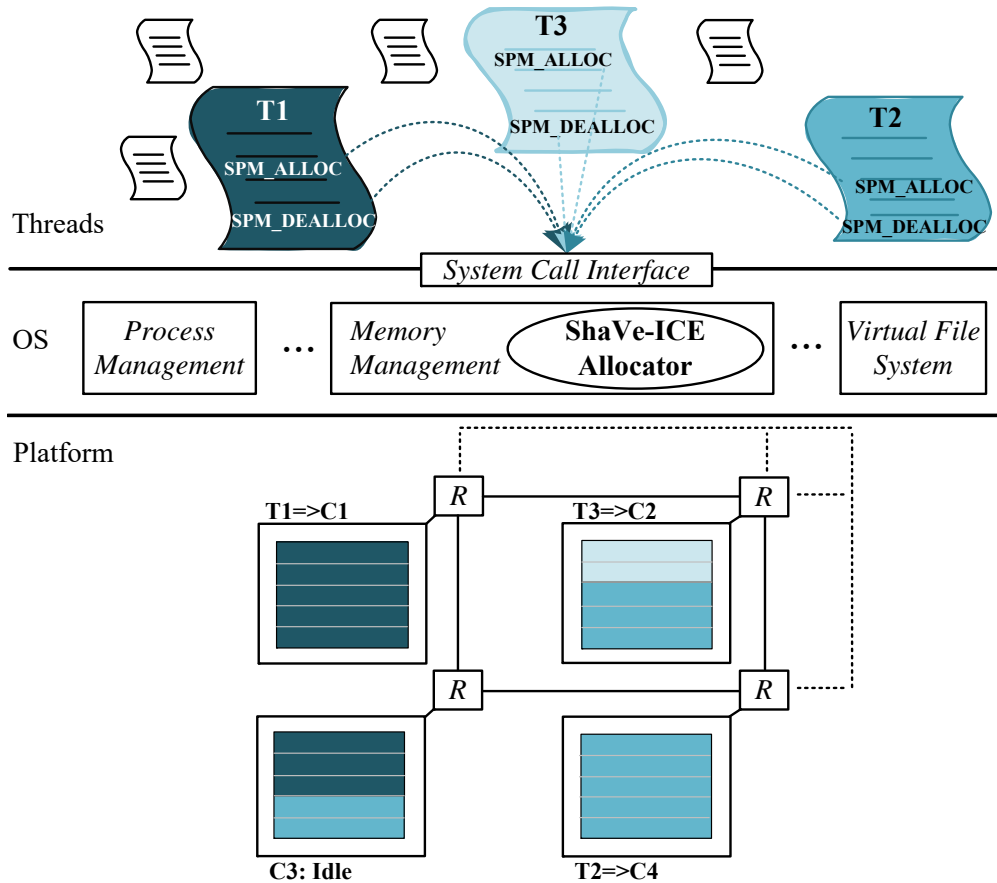


Figure 2.7: ShaVe-ICE allocator is invoked when a thread T_i mapped on core C_j calls any of the SPM APIs. It virtualizes and ultimately shares the entire distributed SPM space between all the concurrently running threads. Dedicated hardware assist enables remote allocations and remote memory accesses.

T1 enters the system in cycle 0 and is scheduled to run on C1. This thread requests 12KB of SPM space, however with local-only allocator, only 8KB of SPM can be given to this thread although C2, C3, and C4 are all idle (Figure 2.8.a). However, if we allow remote allocation on neighboring cores, all 12KB can be granted to T1 (Figure 2.8.a').

Table 2.2: Details of threads in the motivational example of Figure 2.8

Thread	Core	Entrance Cycle	Working Set Size	#Accesses	#Execution Cycles if all accesses are hit with access latency = 1 cycle
T1	C1	0	12KB	18000	60K
T2	C2	20K	20KB	24000	65K
T3	C4	45K	6KB	3000	40K

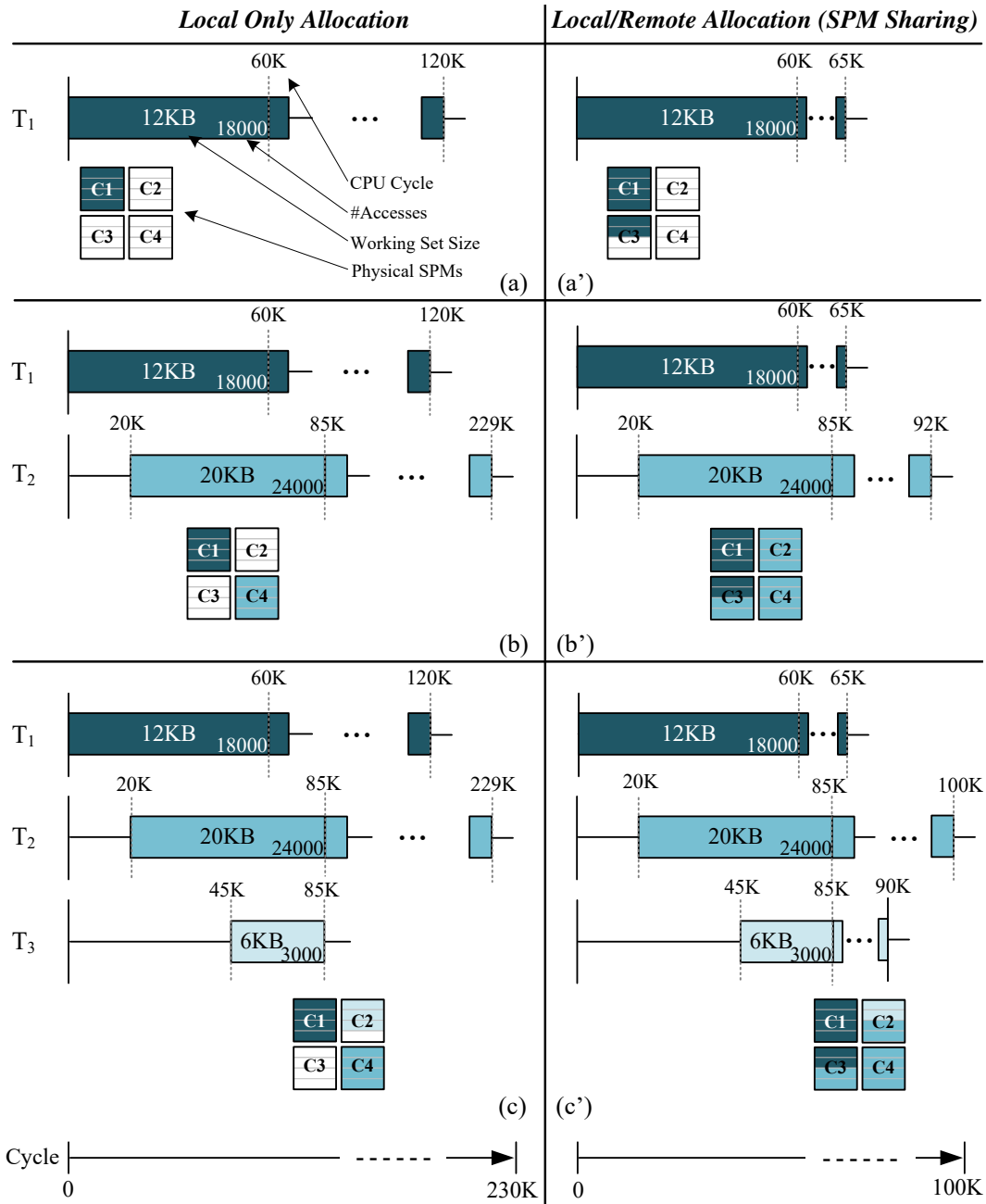


Figure 2.8: An example showing how sharing the entire SPM space improves the overall performance of a 2X2 system with 3 thread running on it. The left-hand side column shows the state of SPM allocations when allocations are limited to the local SPM only; and right-hand side column shows the same scenario when remote allocations are also possible in addition to local allocations. (a)&(a'): only thread T1 is running, (b)&(b'): thread T2 starts execution after 20K cycles while T1 is still running, (c)&(c'): thread T3 starts execution after 25K cycles while both T1 and T2 are still running. Average and maximum execution times among all three threads are improved by about 2X and 2.9X respectively.

T2 gets scheduled on core 4 after 20 kilocycles. This thread needs 20KB of SPM space. While with local-only allocator, a maximum of 8KB can be given to this thread (Figure 2.8.b), remote allocation allows us to accommodate all 20KB by borrowing space from neighboring idle cores namely C3 and C2 (Figure 2.8.b').

Finally, T3 gets scheduled on core 2 after 45 kilocycles. This thread only needs 6KB of SPM space which can be granted with the local-only allocator (Figure 2.8.c). However, as we saw in the previous case, with a local/remote allocator, T2 occupied the entire SPM belonging to core 2. Therefore, as shown in (Figure 2.8.c'), half of the SPM space on core 2 is made available for T3 by evicting some of the pages that were previously allocated to T2.

To summarize, when we are limited to local SPM allocations (left panel of Figure 2.8), all three threads complete their execution after 229 kilocycles, and on average each thread takes about 123 kilocycles to finish, 2.2X more than the ideal case.

But when we virtualize the entire available SPM space and enable all threads to use it (right panel of Figure 2.8), all three threads complete their execution after 100 kilocycles. In this case, each thread takes about 60 kilocycles on average, which is 2X faster than the previous case. The reduction in off-chip memory traffic also results in energy savings.

This simple scenario shows how virtualizing the entire SPM space and allowing threads to compete for memory resources distributed across a chip can help improve the *overall* performance of threads running on manycore systems with SPMs.

2.5 ShaVe-ICE Details

ShaVe-ICE is an operating system implementation to support virtualization and sharing of on-chip SPMs, and includes required architectural support. Software consists of the

SPM allocator (Sections 2.5.1 and 2.5.2) and hardware support includes custom memory management units along with a network protocol (Section 2.5.3).

2.5.1 SPM Allocator

The SPM allocator, as part of the operating system’s memory manager, is the core of the ShaVe-ICE scheme. An SPM API call from any thread invokes the operating system and this call will be forwarded to the ShaVe-ICE’s SPM allocator to make a decision about the request. Currently, we assume all API calls are done by a centralized SPM manager that runs on the same core as the request was made. The allocation is done at the granularity of a page. The allocator aligns the boundaries of an allocation request and also makes sure that a page is not allocated more than once. The allocator, based on its policy (discussed in Sec. 2.5.2), determines the set of actions required and subsequently broadcasts proper SPM management messages throughout the platform to accommodate the request. The requests are served by a centralized manager in a first-come, first-served (FCFS) order and therefore there is no chance for erroneous behavior and/or deadlock in decision makings.

2.5.2 Allocation Policies

ShaVe-ICE’s allocation policies have to balance the memory needs of locally-pinned threads, while allowing for sharing of SPM resources among cores. We present both non-preemptive as well as preemptive heuristics. All policies work in a best effort manner. Their goal is to reduce the average execution time of all threads that are running, possibly degrading some for the benefit of the whole mix. They attempt to map the maximum number of requested pages to on-chip SPMs. Anything that can not be mapped to on-chip SPMs due to capacity limitation, stays in the main memory.

Policy 1: Local Allocator (LA)

```
Input: a new allocation request for P pages from thread running on core[i];  
Output: number of mapped pages M;  
1 R = P;  
2 T = maximum number of contiguous free pages on SPM[i];  
3 while R > 0 and T > 0 do  
4   | allocate min(R,T) pages on SPM[i];  
5   | R -= min(R,T);  
6   | T = maximum number of contiguous free pages on SPM[i];  
7 end  
8 M = P - R;
```

2.5.2.1 Non-preemptive Allocation Heuristics

In non-preemptive allocators, once the data has been allocated on-chip, it stays there until the thread explicitly calls `SPM_DEALLOC()` for that piece of data.

2.5.2.1.1 Local Allocator(LA)

With the *Local* allocator, a thread's allocation request is granted only if there is space available on the local SPM. The pseudo-code of this policy is shown in Policy 1. In each iteration (lines 3...7), the maximum number of free contiguous physical pages on the SPM are found and allocated. Depending on the current allocation status of the local SPM, pages requested by the thread could get allocated contiguously or dispersed throughout the physical SPM. At the end, the number of pages that are successfully mapped is returned.

The Local allocator implements the simplest allocation policy and does not need any hardware support for remote accesses. Since the state-of-the-art techniques for SPM management are limited to the physical SPM local to each core, we will use this policy as the baseline.

Policy 2: Random Remote Allocator (RRA)

```
Input: a new allocation request for P pages from thread running on core[i];  
Output: number of mapped pages M;  
1 R = P - LocalAllocator(P, i);  
2 RL = list of cores;  
3 while R > 0 and RL is not exhausted do  
4   | j = Randomly pick a core from RL;  
5   | R -= LocalAllocator (R, j);  
6 end  
7 M = P - R;
```

2.5.2.1.2 Random Remote Allocator (RRA)

The *Random Remote* allocator grants a thread's allocation requests if there is space available *anywhere* on-chip, and assigns data to random SPMs. This allocator implements the simplest allocation policy for remote SPM allocation. We have intentionally implemented this policy to evaluate the pure advantage of sharing discarding the importance of data placements. The pseudo-code of this policy is shown in Policy 2. It initially behaves like the local allocator trying to allocate as many pages as possible on the local SPM. If there are more pages to allocate, in each iteration (lines 3...6), a core is chosen randomly and the maximum number of free pages on its SPM is allocated for the thread's request.

2.5.2.1.3 Closest Neighborhood Allocator (CNA)

With the *Closest Neighborhood* allocator, the *nearest* available SPM slots are allocated for any allocation request. The pseudo-code of this policy is shown in Policy 3.

The order in which the neighboring SPMs are considered for data allocation (lines 2...6) is based on the hop distance from the core that issued the allocation request. Only cores that are within T radius of the home core will be considered. Figure 2.9 shows one example where the white core in the middle issues an allocation request. First, the allocator searches for free physical pages in the local SPM (hop distance=0). Next candidates are the SPMs distanced within one hop from the core requesting allocation (labeled 1 through 4). If more SPM pages

Policy 3: Closest Neighborhood Allocator (CNA)

Input: a new allocation request for P pages from thread running on core $[i]$,
hop-threshold = T ;

Output: number of mapped pages M ;

```

1  $R = P - \text{LocalAllocator}(P, i)$ ;
2  $DL = \text{list of cores within } T \text{ radius of core}[i] \text{ sorted based on distance from that core}$ ;
3 while  $R > 0$  and  $DL$  is not exhausted do
4   |  $j = \text{next element in } DL$ ;
5   |  $R -= \text{LocalAllocator}(R, j)$ ;
6 end
7  $M = P - R$ ;

```

are needed, the allocator will search the SPMs within hop distance two (labeled 5 through 12). And this process continues until all pages are allocated or all candidates are exhausted.

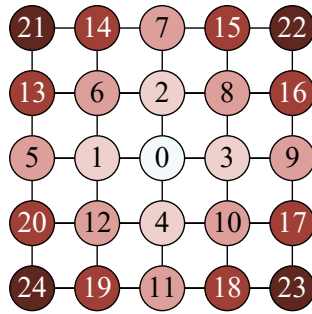


Figure 2.9: Hop-distance-based search for empty SPM space in Closest Neighborhood allocation policy.

2.5.2.2 Preemptive Allocation Heuristics

Preemptive allocation heuristics may allow for more efficient sharing of memory resources, but could affect previously allocated data by (1) migrating it to a different place on-chip or (2) evicting it to main memory.

2.5.2.2.1 Closest Neighborhood with Guaranteed Local Share Allocator (CNGLSA)

This heuristic is a modified version of the Closest Neighborhood allocator that finds the

nearest available SPM slot for any allocation request. However, it also guarantees that a predefined percentage of each SPM is allocated for requests from the thread running on that SPM's local core, *should that thread need it*. The pseudo-code of this policy is shown in Policy 4. Note that this policy does not reserve the SPM ahead of time, rather it frees up the required SPM space by relocating the data from there to a different SPM or to off-chip memory.

With this policy, if a new thread is assigned to an idle core while that core's SPM was already allocated for another thread, all or a portion of that SPM will be relocated/deallocated in order to be used by the new thread. Currently, we support two ways of determining the page(s) that should be relocated (line 4): (1) least recently accessed page, (2) least frequently used page. The assumption is that there are hardware counters that can keep track of these statistics at runtime. Once the page is selected, we attempt to relocate that page to a free SPM nearest to its owner. If the entire on-chip SPM space within a certain hop distance is

Policy 4: Closest Neighborhood with Guaranteed Local Share Allocator (CNGLSA)	
	Input: a new allocation request for P pages from thread running on core[i], hop-threshold = T, guaranteed local share = X%;
	Output: number of mapped pages M;
1	R = P - LocalAllocator(P, i);
2	S = local SPM share;
3	while R > 0 and S < X do
4	EP = select a page which doesn't belong to the thread running on core[i] to be evicted;
5	core[j] = owner of EP;
6	relocate EP to the closest free SPM to core[j] or deallocate to off-chip-memory if no SPM space is available;
7	allocate one page on SPM[i] for the new request;
8	R -= 1;
9	S = local SPM share;
10	end
11	if R > 0 then
12	R -= ClosestNeighborhoodAllocator(R,i);
13	end
14	M = P - R;

utilized, we evict that page to the main memory.

2.5.3 Hardware Support

Many of the previous allocation policies require some hardware support to manage the distributed memory mappings as well as to handle remote accesses via the NoC. This subsection explains all the required hardware assists.

2.5.3.1 Distributed Memory Management Units

MMUs are core on-chip components enabling all the memory-related communications. A MMU sends/receives all memory-related network messages from/to their local SPM. They are distributed throughout the chip, one MMU per core. Every memory request from the CPU is handled by the MMU. Each MMU has its own SPM Address Translation Table (SATT) which holds all the virtual to physical address translations for the thread on that core. Each entry of an SATT (Figure 2.10) holds five kinds of information: (1) the virtual page address, (2) the id of the core hosting that page, (3) the physical address on host SPM, (4) a bit flagging if the entry is a valid entry, and (5) a flag bit indicating if the information in that row is shared data or private data. If no mapping exists on SATT for the memory location requested by CPU, the data cannot be found on-chip; the address translation will be done through the Translation Lookaside Buffer (TLB) and the request is forwarded to off-chip memory.

valid	virtual page	host core ID	physical SPM address	shared data
-------	--------------	--------------	----------------------	-------------

Figure 2.10: SATT entry

2.5.3.2 Network Protocol

Table 2.3 lists all the network messages required to implement the SPM-related communications between MMUs via the NoC. There are 6 kinds of request messages and five types of response messages. SPMReq_READ and SPMReq_WRITE are used for remote memory read and write accesses which are respectively responded with SPMResp_DATA and SPMResp_WRITE_ACK. SPMReq_ALLOC and SPMReq_DEALLOC are also used for allocating and deallocating a virtual page and are both responded with SPMResp_GOV_ACK. In certain scenarios, allocating a page for a thread requires relocating a page from another thread. In such cases, the core that has to relocate its page sends a SPMReq_RELOCATION_READ to the current host of that page. When the data is read from SPM, the current host will forward the data with a SPMReq_RELOCATION_WRITE to the future host of that page while sending back a SPMResp_RELOCATION_HALFWAY to the owner core. The owner then can signal the core who wants to allocate a new page that the space is freed up. Future host node receives the request and after copying the data on its SPM sends a SPMResp_RELOCATION_DONE to the owner core so it can resume its execution.

Table 2.3: SPM-related network messages in ShaVe-ICE

SPM Request Messages		SPM Response Messages	
Type	Purpose	Type	Purpose
SPMReq_READ	Request for a data read	SPMResp_DATA	Data for read requests
SPMReq_WRITE	Request for a data write	SPMResp_WRITE_ACK	Acknowledgment for write requests
SPMReq_ALLOC	Request for allocating a page on a physical SPM	SPMResp_GOV_ACK	Acknowledgment for ALLOC/DEALLOC requests
SPMReq_DEALLOC	Request for deallocating a page from a physical SPM		
SPMReq_RELOCATION_READ	Request for read part of an on-chip page migration	SPMResp_RELOCATION_HALFWAY	Used to notify a MMU that data has been relocated from specific SPM slots and those slots are now free
SPMReq_RELOCATION_WRITE	Request for write part of an on-chip page migration	SPMResp_RELOCATION_DONE	Used to signal the end of a relocation request

2.5.4 Data Movements in ShaVe-ICE

Three types of events encompass all possible scenarios for managing SPM data in ShaVe-ICE: allocation/deallocation, access, and migration.

2.5.4.1 Allocations/Deallocations

Allocations and deallocations are initiated via explicit API calls in the application. An allocation request with its related information (address, size) is sent to the SPM allocator, which holds the state of all on-chip memory and can determine whether or not the request will be granted. If the SPM allocator can grant the allocation request it sends one or more message(s) back to the requesting core. Every message contains a number of allocation information: number of pages granted, core hosting the allocated SPM space, base physical address on the target SPM, etc. The requesting node updates its SATT entry for the virtual page(s), it communicates with the host core and finally the host core's MMU initiates DMA transfers to fulfill the request.

Deallocation follows a similar flow: upon receiving the deallocation request, the SPM allocator notifies the requesting node so it can invalidate the associated SATT entry, and also notifies the host node so that it can invalidate the SPM space and initiate a write-back to main memory.

2.5.4.2 Accesses

Every memory read/write request is initially forwarded to the local MMU. MMU issues a SATT lookup in parallel with the TLB. Since SATT stores all the SPM mappings for its host core, this lookup yields the location of the object. If the look-up is a miss, the data is stored in the cache hierarchy or main memory. If the look-up is a hit, the data exists somewhere on

the distributed physical SPM – either the local SPM or any of the remote SPMs.

If the page containing the requested data exists on the local SPM, the MMU provides the physical SPM address and forwards the request to the local SPM, which returns or updates the data.

If the page is on a remote SPM, the MMU initiates a request (SPMReq_READ or SPMReq_WRITE) for the remote MMU on the node that holds the data while the requesting CPU stalls. Upon receiving the request from the NoC, the remote MMU fetches or updates the data, and returns a message (SPMResp_DATA or SPMResp_WRITE_ACK) to the requester. Finally, the requesting MMU notifies its local core upon receiving the response and execution continues.

2.5.4.3 Data Migration

Under certain conditions, we may need to relocate a page from one SPM to another SPM or evict it to main memory. This is usually needed in order to free space for a thread that has more priority to use a physical SPM in the case of preemptive allocation policies. Migrations are initiated by the allocator, and carried out by network messages (SPM_Req_RELOCATION_READ, SPM_Req_RELOCATION_WRITE, SPM_Resp_RELOCATION_HALFWAY, SPM_Resp_RELOCATION_DONE) exchanged between the affected MMUs; namely: owner of that page, current host of that page, and future host of that page. During migration, the thread whose data is being migrated is stalled to make the execution safe.

2.6 SPM Sharing Evaluations

We conducted experiments to evaluate the capability of ShaVe-ICE for improving the overall performance and power consumption of the memory sub-system for a SPM-based platform by sharing the distributed SPM space.

2.6.1 Experimental Setup

We prototyped the ShaVe-ICE scheme in the gem5 architectural simulator [29]. We extended gem5 by adding the SPM, MMU, and SATT components; defining a new network protocol for SPM-related communications; and adding support for SPM API calls through pseudo-instructions. We ran our experiments in the system emulation (SE) mode of gem5 using in-order alpha cores to configure mesh NoCs using the gem5’s simple network to model the contention delay. Every SPM API call invokes the ShaVe-ICE manager implemented inside the simulator to emulate the behavior of a real operating system.

In the following experiments, all threads begin their execution at the same time and we set the hop distance threshold to 8.

We used Noxim [34] to estimate the energy consumed within the NoC and CACTI [111] to estimate the SPM access energy. DRAM access energy is estimated using the results from [117].

2.6.2 Remote vs. Off-chip Access Latencies

While on-chip accesses tend to be faster than off-chip accesses, if the hop distance between the home core and the host core becomes too long, which is quite possible in larger platforms, the network delay could overcome the benefit of remote allocations. In the most extreme cases,

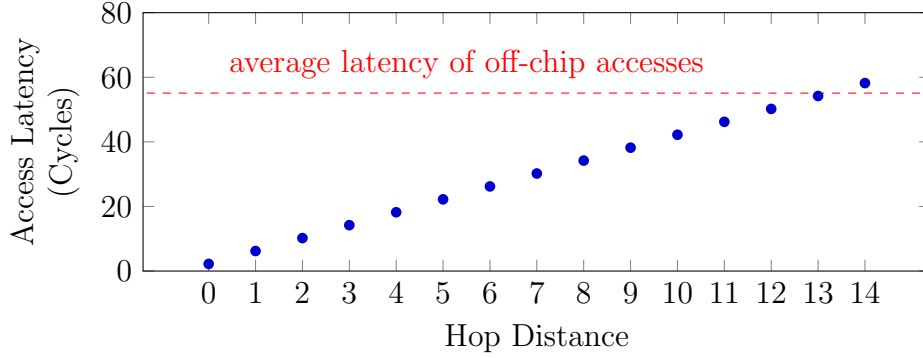


Figure 2.11: Average access latency for various hop distances.

an on-chip memory access may become more costly than an off-chip access. The exact hop distance that could result in degraded performance is dependent on the implementation of the network as well as the latency of off-chip DRAM. We conducted experiments to measure the pure effect of network delay on remote accesses in a 8X8 mesh NoC. For that, we run a thread on corner of the mesh (i.e., core (0,0)) accessing a 4KB stack array within a loop performing some basic arithmetic operations on elements of that array. All other cores are idle. In 15 different runs, we allocate this 4KB array on all 15 possible hop distances from the home core (i.e., 0 to 14). In another run, we allocate that array on the off-chip DRAM.

Figure 2.11 shows the average memory access latency of this thread for different hop distances. The dashed red line shows the average access latency when the data is accessed from off-chip DRAM. On average, every hop adds 4 cycles to the latency. After 13 hops, the latency experienced by the thread to access on-chip SPM will exceed the latency of off-chip DRAM. This analysis is done assuming there is no other network traffics generated by other cores and therefore a more conservative threshold has to be selected. In our experiments we set the hop distance threshold to 10^2 .

²This analysis was done just to highlight the pure effect of the network on the latency of remote access. All the comparisons in Section 2.6 are done with the network traffic being generated by multiple threads running concurrently.

2.6.3 Memory Microbenchmark

Since our goal is to evaluate the efficiency of different allocation policies, we generated synthetic threads using an in-house configurable memory microbenchmark (mem-ubench) generator designed to stress the memory resources with different intensities (Figure 2.12). This microbenchmark has three phases (prologue, body, epilogue) that are repeatedly executed and can be independently adjusted in order to make them more compute-bound or memory-bound. The prologue and epilogue consist of mainly regular and random memory operations respectively; while the body consists mainly of compute operations. Each function operates on three arrays: two globally allocated arrays, and a third local to the function. This way, we can exercise many diverse scenarios using the coarse-grained intensity setting in combination with the finer-grained duration settings.

The list of configuration parameters for mem-ubench is:

- length: This parameter controls the length of execution.
- working-set-size: This parameter sets the amount of data that the benchmark allocates and accesses.
- mem/comp-intensity: This parameter defines the ratio of memory operations to compute operations for all phases.
- prologue-duration: This parameter controls the relative duration of the first phase of execution.
- body-duration: This parameter controls the relative duration of the second phase of execution.
- epilogue-duration: This parameter controls the relative duration of the third phase of execution.

In this early investigation of SPM-based manycore system, our goal was to explore the key factors that could affect the performance and power of such systems. Towards this end, we

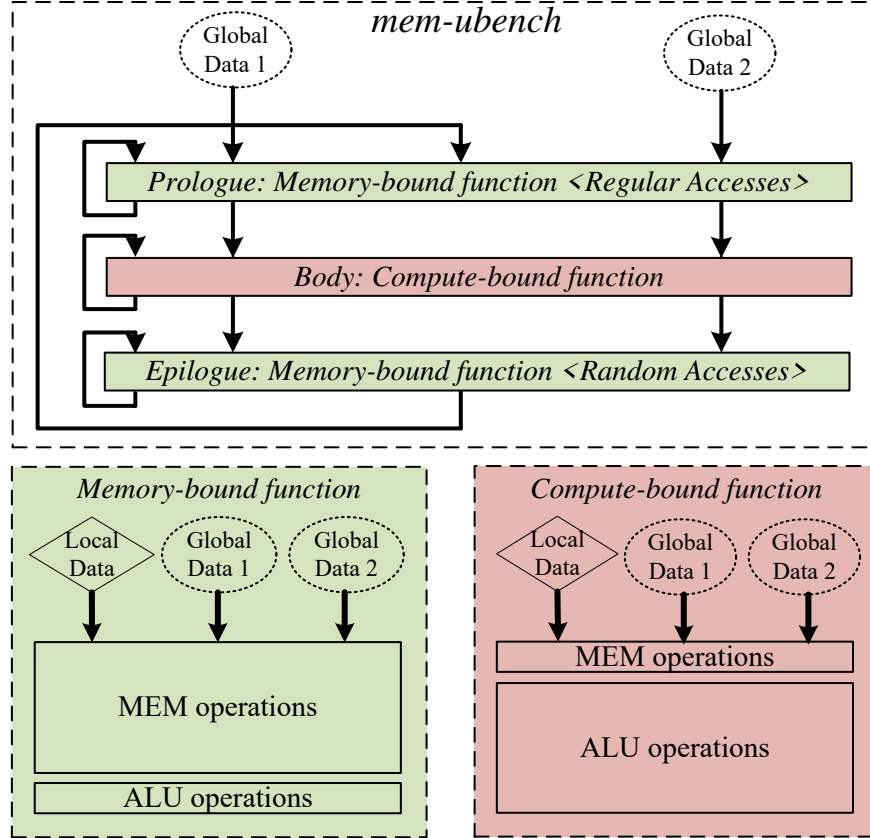


Figure 2.12: Overall flow of *mem-ubench* execution: each phase (prologue, body, epilogue) is either memory or compute intensive to a configurable degree. Operations are performed on three data arrays of configurable size with both regular (strided) and randomized access patterns. The number of executions of each phase as well as the entire 3-phase loop are configurable.

use only combinations of the memory microbenchmark that stimulate specific behaviors to understand these key factors and relate them to the overall behavior of the system. Therefore, we avoided using all combinations of configurations that could result in randomly mixing different kinds of microbenchmarks. In particular, we developed configurations to evaluate the following policies: (1) Local Only, (2) Random Remote, (3) Closest Neighborhood, (4) Closest Neighborhood with Local Reservation. In each policy comparison, we highlight a specific opportunity that is exploited to improve performance/power. For each policy comparison we combined specific configuration(s) of the microbenchmark to exacerbate the advantage of the superior policy, and allowed us to isolate the sources of improvement so as to more clearly explain them. The following two subsections present the result of our performance

and energy evaluations.

2.6.4 Performance Comparisons: ShaVe-ICE vs. Software Cache

In the first set of experiments, we compare ShaVe-ICE with a software cache. A software cache consists of a simple memory array (similar to SPM) and a software system that is capable of automatically managing that memory to behave similar to hardware-managed cache. Instead of using specially-designed hardware for cache management, a software cache uses general-purpose instructions.

Here, we compare the performance of a system using ShaVe-ICE with a similar system that uses software caching. The software cache has the same size as the SPMs in ShaVe-ICE. It implements a direct-mapped cache. The hit and miss latencies of this cache are 11 and 430 cycles respectively. These latencies account for the execution times of lookup and the miss handling routines and are extracted from [118]. A SPM access which hits the local SPM takes two cycles: one cycle to translate the virtual address to physical SPM address and one cycle to access the SRAM array.

We consider two cases. In both cases, we configure a 5X5 mesh-based manycore platform with a utilization of 50%: Case A) The size of the SPM and software cache is larger than the thread's working-set, therefore ShaVe-ICE allocates every memory object on the local SPMs and the software cache has a very low miss rate (Figure 2.13). The address translation for hit accesses in a software cache has to be done in software as opposed to the address translation table (SATT in Figure 2.5) which is used in ShaVe-ICE. Because of that, although most accesses are hit in both cases, the thread's performance with ShaVe-ICE is significantly better compared to a system that uses software cache. Case B) The working-set-size of some threads is larger than local SPM and the software cache (miss rate = $\sim 15\%$) and some threads' working-set are smaller than the local SPM and the software cache (Figure 2.13).

Figure 2.13 shows the improvements in the average and maximum execution time among all running threads for both cases. More improvements are observed in case A compared to case B, because in case A, ShaVe-ICE is allocating all data objects on the local SPM and therefore hit accesses are faster.

2.6.5 Performance Comparisons: ShaVe-ICE Policies

In the second set of experiments, we evaluate various ShaVe-ICE policies with a workload composed of different mem-ubench configurations. In all of the following experiments, we consider an 8X8 mesh-based manycore with equally distributed physical SPMs. To explore the design space, we vary local SPM size from 4KB to 16KB and core utilization from 25% to 100%. We consider four different scenarios to show the benefits of ShaVe-ICE policies under various conditions. Scenario 1 evaluates ShaVe-ICE in the face of core underutilization. Scenario 2 evaluates ShaVe-ICE when threads have different working set sizes. Scenario 3 explores the effect of reducing network traffic for remote memory accesses. Scenario 4 explores scenarios that guarantee a share of each SPM for its locally-pinned thread, thereby improving performance in some cases.

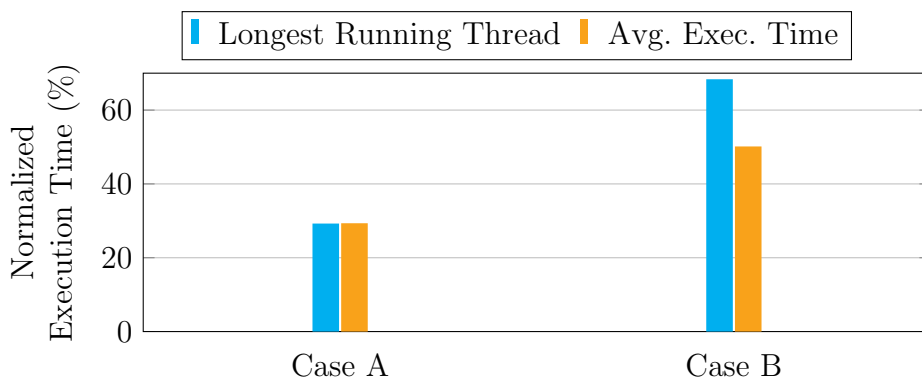


Figure 2.13: ShaVe-ICE vs. software cache – Case A: thread’s working-set is smaller than software cache and SPM, Case B: thread’s working-set is larger than software cache and SPM. Lower values are better.

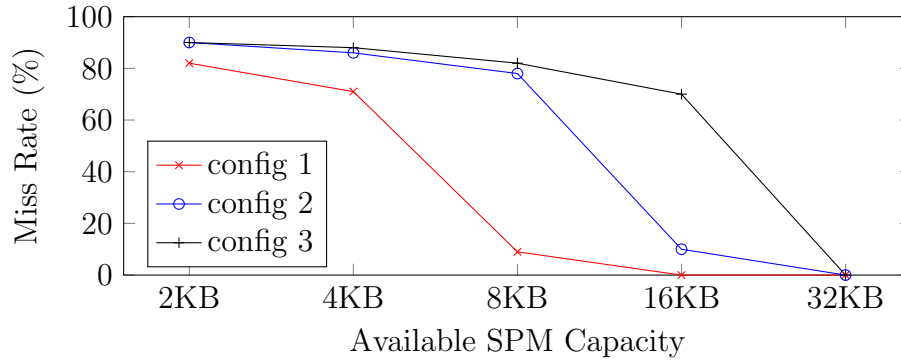


Figure 2.14: Miss rate of various mem-ubench configurations for different available SPM capacities. Miss rate does not have a linear relationship with SPM capacity since not all the data is of the same importance.

For these evaluations, we use three configurations of mem-ubench with different working set sizes. Figure 2.14 shows the miss rate of each of these configurations based on the available SPM capacity. In all three configurations, around 28% of executed instructions are memory operations. These three configurations are chosen to exercise different memory workloads by combining various working set sizes and different sensitivities to the available SPM size. Note that the improvement in miss rate does not have a linear relationship with the SPM size.

For all core utilizations, thread mappings are done in a way that the workload is evenly distributed across the platform. This also holds true when a mix of two types of mem-bench are used to compose the workload.

For the following studies, we discuss the interesting trends observed in the results. However, we note of course that due to the dynamism in multiple dimensions (e.g., type of mem-ubench threads, importance of data objects, arrival time of requests, etc.) it is difficult to identify the cause of specific trends over all cases due to their complex interactions.

For instance, the dynamism is complicated by different memory objects accessed with different levels of importance (i.e., number of accesses to each data object). In our mem-ubench this is the order of importance of the data objects: stack-data ζ global-heap-data ζ local-heap-data. However, the fact that stack-data is more important than other data bjects does not mean

that if the SPM space is limited, only that piece of data will be allocated. This is because allocations requests are served on a best effort manner (first-come-first-served basis) without knowing the future demands. In this work, we have no support for prioritizing more important data over less accesses data. We have stated in our future work that we are working on a light-weight monitoring of accesses through some counters to make sense of the importance of their data objects. The fact that not all data is of the same importance is also corroborated in 2.21. The miss rate drop with additional SPM capacity is not constant as the available SPM space increases. Due to the varying arrival time of the allocation deallocation request and the system-level status of allocations, one policy may allocate more important data when the SPM size is smaller and/or utilization is higher, while the other policy may allocate a less important data with a larger SPM size and/or lower utilization.

2.6.5.1 Scenario 1: Core Underutilization

The first scenario we examine is when the number of threads running on the platform is smaller than the number of cores (i.e., core underutilization). In this case not all CPUs are active, therefore some cores are not occupied and traditional SPM management techniques result in idle cores' SPM space not being used. For this experiment, we replicate identical config 2 mem-ubenchs to generate the workload.

Figure 2.15 shows the average execution time of all threads in the workload when sharing is enabled using the simplest remote allocation (i.e., RRA) normalized to the same metric when threads can only use their local SPM (i.e., LA). This ratio is reported for three different core utilizations from 25% to 75%. For SPM sizes of 16KB, improvements are minimal to none because 16KB can fit most of the config 2's working set according to Figure 2.14. In all cases, when SPMs are sized smaller, significant improvement in execution time is observed. This is due to SPM sharing – when SPMs are smaller than a thread's working set, sharing of remote SPMs prevents off-chip memory accesses that are otherwise required for allocation

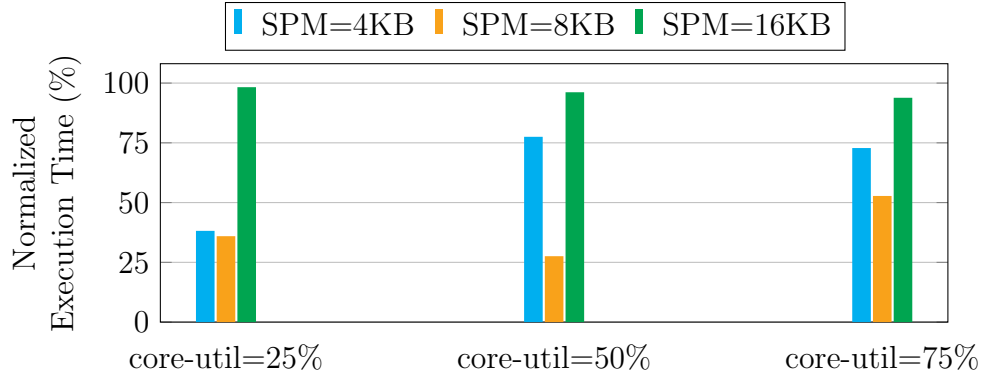


Figure 2.15: Scenario 1: Average execution time among all threads in an 8x8 system - RRA normalized to LA. Core underutilization provides the opportunity to RRA to allocate more SPM space for the threads that have large working set size.

policies that only allow threads to use their local SPMs.

Highest improvements are observed for 8KB SPMs because the miss rate drops significantly when the available capacity becomes larger than 8KB. This is justified by observing the sharp decline in the miss rate for config 2 when moving from 8KB to 16KB SPM space. The maximum improvement happens when SPM is 8KB and the utilization is 50%. This is because at this utilization and with the local allocator, 32 threads have most of their working set off-chip. Therefore, there is a high traffic on the off-chip memory side further increasing the off-chip access latency that each thread experiences. With remote random allocator, all threads get most of their working set allocated on-chip, reducing the number of off-chip accesses which in turn reduced the off-chip access latency as well. At 75% core utilization, not all thread gets their working set allocated, therefore improvement is smaller.

On the other hand, when sharing is enabled with 4KB SPMs, in lower utilizations (e.g., 25%) a lot of nearby SPM space is available to compensate for the shortage of local SPM space. The improvements are expected to be slightly lower than when SPMs are 8KB. With 8KB SPMs, the miss rate of config2 mem-ubench is about to significantly drop if it gets remote SPM space. Remote SPMs are also in hop distance one so the execution time improves significantly. With 4KB SPMs, we require a lot of remote SPMs for the miss rate to drop significantly. In this

case more accesses will be remote with higher average latency. As expected, lower utilizations generally result in larger improvements because more remote SPM space is available for threads.

2.6.5.2 Scenario 2: Variation in Memory Working Set Size

Next, we examine a scenario that unlike scenario 1, there is a variation in the memory working set size of threads concurrently running on the platform. This would result in some cores not utilizing their entire SPM space, and can therefore lend SPM space to other cores in need even if the core utilization is high. We examine this scenario under different core utilizations. For this experiment, we use a mix of config 1 and config 3 mem-ubenchs where config 3 has a significantly larger working set size compared to config 1. However, both types of threads have the same memory utilization (i.e. percentage of dynamic instructions that were memory accesses) equal to 28%.

Figure 2.16 shows the average execution time of different threads in the workload when sharing is enabled using the simplest remote allocator (i.e., RRA) normalized to the same metric when threads can only use their local SPM (i.e., LA).

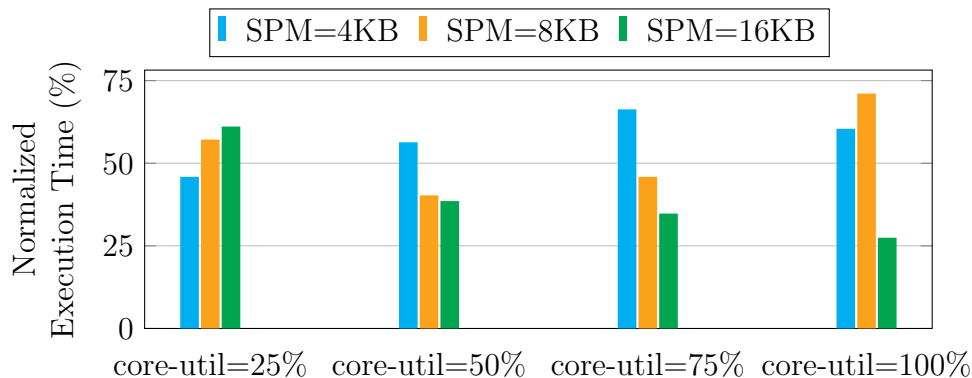


Figure 2.16: Scenario 2: Average execution time among all threads in an 8x8 system - RRA normalized to LA: Threads with small working set size provide the opportunity to RRA to utilize the SPM space unused by the locally-pinned thread in order to allocate more SPM space for the threads that have large working set size.

The trend of execution time ratios with 16KB SPMs is noteworthy. Improvements increase at higher utilizations. With 16KB, all config 1 threads have enough space but config 3 threads are close to their tipping point. On the other hand, with the local allocator, at higher utilizations a significant memory traffic is generated which degrades the performance of config 3 threads dramatically because off-chip accesses take a long time to complete. The random remote allocator allocates more space for config 3 threads, alleviating both issues.

The execution time reductions when even all cores are utilized shows the benefit of SPM sharing for workloads with diverse memory requirements. This is due to the fact that although half of the threads require more SPM space than what is locally available, the other half do not fully occupy their local SPMs and can lend it to more demanding threads.

2.6.5.3 Scenario 3: Reducing Network Traffic

The third set of experiments are devised to show the effect of hop distance on memory latency when borrowing space from remote cores. Like scenario 2, we use a mix of config 1 and config 3 mem-ubenchs for this experiment.

Figure 2.17 shows the average execution time of the workload for the neighborhood allocator normalized to the random remote allocator. Improvements are minimal for 4KB SPM since a lot of them are already occupied by their locally-pinned thread especially when core utilization is higher than 25%. At the low core utilization of 25% – because there is more available SPM space in the pool of free SPMs – the execution time will be reduced more significantly when the remote allocations are done as close as possible to the owner core compared to when they are done at random hop distances.

The simple neighborhood allocator performs better than the random remote allocator in almost all cases except one. The execution time becomes higher at the highest utilization and the smallest SPM size because of the config 3 threads having a higher probability of

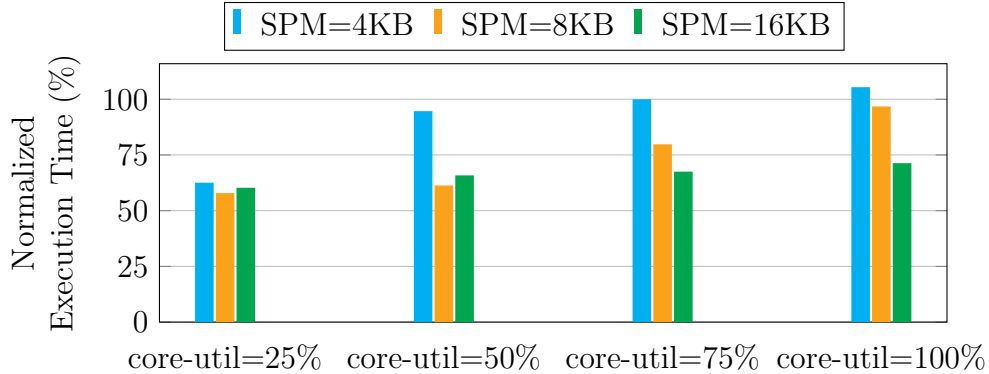


Figure 2.17: Scenario 3: Average execution time among all threads in an 8x8 system - CNA normalized to RRA: CNA allocates remote pages as close as possible to the owner core resulting in reduced network traffic, faster access latency and ultimately better overall performance compared to RRA.

occupying the SPM space that otherwise would have belonged to config 1. As shown in Figure 2.14, config 1 threads can benefit more of 4KB of SPM compared to config 3 threads. Because of the way that two types of threads are evenly distributed, every config 3 thread has four config 1 threads within hop distance of 1. Therefore with the closest neighborhood allocator, there is a 100% probability that a config 3 thread fills in the SPM of one of its config 1 neighbors should they have free space. With the random remote allocator there is a 50% chance that a config 3 fill in a config 1's local SPM.

2.6.5.4 Scenario 4: Guaranteeing SPM Share for Locally-pinned Thread

The neighborhood allocator looks for available SPM space with the shortest hop distance. With this policy, it is possible that a core occupies the entire SPM that is local to another core while that core itself needs it. In the fourth set of experiments, we evaluate if guaranteeing a portion of each SPM for its local thread helps improve the overall performance. Again, we use a mix of config 1 and config 3 mem-ubenchs for this experiment. The local SPM share is set to 100%.

The top part of Figure 2.18 shows the difference in average local hit ratio of different threads

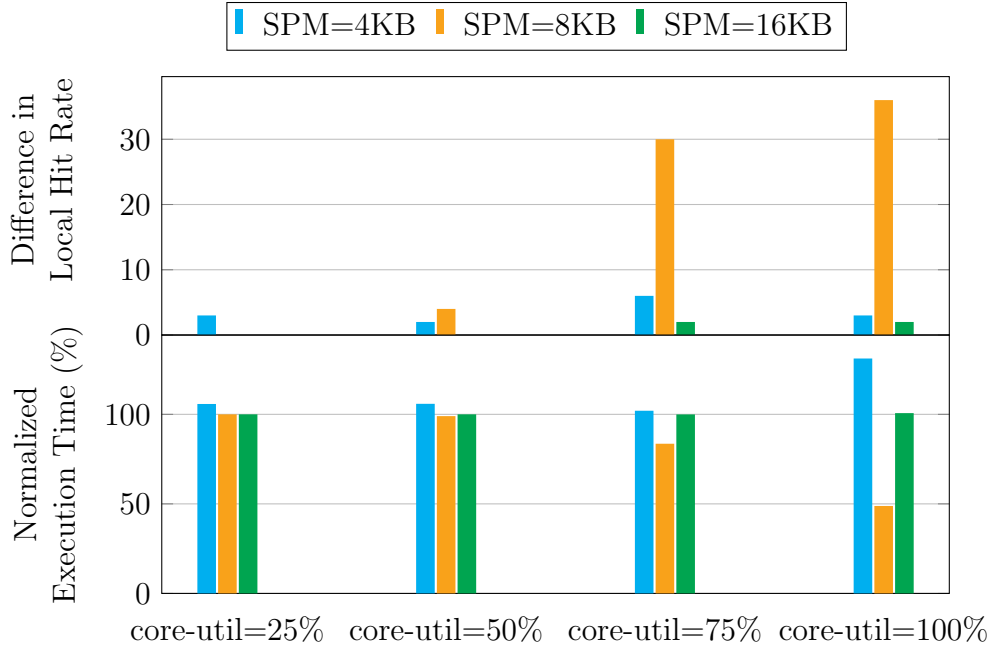


Figure 2.18: Scenario 4: Difference in the local hit ratio - CNGLSA-CNA and average execution time among all threads in an 8x8 system - CNGLSA normalized to CNA: CNGLSA returns the remotely allocated SPM space to the locally-pinned thread should that thread needs it. In some cases, this results in improved local hit ratio and decreased average memory access latency compared to CNA.

running on a system with the neighborhood allocator with and without guaranteed local share. The improvement in local hit rate as utilization increases indicates that this improvement is due to reduced remote SPM accesses. Because the threads are intentionally mapped as sparsely as possible, at lower utilizations there are very few conflicts between neighboring cores competing for SPM space, and therefore local hit rates are nearly identical with both policies. However, this increase in local hit rate does not always turn into execution time reduction. Sometimes, the overhead associated with on-chip data relocation might offset the benefit of more local hits if the data that is being allocated locally is not accessed frequently enough.

The lower part of Figure 2.18 shows the corresponding normalized execution times and as it can be seen only in two cases a noticeable execution time reduction is the gained. The large improvement when SPM size is 8KB at higher utilization is due to config 1 threads. At

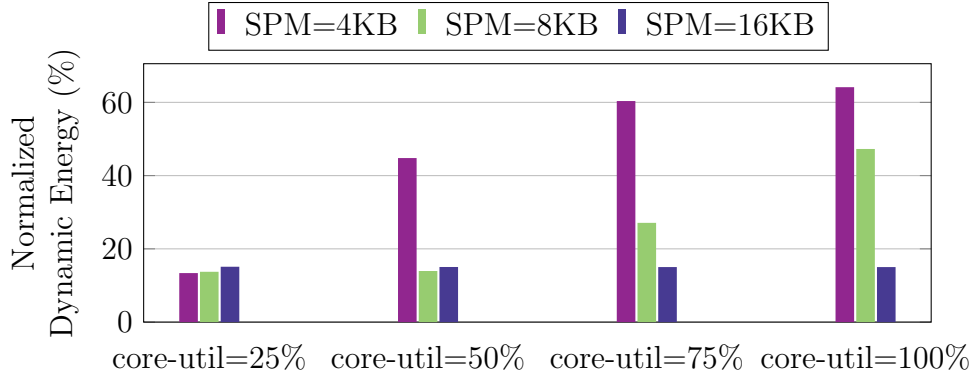


Figure 2.19: Dynamic energy in memory subsystem and network: CNA normalized to LA.

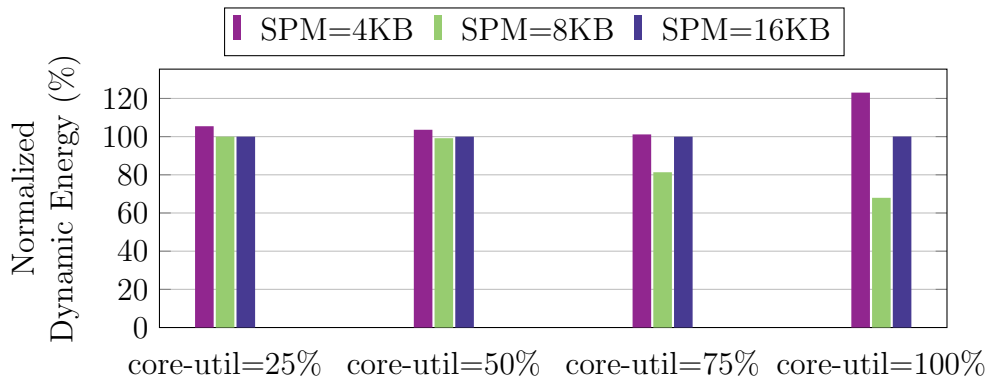


Figure 2.20: Dynamic energy in memory subsystem and network: CNGLSA normalized to CNA.

that SPM size and with the neighborhood allocator, the neighboring config 3 thread(s) that need more than 8KB of SPM space, immediately fill up the entire platform leaving nothing for the config 1 threads. But when neighborhood allocator with guaranteed local share is used, config 1 threads will get all 8KB SPM space that they need which results in dramatic reduction of their execution time.

2.6.6 Energy Comparisons: ShaVe-ICE Policies

Access to off-chip memory is an order of magnitude more costly than on-chip accesses in terms of energy consumption. Although ShaVe-ICE primarily targets improving the overall performance, it can also help reduce the total energy consumed by the memory subsystem.

Figure 2.19 shows total dynamic energy consumed for SPM accesses, off-chip DRAM accesses, and the NoC when distributed SPMs are shared using the closest neighborhood policy normalized to the same metric when allocations are limited to local SPMs only. As expected, the energy savings trend is similar to the performance improvement, as both are primarily a result of reducing off-chip memory accesses.

A similar comparison is made in Figure 2.20 between the neighborhood allocator with and without the use of guaranteed local share. Similar to the performance comparisons in Figure 2.18, only when the benefit of accessing the data locally is higher the cost of migration energy savings can be obtained.

2.6.7 Experiments with Real Workload Mixes

In the last set of experiments, we configure a 4x4 system and run a number of real benchmarks on the platform under different core utilizations. Every core’s SPM is 32 kilobytes. The benchmarks and their respective sources are listed in Table 2.4.

In order to annotate the source code of these benchmarks with ShaVe-ICE APIs: (1) We run the benchmark on gem5 to collect memory traces. These memory traces are generated independent of the memory hierarchy. In the same run we collect the cycles each function was called and when it returned. We also collect the virtual address boundaries of various data

Table 2.4: List of benchmarks for ShaVe-ICE experiments

Benchmark	Acronym	Source
fast fourier transform	FFT	[63]
huffman encoding	HUF	[1]
wikisort	WS	[1]
nbody	NB	[2]
susan-cornering	SU-C	[63]
susan-edging	SU-E	[63]
susan-smoothing	SU-S	[63]

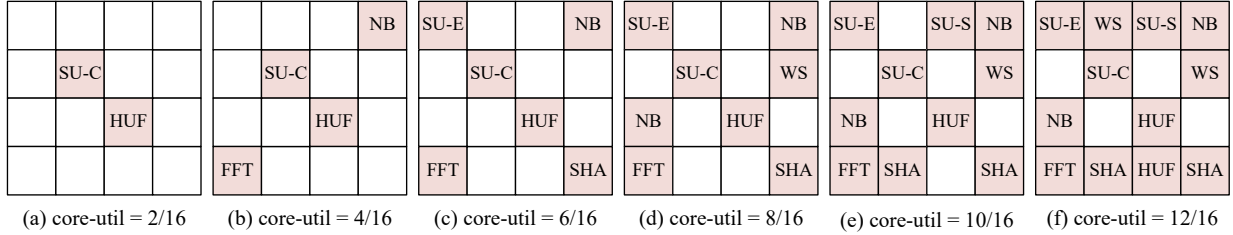


Figure 2.21: Workload mixes with various core utilizations in a 4X4 platform.

objects (function stacks, heap objects, etc) (2) We feed the trace along with this information into our memory trace analyzer which analyzes the trace knowing each access in the memory trace maps to which data object tagged to its corresponding function. The report generated by the analyzer is finally used to add annotations for a subset of the most frequently accessed data objects within proper places in the source code.

We evaluate this system under different core utilizations, various working set sizes, and with two local and closest neighborhood with guaranteed local share allocators. Figure 2.21 shows all the mixes of benchmarks running at the same time. We start from having only two benchmarks running and in each step we add two more benchmarks to the workload mix until the core utilization becomes 75% (i.e., 12/16).

Figure 2.22 shows the execution time of workload with closest neighborhood with guaranteed local share allocator normalized to the execution time with local allocator which is our baseline. It reports four different values, namely: (1)normalized longest execution time among all threads, (2)normalized average execution time among all threads (3)normalized execution time of the thread benefiting the most and (4)normalized execution time of the thread benefiting the least.

The longest running thread in all first five mixes ((a) through (e)) is HUF where its execution time is reduced to around 50% in all cases. In mix (f) another thread of HUF benchmark with a larger input/working set size is the longest running thread and its execution time is reduced to 43.5%. The second HUF benchmark benefited more because of its larger working

set size and the availability of remote SPM capacity when all other threads have finished execution.

The average execution time has reduced to 43% - 92% depending on the core utilization and the working set size demands. Generally less core utilization means more performance improvement. But it also depends on the mix of the threads that are running at the same time. As shown in Figure 2.22, going from mix (e) to mix (f) results in the average execution time to be reduced due to the extra advantage that the second HUF thread receives, as discussed earlier.

The highest improvement bars report the best performance improvement experienced by any of the threads in the mix. Similarly, the lowest improvement bars show the the least benefit experienced by a thread. Except for mix (a), in all other cases there is at least one thread whose execution time is not reduced by using the closest neighborhood with guaranteed local share allocator. And that is because there are threads whose working set size is smaller than each core’s local SPM. In fact, in mixes (c) through (f) there is a thread whose execution time is increased by 0.4%. Although the local SPM share was set to 100% for this experiment, there are times that a thread has to wait for guest thread’s pages to be relocated which incurs

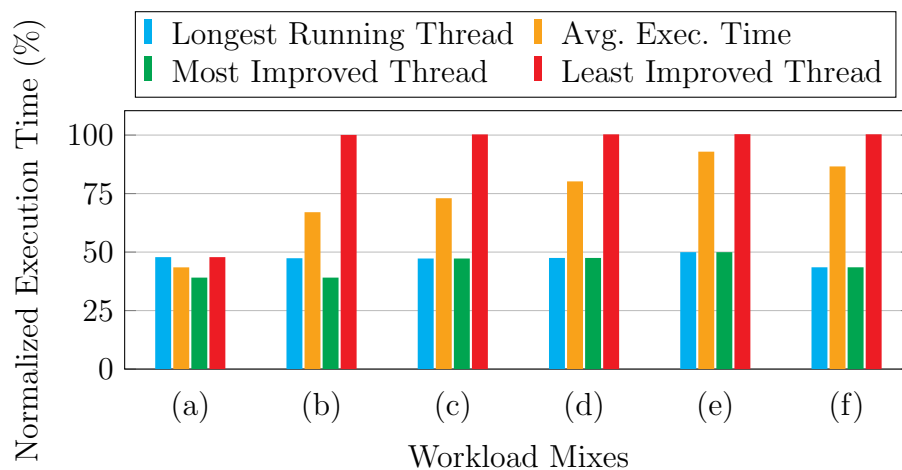


Figure 2.22: Execution time comparison for various workload mixes as shown in Figure 2.21: CNGLSA normalized to LA. Lower values are better.

some small overhead.

2.7 Discussion on Overheads

Not all components of ShaVe-ICE are unique to the solution. SPM-based platforms require an API and a mechanism for address translation to enable allocation and access for local SPMs. However, additional hardware and software components of ShaVe-ICE are required to enable sharing of remote SPMs.

The software component that incurs additional overhead for ShaVe-ICE is the SPM allocator. All SPM-based platforms require a mechanism for allocation and a policy to resolve contention (e.g. *sharing* or *over-subscription* of the local SPM). For an allocator that does not share the entire SPM space, if we assume the SPM page size is P and the allocation size is S , the worst-case time complexity is $O(P*S)$. Since SPM is not shared, the time complexity is not dependent on the number of cores. For allocators that share the entire SPM space between threads, if we assume the SPM page size is P , the allocation size is S , and the number of remote SPMs accessible by the core requesting allocation is N , the worst-case time complexity is $O(P*S*N)$. Accurate analysis of runtime overhead requires implementing the policies inside operating system which is the topic of our future work. To estimate the upper bound of execution times, we measured the time spent in the allocation and deallocation routines inside gem5 every time a thread calls an SPM API. Figures 2.23 and 2.24 show the runtime of these routines for different page sizes and allocation sizes.

The SPM allocator must additionally maintain system-wide SPM state, which includes information specifying each unique virtual page occupying each on-chip SPM page, as well as a list of free pages.

Hardware overhead associated with ShaVe-ICE manifests in the form of address translation

and network communication. Similar to the allocator algorithm, storage for translation required in SPM systems depends on the number of pages per SPM: each core must maintain address translation for its local SPM. Supporting sharing means that each core must now have the capacity to additionally store translation information for remote SPM pages it has access to. In order to mitigate this storage overhead, we can adjust the page size to reduce the number of ATT entries, or limit the amount of total SPM space a core can allocate to a subset of the entire physical SPM space.

As illustrated in Section 2.6, the additional on-chip communication required by ShaVe-ICE yields an overall improvement in average access latency as well as system energy consumption. Although more transactions are required in the NoC to allocate, access, and update local ATT entries, the reduction in off-chip accesses mitigates this penalty. However, as the size of the platform grows to 10s or 100s of cores, the latency penalties incurred by excessive hop distances between cores and remote SPMs could become comparable to off-chip accesses. Therefore, there needs to be a platform-specific threshold for the maximum hop distance when an allocation policy searches for available space during a remote SPM allocation.

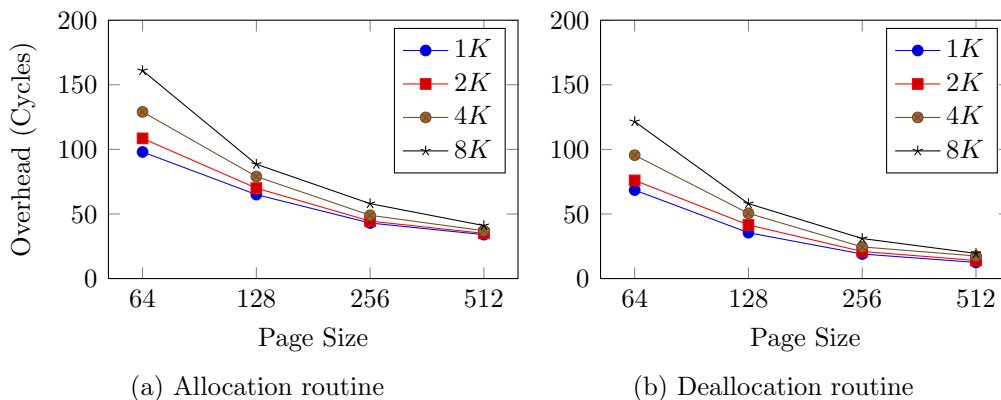


Figure 2.23: Runtime overhead of SPM allocator for different combinations of allocation size and page size: local policy (simplest policy).

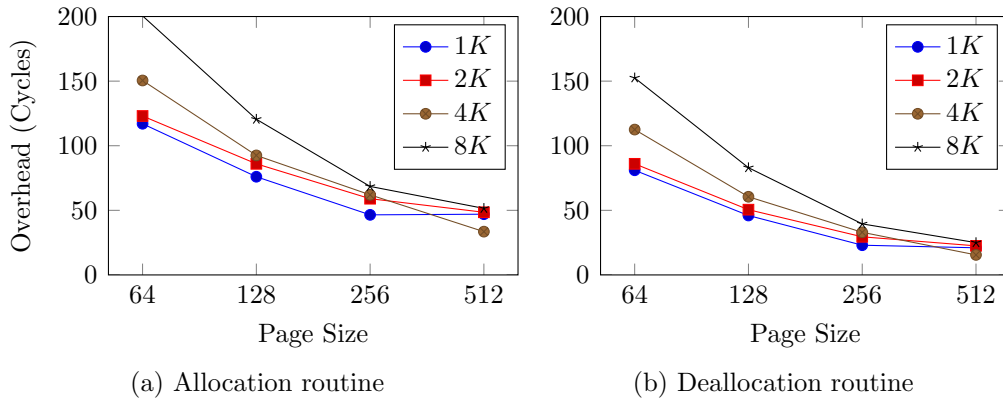


Figure 2.24: Runtime overhead of SPM allocator for different combinations of allocation size and page size: closest neighborhood with local reservation policy (most complex policy).

2.8 Cache+SPM and Shared Data Support Evaluations

In this section we present experiments that compare the hybrid memory hierarchy with a cache-based hierarchy implementing a cache coherence protocol (MESI) and show opportunities for energy saving and/or execution time improvement. We ran our experiments in the system emulation mode of gem5 using in-order ARM cores to configure a 4*4 mesh NoC with one level of on-chip memory. We present preliminary results using use a number of synthetic microbenchmarks – each designed to exercise different memory behavior – (Table 2.5)

Table 2.5: List of microbenchmarks

Benchmark	Parameters	Description
counter_inc	num-threads	An array of counters are incremented by a number of threads. Every thread increases its own counter so there is no true data sharing.
histogram	num-threads data	It computes the histogram of a large array of data. Each worker thread get a portion of the array and updates the histogram for that chunk.
regular_access	num-iterations	It generates regular(strided) accesses to an array of integer data.
irregular_accesses	num-iterations	It generates irregular(random) accesses to an array of integer data.

to highlight the opportunities.

2.8.1 Experiment 1: Coherence Overhead Due to False Sharing

The first experiment compares both hierarchies when false sharing exists. False sharing happens when threads unwittingly impact the performance of each other while modifying independent variables sharing the same cache line. In this experiment we use the *counter_inc* microbenchmark (designed to stress false sharing), varying number of threads from 2 to 10. With the cache-based memory hierarchy, counters share the same cache line causing a false sharing every time a thread accesses its counter. With the software-assisted hierarchy, we allocate the array holding counters on the SPM space and all threads perform remote accesses to the same physical copy. For this letter, we simply allocate the single copy on the SPM of the core spawning the threads; data placement could be further optimized in future work.

Figure 2.25 shows the execution times and network traffic of SAM hierarchy normalized to that of the cache-based hierarchy. As the number of threads increases, the cache coherence overhead becomes more pronounced; indeed we see from Figure 2.25 more improvements for higher number of threads by using a software-assisted memory hierarchy that manages shared data through SPMs.

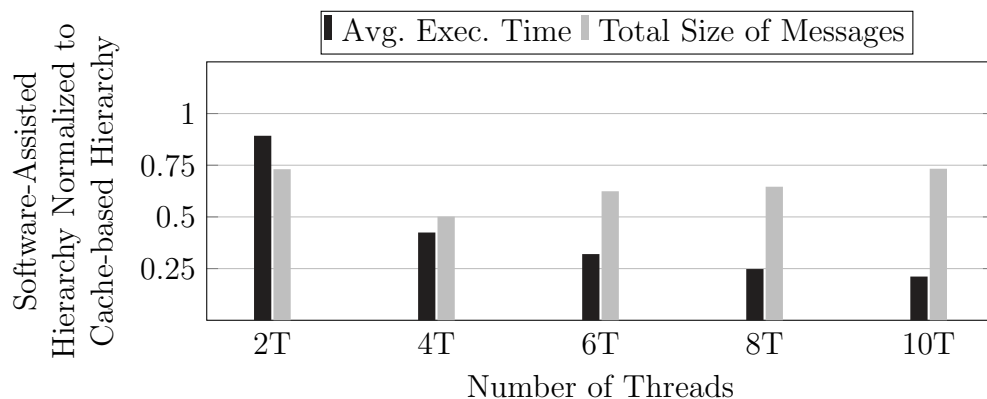


Figure 2.25: Comparing software-assisted memory hierarchy with cache-based hierarchy implementing MESI protocol when false data sharing exists.

2.8.2 Experiment 2: Coherence Overhead Due to Shared Data

In the second experiment, we evaluate how keeping a single copy of shared data on-chip and forwarding all accesses to that copy could improve the average performance of threads compared to when a cache coherence protocol is used to guarantee data correctness. In this experiment we use the *histogram* microbenchmark (designed to generate accesses to shared data). While the data array is read by all threads, it is never written to by any of threads resulting in no coherence issues for that data object. The shared data object that has to be kept coherent across threads is the data structure keeping the computed histogram thus far. Mutex objects are also shared between different threads. We allocate these two data objects on the SPM space and let cache handle other objects. Figure 2.26 shows the execution times and network traffic of SAM hierarchy normalized to that of the cache-based hierarchy. Similar to the previous experiments, improvements are more significant when more number of worker threads are spawned.

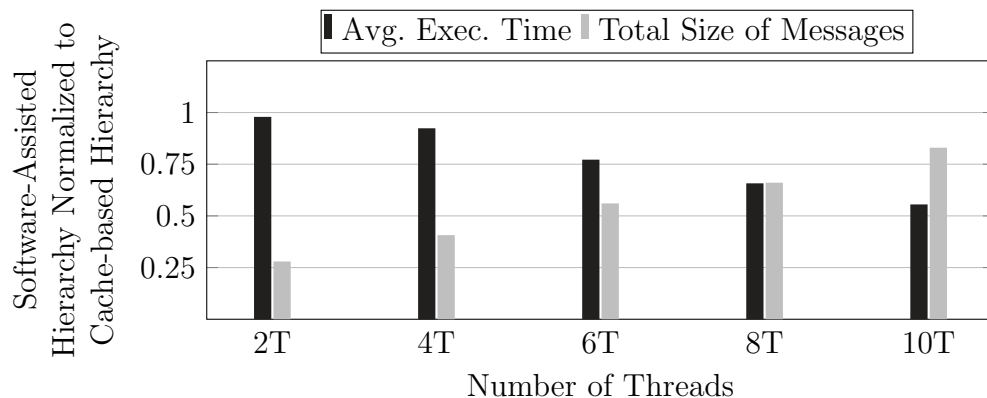


Figure 2.26: Comparing software-assisted memory hierarchy with cache-based hierarchy implementing MESI protocol when true data sharing exists.

2.8.3 Experiment 3: Dynamic Partitioning of Local Memory

The third experiment explores opportunities to improve performance/energy consumption if we are able to dynamically partition the local memory into SPM and cache. For this experiment, we use *regular_access* and *irregular_access* microbenchmarks that generate accesses to a large array of data. We only allocate all or a portion of the data array on SPM. Other memory accesses, to the unallocated part of data array or other scalar data objects in the program, are left to be handled by cache or main memory. The local data memory size is assumed to be 32Kb which could be partitioned in three different modes: 1) 32-KB cache, 2) 32-KB SPM, and 3) 16-KB cache / 16-KB SPM. SPM and Cache access latencies are assumed to be equal (2 cycles). For different runs, we set the size of the data array to 12-KB

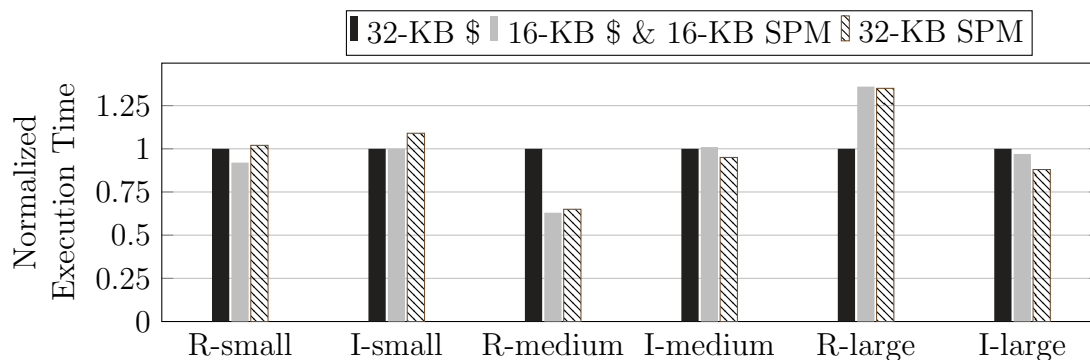


Figure 2.27: Comparison of execution time with different configurations of a fixed size local memory: cache only, SPM and cache, SPM only. Lower is better.

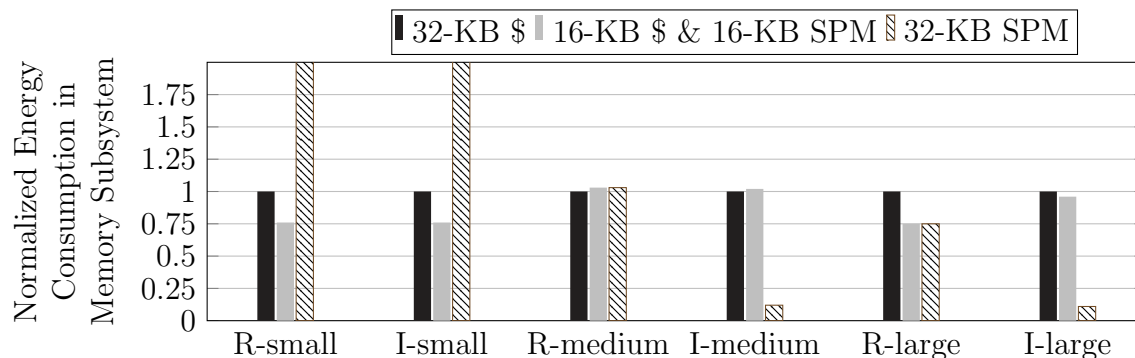


Figure 2.28: Comparison of energy consumption with different configurations of a fixed size local memory: cache only, SPM and cache, SPM only. Lower is better.

(small), 48-KB (medium) and 64-KB (large) in microbenchmarks. For energy consumption, we consider total dynamic energy consumed in caches, SPMs, NoC, and DRAM. Figures 2.27 and 2.28 show the execution time and energy consumption of each configuration normalized to cache only configuration. Depending on the relative sizes of working set to the local memory size and access pattern, in each case, different partitionings result in the best energy consumption, execution time, or both. A runtime manager with this knowledge, through both static analysis as well as runtime monitoring, can exploit and tune the partitioning to improve desired objectives.

2.8.4 Experiment 4: Sharing SPMs Between Cores

In Experiment 3, we assumed each core can allocate memory on its local SPM only. If we virtualize the distributed SPM space and let threads do remote allocations, a better performance/energy trade-off might be achievable when there is core underutilization or a thread does not need its entire local memory. We redo Experiment 3 for the irregular_access microbenchmark when working set is larger than the local memory (I-medium and I-large) and also allowing remote SPM allocations of sizes 16-KB and 32-KB on neighboring cores.

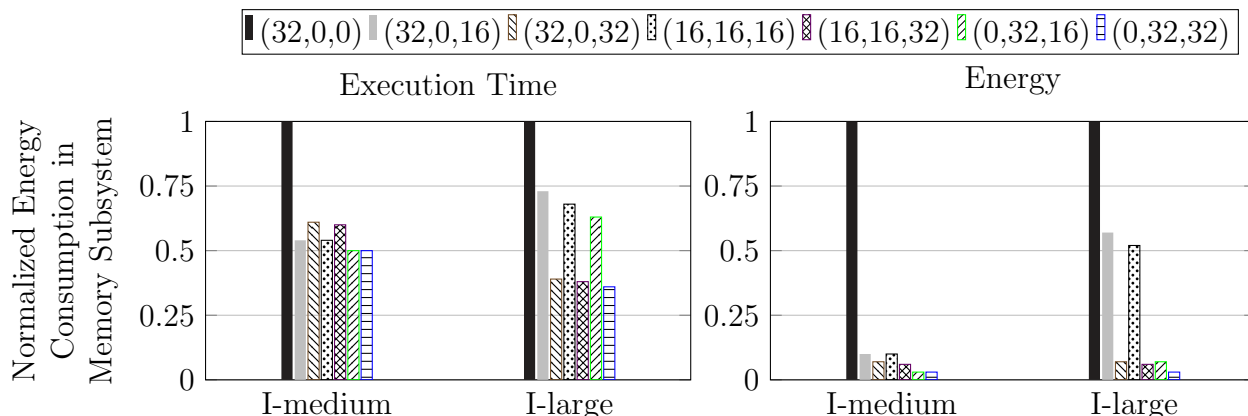


Figure 2.29: Comparison of execution time and energy consumption when remote SPMs are available for allocation. Legends are shown as: (local cache, local SPM, remote SPM at hop distance 1). Lower is better.

Figure 2.29 shows the execution time and energy for different allocation scenarios normalized to cache only hierarchy. Comparing Figure 2.28 and Figure 2.29 shows the opportunity to both improve performance and save energy in memory hierarchy enabled by sharing SPMs which results in reducing off-chip accesses.

Chapter 3

Approximate On-chip Data Storage

3.1 Introduction

Approximate computing leverages the intrinsic resilience of applications to inexactness in their computations, to achieve a desirable trade-off between efficiency (i.e., performance, energy or both) and acceptable quality of results. The need for approximate computing is driven by two factors: a fundamental shift in the nature of computing workloads, and the need for new sources of efficiency.

Many important application domains such as computer vision, machine learning, signal processing, web search, augmented reality, big data analytics, etc. can inherently tolerate inaccurate computation [38]. Today's systems waste time, energy, and complexity to provide uniformly pristine operation for applications that do not require it.

In embedded systems, where battery life or other resources are constrained many applications can live with inexactness. This is true especially where the *input is analog*. For example, speech recognition applications turn analog input signals into a sentence, and navigation

software turns maps and location estimates from a GPS into driving directions. This is also true where the *output is analog* such as in audio, image and video processing, since human perception is inherently inaccurate.

Motivated by the benefits available via relaxing stringent correctness and precision requirements, researchers have proposed a variety of approximate computing strategies both in software and hardware. Approximation strategies in software include: dynamically selecting between different implementations of an specification that provide varying accuracy-energy-performance tradeoff (Dynamic Knobs [68], Green Framework [16], Levels [83], ViRUS [150], Variable Accuracy Algorithms [10]), skipping some of the loop iterations (Loop Perforation [134] [107]), eliminating synchronization in multi-threaded programs (Dubstep [108], Approx. Data Structures [124]), unsound parallelization of sequential programs (QuickStep [105]), adjusting floating point precision (Precimonious [126]) randomizing deterministic programs ([106, 160]). Different approximation techniques in hardware level are proposed including: underdesigned functional units that produce wrong results occasionally (approximate adders [61, 62, 74, 101], approximate floating point units [5, 31], approximate multipliers [82, 91, 90, 66]), utilizing separate processing units with different levels of reliability (ERSA [87]), memory approximation (Relaxed Cache [132], QuARK [109], Flicker [92], Approx. Storage [130])¹. EnerJ [129] and ACCEPT [128] are two compiler frameworks that provide the required language constructs and a library of approximation strategies to the developer to write and analyze approximate programs. Truffle [51] presents the required architecture support to execute applications written with a language like EnerJ. Although a significant amount of research on approximate computing has been presented in recent years, performing approximation in a controlled manner has remained an open problem. Most of the previous work have attempted to show it is possible to operate at different design points in the accuracy-energy-performance space. Very recently there have been a few work (Green [16], Rumba [79], Proactive Control [139]) on developing online checking of the result

¹A complete overview of related work on approximate memories is presented in Section 3.2.

of approximate computations.

Previous work have shown that designing such systems in a general-purpose computing environment requires a holistic view of all layers from algorithms, programming models, system software, and hardware down to the transistor level, rather than focusing on a single layer.

In this chapter, we explore approximation strategies for on-chip data storage of modern embedded systems.

3.2 Prior Work on Memory Approximation

3.2.1 Taxonomy of Prior Research on Approximate Memory Management

The majority of work on approximate computing focuses on computation. However, the idea of quality-energy-performance trade-offs lends itself to storage as well. This includes both transient data stored on-chip on in the main memory or persistent data stored in the secondary storage. Here, we present a taxonomy of research on approximate memory management. Accordingly, we classify techniques along five axes:

1. Memory Technology
2. Memory Component
3. Approximation Objectives
4. Approximation Strategy
5. Abstraction Layers Involved

Depending the specific memory hierarchy component that is subject to approximate storage, different layers of the system abstraction come into play. Table 3.1 depicts the abstraction layers covered by recent efforts. It can be seen that many of the previous work leverage multiple layers of abstraction. While majority of approaches target reducing energy consumption as the main objective, some approaches attempt in increasing the performance and/or increasing the lifetime of the memory substrate (Table 3.2). Approximations can be applied to memory structures abstractions (Table 3.3) such as L1/L2 cache [132] [114, 131] [103] [104] [86] [56] [127] [80] [102] [72], scratchpad memory [122], main memory [92] [119] [52] [121] [156] [155],

Table 3.1: Prior research In approximate memory management classified based on abstraction levels involved

Related Work	Abstraction Layer				
	Application/ Algorithm	Compiler/ ISA	OS / Runtime	Architecture	Device
Shoushtari et al. - 2015 [132]					
Monazzah et al. - 2017 [109]					
Lee et al. - 2006 [86]					
Liu et al. - 2011 [92]					
Sampson et al. - 2013 [130]					
Oboril et al. - 2016 [114, 131]					
Ranjan et al. - 2015 [122]					
Raha et al. - 2017 [119]					
Thwaites et al. - 2014 [145, 154]					
Miguel et al. - 2015 [103]					
Rinard et al. - 2013 [124]					
Tian et al. - 2015 [146]					
Miguel et al. - 2014 [104]					
Ganapathy et al. - 2015 [56]					
Cho et al. - 2014 [41]					
Fang et al. - 2012 [52]					
Sampaio et al. - 2015 [127]					
Guo et al. - 2016 [59]					
Jevdjic et al. - 2017 [73]					
Xu et al. - 2015 [153]					
Kislal et al. - 2016 [80]					
Arumugam et al. - 2015 [13]					
Chen et al. - 2016 [37]					
Miguel et al. - 2016 [102]					
Ranjan et al. - 2017[121]					
Zhang et al. - 2017[155]					
Jain et al. - 2016[72]					
Zhao et al. - 2017[156]					

secondary storage [153] [130] [59] [73], in a single-layer or cross-layer fashion. These techniques have three main objectives: making a trade-off between the quality of the generated output and (1) energy consumption [109] [122] [132] [92] [119] [114, 131] [103] [86] [104] [52] [56] [41] [146] [127] [153] [102] [121] [155] [156], (2) performance of the memory subsystem [145, 154] [130] [114, 131] [104] [56] [124] [59] [73] [153] [80] [13] [37] [102] [121] [155] [72], and (3) improving their lifetime [52] [130].

These objectives are often dependent on the memory technology used (Table 3.4). Different memory technologies have different limitations, for example, SRAM and (e)DRAM consume

Table 3.2: Prior research In approximate memory management classified based on approximation objective

Related Work	Goal		
	Energy	Perf.	Lifetime
Shoushtari et al. - 2015[132]	■		
Monazzah et al. - 2017[109]	■		
Lee et al. - 2006[86]	■		
Liu et al. - 2011[92]	■		
Sampson et al. - 2013[130]		■	■
Oboril et al. - 2016[114, 131]	■	■	
Ranjan et al. - 2015[122]			
Raha et al. - 2017[119]	■		
Thwaites et al. - 2014[145, 154]		■	
Miguel et al. - 2015[103]	■		
Rinard et al. - 2013[124]		■	
Tian et al. - 2015[146]	■		
Miguel et al. - 2014[104]	■	■	
Ganapathy et al. - 2015[56]			
Cho et al. - 2014[41]	■		
Fang et al. - 2012[52]	■		■
Sampaio et al. - 2015[127]	■		
Guo et al. - 2016[59]		■	
Jevdjic et al. - 2017[73]	■		
Xu et al. - 2015[153]	■	■	
Kislal et al. - 2016[80]		■	
Arumugam et al. - 2015[13]		■	
Chen et al. - 2016[37]		■	
Miguel et al. - 2016[102]	■	■	
Ranjan et al. - 2017[121]	■	■	
Zhang et al. - 2017[155]	■	■	
Jain et al. - 2016[72]		■	
Zhao et al. - 2017[156]	■		

Table 3.3: Prior research In approximate memory management classified based on the memory component

System Component	Prior Research
Cache	Shoushtari et al. - 2015 [132] Oboril et al. 2016 [114, 131] Miguel et al. - 2015 [103] Miguel et al. - 2014 [104] Lee et al. - 2006 [86] Ganapathy et al. - 2015 [56] Sampaio et al. - 2015 [127] Kislal et al. - 2016 [80] Miguel et al. -2016 [102] Jain et al. -2016 [72]
SPM	Ranjan et al. - 2015 [122]
Main Memory	Liu et al. - 2011 [92] Raha et al. - 2017 [119] Fang et al. - 2012 [52] Ranjan et al. - 2017 [121] Zhang et al. - 2017 [155] Zhao et al. - 2017 [156]
Secondary Storage	Xu et al. - 2015 [153] Sampson et al. 2013 [130] Guo et al. - 2016 [59] Jevdjic et al. - 2017 [73]

Table 3.4: Prior research In approximate memory management classified based on the memory technology

Memory Technology	Prior Research
NVM	Xu et al. - 2015 [153] Fang et al. - 2012 [52] Sampson et al. - 2013 [130] Ranjan et al. - 2015 [122] Monazzah et al. - 2017 [109] Oboril et al. 2016 [114, 131] Sampaio et al. - 2015 [127] Guo et al. - 2016 [59] Jevdjic et al. - 2017 [73] Zhao et al. - 2017 [156]
DRAM/eDRAM	Liu et al. - 2011 [92] Raha et al. - 2017 [119] Cho et al. - 2014 [41] Ranjan et al. - 2017 [121] Zhang et al. - 2017 [155]
SRAM	Shoushtari et al. - 2015 [132] Ganapathy et al. - 2015 [56]

high leakage and refresh power, respectively, while NVMs have high write energy/latency and/or low write endurance. To reduce energy consumption of these memories and improve

Table 3.5: Prior research In approximate memory management classified based on the approximation strategy

Approximation Strategy	Prior Research
Precision scaling	Miguel et al. - 2015 [103] Tian et al. - 2015 [146] Ranjan et al. - 2017 [121] Jain et al. - 2016 [72] Zhao et al. - 2017 [156]
Load value approximation	Thwaites et al. - 2014 [145, 154] Miguel et al. - 2014 [104] Miguel et al. - 2016 [102]
Memory access skipping	Thwaites et al. - 2014 [145, 154] Fang et al. - 2012 [52] Kislal et al. - 2016 [80]
Using faulty or unprotected memory	Oboril et al. - 2016 [114, 131] Sampson et al. - 2013 [130] Lee et al. - 2006 [86] Ganapathy et al. - 2015 [56] Sampaio et al. 2015 [127] Guo et al. - 2016 [59] Jevdjic et al. - 2017 [73] Xu et al. - 2015 [153]
SRAM voltage scaling / DRAM refresh rate reduction / NVM current amplitude or duration reduction	Monazzah et al. - 2017[109] Shoushtari et al. - 2015 [132] Liu et al. - 2011 [92] Raha et al. - 2017 [119] Ranjan et al. - 2015 [122] Sampson et al. - 2013 [130] Cho et al. - 2014 [41] Zhang et al. - 2017 [155]

the lifetime of NVMs, approximate storage techniques sacrifice data integrity by reducing supply voltage in SRAM [132] [56] and refresh rate in (e)DRAM [92] [119] [41] [121] [155], and by relaxing or skipping read/write operation in NVMs [153] [52] [130] [122] [109] [114, 131] [127] [59] [73] [156] (Table 3.4).

These objectives are achieved by a number of general strategies (Table 3.5), namely: (1) precision scaling [103] [146] [121] [72] [156], (2) approximating load operations [145, 154] [104] [102], (3) skipping store operations [145, 154] [52] [80], (4) using faulty or unprotected memory substrate [114, 131] [130] [86] [56] [127] [59] [73] [153], (5) tweaking technology-dependent reliability-energy knobs [109] [132] [92] [119] [122] [130] [41] [155].

3.2.2 Overview of Prior Research and Practices

This section overviews related work on approximate memory management.

■ **Approximate Spintronic SPM:** Ranjan et al. [122] explore the energy-quality trade-off in STT-MRAM memories. They study three types of approximations: (1) approximations through incorrect read decisions (2) approximations through read disturbs (3) approximations through incomplete writes. They design a quality-configurable memory array in which data can be stored to varying levels of accuracy based on application requirements. For that they employ an additional peripheral circuitry, which allows adjusting the degree of these approximation mechanisms. They integrate the quality-configurable array as a scratchpad in the memory hierarchy of a programmable vector processor and expose it to software by introducing quality-aware load/store instructions within the ISA. They also develop an auto-tuning framework that utilizes gradient descent search to determine the quality fields of the load/store instructions in a given application program so as to minimize energy for a desired application output quality.

■ **Approximation-Aware Multi-Level Cells STT-RAM Cache Architecture:** Sampaio et al. [127] propose a partially-protected cache architecture based on MLC STT-RAM. They consider an n way set-associative cache where the last way stores Error Correction Codes (ECCs) for error protection and is implemented by SLC cells. Thus, at every read and write operation, these ECCs must be accessed to serve as input for the error correction unit. They realize approximate storage by avoiding error protection of applications' non-critical data. In this case, the last cache way is enabled for data storage, dynamically increasing the associativity of the cache which results in improved performance.

■ **Relaxing Non-volatility of STT-MRAM Caches:** The authors of [114, 131] also propose approximate STT-MRAM but unlike [109] their knob is the non-volatility property of STT-MRAM. Reducing the the thermal stability factor decreases the access latency and

since the current required to switch the bit-cell content is applied for a smaller duration, the write energy will become significantly better. To protect the critical data, they propose to use multiple copies of the content of this data, such as dual or triple modular redundancy, or to use ECC, which protects the data by adding check bits. In order to choose the optimal thermal stability factor and application they calculate the impact of various thermal factors on failure rates, write latency and write energy at device level. These values are then projected at the architecture level to estimate per access metrics. Finally at the system-level, the application output is evaluated using the Signal to Noise Ratio (SNR) metric.

■ **Selective Data Protection:** Lee et al. [86] propose selective data protection for mitigating failures caused by soft errors in multimedia embedded applications when power consumption is a concern. They extend the concept of horizontally partitioned cache by incorporating two caches at the same level of the memory hierarchy. One of the caches are protected against soft errors with Single-Error Correction and Double-Error Detection (SECDED) and the other one is left unprotected. Each memory address is mapped exclusively to one of these caches. They partition the application’s data objects into failure critical and failure non-critical and map the first data category to the protected cache while the second data category is mapped to the unprotected cache. Their approach minimizes the protection overheads while achieving a failure rate close to an architecture with a single fully protected cache.

■ **Flikker:** Flikker [92] allows parts of a DRAM module to be refreshed at a much lower rate than the standard refresh rate, allowing retention errors to happen but saving significant DRAM refresh power. The programmer identifies critical and noncritical data. During execution, data objects are allocated to different parts of the DRAM. The critical data are mapped to the regions that are refreshed at regular rates, while the noncritical data are mapped to the regions that are refreshed at a lower rate.

■ **Quality Configurable Approximate DRAM:** Raha et al. [119] propose the notion

of a quality-aware approximate DRAM and develop a novel data allocation scheme for the proposed approximate DRAM. The core idea is that at sub-optimal refresh rates, DRAM physical pages can be split into a number of quality bins based on the characteristics of the errors seen in each page. Approximate data can then be allocated to pages belonging to the bins in decreasing order of quality, ensuring that they always allocate to the least erroneous pages. The location and nature of the bit-errors are obtained through extensive error characterization of off-the-shelf DRAM ICs at various refresh rates. These errors correlate well with the eventual application-level output quality and hence, are used to guide the allocation of application data to DRAM pages based on the output quality specification. Their proposed mechanism is inherently quality configurable since it has the provision of increasing the refresh rate as needed, which increases the number of pages in higher quality bins (with lower errors), leading to better quality. Compared to [92], their work requires only a single refresh interval for the entire DRAM. Therefore, it is simpler to implement and results in a better energy-quality trade-offs.

■ **DrMP: Mixed Precision-aware DRAM:** Zhang et al. [155] propose DrMP to exploit process variations within and across DRAM rows to save data with mixed precision. They describe three variants of the approach: DrMP-A, DrMP-P, and DrMP-U.

(1) DrMP-A is to achieve high performance approximate computing. It reduces restore time and maps important data bits of different data types to fast reliable row segments for improved performance. (2) DrMP-P pairs memory rows together to reduce the average restore time for precise computing. (3) DrMP-U combines DrMP-A and DrMP-P to support both approximate and precise computing.

■ **Approximate Storage in MLC NVM:** Sampson et al. [130] introduce two techniques enabling applications to store data approximately in PCM memories in order to improve their lifetime, density, and performance. Increasing the number of levels, improves the density of storage, however multilevel cells (MLC) are slower to write due to the need for tightly

controlled iterative programming. The first technique they introduce reduces the number of programming pulses used to write to an MLC. This allows errors in multi-level cells but improves performance and energy efficiency. On the other hand, this technique can also be used to improve the storage density for a fixed power budget or performance target. Their second technique extends memory lifetime by mapping approximate data onto blocks that have exhausted their hardware error correction resources. In order to reduce the effect of erroneous bits on the final result, higher priority is given to the correction of higher-order bits compared to lower-order bits. Their evaluations show that approximate writes in MLC PCM are faster than precise writes. Also using faulty blocks improves the endurance of the memory module with bounded quality loss.

■ **SoftPCM:** Fang et al. [52] propose SoftPCM that reduces the number of writes to a PCM-based main memory by taking advantage of the error-tolerance property of video applications. SoftPCM compares the new data with the old data and when the new data to be written are same as the existing stored data, this technique terminates the write operation and takes the old data as the new data. SoftSPM provides significant energy savings and lifetime extensions for PCM memories with a negligible reduction in video quality.

■ **SoftFlash:** Xu and Huang [153] leverage error-tolerance capability of data-centric applications for reducing ECC overhead in flash-based Solid-State Drives (SSD). They design a framework for monitoring and estimating soft-error rates of Flash SSD at runtime. They also carry out extensive fault-injection experiments on a wide range of applications including multimedia, scientific computation, and cloud computing to understand the requirements and characteristics of data level error tolerance. These two studies show that their target applications can tolerate a much higher error rate than that targeted by Flash SSDs. Depending on the difference between the error rate of the SSD obtained using their monitoring framework and the error rate that the application can tolerate, SoftFlash dynamically lowers the ECC protection or avoids using ECC altogether. They show that their approach provides

significant performance and energy efficiency gains with an acceptable quality.

■ **Synchronization-Free Approximate Data Structures:** Rinard [124] presents approximate data structures with construction algorithms that execute without synchronization. The data races present in these algorithms may cause them to drop inserted or appended elements. Nevertheless, the algorithms 1) do not crash and 2) may produce a data structure that is accurate enough for its users to use successfully. Rinard advocates an approach in which the approximate data structures are composed of basic tree and array building blocks with associated synchronization-free construction algorithms. These data structures have been engineered to execute successfully in parallel contexts despite the presence of data races. Rinard shows that using these data structures results in significant speedup in time.

■ **Approximate Memory Compression:** Ranjan et al. [121] propose a technique to reduce off-chip memory traffic and energy by leveraging the intrinsic resilience of emerging workloads such as machine learning and data analytics. Their objective is to reduce memory traffic rather than overall memory footprint. In order to realize approximate memory compression, they enhance the memory controller to be aware of memory regions that contain approximation-resilient data. They propose an application programming interface to expose approximation-tolerant memory regions to the memory controller. The memory controller then transparently compress/decompress the data written to/read from these regions. To provide control over approximations, the quality-aware memory controller conforms to a specified error constraint for each approximate memory region. They also incorporate a runtime framework to dynamically modulate the error constraints for each region.

■ **ApproxMA:** Tian et al. [146] present ApproxMA, a technique for dynamic precision scaling of off-chip accesses. They apply their technique to a mixed-model-based clustering problem, which is inherently error-resilient and needs large amount of data accesses from off-chip memory. ApproxMA is comprised of runtime precision controller and memory access

controller. For data accesses, runtime precision controller firstly generates the customized bit-width according to runtime quality requirements, and then memory access controller loads the scaled data from off-chip for computations. The runtime precision controller works based on the fact that in a clustering algorithm, a functional error happens only when a sample is assigned to a wrong cluster. Based on this, the precision can be lowered as long as the relative distances between clusters and samples are still in correct order, so that no functional error happens. From the point view of error resilience capability, an appropriate amount of functional errors are acceptable. The iterative nature of clustering algorithm is to continuously correct the mixture models and re-label all the samples by using higher precision in later iterations. By analyzing subset of data and/or intermediate computational results, runtime precision controller calculates precision constraints (when there will be a functional error) and error resilience capability (how many functional errors are tolerable), and then decides the required bit-width for the current data access. Since use of approximation can lead to fluctuation in membership, on detecting such a situation, their technique increases the precision of data. To realize loading certain most significant bits of data from off-chip memory, the memory controller reorganize the data. Bits of the same significance in different words are combined to form new words and stored in off-chip memory. They show that, compared to fully precise off-chip access, their technique saves significant energy with a negligible loss in accuracy.

■ **Sorting Algorithms on Approximate Memory:** Chen et al. [37] study sorting on a hybrid storage system with both precise storage and approximate storage. Unlike many other works, the sorting algorithm cannot accept any error. This work is an example of precise computing on approximate storage. Their approximate storage is an approximate MLC configuration that reduces the guardband to provide better write performance and energy efficiency. They propose an approx-refine execution mechanism to improve the performance of sorting algorithms on the hybrid storage system to produce precise results. A lightweight refinement stage transforms the nearly sorted output to a totally sorted output.

Their optimization gains the performance benefits by offloading the sorting operation to approximate storage, followed by an efficient refinement to resolve the unsortedness on the output of the approximate storage.

■ **Inexact Memory Aware Algorithm Co-design:** Arumugam et al. [13] introduce inexact memory aware algorithm design and presents several relevant algorithms for sorting and string matching.

■ **Mitigating the Impact of Faults in Unreliable Memories:** Ganapathy et al. [56] present a technique for minimizing the magnitude of errors when using unreliable memories. In contrast to ECC techniques that correct the errors, the proposed approach minimizes the magnitude of the error (caused due to a faulty cell) quantified in terms of a suitable metric. This is ensured by placing bits of lower significance into the faulty cells. To achieve that, on each write their approach shifts the data circularly in such a way that the least significant bits are stored in the faulty cells. By controlling the granularity of the shuffling, the proposed technique enables trading-off quality for power, area, and timing overhead. Compared to error correction codes, their approach improves latency, power and area. Also, use of their technique allows tolerating a limited number of faults, which reduces the manufacturing cost compared to the conventional zero-failure yield constraint.

■ **Cache-Aware Approximate Computing for Decision Tree Learning:** Kislal et al. [80] exploit the flexibility inherent in decision tree learning based applications regarding performance and accuracy trade-offs and propose a framework to improve performance with negligible accuracy loss. This framework employs a data access skipping module that skipss costly cache accesses according to the aggressiveness of the strategy specified by the user and a heuristic to predict skipped data accesses to keep accuracy losses at minimum. They make two important observations: 1) The inaccuracy resulting from skipping data accesses depends more on the number of data accesses skipped, rather than which specific data accesses are skipped. 2) In contrast, the performance benefits achieved through data access skipping

depends strongly on which specific data accesses are skipped.

■ **Concise Loads and Stores:** Jain et al. [72] aim at reducing the pressure on the memory subsystem by enabling concise storage – a storage paradigm where the data elements are stripped of their marginal bits, removing the movement and storage costs associated with those bits in the memory subsystem. They propose asymmetric compute-memory extension (ACME) for conventional architectures. In ACME, data can be treated asymmetrically; computation is done on conventional 32-bit IEEE 754 single precision values – while data is stripped of its marginal bits before being used in the memory hierarchy. ACME includes a simple ISA extension that can be leveraged by the programmer and compiler, adding two new instruction classes to the ISA to operate on concise data – load-concise and store-concise – to perform conversions between concise and single precision format.

■ **High-Density Image Storage Using Approximate Memory Cells:** Guo et al. [59] targets storing images in approximate solid-state memories with multi-level cell (MLC) capability. They make this observation that changes in different bits in the compressed image file lead to different kinds of distortion in the output image. Hence, to preserve quality in the decoded image, different bits should be subject to different levels of approximation. The key idea is to determine the relative importance of encoded bits created by the encoding algorithm, and store them into separate regions of approximate storage, each of which tuned to match the error tolerance of the bits it stores. Their analysis shows that: (1) Lower frequency coefficients, often higher in value, are the most important coefficients for image quality; (2) control bits affect output quality more than the run-length bits, and run-length bits affect the output quality significantly more than refinement bits. Based on this analysis they store different parts of the image file into different parts of storage with appropriate level of approximation.

■ **VideoApp:** Jevdjic et al. [73] propose VideoApp, a methodology to compute bit-level reliability requirements for encoded videos by tracking visual and metadata dependencies

within encoded bitstreams. They add an analysis framework to the video encoder as a post-processing step which computes the importance of each bit with respect to the visual damage flipping that bit would cause. The encoded videos are then partitioned into multiple streams, where each stream is stored with a different level of error correction depending on its reliability requirements.

■ **Approximate Image Storage with MLC STT-MRAM Main Memory:** Zhao et al. [156] target energy-efficient data movement in image processing applications, by leveraging 2-bit MLC STT-MRAM technology. 2-bit MLC STT-MRAM cells have a unique property writing to the two bits (a soft bit and a hard bit) can be performed by asymmetric write current. The soft bit has a higher resistance than the hard bit and it requires a smaller switching current. Their mechanism skips the writes of neighbor pixels that are sufficiently similar. For example when 4 neighboring pixels have similar values, they only store the code of one pixel into the soft bits of each MLC STT-MRAM cell with small write current, when the image is initially written or updated in the memory. The other three memory locations are reserved for other three pixels but are not written to. When they are read by the application, the values are copied from the first pixel. This copying consumes less energy than writing all four precise pixel values, since similar values on soft and hard bit only require a single write operation as opposed to two that is required for writing precise values. They report significant energy saving is possible with their technique for various image processing functionalities.

■ **eDRAM-based Tiered-Reliability Memory:** Cho et al. [41] proposes a technique for saving refresh energy in eDRAM-based frame buffers for video applications. The human visual system is known to be more sensitive to a change in the higher-order bits of a pixel value than the lower-order bits. Their technique divides the memory array into different segments, each of which can be refreshed at different periods. The basic idea is that the higher order bits of a pixel are allocated to the more reliable segments of the memory. Based on the application characteristics and user preference, the number of segments and their refresh

rates can be changed. Their technique saves significant refresh power without incurring loss in visual perception quality of the video.

■ **Doppelgänger Cache:** Miguel et al. [103] introduce Doppelgänger cache which takes advantage of approximate similarity of data blocks stored in the last-level cache (LLC). They identify cache blocks that are approximately similar and associate their tags to a single entry in the data array. Multiple blocks share a single data entry, which serves as an acceptable approximation of their values. This reduces the footprint of data that LLC needs to store. To determine whether a similar cache block exists prior to inserting an incoming block, Doppelgänger cache computes maps for each block using a hash function of data; the hash function is chosen such that similar blocks generate the same maps. They show that their cache design uses the available capacity more efficiently, which results in substantial area and energy savings.

■ **Bunker Cache:** Miguel et al. [102] propose the Bunker Cache, a design that exploits an application's spatio-value similarity to reduce both off-chip memory accesses and cache storage requirements. Spatio-value similarity refers to their observation that data elements that are approximately similar in value tend to be stored at regular intervals in memory. Bunker cache works by storing similar data blocks in the same location in the cache since substituting one for the other still yields acceptable application quality. The blocks that exhibit spatio-value similarity are usually stored at spatially regular intervals (or strides). Therefore, the implementation of the Bunker Cache is contained entirely within the cache index function. Bunker cache is an approximate computing technique that can achieve efficiency gains with mostly commodity hardware.

■ **LVA:** Miguel et al. [104] presents a load value approximator, a hardware mechanism that estimates memory values. By approximating the load value on a cache miss, the processor can immediately proceed without waiting for the cache response. Traditional load-value predictors fetch a predicted block on every cache miss to confirm the correctness of prediction. LVA

only fetches the approximated blocks occasionally just to train the approximator. Unlike traditional load-value predictors which perform rollbacks upon mispredictions, with this technique rollbacks are not required since the applications can tolerate errors. They show that, with negligible degradation in output quality, their technique provides significant speedup and energy saving.

■ **RFVP:** The authors of [145, 154] present an approximate computing approach to deal with the fact that the performance of modern GPUs is significantly limited by the available off-chip bandwidth. RFVP relies on the programmer annotating the code to identify the memory accesses that are not critical. Of these, the load accesses that result in a large fraction of misses are selected. The impact of approximating each of these loads on quality is measured and a subset of loads that lead to a smaller quality degradation are finally selected for approximation. When these loads miss in the cache, the requested values are predicted and the prediction is not checked against the actual value later on to avoid pipeline flush overheads. Controlling the drop rate can be used as a knob to balance quality degradation and performance improvement. Since GPUs execute in SIMD mode, load accesses happen simultaneously for multiple concurrent threads. To avoid the overhead of having a value predictor for each thread separately, they leverage the value similarity across accesses in adjacent threads to design a multi-value predictor that has only two parallel specialized predictors, one for threads 0 to 15 and another for threads 16 to 31. Use of special strategies for the GPU architecture, such as use of a multi-value predictor, distinguishes their technique from that of Miguel et al. [104]. They show that their technique improves performance and energy efficiency with bounded QoR loss in both the GPU and CPU.

3.3 Partially-Forgetful Memories

This chapter proposes a class of on-chip memories for approximate computing called Partially-Forgetful Memories (PFMs). Potential advantages of deploying PFMs include: (1) better energy efficiency for memory accesses, (2) reduced average memory access latency, and (3) increased lifetime of the memory subsystem.

The following challenges need to be addressed to effectively exploit PFMs for approximate computing:

1. *Data Partitioning*: Data structures can be defined as critical (i.e., any corruption leads to catastrophic failure or significant quality degradation) or non-critical (i.e., quality degradation resulting from corruption may be acceptable). This concept has been used before in [92]. A simple partitioning of data – into critical and non-critical – has been shown to improve DRAM energy efficiency [92]. Further categorization of the non-critical data can lead to even more energy efficiency. For example: (i) integer data can be partitioned based on the magnitude of tolerable error, and (ii) floating point data can be partitioned based on a limit on precision. These categorizations can be reflected in the program by annotating different data structures. One approach [129][33] defines type qualifiers and dedicated assembly-level store and load instructions for this purpose. We present another approach for PFMs that uses dynamic declarations which are enforced by a runtime system, as shown by an example in Section 3.4.
2. *Data Mapping*: The data partitioning annotations in the previous step should guide mapping of each data category to an appropriate part of memory based on its reliability characteristics. Depending on the type of memory, this mapping can be done by the hardware, application/compiler, or operating system. For caches, the cache controller decides where to map a new incoming cache block. In case of software-controlled memories (e.g., scratchpads), the compiler aggregates data with the same reliability

requirement in tagged groups, and then a runtime system (having knowledge about underlying hardware) does the actual mapping based on this tagging. For main memories, the operating system should do the mapping during logical-to-physical mapping.

3. *Controlling Exposed Error Rate to Program:* Most of the previous attempts at utilizing relaxed-reliability memories [92][130] lack the ability to dynamically adjust the exposed error-rate to the program. Taassori et al. [141] have shown how the DRAM timing margin can be dynamically changed to make a precision-performance trade-off. We believe this is necessary because different applications have different levels of tolerance to errors. Even within an application, one execution phase may have more tolerance to errors (i.e., less criticality) than another phase. Therefore, it is important to enable adjustment of the guardbanding knobs based on the characteristics of application. Examples of these knobs are: DRAM Refresh Rate, SRAM Voltage, STT-RAM Retention Time, I_{RESET} Current in PCM, etc.
4. *Adaptation to Phasic Behavior of Applications:* There are three main reasons to change memory guardbanding knobs during runtime: (1) Depending on the (set of) applications that are executed, we might tolerate different levels of error, (2) We may need to change the ratio of reliable to unreliable memory capacity to prevent performance degradations for specific situations (e.g., where most of the working set objects should be mapped to reliable parts of memory), and (3) The programmer can save power by disabling idle parts of the memory for computation-bound application phases (this works even for applications that do not tolerate any errors).

In this chapter, we present exemplars of PFMs for different memory components and technologies. Section 3.4 presents Relaxed Cache for L1 SRAM data caches, Section 3.5 presents Quark Cache for L2 STT-MRAM caches and Section 3.6 presents Write-Skip for

data STT-MRAM SPMs. Finally in Section 3.7 we discuss the idea of using formal control theory to control the quality of PFM.

3.4 Relaxed Cache

3.4.1 Introduction

As the semiconductor industry continues to push the limits of sub-micron technology, the ITRS expects hardware variations to continue increasing over the next decade [71]. The memory subsystem is one of the largest components in today's computing systems, a main contributor to the overall power consumption of the system, and therefore one of the most vulnerable components to the effects of variations. Device manufacturers have partially masked the presence of variability by *guardbanding* mechanisms [60].

Guardbanding leads to over-design with less than optimal power and/or lifetime for different memory technologies. In SRAM-based on-chip memories, the desire to operate at low voltages necessitates the need for guardbanding because of significant threshold voltage variations in those regimes. This guardbanding is usually a combination of using: (1) higher supply voltage levels; (2) larger transistors; (3) complex logic for error detection/correction, and (4) spare cells. Here, we focus mainly on the first technique.

There is a large body of work on fault-tolerant voltage-scalable SRAM cache designs that attempt to use the best combination of aforementioned guardbanding techniques which minimize power overhead while satisfying a yield threshold [9, 151, 152, 8, 21]. All of these approaches are application-agnostic and do not adapt the guardbanding to the requirements of application.

Findings of [148] show that masking all variabilities in 90nm SRAM requires an increase

in data retention voltage from nominal value of 0.35V to 0.7V. Reddi et al. [123] show that as feature size becomes smaller, more guardbanding is required. While 20% voltage guardbanding can improve error rate in 45nm by an order of magnitude, it can improve error rate of 16nm SRAM cell only by 3X.

We propose the idea of removing or reducing the typical guardbanding for the memories, more specifically for SRAM memories. Reducing guardbanding causes some faults to appear in memory. This class of memories can be used for the systems running approximate applications that can tolerate some level of error.

We present *Relaxed Cache* which exploits the application's behavior for adaptive relaxation of guardbands in cache memories to save energy. Unlike previous efforts on memories for approximate computing [130, 92], here this relaxation is done in a *disciplined* manner. Using language extensions, Relaxed Cache provides two knobs to the software programmer, which then (s)he uses for controlling the amount of guardbanding during different phases of execution. These knobs are: (1) SRAM array voltage (VDD), and (2) number of acceptable faulty bits in a cache block (AFB). The application developer also tags the approximate data objects. Later, based on this tagging, the cache controller knows if a piece of data is critical and should be protected, or if relaxed caching is acceptable and accordingly performs the block replacement. The knob adjustment along with this data tagging enable the programmer to guide the system to dynamically alternate between different cache operational points during runtime seeking the optimal point in each phase of execution. This optimality can be defined based on various metrics including energy, performance, output fidelity or a combination thereof.

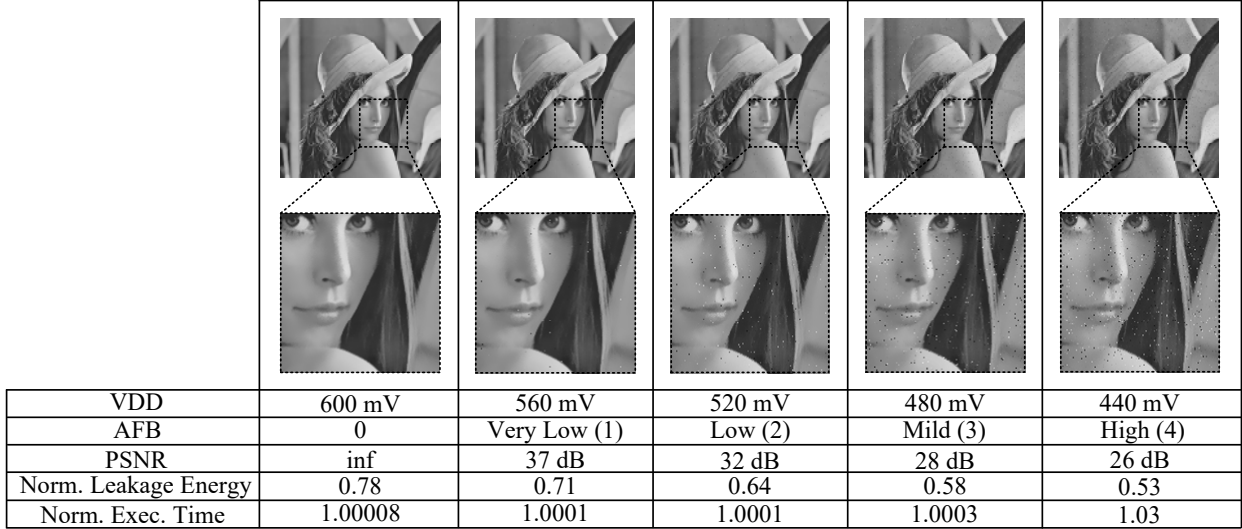


Figure 3.1: Exploring performance-energy-fidelity space for Image Smoothing benchmark by adjusting Relaxed Cache controlling knobs (leakage energies and execution times are normalized to a baseline that Uses 700 mV for SRAM array supply voltage).

3.4.2 Motivation

Here, using the Susan/Image-Smoothing benchmark from MiBench [63], we show how these knobs can be used to save leakage energy in the underlying cache with minimal impact on the performance and output fidelity of the application. This benchmark is commonly used for smoothing images in order to remove specks of dust and artifacts from scanning. Figure 3.1 shows the output of smoothing the Lena test image with different levels of AFB and VDD. Figure 3.1 also shows normalized leakage energy and execution time w.r.t. the baseline system that uses a cache with 700mV supply voltage. This example shows that depending on the acceptable quality for the output of this benchmark, the programmer can achieve up to 47% leakage energy saving by adjusting controlling knobs at the application-level.

Before detailing the Relaxed Cache approach, we present the rationale behind the performance-energy-fidelity trade-off offered by adjusting controlling knobs in caches. To increase memory density, memory bit-cells are typically scaled to reduce their area. High density SRAM bit-cells use the smallest devices in a technology, making SRAMs more

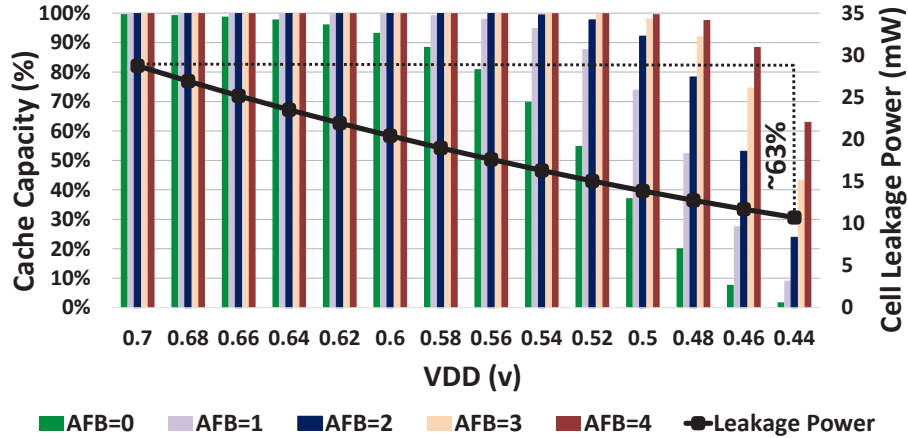


Figure 3.2: Trade-off between cache capacity, VDD, AFB and bit-cell leakage power (cache block size=64-byte, technology=45nm).

vulnerable to manufacturing variations. On the other hand, static power, dominated by sub-threshold leakage current, is the primary contributor of power in memories and has an exponential dependence on supply voltage [53]. Reducing supply voltage for saving power makes the SRAM even more vulnerable to variations, especially in 65nm and below, which results in an exponential increase in the probability of cell failure [149].

Figure 3.2 shows that the capacity of a cache with traditional assumption of 100% fault-free behavior (i.e., AFB=0), drops exponentially as we lower the voltage, which can dramatically degrade the performance of execution due to too many misses. This prevents the cache manufacturers to set the Vdd-min below a certain threshold. But if we consciously allow the SRAM to violate data integrity by increasing AFB to a value more than zero (as we did in the example shown in Figure 3.1), we can effectively lower this threshold depending on the level of tolerable errors in the application. Therefore, it's advantageous to let the application determine the (VDD, AFB) operational points of the cache. Even when no error is tolerable, the programmer can exploit the application's phasic behavior, e.g. lowering the cache voltage for an application's phase that will not require a large cache capacity. Note that moving from a voltage level to a lower voltage, even with same AFB level, helps the system to save energy.

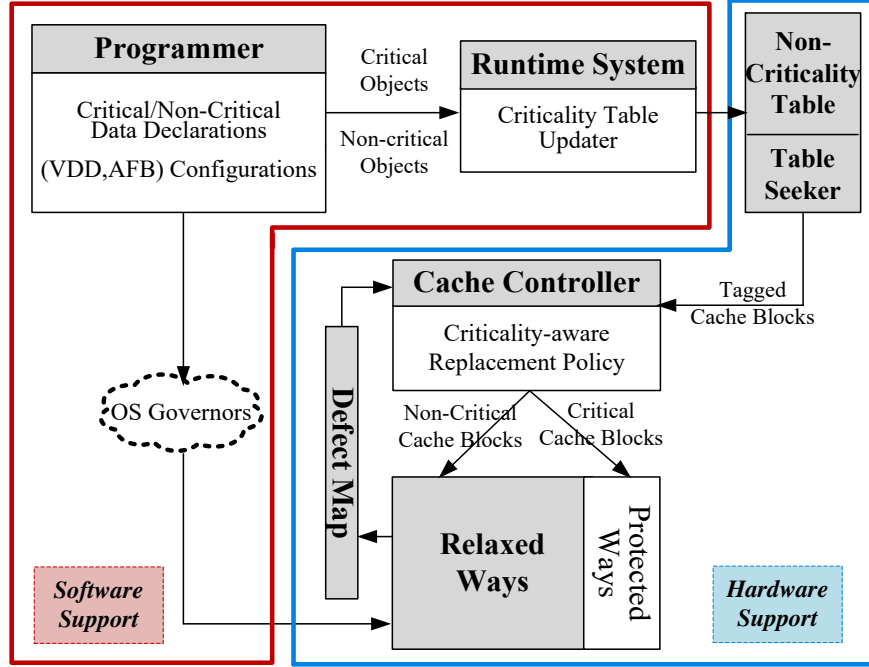


Figure 3.3: High-level diagram showing HW/SW components of Relaxed Cache and their interactions.

Relaxed Cache lets the application programmer adjust the cache guardbanding knobs and therefore adapt its reliability and capacity to the application’s demands. Accordingly, the application programmer needs to identify critical memory objects as well as major criticality and computational phases of the application.

Figure 3.3 shows the high-level overview of the Relaxed Cache scheme. Our scheme requires some small modifications to both the traditional hardware and software components of a system (shaded component of Figure 3.3). In the following, we describe the changes required for hardware and software support.

3.4.3 Hardware Support

Here, we present the hardware support for Relaxed Cache that requires the following minor changes to traditional caches (shaded components in the hardware blocks of Figure 3.3):

1. Tuning *Relaxed Ways* based on the architectural knob settings (AFB and VDD).
2. Modifying the *Defect Map* to store special defect information required by Relaxed Cache.
3. Adding a piece of on-chip memory that keeps the *Criticality Table* and comparators to search it.
4. Making the *Cache Controller* aware of critical vs. non-critical blocks.

We describe each in more detail below.

3.4.3.1 Tuning Relaxed Ways Based on Architectural Knobs

Consider a cache that has N ways. A fixed small number (e.g., $\log N$) of these ways are protected, by using a combination of high voltage and error correction codes that assure their fault-free storage(Protected Ways). The remaining cache ways in a cache set work with a lower dynamically adjustable voltage.

The operation of the SRAM array for relaxed ways is controlled by two architectural knobs: (1) Supply voltage (VDD), and (2) Number of acceptable faulty bits (AFB) per each cache block. Every block with more than AFB faults is disabled. The definition of AFB allows us to relax the guardbands for a majority of cache ways while controlling the level of error that is exposed to the program. The combination of AFB and VDD also determines the active portion of the cache, hence can be tuned for required performance.

According to these definitions, we can have four types of cache blocks for each (VDD, AFB) setting:

- Protected Block (PB): All blocks in the protected ways.

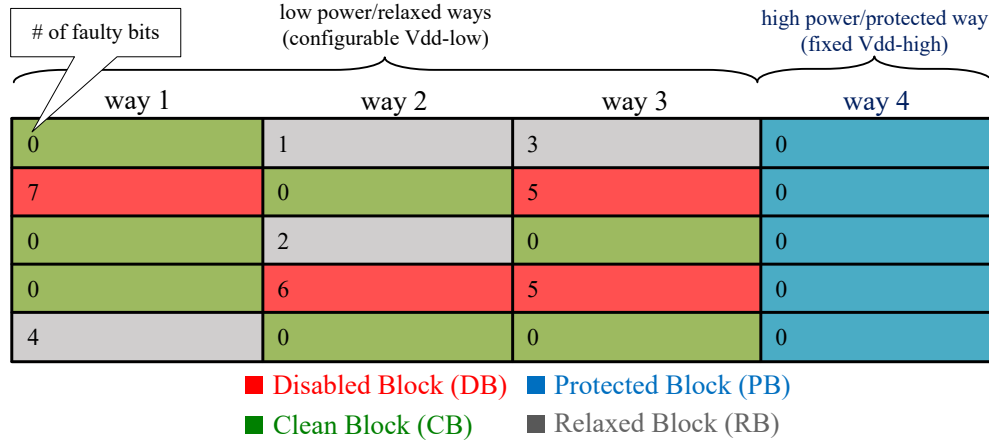


Figure 3.4: A Sample 4-way Cache with VDD=580mV and AFB=4.

- Clean Block (CB): All fault-free blocks in the relaxed ways.
- Relaxed Block (RB): All blocks in the relaxed ways that have at least one but no more than AFB number of faults.
- Disabled Block (DB): All blocks in the relaxed ways that have more than AFB number of faults.

Figure 3.4 demonstrates these terminologies using a sample 4-way cache when AFB=4.

3.4.3.2 Defect Map Generation and Storage

Disabling those blocks that have more than AFB number of faults requires a defect map for bookkeeping as in previous fault-tolerant voltage-scalable caches (e.g.,[21]). To populate the cache defect maps, we can use any standard Built-In Self-Test (BIST) routine that can detect memory faults (e.g., March Tests [64]). If BIST is done once at test-time, then a non-volatile (NV) storage will be needed to store the defect map for the duration of the device lifetime. If it is done at every power-up of the chip, then no extra NV storage is necessary.

The defect map bookkeeping requires a few additional bits in the tag array of each cache

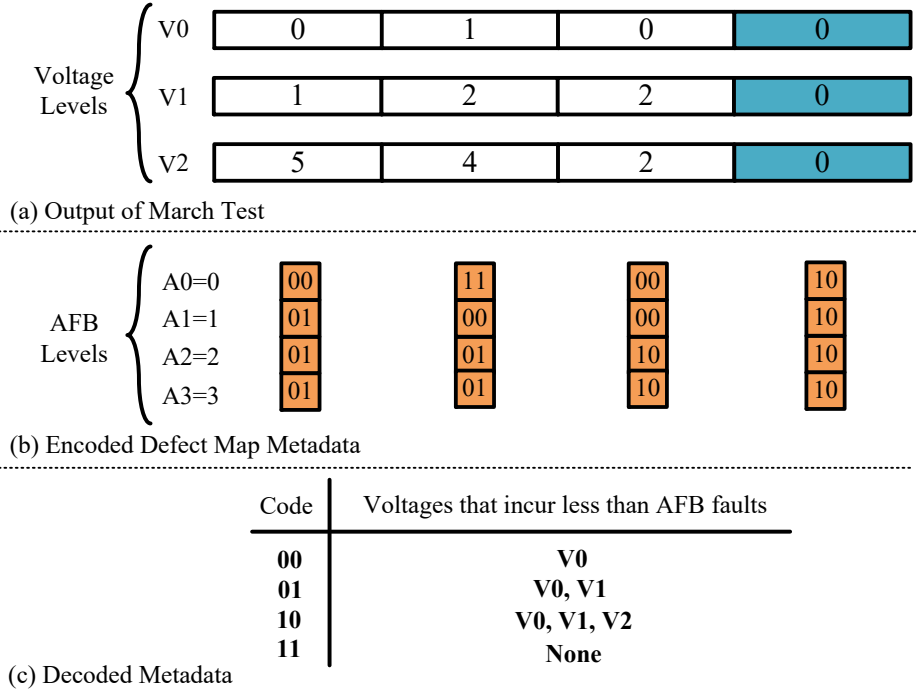


Figure 3.5: Encoding and decoding defect map info in Relaxed Cache

block. However, if the number of AFB and VDD levels are limited, this overhead is negligible. For instance, assume that we have four levels of AFB and three levels of VDD. We can show that in this case the defect map overhead is about 1.5% for a cache with block size of 64-byte. Assume that our VDD levels are $V0 > V1 > V2$. Similarly we have $A0 < A1 < A2 < A3$ for AFB levels. Figure 3.5(a) shows the result of running a march test on four blocks of a set in a 4-way cache for three different VDD levels. The digits inside blocks represent number of faulty bits for each block at that VDD. Note that blue blocks belong to a protected way, therefore they are fault-free for all VDDs. The fault inclusion property [58] states that the bits that fail at some supply voltage level will also fail at all lower voltages. Therefore by decreasing the voltage, number of faulty bits increases monotonically. We use this property and for each $AFB=A_i$ encode the lowest voltage level that results in A_i number of faults or less (Figure 3.5(b)). In this manner, for each 64-byte block we use 8 bits to capture defect status of that block in all available (VDD, AFB) pairs, resulting in a 1.5% area overhead for a traditional cache.

3.4.3.3 Non-Criticality Table

A dedicated piece of on-chip memory is used to store the application’s virtual addresses that can be approximated (a sample is shown in Table 3.6). Before replacing a block with an evicted block, this table is searched by hardware comparators to check if the block’s address is contained in any of the address ranges in this table. Accordingly the block is tagged.

The number of entries in the table is a design choice, and our experiments show that approximately 30 entries are sufficient. Thus the storage overhead is $30 * (32+32+1)$ bits \approx 4 cache blocks, which is less than 1% of cache capacity.

Table 3.6: A sample criticality table for Relaxed Cache

Valid	Start	End
1	0x100ef310	0x100ef79c
1	0x100ec940	0x100efdcc
0
0
0

Hardware comparators check if the block’s address is contained in one of the table entries. While the data is being fetched from L2 or main memory, these comparators sequentially search for the address inside the table. One comparator checks if the block’s start address is greater than Start and the other one checks if the block’s end address is less than End. The output of both comparators are ANDed to determine if the block is found in the table. The number of comparators is also a design choice. To keep the performance overhead of tagging low, more than 1 pair can be used. Note that since the access latency of L2 cache is in the order of 10 cycles, for a 30-entry table, the delay of the table searching can be masked using 3 pairs of comparators that would effectively make the performance overhead zero. These comparators will be activated only during cache misses (<10% of accesses). Our power analysis using Synopsys Design Compiler shows that total (dynamic and leakage) power consumption of this comparators will be about 1% of the total leakage power in baseline

mode. This means any leakage saving of more than 1% would effectively compensate this power overhead.

3.4.3.4 Making Cache Controller Aware of Block’s Tag

Relaxed Cache’s replacement policy needs to discriminate between critical and non-critical blocks. Whenever a miss occurs and the missed data block is tagged as a non-critical block the replacement policy should select a victim block from the relaxed ways of the corresponding cache set. However, a critical block is always allocated in a protected way. Note that for very aggressively-scaled voltage levels, it’s possible that all of the blocks in the relaxed ways become disabled based on the specified AFB. In this case, the replacement policy uses a block in one of the protected ways for bringing a non-critical block into the cache. This allows us to operate at very low voltage levels and trade cache capacity for power saving for applications that are not memory-bound in certain phases of their execution. Because block replacement logic is not in the critical path for hit read/write accesses, this minor modification does not affect cache access latency.

3.4.4 Software Support

We now describe changes to software components in Figure 3.3:

3.4.4.1 Programmer-Driven Application Modifications

3.4.4.1.1 Data Criticality Declaration

In order to utilize relaxed ways, the software programmer should identify non-critical data structures in the program to be mapped to those relaxed ways. We believe a software programmer can easily make this distinction since (s)he is aware of the functionality and

semantics of different parts of the application and related data structures. Frameworks like Rely [33] and ACCEPT [128] could also assist the programmer in identifying non-critical data objects in a program.

The memory footprint of an application has four segments: Code, Global, Stack, and Heap. The code segment usually can not tolerate any errors. Global, stack and heap can contain both critical and non-critical data. In our experiments we found most of the non-critical data to be allocated on heap. However, our implementation supports non-critical data in all memory segments.

The constructs `ADD_APPROX(Start-VA, End-VA)` and `REMOVE_APPROX(Start-VA, End-VA)` are used to declare and undeclare data non-criticality dynamically within code. `Start-VA` and `End-VA` are the virtual address boundaries of target region. The additional capability of undeclaring a data region becomes important in two cases: (1) When an error bounding procedure should be performed on non-critical data object before passing it to the next stage of program in order to reduce the propagated error magnitude; here, it is usually desirable to stop introducing any further errors in data after error bounding. (2) When a critical procedure should be done on a piece of non-critical data (e.g., computing checksum of pixels at the end of image compression).

3.4.4.1.2 Cache Configuration

The intuition behind letting the programmer configure the cache settings, is a key insight in today's energy management techniques for systems and applications: *they respond to phases*. Accordingly, our scheme leverages the programmer's knowledge for managing energy consumption. The programmer, with in-depth knowledge of the software is in the best position to identify different phases of application and based on that manages the settings of the underlying hardware. On the other hand, a programmer can respond to the system's operational mode by devising provisions that adapt the execution to these modes [42]. For

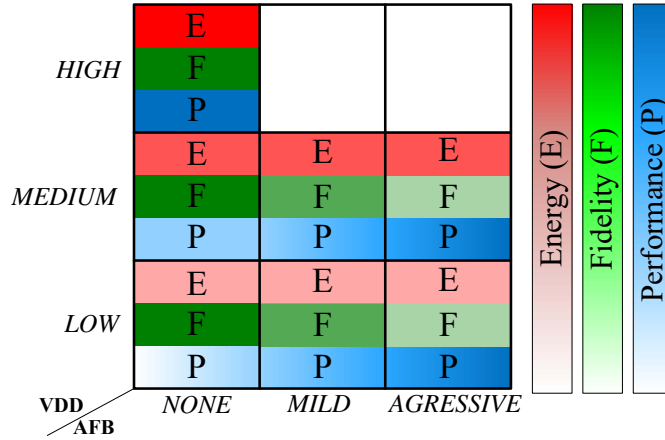


Figure 3.6: Abstracting Relaxed Cache knobs up to metrics familiar to a software programmer (i.e., performance, fidelity, and energy consumption). Note that (VDD = High, AFB = Mild) and (VDD = High, AFB = Aggressive) combinations are sub-optimal, hence not applicable.

example, a programmer can decide to render a high-fidelity image when the user’s smart-phone is fully charged, and only a low-fidelity image when the battery level of the phone falls below 25%.

To utilize the capabilities offered by Relaxed Cache, the software programmer can annotate the code regions with cache configuration hints. These hints are then used by the system to dynamically control the VDD and AFB knobs to adjust the guardbanding to the current phase of execution. The programmer, with in-depth knowledge of the software is in the best position to identify application phases with different criticality and/or memory-insensitivity and based on that guide the adaptive guardbanding in hardware.

To accommodate programmers with different levels of hardware familiarity, we can abstract the Relaxed Cache knobs as shown in Figure 3.6. An embedded system programmer, with fairly good understanding of hardware details, can explicitly set Relaxed Cache’s knobs to fully exploit its capabilities. On the other hand, for a traditional software developer (with little knowledge of hardware), we can abstract the hardware knobs settings into discretized “levels” allowing the programmer to qualitatively set the knobs based on their effects (e.g., low power setting, or high-fidelity setting), as shown in Figure 3.6.

```

#include "Approximations.h" // Enables approximation annotations

int main(int argc, char**argv){

    // Capturing Raw Image on Camera
    // Compress Raw Image

    /*----- Image Scaling -----*/
    CONFIG_CACHE(MEDIUM VDD, MILD AFB);

    int *src, *dest;
    dest = (int*) malloc(*num_elmnts*sizeof(int));

    #ifdef APPROX_SRC
        ADD_APPROX((uint64_t)src, (uint64_t)(src+num_elmnts));
    #endif

    #ifdef APPROX_DEST
        ADD_APPROX((uint64_t)dest, (uint64_t)(dest+num_elmnts));
    #endif

    src = read_image(in_filename,&sw,&sh,&src_dim);
    dest = allocate_transform_image(scale_factor,sw,sh,&dw,&dh,&dest_dim);

    scale(scale_factor, src, sw, sh, dest, dw, dh);

    #ifdef APPROX_SRC
        REMOVE_APPROX((uint64_t)src, (uint64_t)(src+num_elmnts));
    #endif

    #ifdef APPROX_DEST
        REMOVE_APPROX((uint64_t)dest, (uint64_t)(dest+num_elmnts));
    #endif

    free(src);
    /*-----*/

    // Other Image Transformations/Editing

}

```

Figure 3.7: A sample code showing programmer’s data criticality declarations and cache configurations for Relaxed Cache.

Language extensions then let the programmer use `CONFIG_CACHE(VDD-LEVEL, AFB-LEVEL)` for configuring the cache knobs. Since this reconfiguration incurs a penalty of several cycles, it should be done only in certain phase transitions of an application.

Figure 3.7 shows an image resizer code fragment that uses declarations and cache configuration

to save energy. In this example, two large regions of data that can tolerate some level of errors are the original image’s data structure (pointed to by `src`), and the data structure for storing up-scaled image (pointed to by `dest`); the programmer has declared both regions as candidates for relaxed caching. After returning from the `scale()` function, and before moving to the next program phase, (s)he has undeclared both regions.

3.4.4.2 Runtime System

We assume a runtime system uses the programmer’s declarations to keep and update a table of virtual addresses that contain non-critical data objects for each application. During a miss, while the data is fetched from the lower level of memory (L2 data cache or main memory), the comparators in the table seeker sequentially search for the block’s address inside the table; the table seeker then tags the block as critical/non-critical. This tagging is used by cache controller’s replacement mechanism to find an appropriate cache way for replacement. Note that, in order for a block to be tagged as non-critical, the entire data block should hold non-critical data. A data block with mixed critical/non-critical data would still be tagged as critical. Further optimization is possible through criticality-aware data placement during compilation to avoid having mixed criticality blocks, which is out of the scope of this work.

3.4.5 Evaluations

3.4.5.1 Experimental Setup

We modified the cache architecture in the gem5 framework [29] to implement our scheme in detail and used gem5’s pseudo-instruction capability for implementing the required language extensions. The common gem5 parameters used in all our simulations are summarized in Table 3.7. We used the *Random* block replacement policy that is commonly used in embedded

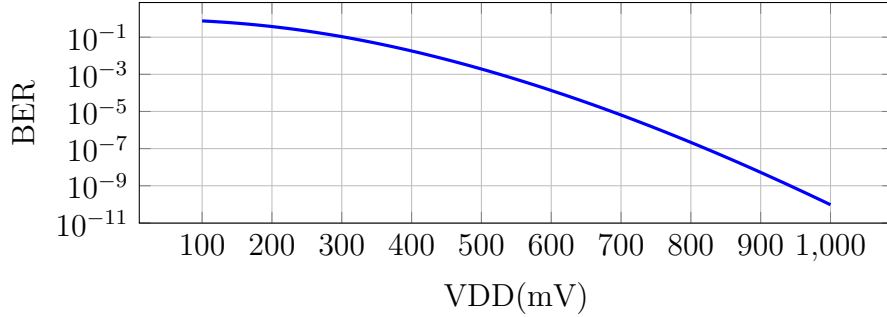


Figure 3.8: SRAM BER for 45nm using models and data from [149]

processors [11, 12] due to its minimal hardware and power cost.

Table 3.7: gem5 settings for Relaxed Cache experiments

Parameter	Value	Parameter	Value
ISA	Alpha	Replacement Policy	Random
CPU Model	Detailed (OoO)	Cache Block Size	64B
No. Cores	1	L1\$ Size, Assoc.	32KB, 4-way
Cache Config.	L1 (Split), L2	L2\$ Size, Assoc.	256KB, 8-way
Simulation Mode	Syscall Emul.	Main Memory	LPDDR3, 512MB

We injected errors into different cache blocks randomly, with uniform distribution over different bit positions. Our SRAM bit error rates (BER) are shown in Figure 3.8. These were computed by [58] using the data and models from [149], which performed a detailed analysis of SRAM noise margins and error rates for a commercial 45nm technology. Using these BERs, we found the distribution for number of faults for each data array voltage, thus allowing us to compute the expected cache capacity in each AFB.

Cache power consumption is estimated based on the model in [58]. CACTI 6.5 [111] is modified to extract static power for the baseline and Relaxed Cache. Note that baseline cache does not have any fault tolerance and does not support voltage scalability.

3.4.5.2 Benchmarks

We selected a number of multimedia benchmarks to examine the energy savings enabled by Relaxed Cache.

1) **Scale**: Scale is an image resizer application. We use Peak-Signal-to-Noise Ratio (PSNR) as the fidelity metric for this application. Table 3.8 shows the relation between PSNR and perceptual quality.

Table 3.8: Relation between PSNR and perceptual quality in image processing domain

PSNR	Perceptual Quality
< 28.0	Low
$28.0 \leq \dots < 30.0$	Acceptable
$30.0 \leq \dots < 33.0$	Good
≥ 33.0	Excellent

2) **Susan**: Susan is an image recognition package from MiBench [63]. We used Image-Smoothing and Edge-Detection kernels from this package. PSNR is used as the fidelity metric for Image-Smoothing. For Edge-Detection, accuracy metric is defined as the fraction of detections that are true positives rather than false positives. A value of 0.8 or more is usually considered acceptable.

3) **x264**: This application is an H.264 video encoder [28]. PSNR of 32dB can provide satisfactory video quality for many types of videos.

3.4.5.3 Experimental Results

3.4.5.3.1 Leakage Energy Savings

Figure 3.9 summarizes the achievable leakage energy savings for different Relaxed Cache knob settings (VDD, AFB). The baseline cache is assumed to use 700mV for supply voltage. We assume power-gating for disabled blocks. Consequently, if we keep voltage constant

and increase AFB (effectively reclaiming faulty memory blocks), the leakage energy saving decreases. Also note that for voltages $\geq 540\text{mV}$ increasing AFB does not reduce energy savings. This is due to the low number of faulty blocks at those voltages. This analysis shows that we can decrease leakage energy up to 74% by adjusting the Relaxed Cache knob settings.

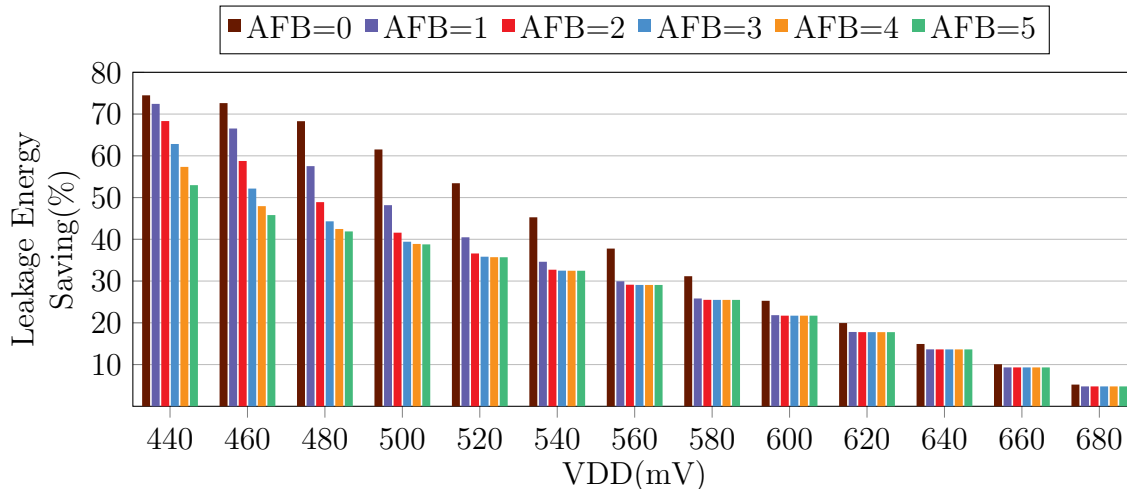
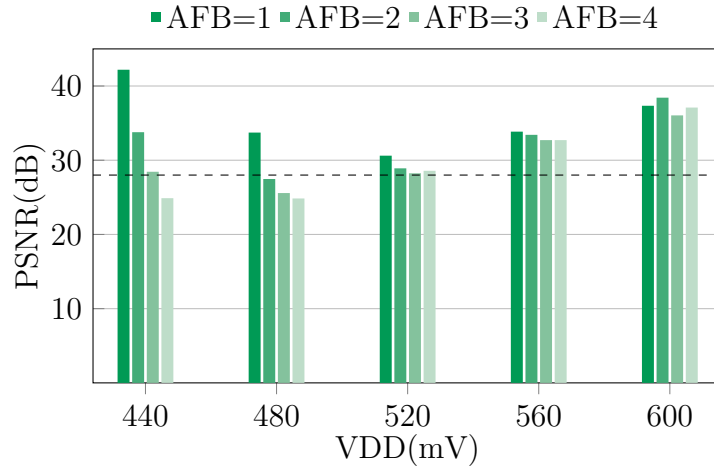


Figure 3.9: Leakage energy savings for a 4-Way L1 cache with 3 relaxed ways (energy savings are normalized to a baseline cache that uses 700mV).

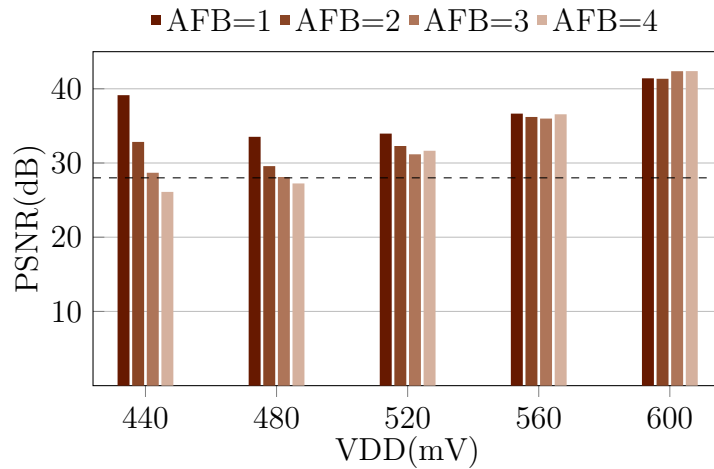
3.4.5.3.2 Fidelity Analysis

Figure 3.10a shows the PSNR difference between the images up-scaled using Scale benchmark running on a system with energy-efficient Relaxed Cache versus a system that uses a guardbanded baseline cache.

The PSNR values for Image-Smoothing kernel of Susan package are reported in Figure 3.10b. Same as Scale benchmark, many of the (VDD, AFB) settings result in an acceptable output. However, it is interesting to note that on average the PSNR values are larger than the ones reported for Scale. This shows that the Image-Smoothing is more error-tolerant and therefore the programmer can use a more aggressive policy for that kernel. This observation confirms that different applications have different levels of error-tolerance which can be exploited for energy saving.



(a) Scale



(b) Image-Smoothing

Figure 3.10: Fidelity results for (a) Scale and (b) Image-Smoothing benchmarks.

Susan/Edge-Detection is the most error-tolerant application that can produce good results even at high error rates. Figure 3.11 shows the result of Susan/Edge-Detection on the Lena test image with different AFBs while VDD is set to 480mV. We can see that even AFB=6 (which reclaims 99% of the Relaxed Cache faulty blocks at that voltage) can result in an acceptable output.

3.4.5.3.3 Performance Analysis

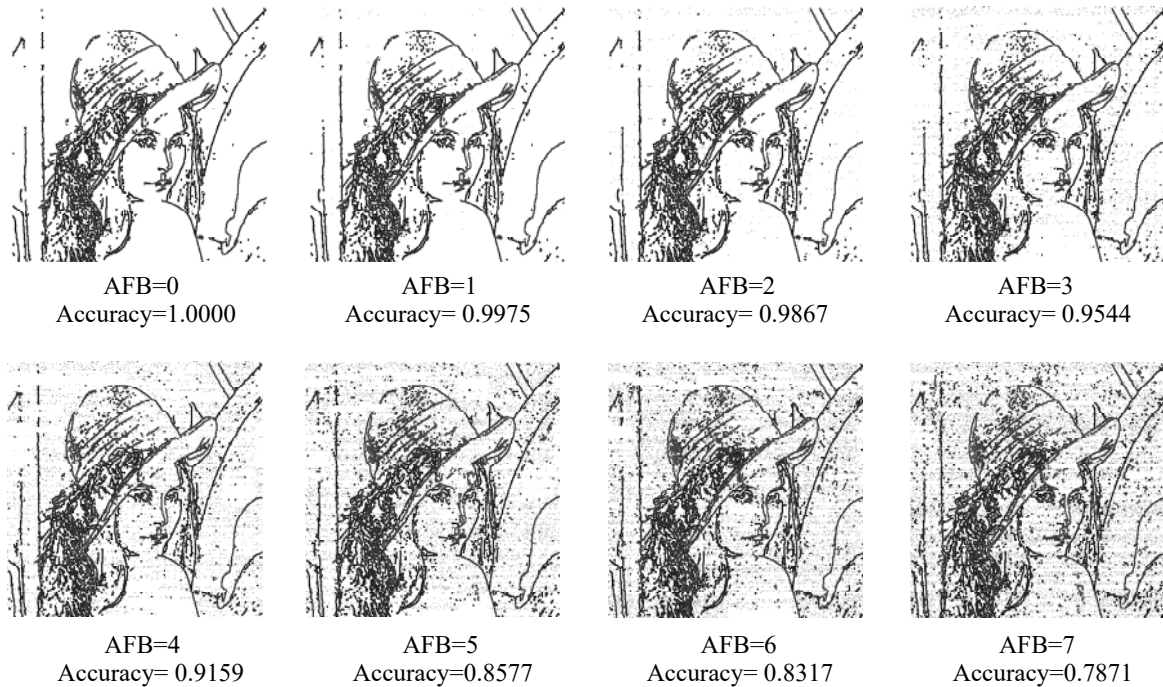


Figure 3.11: Fidelity results for Edge-Detection benchmark (VDD=480mV).

Operating at low voltages effectively disables cache regions affected by errors. Reduced cache capacity has minor impact on the performance of programs with small working set size. The degradations in total execution times are bounded by 3% for Scale and Edge-Detection benchmarks and 2% for Image-Smoothing benchmark over all (VDD, AFB) settings in our experiments. However, for applications with big working set size, the increased cache miss rate due to reduced cache capacity considerably degrades the performance. But, Relaxed Cache’s AFB knob allows us to reclaim parts of the lost cache capacity, thereby increasing performance of the application. This is particularly useful for application instances where a partial/less-accurate result generated within a deadline is more valuable than a late-but-perfect result. Relaxed Cache provides the means for a programmer to explore this trade-off space, by deciding when higher performance is desirable and when higher output fidelity is preferable.

For instance, Figure 3.12 shows the adverse effect of reducing the voltage for the H.264 encoder: x’s correspond to PSNR loss and circles mark FPS loss. Due to the reduced cache

capacity, a normal cache (Baseline in blue) would see 38% decrease in FPS, while maintaining the PSNR quality. On the other hand, Relaxed Cache (in red) trades off the PSNR quality for less degradation in FPS (up to 9.5%). However, in all the experiments the PSNR of degraded-quality video is still above 32dB threshold, yielding acceptable quality for the user.

3.5 QuARK Cache

3.5.1 Introduction

Deployment of SRAM memories in embedded systems is greatly challenging in advanced VLSI technologies in particular in sub-45nm features sizes [120] due to high leakage power, reliability issues as well as low density, making them less appealing for modern embedded systems. These limitations have led to the development of alternative memory technologies with different energy-performance-reliability characteristics such as Spin Transfer Torque Magnetic RAM (STT-MRAM). STT-MRAM offers a high-density, high-speed, non-volatile choice of random access memory, however, it suffers from a critical reliability issue, called *stochastic switching* [46], which if properly handled can make STT-MRAM a promising

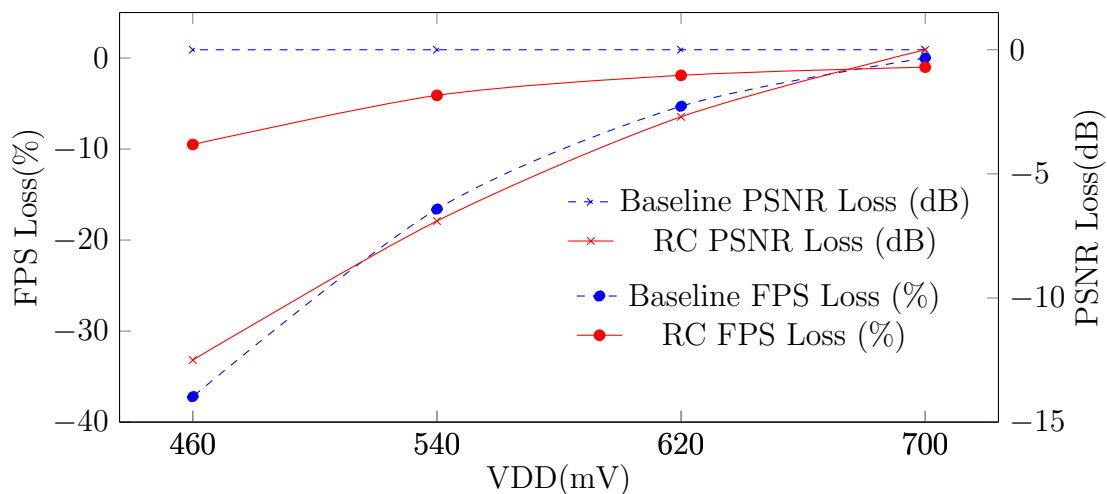


Figure 3.12: FPS-PSNR trade-offs with and without Relaxed Cache scheme (AFB=4).

replacement for SRAM in many applications.

Stochastic switching affects both read and write operations. It affects the read operation by causing the *read disturbance* phenomenon which is defined as the process of unintentionally changing the stored value in a STT-MRAM cell while reading the cell. Similarly, a write operation may not succeed in correctly changing the value of the cell and a *write failure* occurs.

Conventional approaches that address these sources of unreliability to preserve data integrity use a conservatively high current for write operation as well as incorporating ECC to recover the potential errors [140, 77]. Both solutions significantly exacerbate the energy consumption of STT-MRAMs, counteracting some of the advantages of STT-MRAM over SRAM.

We present QuARK Cache, a hardware/software approach for trading reliability of STT-MRAM caches for energy savings in the on-chip memory hierarchy of systems running approximate applications. QuARK Cache utilizes fine-grained actuation knobs to efficiently control reliability-energy trade-offs for individual accesses of concurrently running applications. The STT-MRAM approximation knobs considered in QuARK Cache are read and write current amplitudes.

Compared to Relaxed Cache (Section 3.4) for SRAM caches that uses cache-way-level knobs, QuARK presents a more fine-grained actuation capability enabling it to offer the following advantages: (1) the knob actuations do not affect any other cache block unlike the actuation in Relaxed Cache which requires flushing all the affected blocks, (2) multiple applications with different degrees of reliability can share the same cache without affecting each other's guaranteed level of reliability, and (3) the reliability level requested for a piece of data can be changed at runtime, if the quality of output is not satisfactory.

3.5.2 STT-MRAM Reliability/Energy Trade-off Knobs

In this section, we first review the structure of a STT-MRAM cell and the mechanisms of reading from and writing to it. Then, we discuss the available circuit-level knobs in STT-MRAMs which can be used to trade accuracy for energy.

3.5.2.1 STT-MRAM Basics

The standard STT-MRAM cell (1T-1J) includes a Magnitude Tunnel Junction (MTJ), and an access transistor. MTJ consists of an oxide barrier layer that is sandwiched between two ferromagnetic layers. One of the ferromagnetic layers has fixed magnetic field direction (i.e., reference layer), while the magnetic field direction of the other layer can be changed (i.e., free layer). Relative magnetic field direction of these layers delivers different resistances use to store values. STT-MRAM cell stores a value, based on the resistance of MTJ. If the magnetic field directions of two ferromagnetic layers are in parallel state, MTJ delivers a low resistance. Otherwise it manifests as a high resistance. Read and write operations in STT-MRAM cell are performed by applying either a small current to read MTJ resistance by sense amplifier, or a high current to change the resistance (i.e., write a new value) in the MTJ.

To read the value of a STT-MRAM cell, a small current is applied to measure the resistance of MTJ by a sense amplifier. Similarly, during a write operation, current should be applied to MTJ to change the magnetic field direction of the free layer in order to write a value in the STT-MRAM cell. The direction of applied write current is determined based on the required magnetic field direction in the free layer.

3.5.2.2 Reliability-Energy Knobs

As noted previously, STT-MRAM switching is a stochastic operation. Therefore, read and write operations can be performed at different levels of reliability and consequently energy consumption [110, 88].

Write Operation: During a write operation, depending on the amount and duration of the applied write current, the direction of the magnetic field of the free layer in MTJ may not change, which can result in a write failure. The write failure probability can be calculated by Equation (3.1) [97]:

$$P_{wf}(t_w) = \exp\left(-t_w \times \frac{2\mu_B p (I_w - I_{C_0})}{(c + \ln(\Pi^2 \frac{\Delta}{4})) \times (em(1 + p^2))}\right) \quad (3.1)$$

where Δ is the thermal stability factor, I_{C_0} is the critical MTJ switching current at 0°K, c is the Euler constant, e is the magnitude of electron charge, m denotes the magnetic momentum of the free layer, p is the tunneling spin polarization, μ_B is the Bohr magneton, I_w is the write current, and t_w is the write pulse width. I_w is one of the effective circuit-level knobs available to control the reliability-energy trade-off during a write operation. Generally, a lower I_w decreases the write energy, but it also amplifies the probability of a write failure.

Based on the data reported in [122], we modeled a 1MB STT-MRAM cache in NVSim [48]. Figure 3.13(a) shows the write energy vs. write error rate of this cache at different write currents. We see that modulating I_w leads to savings in write operation energy, e.g., 2X improvement for an error probability of 9×10^{-4} .

Read Operation: During a read operation, depending on the amount and duration of the applied read current, an unintentional magnetic field direction flip in the free layer of MTJ may happen due to stochastic switching which can result in read disturbance. The read disturbance probability can be calculated using Equation (3.2) [144]:

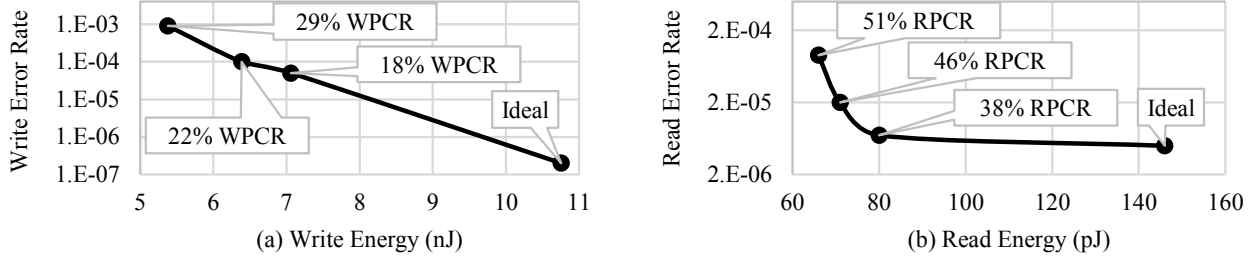


Figure 3.13: STT-MRAM knobs for reliability-energy trade-off in 1MB cache. (a) Write Pulse Current Reduction (WPCR), and (b) Read Pulse Current Reduction (RPCR).

$$P_{rd} = 1 - \exp\left(-\frac{t_r}{\tau} \times \exp\left(-\frac{\Delta(I_{C_0} - I_r)}{I_{C_0}}\right)\right) \quad (3.2)$$

where τ is the attempt period, I_r is the read current, and t_r is the period of read pulse. Unlike write operation, both the read energy consumption and the read disturbance probability are decreased by reducing I_r . However, another constraint prevents us from applying a very small read current. Lowering the read current amplitude below a pre-defined sense amplifier threshold leads to different type of read errors in STT-MRAM which is called *decision failure*.

Accordingly, I_r is one of the effective circuit-level knobs to control the reliability-energy trade-off during a read operation. Figure 3.13(b) depicts the reliability-energy trade-off for read operation of aforementioned 1MB STT-MRAM cache. We see that decreasing the read pulse current results in energy savings for read operation. For example, modulating I_r leads to over 2.2X improvement in read energy consumption for an error probability of 9×10^{-5} .

3.5.3 The QuARK Approach

QuARK Cache enables a system with STT-MRAM caches running approximate computing applications to trade accuracy of storage in on-chip memory for energy savings. It captures the

information about non-critical data objects from software and adjusts the reliability-energy knobs accordingly. This section presents the details of software and hardware supports for QuARK Cache.

3.5.3.1 Software Support

The software support required for QuARK Cache is very similar to the one described for Relaxed Cache (Section 3.4.4). QuARK Cache provides two API calls to the programmer: `ADD_APPROX` and `REMOVE_APPROX`. Table 3.9 shows their formats. `BaseVA` is the base address of the non-critical data object, `Size` is the size of non-critical data object, and `ReliabilityLevel` is the required reliability guarantee for that data object. Note that `ADD_APPROX` receives an additional parameter (i.e., `ReliabilityLevel`) in comparison to Relaxed Cache.

The QuARK Cache APIs communicate with the QuARK Cache hardware support (introduced in Section 3.5.3.2) to pass the information about non-critical data objects and their acceptable reliability level. There are two possible approaches for integrating these two APIs into the system: 1) The ISA of the processor can be modified to support the QuARK Cache APIs. With this approach, the processor has to be modified to be able to directly communicate with the QuARK Cache hardware support. 2) The QuARK Cache APIs can be implemented within a special runtime library. With this approach, hardware components of QuARK Cache become memory-mapped interfaces. The runtime library uses normal read/write instructions

Table 3.9: QuARK Cache APIs

Method	Parameters	Note
<code>ADD_APPROX</code>	<code>BaseVA</code> <code>Size</code> <code>ReliabilityLevel</code>	Base virtual address of approx. memory region Size of the approx. memory region Required reliability guarantee
<code>REMOVE_APPROX</code>	<code>BaseVA</code> <code>Size</code>	Base virtual address of approx. memory region Size of the approx. memory region

```

unsigned *image;
int reliability_level = RL0;
...
image = (unsigned int*)malloc(WIDTH*HEIGHT*sizeof(unsigned));
if(approximation_enabled) {
    switch(get_environment_lighting()) {
        case 0~30 :
            reliability_level = RL0;
        case 30~70 :
            reliability_level = RL1;
        case 70~100:
            reliability_level = RL0;
    }
}
ADD_APPROX(image,WIDTH*HEIGHT*sizeof(unsigned),reliability_level);
...
load_image(image);
face_detection(image);
...
REMOVE_APPROX(image,WIDTH*HEIGHT*sizeof(unsigned));
...

```

Figure 3.14: A pseudo-code example showing how QuARK Cache APIs can be used in a face detection application.

to transfer the information provided by the APIs to the hardware.

Figure 3.14 shows an example on how QuARK Cache APIs can be used in the source code of a hypothetical face-detection application. According to [54], the environmental lighting affects the quality of face detection algorithm. Thus, considering a reasonable output quality in the worst case environmental lighting (when the lighting is either very low or very high), we can trade quality for energy saving in the cases that lighting is in normal condition (30~70 in this example).

3.5.3.2 Hardware Support

Figure 3.15 shows how the required hardware support for QuARK Cache can be integrated into the architecture. Deploying QuARK Cache for L1 private and L2 shared caches of memory hierarchy in a multicore architecture requires adding two types of modules: (1) QuARK Cache approximation table and (2) QuARK Cache controllers. Additionally, the

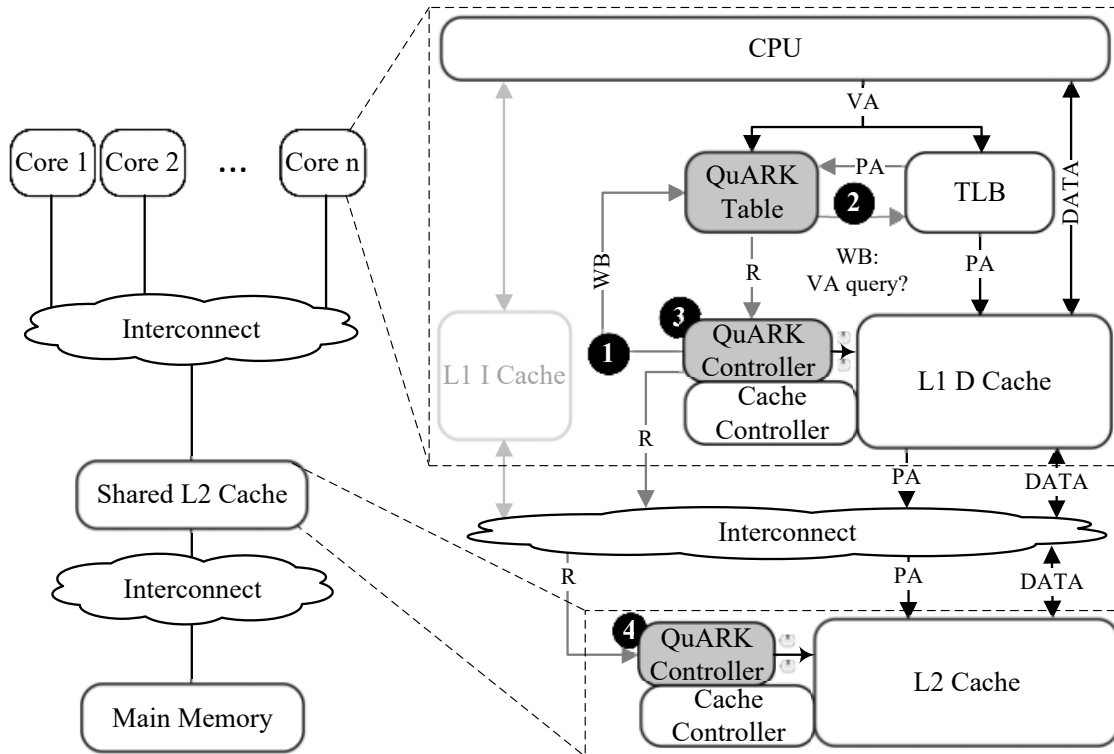


Figure 3.15: Integrating QuARK Cache into the architecture. Required changes are highlighted in gray.

interconnect should be modified to carry the reliability level information. These modifications are highlighted in gray in Figure 3.15.

3.5.3.2.1 QuARK Cache Approximation Table

QuARK Cache approximation table is responsible for storing the information provided by the QuARK Cache APIs. Each core includes a private approximation table. Anytime an `ADD_APPROX` is called in the thread running on that core, its parameters (i.e, baseVA, size, and reliability level) are saved in one of the rows of this table. Similarly, `REMOVE_APPROX` removes all or part of the information stored in a row.

This table is placed next to the Translation Look-aside Buffer (TLB) of each core. For every memory access, and during the TLB lookup, the QuARK Cache approximation table is also searched. If the virtual address of the memory access falls within the boundaries of one of the

rows in the table, the reliability of that access is set according to the reliability level stored in the corresponding row. For accesses that do not hit any row in the table, the reliability level is set to the maximum possible reliability. QuARK Cache controls STT-MRAM knobs at cache block granularity. The virtual address ranges provided by the APIs should be aligned to cache block boundaries, because the cache blocks that contain a mix of critical and non-critical data should not be approximated. An address alignment module embedded in the approximation table performs the required alignment. The addresses provided by the QuARK Cache APIs that will be stored in QuARK Cache approximation table are Virtual Address (VA). However, in most architectures (e.g., ARM), caches work with Physical Addresses (PA).

3.5.3.2.2 QuARK Cache Controller

The reliability level for each access is set by looking up the QuARK Cache approximation table. Then the memory request, now augmented with a reliability level, is passed to QuARK Cache controller(s). QuARK Cache controller sits next to the cache controller and it is responsible for setting the STT-MRAM reliability-energy trade-off knobs (i.e., read current and write current) for each access. The feasibility of designing efficient peripheral circuits for changing the current (voltage) level during the runtime is evaluated in [70, 85, 122]. QuARK Cache controller receives the reliability level set by the approximation table and selects the minimum current setting that satisfies the proper level of reliability for that access. One QuARK Cache controller is needed for each private L1 cache and one for each shared L2 cache.

3.5.3.2.3 Support for Cache Fillings and Write-backs

Special consideration might be needed for interactions between L1 and L2 caches anytime there is an L1 data cache miss.

■ L1 Cache Filling: Anytime a cache miss occurs in L1 data cache, a cache filling is required.

The memory access that resulted in a cache miss is already augmented with a reliability level when the approximate table was searched during TLB lookup. The QuARK Cache controller will use this reliability level to fill the L1 cache line when the request is served by the L2 cache.

■ **L2 Cache Filling:** Cache filling in L2 is required whenever a L1 cache request is missed in the L2 cache. Similar to the previous case, the reliability level of this request is already provided by the approximation table. The QuARK Cache controller of L2 will use this reliability level during the filling operation of L2 cache line from upper-level memories.

■ **Write-backs from L1 to L2:** There are some times that L1 cache controller decides to evict a dirty cache line from L1 cache. This happens when either a new block should be replaced by dirty a one, or the last modified version of the block in L1 cache should be written to upper-level memories. For write-back from L1 data cache to L2 cache, QuARK Cache follows a different flow. Following Figure 3.15, the detailed steps to write back a dirty block from L1 data cache to L2 cache are as follows:

- 1) QuARK Cache controller in L1 data cache sends the PA of dirty block to the QuARK Cache table.
- 2) QuARK Cache table produces the VA queries for all of its entries and checks the TLB PA output to find the PA of dirty block in address intervals of entries. Then, QuARK Cache table determines the reliability level of this write back request.
- 3) The QuARK Cache controller of L1 data cache reads the data of dirty block with the provided reliability level and passes the data and reliability level to L2 cache.
- 4) The QuARK Cache controller of L2 cache adjusts the current level and performs the L1 data write back operation.

■ **Write-backs from L2 to upper-level memory:** Write-backs from L2 cache are issued due

to the same reasons that mentioned for L1 write-back requests. For all of the write-back operations from L2 cache, QuARK Cache considers the highest reliability level for reading the victim blocks from L2 cache and writing them to the upper-level memory.

3.5.4 Evaluations

In the following, we show experiments with a mix of approximate applications to evaluate the QuARK Cache’s capabilities in saving energy in the on-chip memory hierarchy under different levels of accuracy demanded by applications.

3.5.4.1 Experimental Setup

In order to implement our scheme in detail, we modified the cache architecture in the gem5 framework [29] for a multicore architecture. In this work, to assess the efficiency of the QuARK Cache in enhancing the energy characteristics of multicore architectures, we enable QuARK Cache for L2 cache as an exemplar. We used gem5’s pseudo-instructions for implementing QuARK Cache’s language extensions. The common gem5 parameters used in all our simulations are summarized in Table 3.10.

We randomly injected faults into different cache blocks during read and write operations, with uniform distribution over different bit positions. We used bit error rate (BER) and access current data from [122] and modeled a 1MB STT-MRAM cache in NVSim [48] to extract

Table 3.10: gem5 settings for QuARK Cache experiments

Parameter	Value	Parameter	Value
ISA	ARMv7-A	L1 \$ Size, Assoc.	64KB, 4
No. of Cores	4	L2 \$ Size, Assoc.	1MB, 16
Cache Configuration	L1 (Private) L2 (Shared, QuARK Cache-enabled)	Cache Block Size	64B

Table 3.11: Accuracy-energy transducer map for 1MB QuARK Cache-enabled STT-MRAM cache. Energy consumptions are reported for a 64-byte cache line.

Accuracy Level	Read Current	Read Error Rate	Read Energy	Write Current	Write Error Rate	Write Energy
L0 (Baseline)	$49\mu A$	protected	$0.146nJ$	$653\mu A$	protected	$10.755nJ$
L1	$30\mu A$	7×10^{-6}	$0.080nJ$	$529\mu A$	5×10^{-5}	$7.059nJ$
L2	$26\mu A$	2×10^{-5}	$0.071nJ$	$503\mu A$	1×10^{-4}	$6.386nJ$
L3	$24\mu A$	9×10^{-5}	$0.066nJ$	$461\mu A$	9×10^{-4}	$5.378nJ$

Table 3.12: List of approximate applications for QuARK Cache experiments.

Benchmark	Domain	Quality Metric
Corner Detection	Computer Vision	Mean Pixel Difference
Edge Detection	Computer Vision	Mean Pixel Difference
Image Smoothing	Computer Vision	PSNR
Blackscholes	Financial Analysis	Average Relative Error
Image Scale	Multimedia	PSNR
Sobel Filter	Computer Vision	Mean Pixel Difference

energy consumption. The details of error rates and energy consumptions of the modeled cache can be seen in Table 3.11.

3.5.4.2 Benchmarks

Table 3.12 lists the RMS applications we use in our evaluations and their quality metrics. We annotated each application by inserting `ADD_APPROX` and `REMOVE_APPROX` in the source code for non-critical data objects .

Figure 3.16 shows the distribution of overall and approximate read/write operations in L2 cache for the benchmarks listed in Table 3.12 when they run on a single-core configuration. As it can be seen, on average, 80% of accesses to L2 cache in these benchmarks can be done in the approximate mode, showing the significant energy saving potentials in such applications. Furthermore, about 45% of these accesses are for read requests and 35% are for write requests.

To evaluate the efficiency of QuARK Cache, we consider the effects of multi-programming

Table 3.13: List of workload mixes for QuARK Cache experiments.

Workload Mixes	Benchmarks	RLs
<i>Comb1</i>	(Corner Detection, Sobel, Image Smoothing, Blackscholes)	(RL1, RL3, RL2, RL3)
<i>Comb2</i>	(Scale, Blackscholes, Sobel, Image Smoothing)	(RL2, RL2, RL1, RL3)
<i>Comb3</i>	(Sobel, Corner Detection, Scale, Edge Detection)	(RL2, RL3, RL2, RL1)
<i>Comb4</i>	(Blackscholes, Scale, Edge Detection, Image Smoothing)	(RL3, RL1, RL2, RL2)

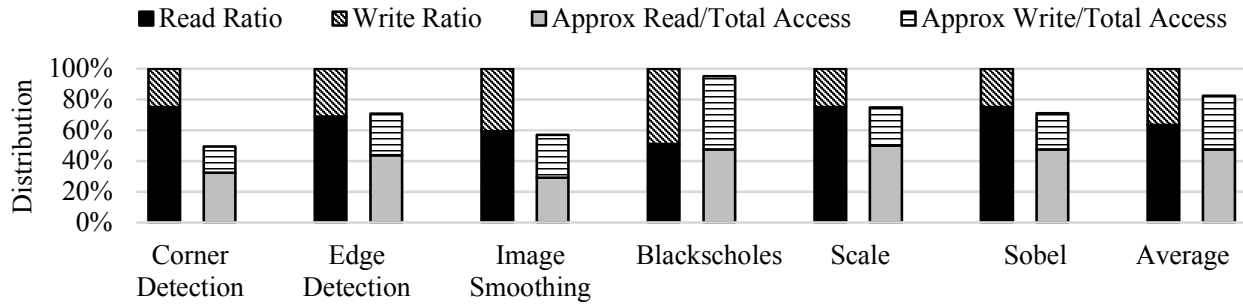


Figure 3.16: Distribution of overall and approximation read and write accesses in L2 cache for the selected benchmarks.

in a shared L2 cache equipped with QuARK Cache. To this end, we run different mixes of benchmarks mentioned in Table 3.13 in two modes: fixed-reliability mode, and mixed-reliability mode. In fixed-reliability mode, we run each workload mix in four reliability levels (RLs): RL0 (i.e., fully-protected), RL1, RL2 and RL3 (i.e., least-reliable) configurations. In the mixed-reliability mode, we consider different levels of reliability for each benchmarks in each combination. Our empirically-chosen criteria to set the RL for each application is based on the user-level QoS desirability. For instance, as shown in the motivational example, RL3 provides an acceptable QoS for Image Smoothing, while for Edge Detection RL2 is more desirable. Note that these RLs can be chosen by the user/designer w.r.t any other policies. The workload mixes along with their reliability levels are listed in in Table 3.13.

3.5.4.3 Experimental Results

Figure 3.17 shows a summary of our experiments. We evaluate QuARK Cache in a multicore platform from these perspectives: 1) flexibility in running different applications with different

reliability levels, 2) energy savings, and 3) delivered quality.

Figure 3.17(a) depicts the average reliability level distribution of accesses to the shared L2 cache equipped with QuARK Cache. To evaluate the flexibility of QuARK Cache in delivering the L2 cache accesses with these various required reliability levels, over the approaches that use coarse-grained actuation knobs, we conduct an experiment, where we divide the execution time of workloads to 10ms epochs. Unlike QuARK Cache that uses fine-grained actuation knobs and assigns the required reliability level to each accesses on-demand, the coarse-grained approaches either should decide about the reliability level of the cache ways (memory banks) statically, or in the best case, dynamically and at the start of each epoch.

Thus, in each epoch, the coarse-grained approaches always should set the reliability level of cache ways, based on the most vulnerable data that should be served by that way. However, both static and dynamic fine-grained approaches can modify cache replacement policy and restrict the traffics of vulnerable data to susceptible ways by paying significant performance penalty. According our experimental results, for all workloads mentioned in Figure 3.17(a), all the 16 ways of the simulated shared L2 cache contains at least one vulnerable block (accessed in RL0 mode) at each epoch. Thus, in all epochs, coarse-grained approaches should pay extra energy overhead to maintain the integrity of vulnerable blocks in the cache ways (even for the blocks that contain imprecise data), while fine-grained actuation knobs in QuARK Cache not only keeps the integrity of vulnerable data, but also provide the opportunity for energy saving in the blocks that contain imprecise data.

Considering the energy consumption reported in Table 3.11, we can find that with the current distribution of approximate read and write operations in the selected benchmarks, and the significant difference between L2 cache read and write energy consumptions, the write energy saving offered by QuARK Cache dominates the gain achieved by approximate read operations in QuARK Cache. Thus, to keep a reasonable trade-off between output quality and energy consumption, we decided to only use the write operation knob in our experiments in following.

It should be noted that QuARK Cache read operation knob would be still useful for the read intensive approximation benchmarks or future STT-MRAM technologies that alleviate the difference between read and write energy consumptions.

From Figure 3.17(b), we see that QuARK Cache delivers up to 40% energy saving for Comb3 at level 3 of fixed-reliability mode. For mixed-reliability mode, it can be observed that the assigned reliability level to each benchmarks as well as the combination of benchmarks in each workload have a major contribution in the energy saving of L2 cache equipped with QuARK Cache. For example, considering mixed-reliability mode, it can be seen that changing the reliability level of Blackscholes from level 3 to 2, and also replacing Corner Detection by Scale are the two most important contributors for less energy saving in Comb2 compared with Comb1.

To compare the quality of the benchmarks, we used the quality metrics introduced in Table 3.12. Figure 3.17(c) depicts the average results of relative error for Blackscholes in all experiments. We see that using level 1 or 2 for the reliability of L2 cache keeps the quality degradation less than 5%. However, high energy saving delivered by using level 3 may not be acceptable for some applications due to high degree of quality degradation (23%). Figure 3.17(d) shows the average PSNR used as quality metric for image smoothing and Scale. We see that Scale is more susceptible to approximation computing. The average calculated PSNR of Scale outputs at level 3 is less than 30dB threshold, which may not be acceptable for some applications (still level 1 or 2 can be selected for these applications). However, the average PSNR of image Smoothing in all of the reliability levels are higher than 30db threshold. Figure 3.17(e) shows the mean pixel difference used as quality metric for Corner detection, Edge detection, and Sobel filter. We see that among these three applications, on average, Corner detection is more susceptible to approximate cache storage. However, its output quality is still degraded by less than 10% using RL3 execution mode.

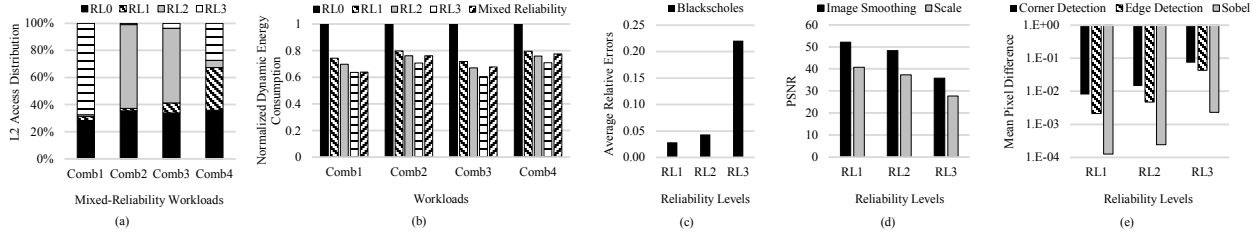


Figure 3.17: QuARK Cache evaluation results: (a) Distribution of accesses in mixed-reliability workloads, (b) Energy savings (normalized to fully-protected STT-MRAM L2 cache), (c) Average relative error for blackscholes benchmark, (d) PSNR for Scale and Image Smoothing benchmarks, and (e) Mean pixel difference for Corner Detection, Edge Detection and Sobel benchmarks.

3.6 Write-Skip SPM

3.6.1 Introduction

STT-MRAM offers a high-density, high-speed, non-volatile choice of random access memory. Despite all these features, high write access energy is still a challenge for its widespread use.

We propose the idea of skipping expensive write operations in spintronic memories when the new data is *approximately*-equal to the old data, thereby trading off accuracy in return for energy savings and performance improvements. Skipping some of the write operations is possible for certain class of applications in approximate computing domain.

Read-before-write schemes were proposed before to remove redundant write operations in phase change memories (PCM) [158] and STT-MRAM memories [30] to enhance the endurance of the bit-cells and reduce the energy consumption respectively. In this work we extend this idea by relaxing the requirement for strict equality instead using approximate equality when avoiding unnecessary write operations.

3.6.2 The Write-Skip Approach

3.6.2.1 Read-Before-Write

Write-Skip reuses the existing read circuitry to read the current data of the memory at a specific address before writing a new data. In general, the read time of STT-MRAM is comparable to that of SRAM and it is significantly less compared to the write time of STT-MRAM [159]. Consequently, there is just a small timing penalty when a read operation is performed before write. The read access energy is also about two orders of magnitude smaller than the write energy [159].

3.6.2.2 Approximate Equality

Approximate equality can be defined for primitive integer and floating-point data types with different bit widths (e.g. `float`, `double`, `int`, `unsigned int`, `long int`, `short int`, etc).

To check approximate equality of two n -bit integers, simply exclusive-oring the $n-k$ upper bits of two data is enough where k is the number of lower bits dropped when verifying equality. Two numbers are approximately-equal if all XOR gates for individual bits generate 0. k is a knob controllable by the software for trading accuracy for energy savings. Higher values of k result in potentially more write operations being skipped, hence higher energy savings, but at the cost of quality degradation.

For floating-point data, we use the fact that adjacent floats have integer representations that are adjacent. That means by dropping the lower k bits of the binary representation of two n -bit floating-point numbers and XORing the upper $n-k$ bits, we can verify their approximate equality.

3.6.3 Evaluations

We evaluate Write-Skip by integrating it into the software-assisted memory hierarchy introduced in Chapter 2. The SAM hierarchy introduced in Chapter 2 uses a distributed on-chip memory composed of both cache and software-programmable memory (SPM). In this work we augment SPMs with Write-Skip and map all the approximate data to SPM. We extend the APIs provided by the SAM hierarchy to pass two kinds of information from software down to on-chip memory hierarchy: (1) data type, (2) number of bits discarded when checking approximate-equality of two consecutive write operations.

Table 3.14 lists the approximate applications we use in our evaluations and their quality metrics. We annotated each application by inserting `add_approx` and `remove_approx` in the source code for non-critical data objects.

Table 3.14: List of approximate applications for Write-Skip experiments

Benchmark	Domain	Quality Metric
Image-Smoothing	Multimedia	PSNR
Sobel Filter	Computer Vision	Mean Pixel Difference
K-means	Machine Learning	Average Distance from Cluster Centers

3.6.3.1 Approximate Value Locality

Figures 3.18, 3.19 and 3.20 show the percentage of write operations, respectively for image-smoothing, sobel and k-means, that are approximately equal to the previous write to the same memory location. The ratios are shown with respect to both total write operations as well as just write operations to SPM that holds approximate data. Both image-smoothing and sobel have some exactly-equal consecutive writes – exploited by the approach in [30] – and dropping a number of bits during equality check increases this ratio even more. K-means’ write accesses have less value locality. At least 14 bits of float data objects in k-means should be dropped during equality check in order to see a significant percentage of approximately-equal

writes.

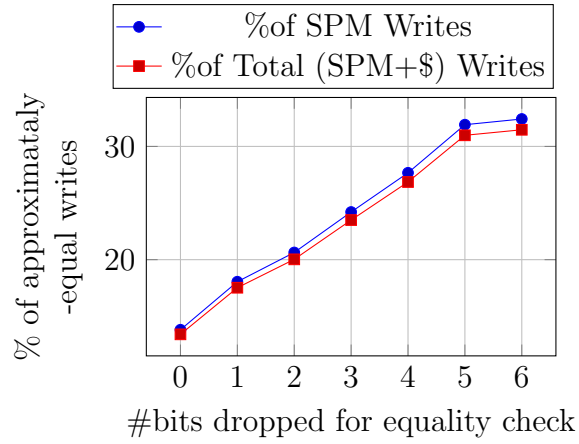


Figure 3.18: Percentage of approximately-equal writes in image-smoothing.

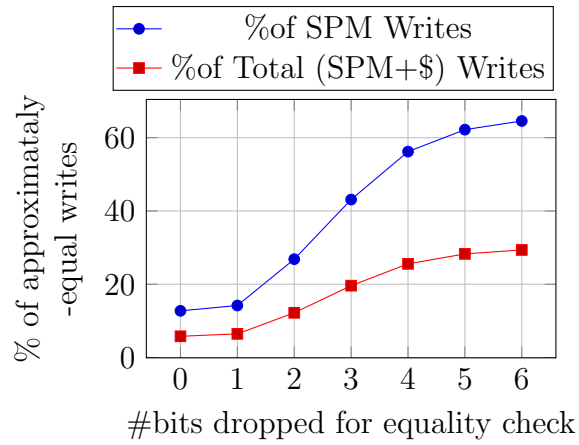


Figure 3.19: Percentage of approximately-equal writes in sobel.

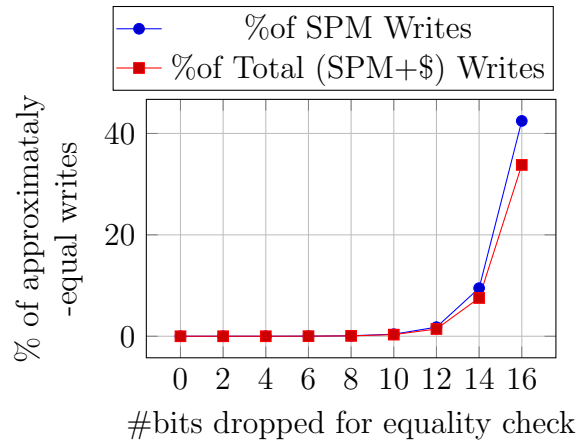


Figure 3.20: Percentage of approximately-equal writes in k-means.

3.6.3.2 Output Fidelities

Figures 3.21, 3.22 and 3.23 show the quality of generated outputs compared to the golden output generated when none of the write operations are skipped. The average PSNRs of output generated by image smoothing in all cases where 1 to 6 bits are dropped during equality check are higher than 30dB acceptable threshold. The worst case mean pixel difference between the approximate output and the golden output for sobel is less than 0.015. And for k-means, the output generated when discarding lower 16 bits during equality check only increases the average euclidean distance of all data points from the cluster centers by about 6%. These preliminary results exhibit the opportunity to skip up to 34% of write operations for certain applications and still produce outputs with acceptable quality.

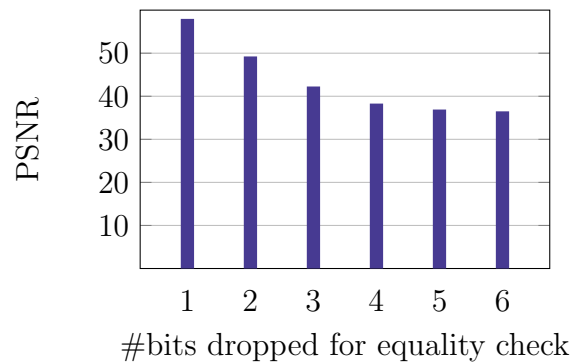


Figure 3.21: Output fidelity for image-smoothing.

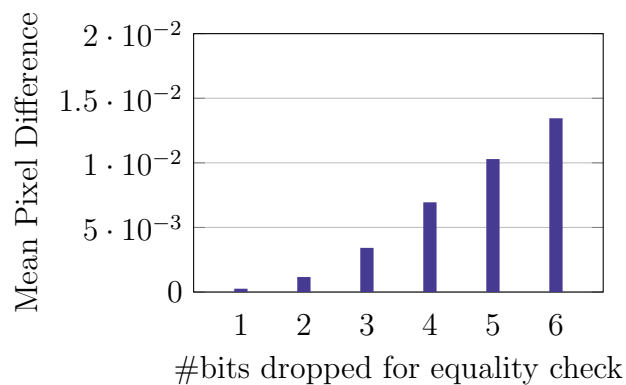


Figure 3.22: Output fidelity for sobel.

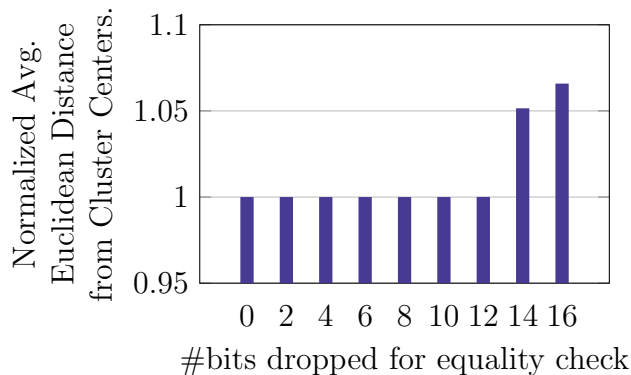


Figure 3.23: Output fidelity for k-means.

3.6.3.3 Energy Consumption in On-chip Memory

Figures 3.24, 3.25 and 3.26 show the energy consumption in both SPM and Cache of SAM hierarchy, respectively for image-smoothing, sobel and k-means, normalized to the baseline where none of the write operations are skipped. The Write-Skip approach is applied only to SPM and the energy consumed in cache for read and write operations is not affected by our approach. However the energy comparisons are done for the total energy consumed in both SPM and cache. Write-Skip performs a read before every write and therefore imposes some energy overhead. However since the read operations are much cheaper than write operations from energy consumption point of view, the overhead is offsetted by the savings for most of the cases. In our experiments, only for k-means and when the number of bits discarded during equality check is between 1 to 10, the overhead is not counterbalanced. However the worst-case overhead is less than 0.8%. Our preliminary experiments demonstrate that Write-Skip enables a useful trade-off in spintronic memories: The energy cost of write accesses can be reduced by up to 30% when the application can tolerate approximate storage of its data.

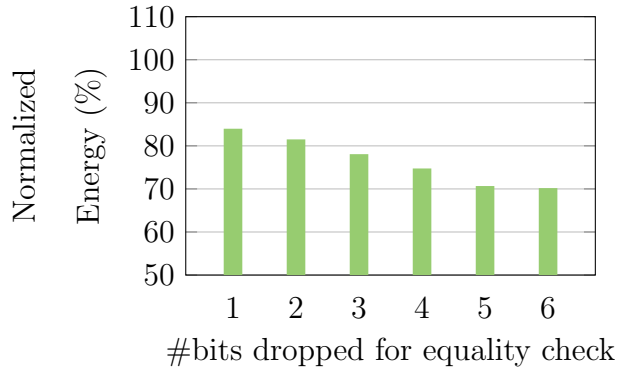


Figure 3.24: Energy consumption in on-chip memory for image-smoothing normalized to the baseline where none of the write operations are skipped.

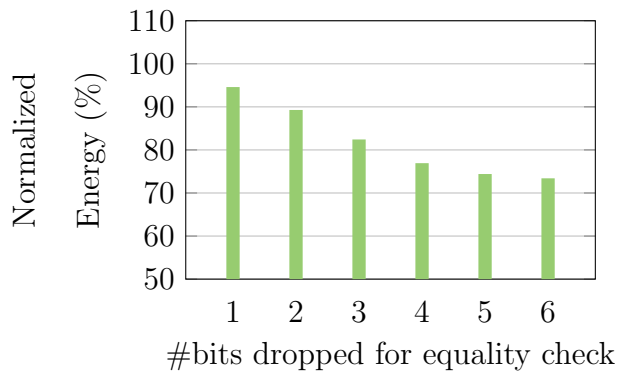


Figure 3.25: Energy consumption in on-chip memory for sobel normalized to the baseline where none of the write operations are skipped.

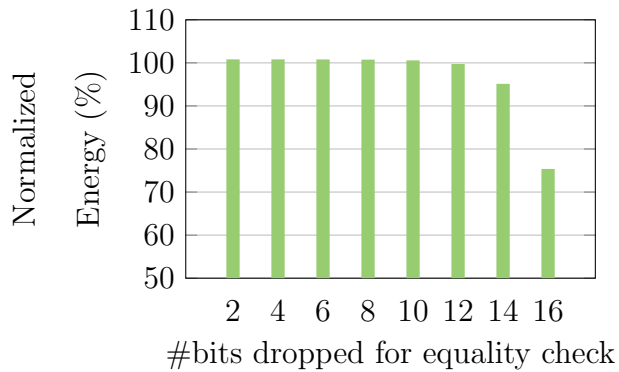


Figure 3.26: Energy consumption in on-chip memory for k-means normalized to the baseline where none of the write operations are skipped.

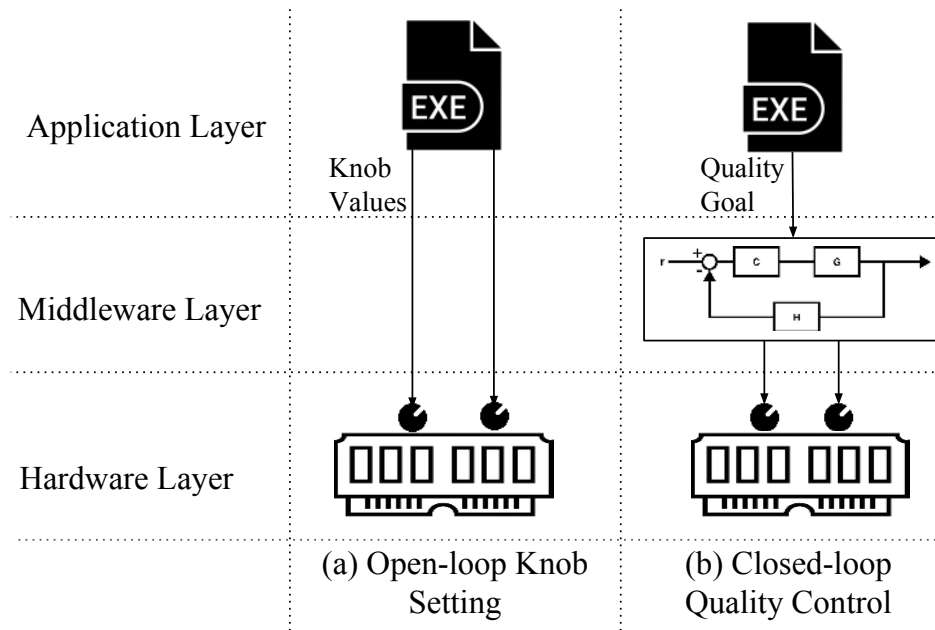


Figure 3.27: Open-loop knob settings (prior works) vs. closed-loop quality control (this work).

3.7 Controlled Memory Approximation

3.7.1 Introduction

Previous work on approximate memories has focused mainly on open-loop and profile-guided approaches for *setting approximation knobs* statically to appropriate values [132, 92, 109]. Based on some design-time analysis the system designer or the programmer set the reliability knob(s) to appropriate values (Figure 3.27(a)).

Our focus here is the quality control problem for systems with a quality-configurable memory running approximate programs. This problem has not been addressed in most of the literature on approximate memories. Instead of putting the burden of setting approximation knob(s) on the system designer or the programmer, we aim at automatically *controlling the quality* by utilizing a feedback controller in the middleware layer that finds the optimal knob values at runtime (Figure 3.27(b)) depending on the requested quality for the application. With our approach, the programmer, instead of setting the knobs directly, only needs to declare

the desired quality. This not only ease up the task of the programmer but also makes the approximation portable across different platforms/memory technologies.

Profile-guided open-loop approaches suffer also from two major disadvantages: (1) They make approximation decisions based on average or worst-case input behavior. These techniques rely on training with inputs that are representative of real-world inputs, which may be difficult to achieve in practice. Laurenzano et al. [84] have shown that the accuracy of approximate programs depend heavily on program input. (2) It is difficult to model an under-designed memory in order to measure the output accuracy at different settings. Temporal faults that are variability-induced, temperature-induced, etc cannot be modeled easily. Unlike, software approximation strategies that are easier to evaluate, hardware approximation requires a more rigorous runtime tuning. A perfect fixed knob(s) to quality metric mapping is unfeasible to obtain through profiling.

In this work, we target streaming applications in which the application is given a sequence of inputs and the results from processing previous inputs can be used to adjust the knobs for the successive inputs because of the correlation of the inputs as well as the temporal behavior of the memory errors. Formal control theory lends itself well to this kind of applications where the history and trend of the previous inputs can be utilized to make predictions for future inputs. In other words, when the system dynamics can be formulated using difference/differential equations.

The closest work to our work is Green compiler [16]. Green targets tuning software approximation and constructs a feedback mechanism for calibrating the knob settings. However it does not propose a systematic strategy to recalibrate the knob(s) settings. The authors of [139] propose a proactive control strategy for non-streaming programs that uses machine learning to learn complex cost and error models offline and utilizes these models at runtime to determine knob settings. These models are not light-weight and not easy to build and do not consider the temporal similarity in the inputs for streaming applications.

We believe that despite the randomness of errors introduced into the execution of the program because of the memory approximation, the system's deterministic dynamic can be captured in way that a formal control-theoretic technique can effectively control the quality of the program even in presence of stochastic (non-deterministic) behavior.

Our solution to this problem employs a controller to construct a closed-loop feedback control mechanism. This controller monitors at the error (i.e., difference between target output quality and measured output quality) and applies a correction accordingly.

Our contributions are as follows:

- We propose a strategy to control the quality of programs with approximate memory and remove the burden of manually setting the approximation knobs from the programmer.
- We show that black-box system identification technique is able to identify the deterministic dynamics of the system in order to create a model based on that.
- We present a prototype of our control strategy for an edge-detection program.

3.7.2 Problem Modeling

3.7.2.1 Application Class

We define streaming application as an application that operates over a long sequence of input data items. The data items are fed to the application from some external source, and each data item is processed for a limited time before being discarded. Most streaming applications are built around signal-processing functions applied on the input data set with little if any control-flow between them. Examples of streaming applications include communication protocols, audio and video processing, cryptographic kernels, and network processing.

We target streaming applications where a stable computation pattern is applied on consecutive inputs. Usually these applications are either state-less, meaning that the inputs are processed completely independent and they have no effect on each other, or they have a limited effect on each other. In the context of approximate computing, this means that the quality degradation of k th input is either purely dependent on the hardware errors happened during processing that input or is just slightly affected by the quality degradation in previous inputs (i.e., 1st through $k-1$ th).

3.7.2.2 Monitoring Quality at Runtime

In order to construct a closed loop feedback mechanism and adjust the memory approximation knob(s) at runtime, the system needs to occasionally sample the current quality of the output with the current knob(s) settings.

This quality measurement method is application dependent and normally the programmer provides a software routine to measure it at runtime. In many cases, this quality measurement would require computing the precise and approximate versions of the output for comparison. Previous work have used the same method for monitoring quality [16, 14].

3.7.2.3 Memory Approximation Knob(s)

Depending on the approximation strategy used and the memory technology, there is always one or more knobs that can tune the degree of approximation storage. For example, in SRAM memories, voltage is the main knob. Refresh rate is the error-energy knob for DRAMs. For non-volatile memories (NVMs) such as STT-MRAM memories, read and write current amplitudes are two key knobs. If the approximation strategy is to skip some of the memory accesses – often stores – or predicting the value of load operations, the percentage of such operations could be the knob that tuning of which would adjust the final quality of the

output.

The implicit assumption is that a more aggressive knob setting makes the memory more unreliable but at the same time lowers down the energy consumption, while a more conservative knob setting enhances the reliability of memory operations but at the cost of higher energy consumption. The ultimate goal is to set the most aggressive knob setting that satisfies the quality goal in order to gain the highest energy savings possible.

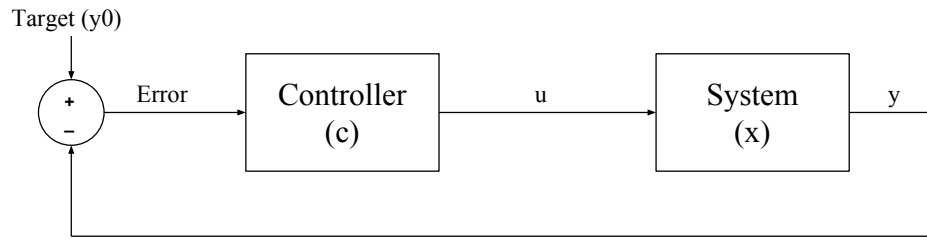
3.7.3 Quality Control with Feedback Control Theory

In control theory, a controlled system is represented as a feedback control loop as in Fig. 3.28(a). At the epoch k , the controller reads the *measured output* $y(k)$ of a system in *state* $x(k)$, compares it against the target value y_{ref} , and based on the difference (or error $e(k)$), generates the *control input* $u(k)$ to actuate on the system and reduce the error.

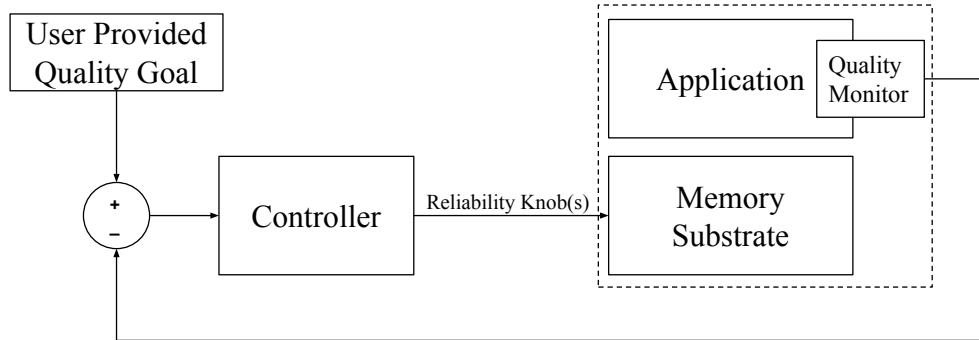
In our case, the system is composed of both quality-configurable memory as well as the software running on a system with this memory. The control input to the system is the value of a knob and the measured output of the system is the quality of the generated output.

The closed loop approach for tuning knobs is shown in Fig. 3.28(b). The application runs on a processor with a quality-configurable memory. With a pre-determined frequency, the quality monitor routine measures the current quality of the output. This quality is compared against the quality goal. A positive difference means there is still room to relax the reliability requirements of the memory and the controller accordingly sets a more aggressive knob setting. A negative difference means that the quality has degraded more than what was intended. The controller accordingly sets a more conservative knob setting.

To formally control such a system, a model of the system dynamics that describes the relation between control input(s) (i.e., knob(s)) and the measured output (i.e., the quality of



(a)



(b)

Figure 3.28: Closed loop approach (this work) for tuning memory approximation knob(s).

the program’s output) as a function of time epoch, is needed.

A common practice to extract the dynamic model of complex systems is through System Identification Theory [93, 94] where we experimentally collect input-output data and use black-box identification methods to isolate the deterministic and stochastic components of the system. In this way, we can build a model entirely from true data for arbitrarily complex systems. In our design, we follow this approach and show that *quality configuration (i.e., tracking) in memory-oriented approximate computing can be modeled as a formal quality control problem*, which can be then addressed by using classical off-the-shelf controllers.

3.7.4 Case Study: Video Edge Detection

3.7.4.1 Application Description and Error Metric

We use canny video edge detection [32] as our case study. Edge detection is the process of identifying sharp changes in image brightness. This is important because it detects physical changes in the objects imaged. For video processing, the edge detection is often conducted on a frame-by-frame basis independently. The temporal similarity between adjacent frames of a scene in a video allows the controller to adjust the quality based on the history of the system from previous frames.

We use miss-classification error as our quality metric, that is the ratio of total number of pixels mistakenly classified as edge/non-edge to the total number of pixels in the frame.

3.7.4.2 Memory Approximation Knob

For simplicity, we assume that all the memory accesses hit the memory substrate in which we are interested to actuate the error rate. This could translate into adjusting knobs for a L1 data cache where the hit rate for approximate data is very close to 100%. We choose write reliability as the tuning knob as the write current amplitude which is shown in previous work to be an appropriate approximation knob for spin-transfer torque memories [109, 122, 130], and there is a relationship between the amount of the current applied for write operations and the write error probability.

3.7.5 System Identification

In order to design a controller for the system, we need to model the system first. The first step towards that is to generate test waveforms from training applications for system

identification. A test waveform contains a series of samples for controller inputs and outputs for a training application, and should exercise as many input permutations as possible. The system dynamics is exercised often by applying a staircase waveform to the control input (e.g., write bit error rate). Such staircase would stimulate system behaviour in response to various levels of control input. In our work, we change write bit error rate from $10E-7$ to $10E-3$ with steps of $10E-7$.

In this method, training sets use varying frequency (e.g., a set of out-of-phase staircase signals for the control input) in order to isolate the deterministic and stochastic aspects of the system. This model is then evaluated to predict the expected data from the identified system. Abnormal behaviour from this model can raise a flag that the controller to be designed from this model might be inaccurate.

We use MATLAB's system identification toolbox for this process [99]. Figure 3.29 shows the result of system identification for canny when a series of 2400 frames are inputted. It can be seen that the predicted model closely fits the measured data.

In this method, training sets use varying frequency (e.g., a set of out-of-phase staircase signals for the control input) in order to isolate the deterministic and stochastic aspects of the system. This model is then evaluated to predict the expected data from the identified system. Abnormal behavior from this model can raise a flag that the controller to be designed from this model might be inaccurate. In our work, we used MATLAB's system identification toolbox for this process [99].

Figure 3.29 shows the result of system identification for canny when a series of 2400 frames are inputted. It can be seen that the predicted model closely fits the measured data.

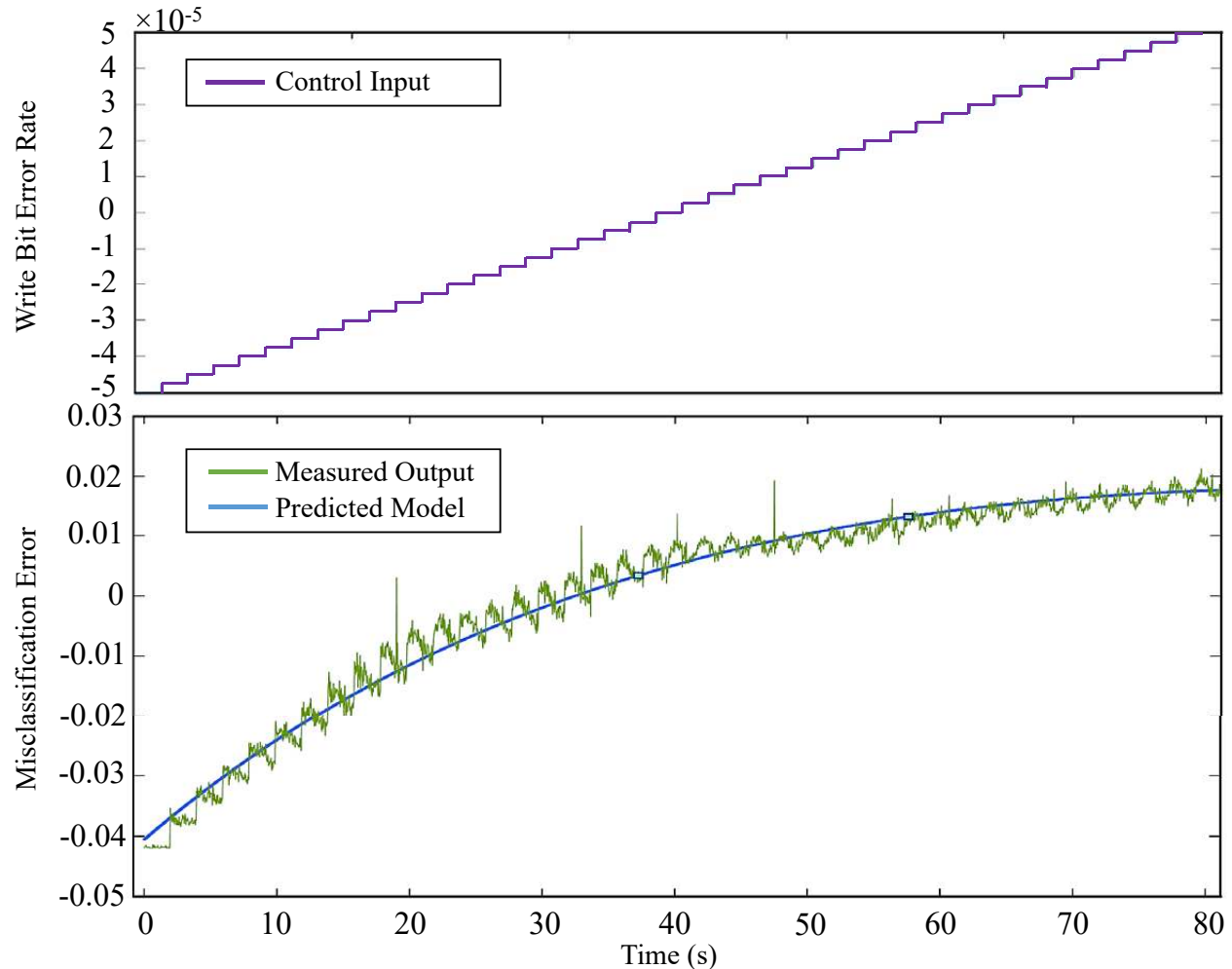


Figure 3.29: Predicted Model vs. Measured Output.

3.7.5.1 Controller

According to the assumptions made in subsections 3.7.4.1 and 3.7.4.2, our system is a simple single-input-single-output (SISO) control system with write bit error rate as control input and edge detection miss-classification rate as measured output². We use a proportional-integral (PI) controller to control this system. The proportional term refers to the fact that the controller output is proportional to the amplitude of error signal, while *Integral* indicates

²Note that while in this case study we utilize a SISO controller, we also acknowledge the possibility of using a multiple-input-multiple output (MIMO) controller for other systems that may provide more than one output quality metric and memories that provide more than one to tune – either because there are multiple knobs per memory component (i.e., STT-MRAM read and write current) or the controller is tuning multiple individual memory components in the memory hierarchy.

that the controller output is proportional to the integral of all past errors [67]. The PI control law has the form:

$$u(k) = u(k - 1) + (K_P + K_I)e(k) - K_P e(k - 1) \quad (3.3)$$

Where K_P and K_I denote the coefficients for the proportional and integral terms, respectively. *Controller design* is a mature field which utilizes many tools that provide off-the-shelf controllers. We use Matlab PID tuner toolbox to design and deploy our controllers.

It is important to note that although derivative control law is helpful to add predictability to the controller, stochastic variations in the system output may cause inaccuracy in the controller. This issue becomes more severe in computer systems as they commonly have a significant stochastic component. Therefore, for computer systems PI controllers are preferred over proportional-integral-derivative (PID) controller [67]. PI control benefits from both integral control (zero steady-state error) and proportional control (fast transient response). In most computer systems a first-order PI controller provides rapid response and is sufficiently accurate [67].

3.7.5.2 Fault Injection Mechanism

To simulate the behavior of a system with approximate memory, we developed a PIN-based [96] memory fault injector. This fault injector is able to inject memory faults into read and/or write operations according to a given error rate. In order to inject faults only into the non-critical data objects of the program, the source code of the program is annotated with `add_approx()` and `remove_approx()` methods to declare the address of those data objects in the program. These methods are called in the program at appropriate places and are captured by the fault injector. The fault injector records these addresses into a table. During the execution, it instruments all the memory accesses. If the virtual address of the access

falls into the any of the given address boundaries, it attempts to inject fault into the part of data referenced by that memory access.

3.7.5.3 Input Dependency

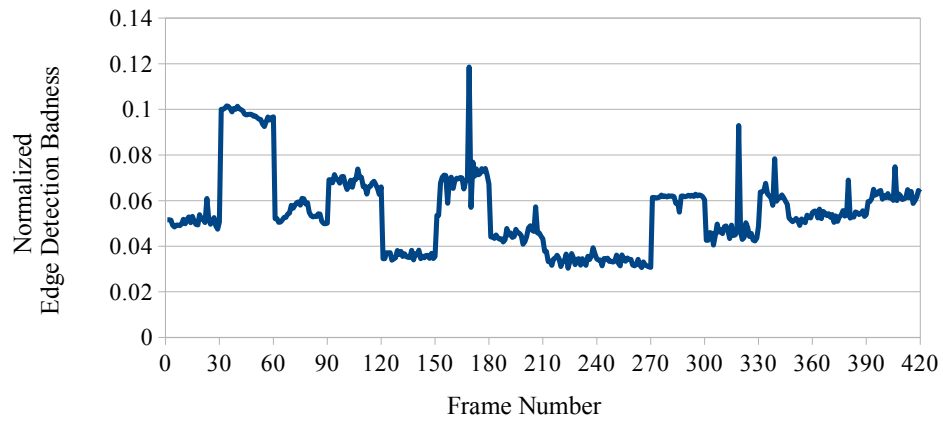
Figure 3.30 shows the variation of the quality of the edge detection in a video composed of multiple scenes when the write bit error rate is set at a constant value, similar to what an open-loop approach would do. This variation confirms the dependency of the quality of a program running on a platform that uses an approximate memory on the input.

3.7.5.4 QoS Tracking

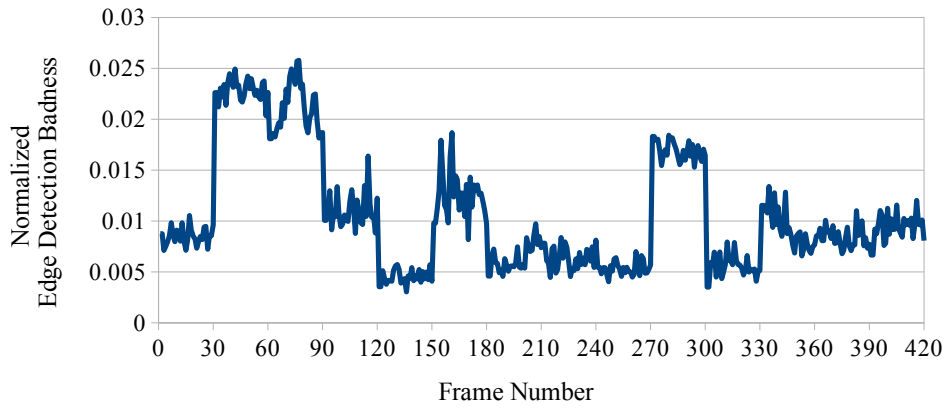
Figure 3.31 shows a sample of how the feedback loop works in practice for different video sequences. The red curve shows the quality goal and the blue curve shows the achieved quality when the PI controller manages the quality by adjusting the knob.

3.7.5.5 Comparison

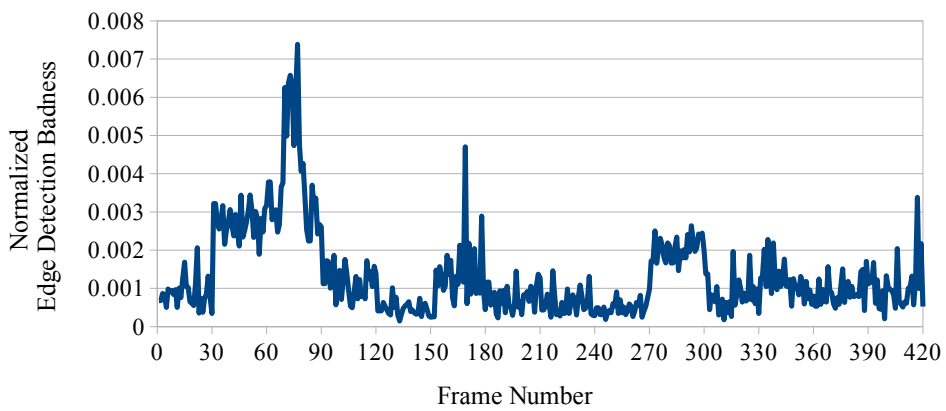
Figure 3.32 compares the performance of a PI controller in tracking target quality with a manual calibration scheme. The manual scheme measures the difference between the desired quality and the current quality. If the difference is within $\pm 10\%$ it does not change the knobs. Otherwise it changes the knob in one direction with certain fine-grained steps until the quality returns back to the acceptable quality region.



(a) BER = 1E-3

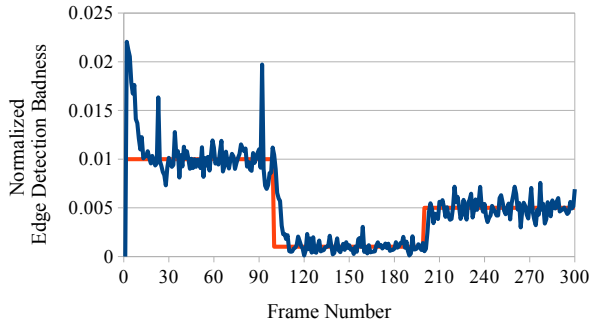


(b) BER = 1E-4

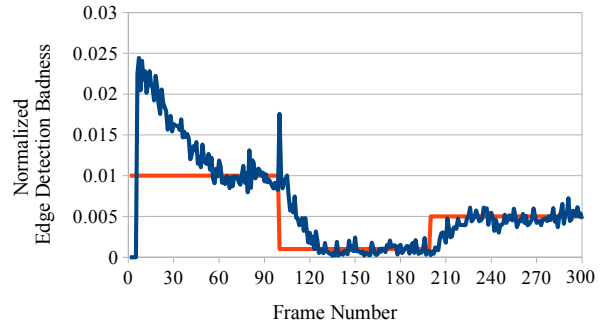


(c) BER = 1E-5

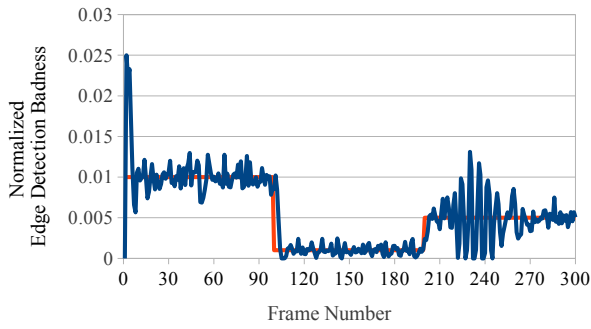
Figure 3.30: Variation of the quality of the edge detection in various video scenes when the bit error rate is constant.



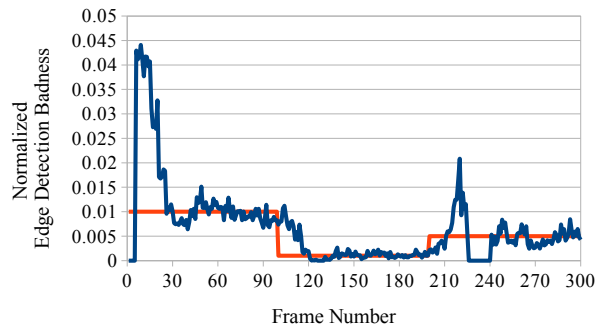
(a) news, $t = 1$



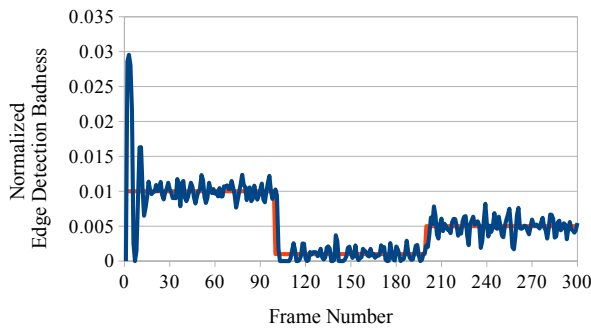
(b) news, $t = 5$



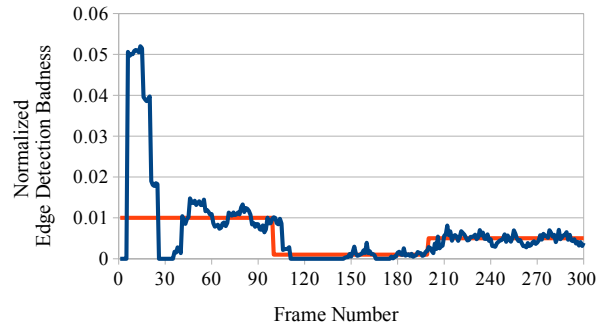
(c) soccer, $t = 1$



(d) soccer, $t = 5$



(e) marymax, $t = 1$



(f) marymax, $t = 5$

Figure 3.31: Quality tracking results. Red curve shows the acceptable error and the blue curve shows the error achieved by the controller.

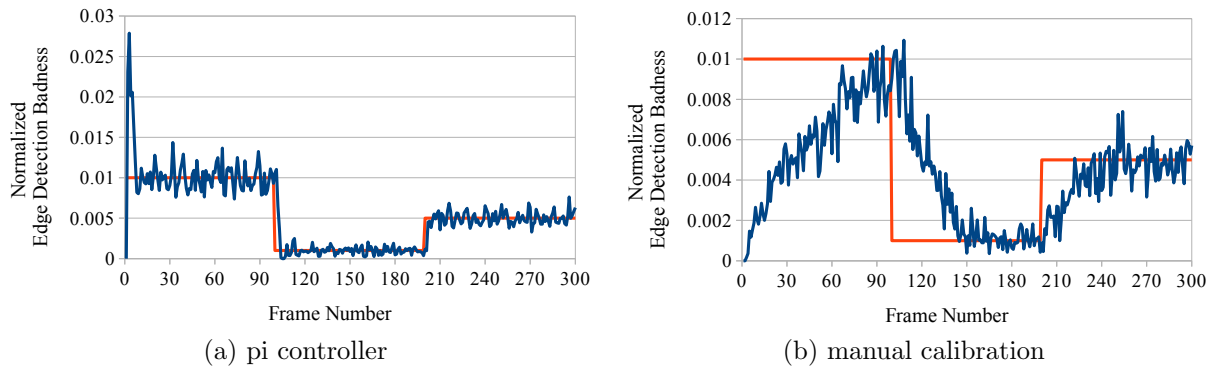


Figure 3.32: Comparing PI controller with manual step-wise re-calibration similar to [16].







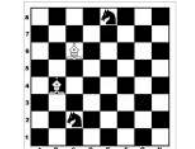
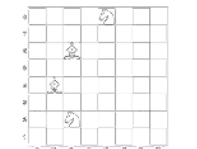

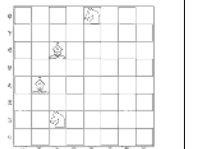
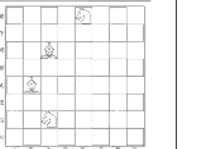
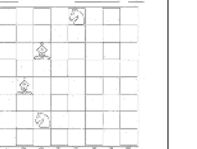





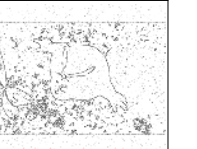
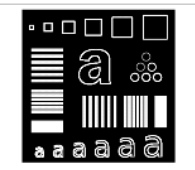


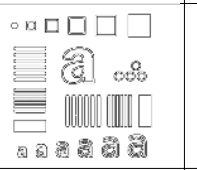
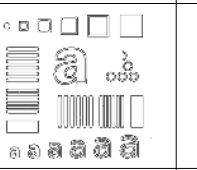
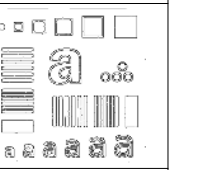
	BER = 0	BER = 1e-6	BER = 1e-5	BER = 1e-4	BER = 1e-3
					
		0.0000098	0.0002661	0.0029438	0.0152044
					
		0.0000000	0.0000049	0.0063853	0.0089849
					
		0.0000423	0.0005944	0.0071542	0.0404427
					
		0.0000000	0.0000501	0.0013885	0.0035272

Figure 3.33: Canny edge detection applied to different images with various write bit error rates resulting in different quality metrics.

Chapter 4

Conclusions and Future Directions

Technology limitations and application demands have driven the emergence of manycore embedded systems. The memory hierarchy of manycore architectures has a major impact on their overall performance, energy efficiency and reliability. We identified three major issues with the traditional memory hierarchies that make them unappealing for manycore architectures and their data-intensive workloads: (1) they are power hungry and not a good fit for manycores in face of dark silicon, (2) they are not adaptable to the workload's requirements and memory behavior, and (3) they are not scalable due to coherence overheads.

This dissertation showed that many of these challenges can be addressed by through different types of software-assists. Application semantics and behavior captured in software can be exploited to more efficiently manage the memory hierarchy.

4.1 Technical Contributions

The novel contributions of this thesis are:

- **SAM:** Hybrid Memory Hierarchy for MES with Software-Managed Coherence
- **ShaVe-ICE:** Software & Architectural Support for Sharing Distributed SPMs in MES
- **Relaxed Cache:** Relaxed Guardbanding for Process-Variation-Affected SRAM Caches
- **QuARK Cache:** Fine-grained Tuning of Reliability-Energy Knobs in Shared STT-MRAM Caches
Write-Skip SPM: Skipping Write Operations in STT-MRAM SPMs
- **CTRL-MEM-APRX:** Controlled Approximation in Quality-Configurable Memories

4.2 Future Directions

There are many future possibilities to extend the software-assisted memory hierarchy proposed in this dissertation:

- The ShaVe-ICE runtime memory manager along with its various allocation policies need to be integrated into a real operating system (e.g, Linux) as a kernel module to explore the overheads.
- The SPM allocation policies introduced in Section 2.5 show promising advantages over the baseline strategy, that limits the allocation to the SPM local to each core. However, we need more intelligent policies that take into account several other factors like functional/power characteristics of heterogeneous memory resources as well as the phasic behavior of applications.
- The current version of the ShaVe-ICE allocator works on a first-come-first-served basis. To make the allocation more fair and also further improve the overall data placements, we need to keep a backlog of unserved allocation requests and judiciously choose between

them and new requests which one to grant when space becomes available. In doing that these allocation policies should have a measure of relative importance in order to prioritize the allocation of different data objects when there is spillovers.

- Although the current version of the software-programmable hierarchy introduced in Chapter 2 relies on explicit annotation by the programmer or the compiler, ideally we would like to remove this burden to some extent and be able to monitor the memory behavior of the application at runtime and protectively decide about which data to bring on-chip and if they should be put in cache or local SPM or remote SPM and also how to partition the available the memory space between cache and SPM depending on workload’s memory behavior.
- For shared data, we considered simply allocating the single copy on the SPM of the core spawning the threads. However, data placement could be further optimized in future work. Worker threads can allocate their own private data on SPM as well, but the assumption is that all shared data objects are allocated on SPM by the boss thread before worker threads are dispatched. Future work could relax this restriction by allowing cores to exchange SATT mappings.
- There is a growing need for mechanisms that adjust memory approximation knobs automatically and coordinate their effect on the final quality of the output. Our approach in Section 3.7 is an initial attempt towards this end. More works need to be done to build better system models and design multi-input-multi-output (MIMO) controllers that coordinate different knobs across the different layers of the memory hierarchy as well the system stack.

Bibliography

- [1] Coyotebench. <https://github.com/Microsoft/test-suite/tree/master/SingleSource/Benchmarks/CoyoteBench>. Accessed: 2016-10-31. (Cited on page 51).
- [2] The Computer Language Benchmarks Game. <http://benchmarksgame.alioth.debian.org/>. Accessed: 2016-10-31. (Cited on page 51).
- [3] NVIDIAs Next Generation CUDA Compute Architecture: Fermi. Technical report, NVIDIA, 2009. (Cited on page 21).
- [4] Adapteva. *Epiphany Architecture Reference*, 2014. Rev 14.03.11. (Cited on pages 2 and 4).
- [5] C. Alvarez, J. Corbal, and M. Valero. Fuzzy Memoization for Floating-Point Multimedia Applications. *IEEE Transactions on Computers (TC)*, 2005. (Cited on page 63).
- [6] L. Alvarez, L. Vilanova, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade. Hardware-Software Coherence Protocol for the Coexistence of Caches and Local Memories. *IEEE Transactions on Computers (TC)*, 2015. (Cited on page 22).
- [7] L. Alvarez, L. Vilanova, M. Moreto, M. Casas, M. Gonzàlez, X. Martorell, N. Navarro, E. Ayguadé, and M. Valero. Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015. (Cited on page 16).
- [8] A. Ansari, S. Feng, S. Gupta, and S. Mahlke. Archipelago: A Polymorphic Cache Design for Enabling Robust Near-threshold Operation. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2011. (Cited on page 82).
- [9] A. Ansari, S. Gupta, S. Feng, and S. Mahlke. ZerehCache: Armoring Cache Architectures in High Defect Density Technologies. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009. (Cited on page 82).
- [10] 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 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2011. (Cited on page 63).

- [11] ARM. *Cortex-A5 processor manual*. <http://www.arm.com>. (Cited on page 96).
- [12] ARM. *Cortex-R4 processor manual*. <http://www.arm.com>. (Cited on page 96).
- [13] G. P. Arumugam, P. Srikanthan, J. Augustine, K. Palem, E. Upfal, A. Bhargava, Parishkrati, and S. Yenugula. Novel Inexact Memory Aware Algorithm Co-design for Energy Efficient Computation: Algorithmic Principles. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015. (Cited on pages 65, 66, and 75).
- [14] Aurangzeb and R. Eigenmann. Harnessing Parallelism in Multicore Systems to Expedite and Improve Function Approximation. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2016. (Cited on page 126).
- [15] O. Avissar, R. Barua, and D. Stewart. Heterogeneous Memory Management for Embedded Systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2001. (Cited on page 14).
- [16] W. Baek and T. M. Chilimbi. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010. (Cited on pages x, 63, 124, 126, and 136).
- [17] K. Bai, J. Lu, A. Shrivastava, and B. Holton. CMSM: An Efficient and Effective Code Management for Software Managed Multicores. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013. (Cited on page 11).
- [18] K. Bai and A. Shrivastava. Heap Data Management for Limited Local Memory (LLM) Multi-core Processors. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODE+ISSS)*, 2010. (Cited on page 15).
- [19] K. Bai and A. Shrivastava. Automatic and Efficient Heap Data Management for Limited Local Memory Multicore Architectures. In *Proceedings of the International Conference on Design Automation and Test in Europe (DATE)*, 2013. (Cited on page 15).
- [20] K. Bai, A. Shrivastava, and S. Kudchadker. Stack Data Management for Limited Local Memory (LLM) Multi-core Processors. In *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2011. (Cited on pages 11 and 15).
- [21] A. BanaiyanMofrad, H. Homayoun, and N. Dutt. FFT-cache: A Flexible Fault-tolerant Cache Architecture for Ultra Low Voltage Operation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011. (Cited on pages 82 and 88).

- [22] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proceedings of the International Symposium on Hardware/Software Codesign (CODES)*, 2002. (Cited on page 11).
- [23] L. Bathen and N. Dutt. HaVOC: A hybrid Memory-aware Virtualization Layer for On-chip Distributed ScratchPad and Non-Volatile Memories. In *Proceedings of the Design Automation Conference (DAC)*, 2012. (Cited on page 16).
- [24] L. A. D. Bathen and N. D. Dutt. SPMCloud: Towards the Single-Chip Embedded ScratchPad Memory-Based Storage Cloud. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2014. (Cited on page 16).
- [25] L. A. D. Bathen, N. D. Dutt, A. Nicolau, and P. Gupta. VaMV: Variability-aware Memory Virtualization. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2012. (Cited on page 16).
- [26] L. A. D. Bathen, N. D. Dutt, D. Shin, and S.-S. Lim. SPMVisor: Dynamic Scratchpad Memory Virtualization for Secure, Low Power, and High Performance Distributed On-chip Memories. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011. (Cited on page 16).
- [27] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. C. Miao, C. Ramey, D. Wentzloff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, 2008. (Cited on pages 2 and 4).
- [28] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008. (Cited on page 97).
- [29] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoab, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011. (Cited on pages 37, 95, and 111).
- [30] R. Bishnoi, F. Oboril, M. Ebrahimi, and M. B. Tahoori. Avoiding unnecessary write operations in STT-MRAM for low power implementation. In *Proceedings of the International Symposium on Quality Electronic Design (ISQED)*, 2014. (Cited on pages 116 and 118).
- [31] V. Camus, J. Schlachter, C. Enz, M. Gautschi, and F. K. Gurkaynak. Approximate 32-bit Floating-point Unit Design with 53Product Reduction. In *Proceedings of the European Solid-State Circuits Conference (ESSCIRC)*, 2016. (Cited on page 63).

- [32] J. Canny. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 1986. (Cited on page 129).
- [33] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013. (Cited on pages 80 and 92).
- [34] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti. Noxim: An Open, Extensible and Cycle-accurate Network On-chip Simulator. In *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2015. (Cited on page 37).
- [35] P. Chakraborty and P. R. Panda. SPM-Sieve: A Framework for Assisting Data Partitioning in Scratch Pad Memory Based Systems. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, 2013. (Cited on page 24).
- [36] P. Chakraborty, P. R. Panda, and S. Sen. Partitioning and Data Mapping in Reconfigurable Cache and Scratchpad Memory-Based Architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2016. (Cited on pages 16 and 21).
- [37] S. Chen, S. Jiang, B. He, and X. Tang. A Study of Sorting Algorithms on Approximate Memory. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2016. (Cited on pages 65, 66, and 74).
- [38] 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 ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013. (Cited on page 62).
- [39] D. Cho, S. Pasricha, I. Issenin, N. Dutt, M. Ahn, and Y. Paek. Adaptive Scratch Pad Memory Management for Dynamic Behavior of Multimedia Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2009. (Cited on page 15).
- [40] D. Cho, S. Pasricha, I. Issenin, N. Dutt, Y. Paek, and S. Ko. Compiler Driven Data Layout Optimization for Regular/Irregular Array Access Patterns. In *Proceedings of the ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2008. (Cited on page 15).
- [41] K. Cho, Y. Lee, Y. H. Oh, G. c. Hwang, and J. W. Lee. eDRAM-based Tiered-Reliability Memory with Applications to Low-power Frame Buffers. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2014. (Cited on pages 65, 66, 67, 68, and 77).

- [42] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy Types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012. (Cited on page 92).
- [43] J. Cong, H. Huang, C. Liu, and Y. Zou. A Reuse-aware Prefetching Scheme for Scratchpad Memory. In *Proceedings of the Design Automation Conference (DAC)*, 2011. (Cited on page 13).
- [44] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Lger, B. Orgogozo, J. Reybert, and T. Strudel. A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. *Procedia Computer Science*, 2013. (Cited on pages 2 and 4).
- [45] N. Deng, W. Ji, J. Li, and Q. Zuo. A Semi-automatic Scratchpad Memory Management Framework for CMP. In *Proceedings of the International Conference on Advanced Parallel Processing Technologies (APPT)*, 2011. (Cited on page 15).
- [46] T. Devolder, J. Hayakawa, K. Ito, H. Takahashi, S. Ikeda, P. Crozat, N. Zerounian, J.-V. Kim, C. Chappert, and H. Ohno. Single-Shot Time-Resolved Measurements of Nanosecond-Scale Spin-Transfer Induced Switching: Stochastic Versus Deterministic Aspects. *Physical Review Letters*, 2008. (Cited on page 101).
- [47] A. Dominguez, S. Udayakumaran, and R. Barua. Heap Data Allocation to Scratch-pad Memory in Embedded Systems. *Journal of Embedded Computing*, 2005. (Cited on page 14).
- [48] X. Dong et al. NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2012. (Cited on pages 104 and 111).
- [49] P. Dubey. Recognition, Mining and Synthesis Moves Computers to the Era of Tera. *Intel Technology Journal*, 2005. (Cited on page 6).
- [50] H. Esmailzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2011. (Cited on page 3).
- [51] H. Esmailzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012. (Cited on page 63).
- [52] Y. Fang, H. Li, and X. Li. SoftPCM: Enhancing Energy Efficiency and Lifetime of Phase Change Memory in Video Applications via Approximate Write. In *Proceedings of the IEEE Asian Test Symposium (ATS)*, 2012. (Cited on pages 65, 66, 67, 68, and 72).
- [53] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2002. (Cited on page 85).

- [54] Forczmański et al. An Algorithm of Face Recognition Under Difficult Lighting Conditions. *Electrical Review*, 2012. (Cited on page 107).
- [55] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An Integrated Hardware/Software Approach for Run-time Scratchpad Management. In *Proceedings of the Design Automation Conference (DAC)*, 2004. (Cited on page 14).
- [56] S. Ganapathy, G. Karakonstantis, A. Teman, and A. Burg. Mitigating the Impact of Faults in Unreliable Memories for Error-resilient Applications. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, 2015. (Cited on pages 65, 66, 67, 68, and 75).
- [57] L. Gauthier, T. Ishihara, H. Takase, H. Tomiyama, and H. Takada. Minimizing Inter-task Interferences in Scratch-pad Memory Usage for Reducing the Energy Consumption of Multi-task Systems. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2010. (Cited on page 14).
- [58] M. Gottscho, A. BanaiyanMofrad, N. Dutt, A. Nicolau, and P. Gupta. Power / Capacity Scaling: Energy Savings With Simple Fault-Tolerant Caches. In *Proceedings of the Design Automation Conference (DAC)*, 2014. (Cited on pages 89 and 96).
- [59] Q. Guo, K. Strauss, L. Ceze, and H. S. Malvar. High-Density Image Storage Using Approximate Memory Cells. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016. (Cited on pages 65, 66, 67, 68, and 76).
- [60] P. Gupta, Y. Agarwal, L. Dolecek, N. Dutt, R. K. Gupta, R. Kumar, S. Mitra, A. Nicolau, T. S. Rosing, M. B. Srivastava, S. Swanson, and D. Sylvester. Underdesigned and Opportunistic Computing in Presence of Hardware Variability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2013. (Cited on page 82).
- [61] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy. IMPACT: IMPrecise Adders for Low-power Approximate Computing. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2011. (Cited on page 63).
- [62] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy. Low-Power Digital Signal Processing Using Approximate Adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2013. (Cited on page 63).
- [63] 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. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC)*, 2001. (Cited on pages 51, 84, and 97).
- [64] S. Hamdioui, A. J. van de Goor, and M. Rodgers. March SS: A Test for All Static Simple RAM Faults. In *Proceedings of the IEEE International Workshop on Memory Technology, Design and Testing (MTDT)*, 2002. (Cited on page 88).

- [65] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim. The IBM Blue Gene/Q Compute Chip. *IEEE Micro*, 2012. (Cited on page 2).
- [66] S. Hashemi, R. I. Bahar, and S. Reda. DRUM: A Dynamic Range Unbiased Multiplier for Approximate Applications. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015. (Cited on page 63).
- [67] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004. (Cited on page 132).
- [68] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic Knobs for Responsive Power-aware Computing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011. (Cited on page 63).
- [69] IBM. The Cell Project, 2005. (Cited on page 4).
- [70] F. Ishihara, F. Sheikh, and B. Nikolić. Level Conversion for Dual-supply Systems. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 2004. (Cited on page 109).
- [71] ITRS. <http://www.itrs.net>, 2015. (Cited on pages xi, 3, and 82).
- [72] A. Jain, P. Hill, S. C. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. Concise Loads and Stores: The Case for an Asymmetric Compute-memory Architecture for Approximation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016. (Cited on pages 65, 66, 67, 68, and 76).
- [73] D. Jevdjic, K. Strauss, L. Ceze, and H. S. Malvar. Approximate Storage of Compressed and Encrypted Videos. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017. (Cited on pages 65, 66, 67, 68, and 76).
- [74] A. B. Kahng and S. Kang. Accuracy-configurable Adder for Approximate Arithmetic Designs. In *Proceedings of the Annual Design Automation Conference (DAC)*, 2012. (Cited on page 63).
- [75] Kalray. MPPA2-256 (Bostan), 2015. (Cited on page 2).
- [76] M. T. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Proceedings of the Design Automation Conference (DAC)*, 2001. (Cited on page 14).
- [77] W. Kang, W. Zhao, J. O. Klein, Y. Zhang, C. Chappert, and D. Ravelosona. High Reliability Sensing Circuit for Deep Submicron Spin Transfer Torque Magnetic Random Access Memory. *Electronics Letters*, 2013. (Cited on page 102).

- [78] A. Kannan, A. Shrivastava, A. Pabalkar, and J.-e. Lee. A Software Solution for Dynamic Stack Management on Scratch Pad Memory. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2009. (Cited on pages 11 and 15).
- [79] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. Rumba: An Online Quality Management System for Approximate Computing. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, ISCA, 2015. (Cited on page 63).
- [80] O. Kislal, M. T. Kandemir, and J. Kotra. Cache-Aware Approximate Computing for Decision Tree Learning. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016. (Cited on pages 65, 66, 67, 68, and 75).
- [81] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve. Stash: Have Your Scratchpad and Cache It Too. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015. (Cited on page 16).
- [82] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In *Proceedings of the International Conference on VLSI Design (VLSID)*, 2011. (Cited on page 63).
- [83] A. Lachenmann, P. J. Marrón, D. Minder, and K. Rothermel. Meeting Lifetime Goals with Energy Levels. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*, 2007. (Cited on page 63).
- [84] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang. Input Responsiveness: Using Canary Inputs to Dynamically Steer Approximation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016. (Cited on page 124).
- [85] D. Lee, S. K. Gupta, and K. Roy. High-performance Low-energy STT MRAM Based on Balanced Write Scheme. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2012. (Cited on page 109).
- [86] 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 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2006. (Cited on pages 65, 66, 67, 68, and 70).
- [87] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra. ERSA: Error Resilient System Architecture for Probabilistic Applications. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2010. (Cited on page 63).
- [88] H. Li, X. Wang, Z. L. Ong, W. F. Wong, Y. Zhang, P. Wang, and Y. Chen. Performance, Power, and Reliability Tradeoffs of STT-RAM Cell Subject to Architecture-Level Requirement. *IEEE Transactions on Magnetics*, 2011. (Cited on page 104).

- [89] L. Li, L. Gao, and J. Xue. Memory Coloring: A Compiler Approach for Scratchpad Memory Management. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2005. (Cited on page 14).
- [90] C. H. Lin and I. C. Lin. High Accuracy Approximate Multiplier with Error Correction. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, 2013. (Cited on page 63).
- [91] C. Liu, J. Han, and F. Lombardi. A Low-power, High-performance Approximate Multiplier with Configurable Partial Error Recovery. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 2014. (Cited on page 63).
- [92] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flicker: Saving DRAM Refresh-power Through Critical Data Partitioning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011. (Cited on pages 63, 65, 66, 67, 68, 70, 71, 80, 81, 83, and 123).
- [93] L. Ljung. *System Identification: Theory for the User*. Prentice-Hall, Inc., 1999. (Cited on page 128).
- [94] L. Ljung. Black-box Models from Input-output Measurements. In *Proceedings of the IEEE Instrumentation and Measurement Technology Conference (IMTC)*, 2001. (Cited on page 128).
- [95] J. Lu, K. Bai, and A. Shrivastava. SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs). In *Proceedings of the Design Automation Conference (DAC)*, 2013. (Cited on pages 11 and 15).
- [96] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005. (Cited on page 132).
- [97] M. Marins de Castro, R. C. Sousa, S. Bandiera, C. Ducruet, A. Chavent, S. Auffret, C. Papusoi, I. L. Prejbeanu, C. Portemont, L. Vila, U. Ebels, B. Rodmacq, and B. Dieny. Precessional Spin-transfer Switching in a Magnetic Tunnel Junction with a Synthetic Antiferromagnetic Perpendicular Polarizer. *Journal of Applied Physics*, 2012. (Cited on page 104).
- [98] A. Marongiu and L. Benini. An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs. *IEEE Transactions on Computers (TC)*, 2012. (Cited on page 15).
- [99] MathWorks. System Identification Toolbox. <https://www.mathworks.com/products/sysid.html>. (Cited on page 130).
- [100] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC Processor: The

- Programmer's View. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010. (Cited on pages 2 and 4).
- [101] J. Miao, K. He, A. Gerstlauer, and M. Orshansky. Modeling and Synthesis of Quality-energy Optimal Approximate Adders. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2012. (Cited on page 63).
- [102] J. S. Miguel, J. Albericio, N. E. Jerger, and A. Jaleel. The Bunker Cache for Spatio-value Approximation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016. (Cited on pages 65, 66, 67, 68, and 78).
- [103] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger. Doppelgänger: A Cache for Approximate Computing. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2015. (Cited on pages 65, 66, 67, 68, and 78).
- [104] J. S. Miguel, M. Badr, and N. E. Jerger. Load Value Approximation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014. (Cited on pages 65, 66, 67, 68, 78, and 79).
- [105] S. Misailovic, D. Kim, and M. Rinard. Parallelizing Sequential Programs with Statistical Accuracy Tests. *ACM Transactions on Embedded Computing Systems (TECS)*, 2013. (Cited on page 63).
- [106] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically Accurate Program Transformations. In *Proceedings of the International Conference on Static Analysis (SAS)*, 2011. (Cited on page 63).
- [107] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of Service Profiling. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, ICSE, 2010. (Cited on page 63).
- [108] S. Misailovic, S. Sidiroglou, and M. C. Rinard. Dancing with Uncertainty. In *Proceedings of the ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, RACES, 2012. (Cited on page 63).
- [109] A.-M. Monazzah, M. Shoushtari, A. Rahmani, and N. Dutt. QuARK: Quality-configurable Approximate STT-MRAM Cache by Fine-grained Tuning of Reliability-Energy Knobs. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2017. (Cited on pages 63, 65, 66, 67, 68, 69, 123, and 129).
- [110] K. Munira, W. H. Butler, and A. W. Ghosh. A Quasi-Analytical Model for Energy-Delay-Reliability Tradeoff Studies During Write Operations in a Perpendicular STT-RAM Cell. *IEEE Transactions on Electron Devices*, 2012. (Cited on page 104).
- [111] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.5: A Tool to Model Large Caches. Technical report, HP Laboratories, 2009. (Cited on pages 37 and 96).

- [112] S. R. Nassif, N. Mehta, and Y. Cao. A Resilience Roadmap. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, 2010. (Cited on page 5).
- [113] N. Nguyen, A. Dominguez, and R. Barua. Memory Allocation for Embedded Systems with a Compile-time-unknown Scratch-pad Size. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2005. (Cited on page 14).
- [114] F. Oboril, A. Shirvanian, and M. Tahoori. Fault Tolerant Approximate Computing using Emerging Non-volatile Spintronic Memories. In *Proceedings of the IEEE VLSI Test Symposium (VTS)*, 2016. (Cited on pages 65, 66, 67, 68, and 69).
- [115] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee. SDRM: Simultaneous Determination of Regions and Function-to-region Mapping for Scratchpad Memories. In *Proceedings of the International Conference on High Performance Computing (HiPC)*, 2008. (Cited on page 11).
- [116] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *Proceedings of the European Conference on Design and Test (EDTC)*, 1997. (Cited on page 14).
- [117] J. Park. *Memory Optimizations of Embedded Applications for Energy Efficiency*. PhD thesis, Dept. Elect. Eng., Stanford, 2011. (Cited on page 37).
- [118] C. Pinto and L. Benini. A Highly Efficient, Thread-safe Software Cache Implementation for Tightly-coupled Multicore Clusters. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2013. (Cited on page 41).
- [119] A. Raha, S. Sutar, H. Jayakumar, and V. Raghunathan. Quality Configurable Approximate DRAM. *IEEE Transactions on Computers (TC)*, 2017. (Cited on pages 65, 66, 67, 68, and 70).
- [120] B. Raj, A. K. Saxena, and S. Dasgupta. Nanoscale FinFET Based SRAM Cell Design: Analysis of Performance Metric, Process Variation, Underlapped FinFET, and Temperature Effect. *IEEE Circuits and Systems Magazine (CAS)*, 2011. (Cited on page 101).
- [121] A. Ranjan, A. Raha, V. Raghunathan, and A. Raghunathan. Approximate Memory Compression for Energy-efficiency. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2017. (Cited on pages 65, 66, 67, 68, and 73).
- [122] A. Ranjan, S. Venkataramani, X. Fong, K. Roy, and A. Raghunathan. Approximate Storage for Energy Efficient Spintronic Memories. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*, 2015. (Cited on pages 65, 66, 67, 68, 69, 104, 109, 111, and 129).

- [123] V. J. Reddi, D. Z. Pan, S. R. Nassif, and K. A. Bowman. Robust and Resilient Designs from the Bottom-up: Technology, CAD, Circuit, and System Issues. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2012. (Cited on page 83).
- [124] M. Rinard. Parallel Synchronization-Free Approximate Data Structure Construction. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Parallelism*, 2013. (Cited on pages 63, 65, 66, and 73).
- [125] A. Ros, M. E. Acacio, and J. M. Garcia. Cache Coherence Protocols for Many-core CMPs. In *Parallel and Distributed Computing*. InTech, 2010. (Cited on page 3).
- [126] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013. (Cited on page 63).
- [127] F. Sampaio, M. Shafique, B. Zatt, S. Bampi, and J. Henkel. Approximation-aware Multi-Level Cells STT-RAM Cache Architecture. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015. (Cited on pages 65, 66, 67, 68, and 69).
- [128] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin. ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing. Technical Report UW-CSE-15-01-01, University of Washington, 2015. (Cited on pages 63 and 92).
- [129] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011. (Cited on pages 63 and 80).
- [130] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate Storage in Solid-state Memories. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013. (Cited on pages 63, 65, 66, 67, 68, 71, 81, 83, and 129).
- [131] N. Sayed, F. Oboril, A. Shirvanian, R. Bishnoi, and M. B. Tahoori. Exploiting STT-MRAM for Approximate Computing. In *Proceedings of the IEEE European Test Symposium (ETS)*, 2017. (Cited on pages 65, 66, 67, 68, and 69).
- [132] M. Shoushtari, A. BanaiyanMofrad, and N. Dutt. Exploiting Partially-Forgetful Memories for Approximate Computing. *IEEE Embedded Systems Letters (ESL)*, 2015. (Cited on pages 63, 65, 66, 67, 68, and 123).
- [133] A. Shrivastava, N. Dutt, J. Cai, M. Shoushtari, B. Donyanavard, and H. Tajik. Automatic Management of Software Programmable Memories in Many-core Architectures. *IET Computers Digital Techniques*, 2016. (Cited on page 14).

- [134] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*, 2011. (Cited on page 63).
- [135] J. Sjödin and C. von Platen. Storage Allocation for Embedded Processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2001. (Cited on page 14).
- [136] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, 2002. (Cited on page 14).
- [137] V. Suhendra, C. Raghavan, and T. Mitra. Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2006. (Cited on pages 11 and 15).
- [138] V. Suhendra, A. Roychoudhury, and T. Mitra. Scratchpad Allocation for Concurrent Embedded Software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2010. (Cited on page 15).
- [139] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali. Proactive Control of Approximate Programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016. (Cited on pages 63 and 124).
- [140] H. Sun, C. Liu, N. Zheng, T. Min, and T. Zhang. Design Techniques to Improve the Device Write Margin for MRAM-based Cache Memory. In *Proceedings of the Great Lakes Symposium on Great Lakes Symposium on VLSI (GLSVLSI)*, 2011. (Cited on page 102).
- [141] M. Taassori, N. Chatterjee, A. Shafiee, and R. Balasubramonian. Exploring a Brink-of-Failure Memory Controller to Design an Approximate Memory System. In *Proceedings of the Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014. (Cited on page 81).
- [142] H. Tajik, B. Donyanavard, and N. Dutt. On Detecting and Using Memory Phases in Multimedia Systems. In *Proceedings of the ACM/IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, 2016. (Cited on pages 6 and 23).
- [143] H. Tajik, B. Donyanavard, N. Dutt, J. Jahn, and J. Henkel. SPMPool: Runtime SPM Management for Memory-Intensive Applications in Embedded Many-Cores. *ACM Transactions on Embedded Computing Systems (TECS)*, 2016. (Cited on page 6).
- [144] R. Takemura, T. Kawahara, K. Miura, H. Yamamoto, J. Hayakawa, N. Matsuzaki, K. Ono, M. Yamanouchi, K. Ito, H. Takahashi, S. Ikeda, H. Hasegawa, H. Matsuoka, and H. Ohno. A 32-Mb SPRAM With 2T1R Memory Cell, Localized Bi-Directional

- Write Driver and ‘1’/‘0’ Dual-Array Equalized Reference Scheme. *IEEE Journal of Solid-State Circuits (JSSC)*, 2010. (Cited on page 104).
- [145] B. Thwaites, G. Pekhimenko, H. Esmailzadeh, A. Yazdanbakhsh, O. Mutlu, J. Park, G. Mururu, and T. Mowry. Rollback-free Value Prediction with Approximate Loads. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, 2014. (Cited on pages 65, 66, 68, and 79).
- [146] Y. Tian, Q. Zhang, T. Wang, F. Yuan, and Q. Xu. ApproxMA: Approximate Memory Access for Dynamic Precision Scaling. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*, 2015. (Cited on pages 65, 66, 68, and 73).
- [147] M. Verma, S. Steinke, and P. Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2003. (Cited on page 14).
- [148] J. Wang and B. H. Calhoun. *Standby Supply Voltage Minimization for Reliable Nanoscale SRAMs*, chapter 6. INTECH, 2010. (Cited on page 82).
- [149] J. Wang and B. H. Calhoun. Minimum Supply Voltage and Yield Estimation for Large SRAMs Under Parametric Variations. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 2011. (Cited on pages ix, 85, and 96).
- [150] L. Wanner and M. Srivastava. ViRUS: Virtual Function Replacement Under Stress. In *Proceedings of the USENIX Conference on Power-Aware Computing and Systems*, 2014. (Cited on page 63).
- [151] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-l. Lu. Reducing Cache Power with Low-cost, Multi-bit Error-correcting Codes. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010. (Cited on page 82).
- [152] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu. Trading off Cache Capacity for Reliability to Enable Low Voltage Operation. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2008. (Cited on page 82).
- [153] X. Xu and H. H. Huang. Exploring Data-Level Error Tolerance in High-Performance Solid-State Drives. *IEEE Transactions on Reliability*, 2015. (Cited on pages 65, 66, 67, 68, and 72).
- [154] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmailzadeh, O. Mutlu, and T. C. Mowry. RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016. (Cited on pages 65, 66, 68, and 79).
- [155] X. Zhang, Y. Zhang, B. Childers, and J. Yang. DrMP: Mixed Precision-aware DRAM for High Performance Approximate and Precise Computing. In *Proceedings of the*

- International Conference on Parallel Architectures and Compilation (PACT)*, 2017. (Cited on pages 65, 66, 67, 68, and 71).
- [156] H. Zhao, L. Xue, P. Chi, and J. Zhao. Approximate Image Storage with Multi-level Cell STT-MRAM Main Memory. In *Proceedings of the International Conference On Computer Aided Design (ICCAD)*, 2017. (Cited on pages 65, 66, 67, 68, and 77).
- [157] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic Cache Contention Detection in Multi-threaded Applications. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2011. (Cited on page 21).
- [158] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009. (Cited on page 116).
- [159] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. Energy Reduction for STT-RAM Using Early Write Termination. In *Proceedings of the International Conference On Computer Aided Design (ICCAD)*, 2009. (Cited on page 117).
- [160] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized Accuracy-aware Program Transformations for Efficient Approximate Computations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012. (Cited on page 63).