Designing Future Low-Power and Secure Processors with Non-Volatile Memory

DISSERTATION

Presented in Partial Fulfillment of the Requirements
for the Degree Doctor of Philosophy
in the Graduate School of The Ohio State University

By

Xiang Pan, B.E., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2018

Dissertation Committee:

Radu Teodorescu, Advisor

Feng Qin

Christopher Stewart

Yinqian Zhang

ABSTRACT

Non-volatile memories such as Spin-Transfer Torque Random Access Memory (STT-RAM), Phase Change Memory (PCM), Resistive Random Access Memory (ReRAM), etc. are emerging as promising alternatives to DRAM and SRAM. These new memory technologies have many exciting characteristics such as non-volatility, high density, and near-zero leakage power. These features make them very good candidates for future processor designs in the power-hungry big data era. STT-RAM, a new generation of Magnetoresistive RAM, in particular is an attractive class of non-volatile memory because it has infinite write endurance, good compatibility with CMOS technology, fast read speed, and low read energy. With its good read performance and high endurance, it is feasible to replace SRAM structures on processor chips with STT-RAM. However, a significant drawback of STT-RAM is its higher write latency and energy compared to SRAM.

This dissertation first presents several approaches to use STT-RAM for future low-power processor designs across two different computing environments (high voltage and low voltage). Overall our target is to take advantage of the benefits of STT-RAM over SRAM to save power and at the same time try the best to accommodate STT-RAM's write drawbacks with novel solutions. In high voltage computing environment, we present a low-power microprocessor framework – NVSleep, that leverages STT-RAM to implement rapid checkpoint/wakeup of idle cores to save power. In

low voltage computing environment, we propose an architecture - Respin, that consolidates the private caches of near-threshold cores into unified L1 instruction/data caches that use STT-RAM to save leakage power and improve performance. On top of this shared L1 cache design, we further propose a novel hardware virtualization core management mechanism to increase resource efficiency and save energy. Although the non-volatility feature of non-volatile memories can be leveraged to build power-efficient designs, it also brings in security concerns as data stored in these memories will be persistent even after system power-off. In order to address this potential security issue, this dissertation deeply studies the vulnerabilities of non-volatile memory as processor caches when exposed to "cold boot" attacks and then proposes an effective software-based countermeasure to eliminate this security threat with reasonable performance overhead.

TABLE OF CONTENTS

Page

Abstract ii

List of Figures vii

List of Tables x

Chapters:

1. Introduction 1

2. Background 5

   2.1 STT-RAM 5
   2.2 Impact of Process Variation on Near-Threshold CMPs 7
   2.3 Advanced Encryption Standard (AES) 8
   2.4 ARM's Cryptographic Acceleration 9

3. NVSleep: Using Non-Volatile Memory to Enable Fast Sleep/Wakeup of Idle Cores 11

   3.1 Introduction 11
   3.2 NVSleep Framework Design 13
      3.2.1 Checkpointing Control 15
      3.2.2 Wakeup Mechanism 16
   3.3 NVSleep Framework Implementation 17
      3.3.1 NVSleepMiss 17
      3.3.2 NVSleepBarrier 20
   3.4 Evaluation Methodology 22
   3.5 Evaluation 23

3.5.1 Application Idle Time Analysis . . . 24
3.5.2 NVSleep Energy Savings . . . 25
3.5.3 NVSleep Overheads . . . 27
3.5.3.1 Performance . . . 27
3.5.3.2 Area . . . 29
3.5.4 Sensitivity Studies . . . 29
3.6 Conclusion . . . 31

4. Respin: Rethinking Near-Threshold Multiprocessor Design with Non-Volatile Memory . . . 32

4.1 Introduction . . . 32
4.2 Near-Threshold CMP With STT-RAM Caches . . . 36
4.2.1 Time-Multiplexing Cache Accesses . . . 39
4.3 Dynamic Core Management . . . 41
4.3.1 Core Virtualization . . . 42
4.3.2 Energy Optimization Algorithm . . . 44
4.3.3 Virtual Core Consolidation . . . 45
4.3.4 Mitigating Core Consolidation Overhead . . . 46
4.4 Evaluation Methodology . . . 48
4.5 Evaluation . . . 50
4.5.1 Power Analysis . . . 51
4.5.2 Performance Analysis . . . 53
4.5.3 Energy Analysis . . . 54
4.5.4 Optimal Cluster Size . . . 57
4.5.5 Shared Cache Impact on Access Latency . . . 58
4.5.6 Dynamic Core Consolidation . . . 60
4.6 Conclusion . . . 62

5. NVInsecure: When Non-Volatile Caches Meet Cold Boot Attacks . . . 63

5.1 Introduction . . . 63
5.2 Threat Model . . . 66
5.3 Cache Aware AES Key Search . . . 67
5.3.1 Technical Challenges . . . 67
5.3.2 Search Algorithm Design . . . 68
5.3.3 Implementation-Specific Considerations . . . 71
5.3.4 Discussion on Scalability . . . 73
5.4 Evaluation Methodology . . . 74
5.5 Attack Analysis . . . 76
5.5.1 Random Information Harvesting . . . 76
5.5.2 Targeted Power-off Attack . . . 83

5.6 Countermeasures 87
5.6.1 Countermeasure Design 87
5.6.2 Countermeasure Effectiveness 90
5.6.3 Performance Overhead 90
5.6.4 Discussion 92
5.7 Conclusion 93

6. Related Work 94

6.1 STT-RAM 94
6.2 Near-Threshold Architectures 95
6.3 Variation-Aware Thread Mapping 96
6.4 Cold Boot Attacks and Defenses 97

7. Conclusion 99

Bibliography 100

LIST OF FIGURES

Figure **Page**

2.1 Basic structure of a Magnetic Tunnel Junction (MTJ) cell in the (a) parallel and (b) antiparallel states. 6

2.2 1T-1MTJ memory cell. 7

2.3 Core-to-core frequency variation at nominal and near-threshold V_{dd}, reproduced from [50]. 8

2.4 AES-128 key schedule with details on key expansion. 9

3.1 SRAM memory structure with STT-RAM backup. 14

3.2 NVSleepMiss hardware-initiated sleep and wakeup. 18

3.3 NVSleepBarrier software-initiated sleep and wakeup. 21

3.4 NVSleepBarrier implementation. 22

3.5 Idle fraction of the execution time for single-threaded benchmarks. . . 25

3.6 Idle fraction of the execution time for multi-threaded benchmarks. . . 25

3.7 Energy consumption of NVSleepMiss for single-threaded benchmarks relative to SRAM baseline. 26

3.8 NVSleepMiss, NVSleepBarrier, and NVSleepCombined energy for multi-threaded benchmarks. 27

3.9 Runtime of NVSleepMiss design for single-threaded benchmarks. . . . 28

3.10 Runtime of different NVSleep designs for multi-threaded benchmarks. 28

3.11 NVSleepMiss and NVSleepBarrier energy for different STT-RAM write latencies, relative to NVNoSleep. 30

3.12 NVSleepMiss energy for different numbers of banks. 30

3.13 NVSleep energy savings in CMPs with 16, 32, and 64 cores. 31

4.1 Dynamic and leakage power breakdown for a 64-core CMP at nominal and near-threshold voltages. 34

4.2 Floorplan of the clustered 64-core CMP design (a) with details of the cluster layout (b). 37

4.3 Example timeline of access requests from cores running at different frequencies to a shared cache running at high frequency. 40

4.4 Overview of the virtual core management system integrated in one cluster. 42

4.5 Greedy selection for dynamic core consolidation. 44

4.6 Power reduction of proposed design for three L2/L3 cache sizes: small, medium, and large. 52

4.7 Relative runtime of SPLASH2 and PARSEC benchmarks for various designs with medium-sized cache. 54

4.8 Energy consumption for small, medium, and large L2 and L3 cache configurations. 55

4.9 Energy consumption for SPLASH2 and PARSEC benchmarks with a core consolidation interval of 160K instructions and a medium-sized L2 and L3 cache. 56

4.10 Shared DL1 cache utilization rate in one cluster. 58

4.11 Fraction of read hit requests serviced by the shared DL1 cache in 1, 2, or more core cycles. 59

4.12 Core consolidation trace of *radix*. 60

4.13 Core consolidation trace of *lu*. 61

4.14 Average number of active cores (and min and max values) using core consolidation for SPLASH2 and PARSEC benchmarks. 62

5.1 Cache view of AES key schedule with 64-byte cache lines. 68

5.2 Details on AES implementation dependent modifications to the key search algorithm. 72

5.3 Probability of finding AES keys in 8MB LLC with various types of benchmarks. 78

5.4 AES key search trace with cumulative LLC miss rate information of (a) *dealII*, (b) *bzip2*, (c) *sjeng*, and (d) *GemsFDTD* benchmarks running on systems without ARM's cryptographic acceleration (NEON) support. 80

5.5 Overall probability of finding AES keys in LLCs with various sizes. . 82

5.6 `poweroff` command triggered operation sequence. 84

5.7 AES key search sequence of normal power-off from start to completion with various sizes of LLCs. 86

5.8 Countermeasure deployment in a system with encrypted storage. . . . 88

5.9 Average countermeasure performance overhead of all sizes of LLCs with different benchmarks. 91

LIST OF TABLES

Table		Page
3.1	Technology choices for NVSleep structures.	15
3.2	NVSleep MSHR with additional fields.	18
3.3	Summary of the experimental parameters.	23
3.4	Energy and area for banked 1KB 32-bit SRAM.	31
4.1	Summary of cache configurations.	48
4.2	CMP architecture parameters.	49
4.3	L1 data cache technology parameters.	50
4.4	Architecture configurations used in the evaluation.	51
4.5	Cluster size impact on performance.	57
5.1	Summary of hardware configurations.	75
5.2	Overview of mixed benchmark groups.	76
5.3	Summary of experiment and workload labels.	77
5.4	Summary of targeted power-off attack results.	85
5.5	Probability of finding AES keys with and without the countermeasure.	90

CHAPTER 1

Introduction

Non-volatile memory (NVM) such as Spin-Transfer Torque Random Access Memory (STT-RAM), Phase Change Memory (PCM), Resistive Random Access Memory (ReRAM), etc. is a promising candidate for replacing traditional DRAM and SRAM memories for both off-chip and on-chip storage [47]. NVMs in general have several desirable characteristics including non-volatility, high density, better scalability at small feature sizes, and low leakage power [26, 79, 85]. The semiconductor industry is investing heavily in NVM technologies and they are getting close to integrating them into products in the near future. Companies such as Everspin [19] and Crossbar [13] are focused exclusively on NVM technologies and have produced NVM chips that are being sold today. A joint effort from Intel and Micron has yielded 3D XPoint [49], a new generation of NVM devices with very low access latency and high endurance, expected to come to market this year. Hewlett-Packard Enterprise's ongoing "The Machine" project [31] is also set to release early this year computers equipped with memristor technology (also known as ReRAM) as part of enabling highly scalable memory subsystems. The industry expects non-volatile memory to replace DRAM off-chip storage in the near future, and SRAM on-chip storage in the medium term.

STT-RAM in particular is an attractive NVM technology with several unique properties including fast read speed, low read energy, good compatibility with CMOS technology, and unlimited write endurance [15, 26, 75, 86]. These characteristics make it a potential candidate for replacing SRAM structures on processor chips. The first part of this dissertation presents NVSleep, a low-power microprocessor framework that leverages STT-RAM to implement fast checkpointing that enables near-instantaneous shutdown of cores without loss of the execution state. NVSleep stores almost all processor state in STT-RAM structures that do not lose content when power-gated. Memory structures that require low-latency access are implemented in SRAM and backed-up by "shadow" STT-RAM structures that are used to implement fast checkpointing. This enables rapid shutdown of cores and low-overhead resumption of execution, which allows cores to be turned off frequently and for short periods of time to take advantage of idle execution phases to save power. We present two implementations of NVSleep: NVSleepMiss which turns cores off when last level cache misses cause pipeline stalls and NVSleepBarrier which turns cores off when blocked on barriers. Evaluation shows significant energy savings for both NVSleepMiss and NVSleepBarrier with a small performance overhead.

The second part of this dissertation looks into combining STT-RAM with multiprocessors operating in the near-threshold voltage range. Near-threshold computing is emerging as a promising energy-efficient alternative for power-constrained environments. Unfortunately, aggressive reduction in supply voltage to the near-threshold range, albeit effective, faces a host of challenges. This includes higher relative leakage power and high error rates, particularly in dense SRAM structures such as on-chip caches. This part of the dissertation presents Respin, an architecture that rethinks

the cache hierarchy in near-threshold multiprocessors. Our design uses STT-RAM to implement all on-chip caches. STT-RAM has several advantages over SRAM at low voltages including low leakage, high density, and reliability. The design consolidates the private caches of near-threshold cores into shared L1 instruction/data caches organized in clusters. We find that our consolidated cache design can service most incoming requests within a single cycle. We demonstrate that eliminating the coherence traffic associated with private caches results in performance boost. In addition, we propose a hardware-based core management system that dynamically consolidates virtual cores into variable numbers of physical cores to increase resource efficiency. We demonstrate that this approach can save additional energy over the baseline while running a mix of benchmark applications.

Although systems with non-volatile memory (e.g. STT-RAM) are not yet commercially available, industry expects non-volatile memory (NVM) technologies to be deployed in production systems in the near future. As such, it is crucial to study the security vulnerabilities of NVM-based computer systems before they are deployed widely. The last part of this dissertation shows that NVMs present new security challenges. In particular, we show that NVM caches are vulnerable to so-called "cold boot" attacks. The original cold boot attacks have demonstrated that once an adversary has gained physical access to a victim computer, he/she could extract the physical DRAM modules and re-plug them into another platform where their content can be read. In contrast to DRAM, SRAM caches are less vulnerable to cold boot attacks, because SRAM data is only persistent for a few milliseconds even at cold temperatures. Therefore, SRAM caches have been generally assumed to be within the security domain of a secure processor because their volatility made these attacks

very challenging. Thus, caches are often used to store sensitive information such as cryptographic keys in various software approaches to defeating cold boot attacks. Our study explores cold boot attacks on NVM caches and defenses against them. Because removing the processor from a system no longer erases the content of the on-chip NVM memory, sensitive information becomes vulnerable. Particularly, this work demonstrates that hard disk encryption keys can be extracted from the NVM cache in cold boot attacks. Multiple attack scenarios are examined in order to evaluate the probability of successful attacks with varying system activities and attack methods. We particularly demonstrate a reproducible attack scenario with very high probability of success. Beyond attack demonstration, we also propose an effective software-based countermeasure. Results show that this countermeasure can completely eliminate the vulnerability of NVM caches to cold boot attacks with a reasonable performance overhead. We hope our work can draw the industry's attention to this specific security vulnerability of NVM caches, and promote integration of countermeasures into hardware and software systems where NVM caches will be deployed.

The rest of this dissertation is organized as follows: Chapter 2 provides required preliminaries related to our work. Chapter 3 details the NVSleep framework, Chapter 4 describes the Respin architecture, and Chapter 5 presents our study on NVM security. Related work will be discussed in Chapter 6. Finally, conclusions will be drawn in Chapter 7.

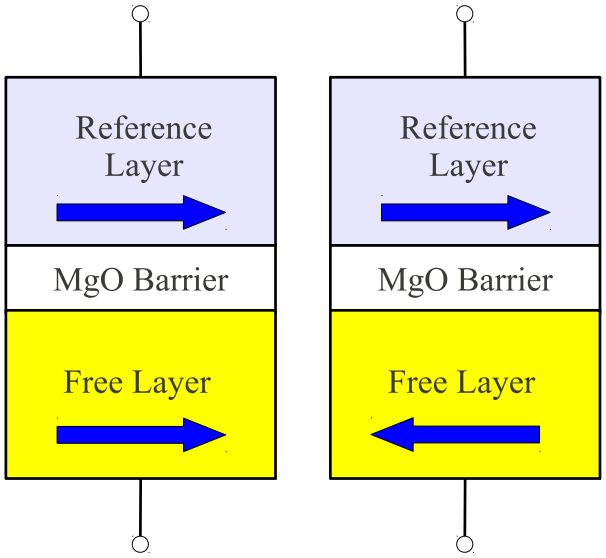
CHAPTER 2

Background

2.1 STT-RAM

STT-RAM memory uses Magnetic Tunnel Junction (MTJ) as the basic storage unit. An MTJ consists of two ferromagnetic layers – a reference layer and a free layer that are divided by a tunnel barrier (MgO) as shown in Figure 2.1. Each layer is associated with a magnetic field where the direction can only be changed in the free layer by passing a large current through the MTJ device. The direction of the free layer relative to the reference layer provides the storage functionality of the MTJ-based memory cell. If the magnetic direction of the free layer matches that of the reference layer, the MTJ cell is said to be in "parallel" state (2.1a). In the parallel state the resistance of the MTJ is low and represents a logical "0". If the magnetic direction of the free layer is reversed, the MTJ is said to be in an "antiparallel" state in which the MTJ has high resistance and represents a logical "1" (2.1b).

The most common STT-RAM cell design is the 1-transistor-1-MTJ (1T-1MTJ) structure shown in Figure 2.2. Each MTJ unit is controlled by a single CMOS transistor connected to the word line (WL) select signal. A source line (SL) is connected to the source of the control transistor and a bit line (BL) is connected to the free



(a) parallel (0) **(b)** antiparallel (1)

Figure 2.1: Basic structure of a Magnetic Tunnel Junction (MTJ) cell in the (a) parallel and (b) antiparallel states.

layer of the MTJ. A read operation involves applying a small current between the SL and BL lines allowing a sense amplifier to detect the drop in voltage over the MTJ which is affected by its resistance. A "0" will be registered for a low resistance state and "1" for a high resistance state. Writing into an STT-RAM cell requires changing the direction of the magnetic field in the free layer and therefore necessitates a large electrical charge to pass through the device. This is achieved by applying higher currents for longer duration. As a result, write operations in STT-RAM are both slower and consume more energy compared to SRAM. A study by Guo et al. [26] suggests STT-RAM to be competitive with SRAM in read latency and energy. However, the write latency and energy were found to be much higher than SRAM when using small

structure sizes. This gap between STT-RAM and SRAM in write characteristics significantly diminishes when the storage structure size becomes large, thus making the use of STT-RAM a feasible technology for modern microprocessors in meeting cache capacity trends.

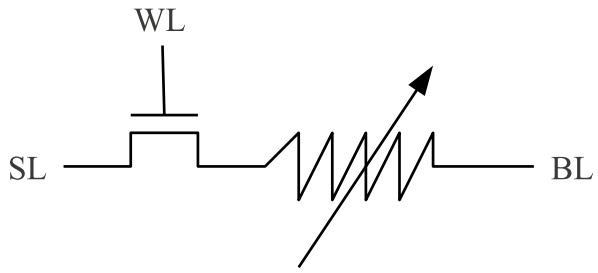


Figure 2.2: 1T-1MTJ memory cell.

2.2 Impact of Process Variation on Near-Threshold CMPs

A growing challenge chip designers face with deep submicron technology is process variation. Process variation effects are introduced to microprocessors during the manufacturing stage and can occur at different levels, including: wafer-to-wafer (W2W), die-to-die (D2D), and within-die (WID). Random effects such as fluctuations in dopant densities impact the effective length (L_{eff}) and threshold voltage (V_t) parameters of transistors which play a key role in controlling device switching speeds. The correlation between V_{dd} and transistor gate delays as indicated by the alpha power model [67] renders processor cores increasingly susceptible to low-voltage operation. Figure 2.3 shows core-to-core variation data reproduced from [50] for a 32-core CMP at nominal and near-threshold voltages. A dramatic increase in variation can

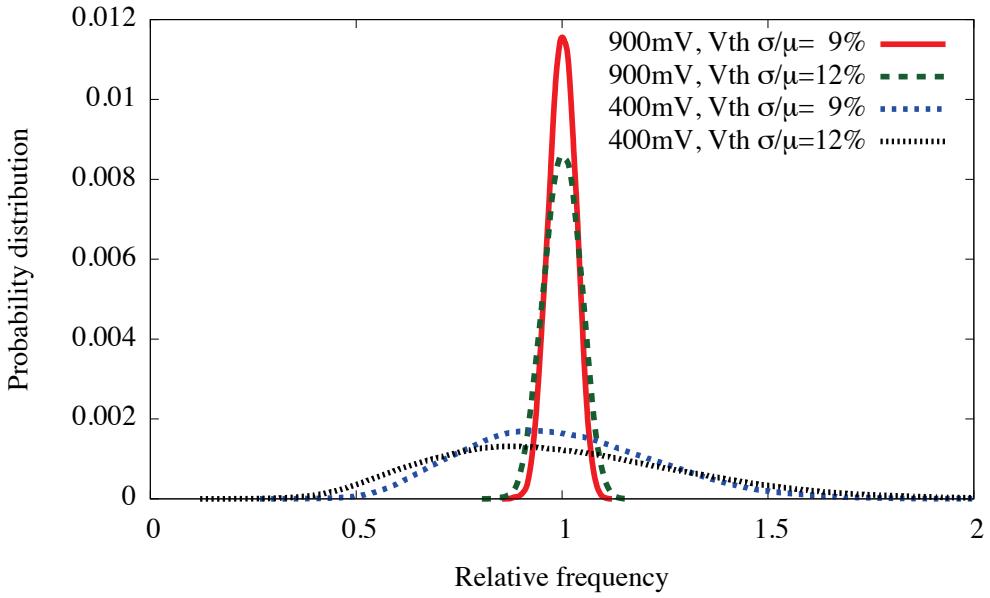


Figure 2.3: Core-to-core frequency variation at nominal and near-threshold V_{dd}, reproduced from [50].

be observed from 4.4% frequency standard deviation divided by the mean (σ/μ) at nominal V_{dd} (900mV) to 30.6% σ/μ at near-threshold (400mV). With this level of variation a subset of cores become twice as fast as others within the same die once supply voltage transitioned to the near-threshold range.

2.3 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) [3] is a symmetric block cipher which encrypts or decrypts a block of 16 bytes of data at a time. Both encryption and decryption use the same secret key. The commonly used AES key size is either 128-bit (AES-128), 192-bit (AES-192), or 256-bit (AES-256). Before performing any encryption/decryption operations, the secret key must be expanded to an expanded key (also known as a key schedule) consisting of individual subkeys that will be used

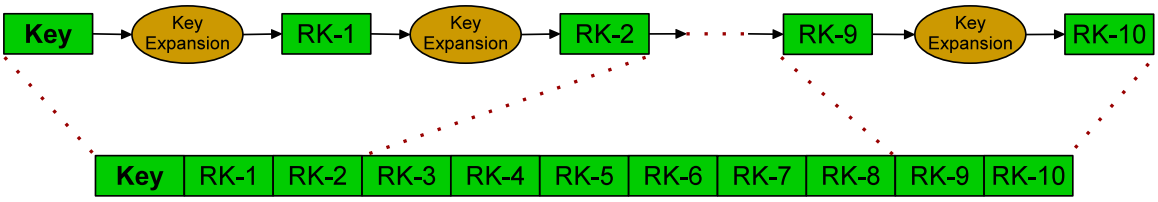


Figure 2.4: AES-128 key schedule with details on key expansion.

for different internal rounds of the AES (Rijndael) algorithm. The key expansion process is shown in Figure 2.4. The key schedule starts with the original secret key, which is treated as the initial subkey. The following subkeys (also known as round keys) are computed from the previously generated subkeys using publicly known functions such as `Rot-Word`, `Sub-Word`, `Rcon`, `EK`, and `K` defined in the key expansion algorithm [8]. In each round of the key generation, the same sequence of functions will be executed. Each newly generated subkey will have the same size, e.g. 16 bytes in AES-128. This process repeats a certain number of rounds (e.g. 10 rounds in AES-128) until the expanded key is completely generated. Each subkey from the expanded key will then be used in separate rounds of AES encryption or decryption algorithms.

2.4 ARM's Cryptographic Acceleration

Many of today's processors include vectorization support in the form of Single Instruction Multiple Data (SIMD) engines. The popularity of this hardware support stems from the multitude of applications that are suitable for vectorization including cryptography, image processing, and speech recognition to name a few. In ARM processors, the SIMD engine is codenamed NEON and was first introduced as part

of the ARMv7 architecture. Later on, additional enhancements were debuted with the ARMv8 architecture. In the ARMv8 architecture, NEON consists of 32 registers (v0 - v31) where each register is 128-bit wide. As such, each NEON register can conveniently load a 128-bit AES key or a full round of the AES key schedule into a single register obviating the need for multiple accesses to the cache or the memory subsystem. In addition, ARMv8 introduces cryptographic extensions that include new instructions that can be used in conjunction with NEON for AES, SHA1, and SHA2-256 algorithms. In the case of AES, the available instructions are: `AESD` for single round decryption, `AESE` for single round encryption, `AESIMC` for inverse mix columns, and `VMULL` for polynomial multiply long. In this dissertation, we explore the use of the NEON engine, in addition to the ARMv8 cryptographic extensions in order to conduct a wholistic study that reflects side channel vulnerabilities associated with encrypted storage when using state-of-the-art ARM processors.

CHAPTER 3

NVSleep: Using Non-Volatile Memory to Enable Fast Sleep/Wakeup of Idle Cores

3.1 Introduction

Power consumption is a first-class constraint in microprocessor design. The increasing core count in chip multiprocessors is rapidly driving chips towards a new "power wall" that will limit the number of compute units that can be simultaneously active. Traditional power management techniques such as dynamic voltage and frequency scaling (DVFS) are becoming less effective as technology scales. Supply voltage scaling has slowed significantly limiting the range and effectiveness of DVFS. Techniques like clock gating are not affected by voltage scaling but they cannot control leakage power which is expected to increase in future technologies [36]. Power gating, a technique that cuts power supply to functional units can be an effective mechanism for saving both leakage and dynamic power. Unfortunately power gating leads to the loss of data stored in the gated units. Stateless units such as ALUs can be restarted with little overhead. However, units with significant storage such as register files, caches, and other buffers and queues hold large amounts of data and state

information. Restoring that information following a shutdown incurs a significant performance overhead.

This work proposes NVSleep, a low-power microprocessor framework that leverages STT-RAM to implement rapid shutdown of cores without loss of execution state. This allows cores to be turned off frequently and for short periods of time to take advantage of idle execution phases to save power. We present two implementations of NVSleep: NVSleepMiss which will turn cores off when last level cache (LLC) misses cause pipeline stalls and NVSleepBarrier which will turn cores off when blocked on barriers.

In both NVSleep implementations all memory-based functional units that are not write-latency sensitive (such as caches, TLBs, and branch predictor tables) are implemented using STT-RAM. These structures do not lose content when power-gated. Other on-chip structures that require low-latency writes are implemented using SRAM and backed-up by shadow STT-RAM structures. A fast checkpointing mechanism stores modified SRAM content into STT-RAM. After the content is saved, the entire structure can be power-gated. When power is restored, the content of the SRAM master is retrieved from the non-volatile shadow. This allows rapid shutdown of cores and low-overhead resumption of execution when cores are powered back up.

Evaluation using SPEC CPU2000, PARSEC, and SPLASH2 benchmarks running on a simulated 64-core system shows average energy savings of 21% for NVSleepMiss in SPEC2000 benchmarks and 34% for NVSleepBarrier in high barrier count multi-threaded workloads from PARSEC and SPLASH2 benchmarks. The energy savings are achieved with a very small performance overhead.

Overall, this work makes the following contributions:

- To the best of our knowledge, NVSleep is the first work to take advantage of the non-volatility feature of STT-RAM to implement rapid pipeline-level checkpointing.
- NVSleep proposes a general and low overhead framework for reducing energy consumption by exploiting short idle execution phases.
- NVSleep is also the first checkpointing framework that is sufficiently fast to allow cores to be shutdown during LLC misses.

3.2 NVSleep Framework Design

NVSleep improves microprocessor energy efficiency by rapidly turning off cores during idle periods and quickly restoring them to full activity when work becomes available. NVSleep is designed to both checkpoint state very quickly and restore execution almost instantly after the core is turned on, without requiring the flushing and refilling of the pipeline. This allows NVSleep to take advantage of short idle execution phases such as those caused by misses in the last level cache.

The NVSleep framework uses two designs for on-chip memory structures. Storage units that are less sensitive to write latency – such as caches and branch predictor tables – are implemented with non-volatile STT-RAM equivalents. When a core is powered off, these units will not lose state. In order to improve the write performance of STT-RAM structures, especially in the presence of bursty activity, we add small SRAM write buffers. A similar optimization was introduced by prior work [26, 74].

Memory structures that are more sensitive to write latency and are frequently updated in the critical path of the execution (such as Register File, Reorder Buffer,

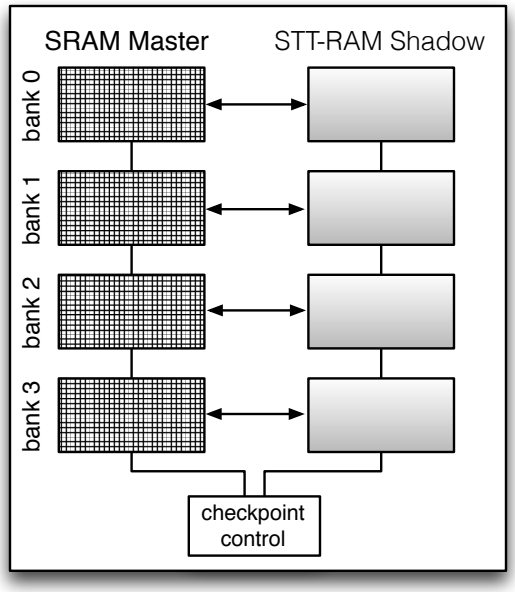


Figure 3.1: SRAM memory structure with STT-RAM backup.

etc.) are implemented using a hybrid SRAM/STT-RAM design. Figure 3.1 illustrates this design. The primary storage elements are implemented using SRAM. The SRAM "Master" banks are backed-up using STT-RAM "Shadow" arrays of equal size. When a checkpoint is initiated, the SRAM entries that have been updated since the last checkpoint are transferred to the STT-RAM Shadow. To speed up the checkpointing process, all hybrid memory structures are banked, allowing all banks to be checkpointed in parallel. Since banking introduces additional area/power overheads, we experiment with various banking options to determine the optimal configuration. To reduce overhead, the shadow and master banks share row decoders. This is possible because during checkpointing and restore the same rows are being accessed in both the master and the shadow. The checkpoint is coordinated by control logic associated with each hybrid structure. The control logic generates addresses for the checkpoint

NVSleep Core Units	Technology
All Caches	STT-RAM
I-TLB and D-TLB	STT-RAM
Branch Prediction Table	STT-RAM
Register File	SRAM + STT-RAM Shadow
Instruction Window	SRAM + STT-RAM Shadow
Reorder Buffer	SRAM + STT-RAM Shadow
Load/Store Queue	SRAM + STT-RAM Shadow
Pipeline Registers	SRAM + STT-RAM Shadow
All Logic	CMOS

Table 3.1: Technology choices for NVSleep structures.

and restore sequence and checks and updates "modified" bits used to identify blocks that have been updated since the last checkpoint.

All pipeline registers outside in the processor are implemented using CMOS flip-flops and backed-up with STT-RAM shadows. Their content is checkpointed in parallel in a single write cycle making banking unnecessary. For completeness, Table 3.1 enumerates the technologies used in the principal components of the NVSleep framework.

3.2.1 Checkpointing Control

NVSleep controls checkpointing, power-down, and wakeup of cores in a distributed fashion across the chip. It relies on the L1 cache controller of each core to help coordinate both the sleep and wakeup process. NVSleep uses two mechanisms for triggering the sleep sequence: one that is entirely hardware-initiated and managed, and one that uses a software API. The hardware-driven mechanism is appropriate for exploiting idle phases caused by events that can be easily identified by the hardware – such as misses in the last level cache.

The software API can be used by the system to request the shutting down of cores. The API relies on a dedicated `sleep()` instruction associated with a reserved memory address `0xADDR` that is tracked by the cache controller. The instruction can be used by the compiler or programmer when an idle execution phase is expected. For instance, a core can be shut down while it is blocked on a barrier, during long latency I/O transfers, or while it is waiting for a lock to be released.

3.2.2 Wakeup Mechanism

Wakeup mechanisms are based on the L1 cache controllers of each core to coordinate the wakeup processes. The cache controllers are not power gated and are therefore available to initiate and coordinate wakeup events. These events include returning misses or wakeup messages from other cores.

For the hardware-initiated sleep, if the sleep event has been triggered by an LLC miss, the cache controller wakes up the core once the missing data makes its way to the L1. The wakeup of a core that was explicitly shut down with the `sleep()` instruction has to be initiated by another core or service processor. The wakeup of a core is triggered by writing to the sleep `0xADDR` address associated with that core. An update or invalidate message for that address will direct the cache controller of the sleeping core to start waking up.

The wakeup overhead depends on how quickly the core can be brought back online and have its state restored. Bringing a system online after sleep can cause ringing in the power supply, which can lead to voltage droops. To prevent large droops we gradually ramp-up core wakeup. In the first phase of wakeup no computation is performed to allow the supply lines to settle. In the second phase, checkpointed data

is restored from shadow STT-RAM structures. Finally, normal execution is resumed. More importantly, we do not allow multiple cores to wakeup simultaneously, limiting the current ramp-up to 1/N of the chip maximum, where N is the number of cores.

3.3 NVSleep Framework Implementation

We developed two applications of the NVSleep framework. The first, which we call *NVSleepMiss* is hardware-controlled and shuts down cores that block on misses in the last level cache. The second, which we call *NVSleepBarrier*, is software-controlled and turns off cores that are blocked on barrier synchronization.

3.3.1 NVSleepMiss

Last level cache misses can lead to hundreds of cycles of stalled execution due to long latency memory accesses. Even though out-of-order processors can hide some of that latency, pipelines will eventually stall when independent instructions are no longer available. Even though stalled cores don't consume as much power as active cores, their idle power is still significant. For instance, modern Intel processors idle at anywhere between 10W and 50W [34]. NVSleepMiss addresses this inefficiency.

NVSleepMiss requires identification and tracking of misses in the last level cache. For this purpose we augment the miss handling status register (MSHR) of the L1 cache controller. We add two new fields to the standard MSHR design, as shown in Table 3.2. The first field, labeled "LLC Miss", is a one-bit tag used to indicate whether this L1 cache miss ends up missing in the last level cache as well. This tag is used to decide when a core should be asked to sleep. The second field is "Pending LD" which is used to keep track of other L1 loads that might still be pending when the core shuts down.

Address	Type/Misc.	LLC Miss	Pending LD
0xAA76...80	...	0	0
0xC342...F7	...	1	0
0xFE34...25	...	0	1

Table 3.2: NVSleep MSHR with additional fields.

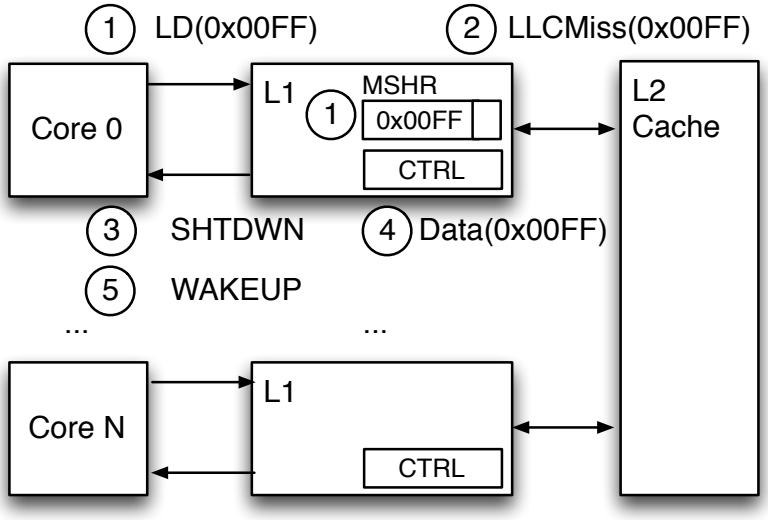


Figure 3.2: NVSleepMiss hardware-initiated sleep and wakeup.

Figure 3.2 illustrates the steps involved in the NVSleepMiss shutdown and wakeup. In this example a load request misses in the L1 cache in step ①. The controller of the last level cache informs the L1 cache controller of an LLC miss event in step ②. The L1 cache controller will find the related entry in the MSHR table and update its "LLC Miss" field to 1. If there are no other pending LLC misses, a shutdown signal will be sent to the core (step ③).

A core initiate the shutdown sequence upon receipt of a sleep signal from the cache controller. To ensure the core wakes up in a consistent state, all instructions that follow the LLC miss in program order are allowed to complete and are retired from the

reorder buffer. Any instructions that are still "in-execution" (meaning they occupy the execution cluster) are allowed to complete and no new instructions are dispatched. The checkpointing sequence begins as soon as the core receives the sleep signal and takes place in parallel with the draining of the execution pipeline. Only SRAM/STT-RAM hybrid structures require explicit checkpointing. The checkpointing control unit copies all modified entries in the SRAM section to STT-RAM. Once the pipeline is drained, a second checkpointing phase is initiated to save any modified entries still remaining (such as those modified while draining the execution cluster).

While draining the execution pipeline after receiving the sleep signal the core could still send load requests to the cache. The core does not need to wait for these loads to be serviced in case they miss in the L1. The cache controller will keep track of what requests have completed while the core is sleeping using the "Pending LD" field of the MSHR. All load requests in the MSHR that are marked as "Pending LD"s will be re-issued to the cache by the cache controller after the core wakes up. This ensures the latest copy of the data is supplied to the core after wakeup. Only loads need to be tracked because stores can complete while the processor is sleeping and do not require any data to be sent to the core.

If data arrives at the L1 from a lower-level cache while the core is sleeping, that data will be written into the cache. If the data corresponds to a load request, the associated MSHR "Pending LD" field will be set to 1. If this is the last pending LLC miss, a wakeup signal will be sent to wake up the sleeping core. This is illustrated by steps ④ and ⑤ in Figure 3.2. Otherwise, the "LLC Miss" tag will be set to 0 but no wakeup signal will be generated. This is because other LLC misses are still pending so there is no reason to wake up the core just yet. After the core is woken

up, all pending loads in the MSHR table will be re-issued to the L1 cache. To avoid a deadlock, these re-issued requests will not ask the processor to sleep, even if they result in an LLC miss.

In the NVSleep framework the cache controller of a sleeping core is never shut down. This allows the cache controller to initiate the wakeup process. It also ensures that all coherence requests continue to be served even when the core is sleeping. As a result, no changes to the coherence protocol are needed.

3.3.2 NVSleepBarrier

Parallel applications often use software barriers to coordinate execution of multiple threads. These threads are sometimes unbalanced for algorithmic or runtime-related reasons. As a result many cores may end up spending a significant portion of time idle while waiting for slow threads to reach barriers. NVSleepBarrier addresses this inefficiency by shutting down cores blocked on barrier synchronization.

The NVSleepBarrier implementation uses the NVSleep API through the `sleep(0xADDR)` instruction. The instruction is treated like a special load instruction that reads from the `0xADDR` address. Figure 3.3 illustrates this process. When the `sleep(0xADDR)` instruction is executed, a load from address `0xADDR` is sent to the cache ①. The cache allocates a line for the data at `0xADDR` and marks it as reserved by setting a dedicated bit in the tag ②. This reserved address will be used to trigger the wakeup process.

The `sleep()` instruction also acts as a memory fence instruction, not allowing any memory access reordering with respect to the load to address `0xADDR`. This will ensure that when the `sleep()` instruction retires, there will be no pending loads or stores in the cache that follow the `sleep()` instruction in program order. This will

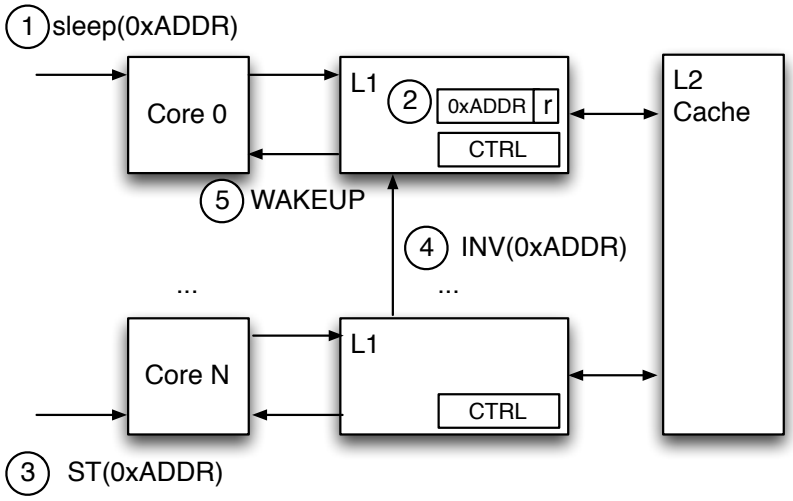


Figure 3.3: NVSleepBarrier software-initiated sleep and wakeup.

allow the core to go to sleep as soon as the `sleep()` instruction retires and no loads will have to be re-issued when the core wakes up.

With this instruction in place, NVSleepBarrier requires minimal changes to the standard software barrier implementation. Figure 3.4 shows code for the implementation of NVSleepBarrier in a generic sense-reversing barrier. The `sleep()` instruction is executed by threads that block on the barrier. The global "sense" variable address is passed as a parameter to the `sleep()` instruction. As a result, the "sense" variable becomes the *wakeup trigger* for all the sleeping cores. Since threads that block on the barrier will only read the "sense" variable, they will not trigger a wakeup. The last thread to reach the barrier will follow the *else* path through the code and will write to the "sense" variable, inverting its direction. The write will also trigger an "invalidate" message to the caches of all blocked cores. This is illustrated in Figure 3.3 by steps ③ and ④. These trigger messages will wake up the sleeping cores, allowing them to resume execution ⑤.

```
void barrier(int count, int sense, int num_threads)
{
 int local_sense;
 // Each core toggles its own sense.
 local_sense = !sense;
 if(count != num_threads-1) {
  // Not the last thread, block.
  while(local_sense != sense) {
    // NVSleep instruction, with sense as
    // wakeup trigger.
    sleep(&sense);
  }
 } else {
  // Last thread in the barrier.
  count = 0;
  // By writing to the sense variable
  // blocked cores are woken up and
  // subsequently released from barrier.
  sense = local_sense;
 }
}
```

Figure 3.4: NVSleepBarrier implementation.

3.4 Evaluation Methodology

We modeled a 64-core CMP in 32nm technology. Each core is a dual-issue out-of-order architecture. We used SESC [63] to simulate the baseline SRAM-based CMP as well as the NVSleep framework. Table 3.3 summarizes the architectural parameters used in our simulations. We used CACTI [57] to extract energy per access for all SRAM memory structures including register file, reorder buffer, instruction window, etc. CACTI was also used to model the energy and area overhead for the banked SRAM memory structures (hybrid register file and reorder buffer, etc.).

For modeling STT-RAM structures we used data from [26] for the access latency and energy, and NVSim [16] to estimate chip area overhead. We also modeled leakage power based on estimated unit area and technology (CMOS vs. STT). We plugged

CMP Architecture	
Cores	64, 32, and 16 out-of-order
Fetch/Issue/Commit Width	2/2/2
Register File	76 int, 56 fp
Instruction Window	56 int, 24 fp
L1 Data Cache	4-way 16KB, 1-cycle access
L1 Instruction Cache	2-way 16KB, 1-cycle access
Shared L2	8-way 2MB, 12-cycle access
Main Memory	300 cycle access latency
STT-RAM Read Time	1 cycle
STT-RAM Write Time	10 cycles
SRAM Read/Write Time	1 cycle
STT-RAM Read Energy	0.01pJ/bit
STT-RAM Write Energy	0.31pJ/bit
SRAM Read/Write Energy	0.014pJ/bit
Core Wakeup Time	30 cycles (10ns)
Coherence Protocol	MESI
Technology	32nm
Vdd	1.0V
Clock Frequency	3GHz

Table 3.3: Summary of the experimental parameters.

these energy numbers into the activity model of the SESC simulator to obtain power consumption and energy.

We ran benchmarks from the SPEC CPU2000, SPLASH2, and PARSEC suites. The benchmark sets include single-threaded and multi-threaded benchmarks. Some parallel benchmarks have heavy barrier activity while others use barriers sporadically or not at all.

3.5 Evaluation

We evaluate the energy and performance implications of the two implementations of NVSleep: NVSleepMiss and NVSleepBarrier. We also evaluate the time and energy cost of checkpointing and show some sensitivity analysis results. We compare NVSleep

with a baseline system (*SRAM Baseline*) in which all memory structures are built with SRAM. The baseline system employs clock-gating of idle functional units to reduce power.

3.5.1 Application Idle Time Analysis

For the purpose of this analysis we define idle time as cycles in which no instruction is retired and no instruction is in execution in a functional unit. During those cycles the processor is virtually stalled. Figure 3.5 shows the percentage of idle time in the total execution time for single-threaded benchmarks. The large number of misses in memory-bound applications like *mcf*, *equake*, *mgrid*, and *swim* leads to idle cycle counts that exceed 50%. Compute-bound applications such as *bzip2* on the other hand have very little idle time. We expect NVSleepMiss to benefit memory-bound applications, with little or no benefit to compute-intensive applications that experience relatively few misses.

For multi-threaded applications, in addition to looking at pipeline stalls due to LLC misses, we also examine idle time spent by cores blocked on barriers. The idle time relative to the total execution time broken down into pipeline stalls and barrier blocked time is shown in Figure 3.6. We find that for the multi-threaded applications we examine, stalls due to LLC misses account for very small fraction of execution time (less than 2% on average). The amount of idle time spent in barriers depends on two factors: the number of barriers and the level of imbalance between work done by individual threads. Applications with large numbers of barriers such as *streamcluster* or with significant imbalance such as *lu* and *fluidanimate* spend up to 89% of their execution time idling inside barriers. *ocean*, on the other hand, is stalled

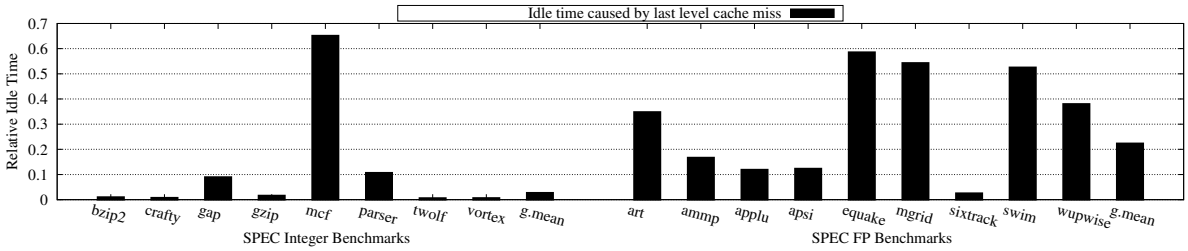


Figure 3.5: Idle fraction of the execution time for single-threaded benchmarks.

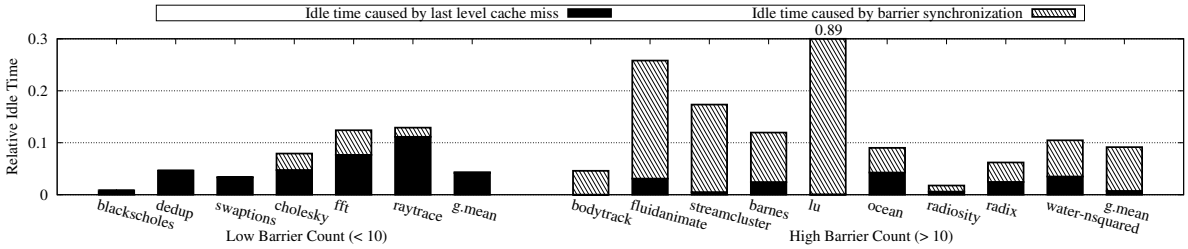


Figure 3.6: Idle fraction of the execution time for multi-threaded benchmarks.

only about 4.7% of its time, even though it runs through about 900 dynamic barrier instances. This is because *ocean* is very balanced, with threads reaching barriers almost simultaneously and leaving them rapidly. We expect unbalanced applications to benefit most from NVSleepBarrier. Naturally, benchmarks that use no barriers such as *dedup* will see no benefit from NVSleepBarrier.

3.5.2 NVSleep Energy Savings

Figure 3.7 shows the energy consumption for NVSleepMiss relative to the SRAM baseline for single-threaded benchmarks. Applications with high number of misses and frequent stalls benefit greatly from NVSleepMiss. For example, *mcf* with the longest idle time achieves 54% energy reduction using NVSleepMiss compared to

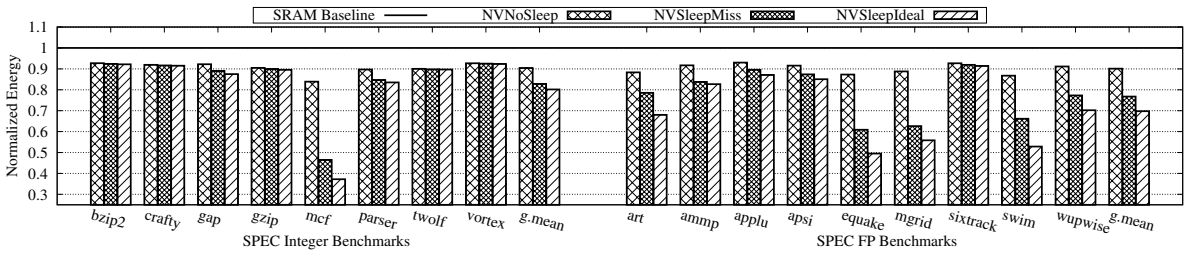


Figure 3.7: Energy consumption of NVSleepMiss for single-threaded benchmarks relative to SRAM baseline.

the baseline. Similar energy reduction is achieved by *equake*, *mgrid*, and *swim*. On average, the energy savings of NVSleepMiss are 17.2% for SPEC Int benchmarks and 23.3% for SPEC FP benchmarks.

We compare NVSleepMiss with two reference designs. One is the NVSleep framework with the sleep option disabled – we call this *NVNoSleep*. In NVNoSleep all energy savings come from leakage reduction from STT-RAM structures. On average, energy is 8-15% higher for NVNoSleep compared to NVSleepMiss. We also compare to an ideal version of NVSleep that has no checkpointing overhead (*NVSleepIdeal*). NVSleepIdeal is 3.3% and 9.1% more energy efficient than NVSleepMiss for SPEC Int and SPEC FP respectively.

For the multi-threaded benchmarks we examine the energy benefits of NVSleepMiss, NVSleepBarrier, and the combined application of the two techniques (*NVSleepCombined*). Figure 3.8 shows the energy savings achieved by the three NVSleep techniques compared to the baseline. Applications with fewer than 10 barriers get virtually no benefit from NVSleepBarrier. For applications with more than 10 barriers the energy savings depend on the level of workload imbalance and the number of barriers. For *lu*, which is extremely unbalanced, the energy reduction exceeds

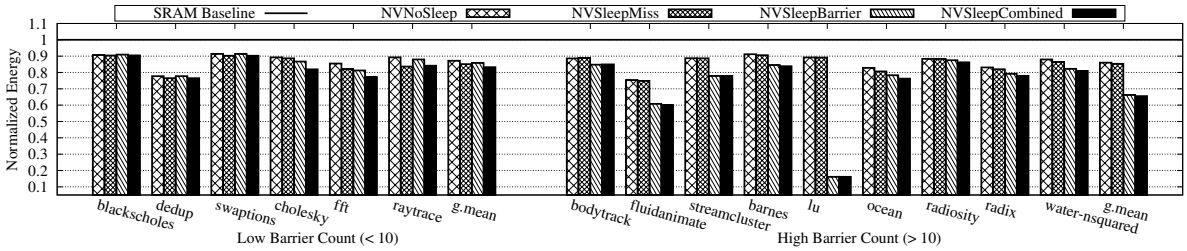


Figure 3.8: NVSleepMiss, NVSleepBarrier, and NVSleepCombined energy for multi-threaded benchmarks.

80%. *streamcluster* has over 4000 barriers but is fairly balanced. Its energy reduction is 22.1%. On average, applications with more than 10 barriers achieve a very significant energy reduction of 33.8% with NVSleepBarrier. This represents a 22.4% improvement over NVNoSleep.

NVSleepMiss does not help much in the case of multi-threaded benchmarks since miss-related idle time is small. As a result NVSleepCombined is only marginally more energy efficient than NVSleepBarrier.

3.5.3 NVSleep Overheads

3.5.3.1 Performance

Figures 3.9 and 3.10 show the runtimes of NVSleep implementations for single-threaded and multi-threaded benchmarks respectively. There are mainly two sources of performance overhead in NVSleep. The first one is caused by the pipeline drain required to shut down cores in consistent states. This drain means that cores resume execution after a shutdown with fewer instructions in their instruction windows than they would otherwise have available. This might slow down execution in some cases after the core is woken up.

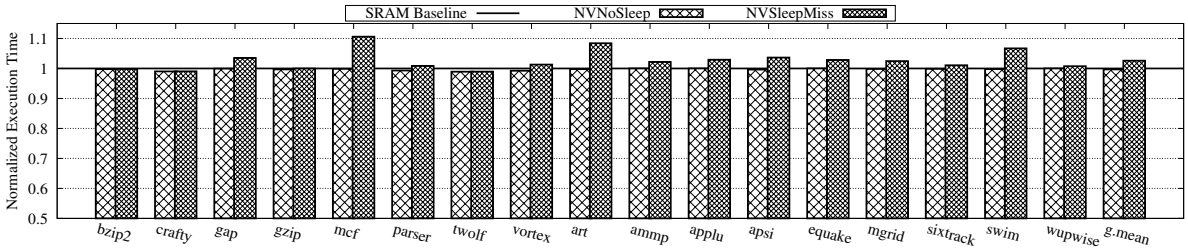


Figure 3.9: Runtime of NVSleepMiss design for single-threaded benchmarks.

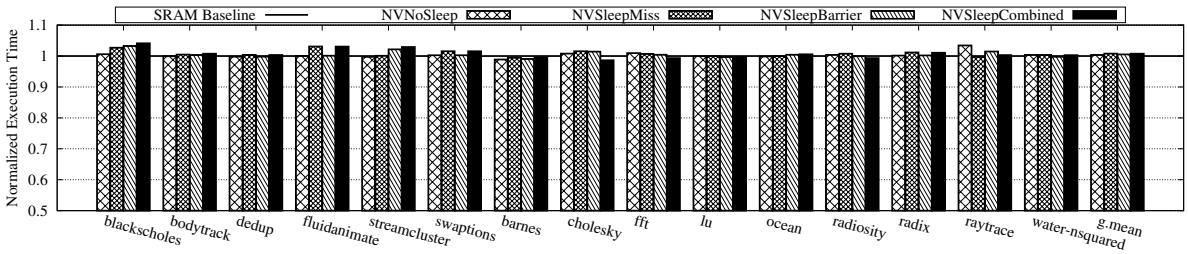


Figure 3.10: Runtime of different NVSleep designs for multi-threaded benchmarks.

The other source of performance overhead is related to the way NVSleep handles multiple LLC misses that occur in close temporal proximity. In NVSleep, a core is only woken up after all LLC misses have returned. As a result, the core will miss the opportunity to work on some instructions that are dependent on data that has become available since the core has been shut down. This explains why we observe more than 5% performance overhead in benchmarks like *mcf*, *art*, and *swim*, in which multiple misses per sleep event are common. On average, the performance overhead of NVSleepMiss is less than 3% and that of NVSleepBarrier and NVSleepCombined is less than 1%.

3.5.3.2 Area

The NVSleep framework has some area overhead due to the banked SRAM blocks and STT-RAM shadow structures. According to CACTI and NVSim simulations with our area model, the 8-bank NVSleep framework design will increase the processor core area by 60%. However, since STT-RAM is much denser than SRAM, the cache area is only 16% of the baseline design. Overall, the total chip area overhead of NVSleep adds up to less than 3% of the SRAM baseline.

3.5.4 Sensitivity Studies

The main overhead of STT-RAM is high write latency. To better understand its impact on overall energy savings, we experimented with various STT-RAM write pulses ranging from aggressive 2ns to conservative 13ns. Figure 3.11 shows that average energy savings for NVSleepMiss with single-threaded benchmarks gradually become smaller as the STT-RAM write latency increases. When the write latency reaches 11ns NVSleepMiss saves almost no energy. On the other hand, since checkpointing is rare in NVSleepBarrier, the increasing STT-RAM write latency has almost no impact on the overall energy savings of NVSleepBarrier. All our previous experiments have assumed an STT-RAM write latency of 3.3ns, also used in prior work [26].

The number of banks in the hybrid SRAM/STT-RAM structures has an impact on the overall energy savings because it directly affects the performance and energy overhead of checkpointing. Figure 3.12 shows the energy consumption of NVSleepMiss with the hybrid memory structures configured with 1, 2, 4, and 8 banks. We show average energy across the SPEC benchmarks. Using higher number of banks reduces performance overhead parallelism because all banks can be checkpointed in parallel.

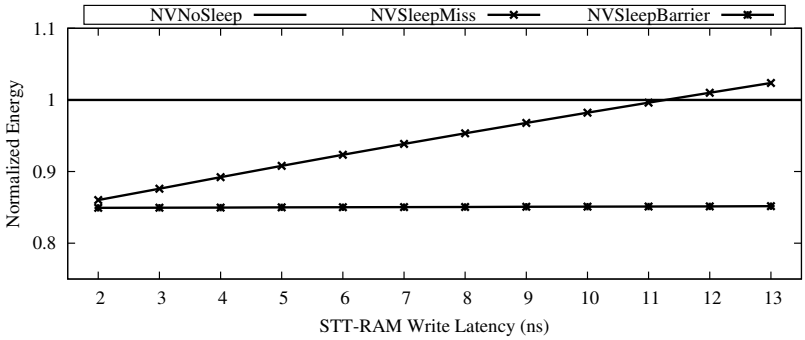


Figure 3.11: NVSleepMiss and NVSleepBarrier energy for different STT-RAM write latencies, relative to NVNoSleep.

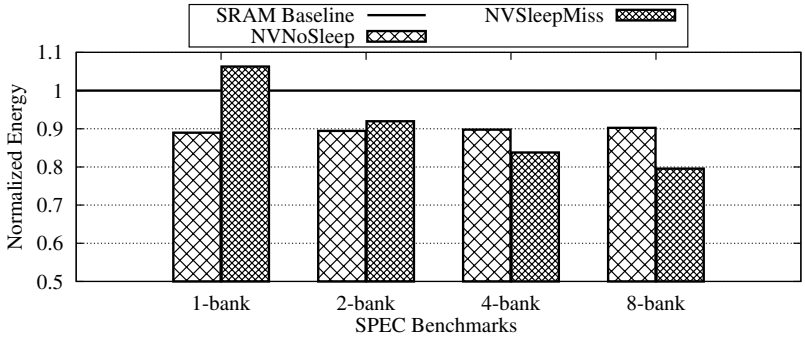


Figure 3.12: NVSleepMiss energy for different numbers of banks.

However, increasing the number of banks also has an energy and area cost. The energy and area sensitivity with the number of banks is shown in Table 3.4. The optimal configuration for our system is the 8-bank design.

To examine the scalability of NVSleepBarrier we run the same experiments on simulated 16 and 32-core systems in addition to the 64-core system. As shown in Figure 3.13, we observe that NVSleepBarrier saves more energy in higher core count systems. This is because barrier idle time and workload imbalance tends to increase with the number of cores. NVSleepBarrier lowers energy by 24% on the 16-core system, by

Num of Banks	Energy/Access (nJ)	Area (mm^2)
1	0.000448	0.007543
2	0.000552	0.012091
4	0.000628	0.018883
8	0.000741	0.029032

Table 3.4: Energy and area for banked 1KB 32-bit SRAM.

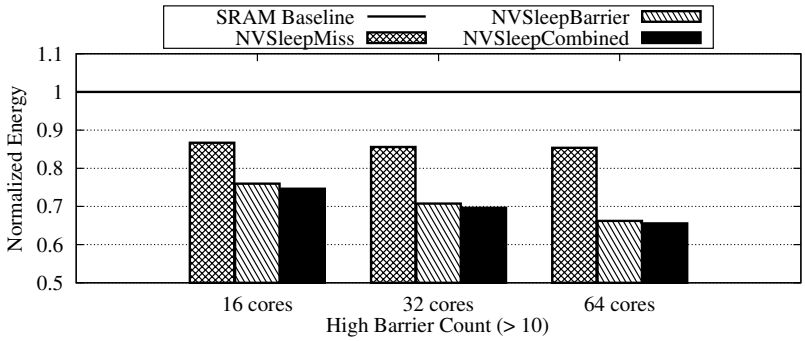


Figure 3.13: NVSleep energy savings in CMPs with 16, 32, and 64 cores.

29.3% on the 32-core system, and by 33.8% on the 64-core system. Energy savings from NVSleepMiss are only marginally higher with increasing number of cores.

3.6 Conclusion

Non-volatile memory can be used effectively to implement rapid checkpoint/wakeup of idle cores. This work has explored a framework for implementing rapid checkpoint using STT-RAM and two applications of that framework: NVSleepMiss and NVSleepBarrier. Evaluation showed average energy savings of 21% for NVSleepMiss in single-threaded applications and 34% for NVSleepBarrier in high barrier count multi-threaded workloads, both with very small performance overhead. In future work we will explore other opportunities for shutting down cores when idle such as spinning due to lock contention or other synchronization events.

CHAPTER 4

Respin: Rethinking Near-Threshold Multiprocessor Design with Non-Volatile Memory

4.1 Introduction

Power consumption is now a primary constraint in microprocessor design spanning the entire spectrum of computing devices. Steady increase in the number of cores coupled with the growing inability to simultaneously activate most units of the chip prompted many to predict the end of traditional multicore scaling [18]. Such predictions emphasize the need to explore energy-efficient architectures that can continue to leverage advancements in process technology. Near-threshold (NT) computing [17,46] has emerged as a potential solution for continuing to scale future processors to hundreds of cores. Near-threshold operation involves lowering the chip's supply voltage (V_{dd}) close to the transistor threshold voltage (V_{th}). Although this approach results in a 10× slowdown in chip speed, power consumption is lowered by 100×, potentially resulting in a full order of magnitude in energy savings. Unfortunately, near-threshold computing suffers from a number of drawbacks. These include decreased reliability, increased sensitivity to process variation, and higher relative leakage power [17,46,51].

Although near-threshold operations dramatically reduce power consumption, the contribution of dynamic and leakage components to the overall savings is not evenly distributed. While dynamic power reduction is cubic as a function of V_{dd} and frequency, leakage power only scales linearly. Caches are leakage dominated structures [43] that can account for 20% to 40% of the overall chip's power consumption depending on their size. Figure 4.1 shows the breakdown of leakage and dynamic power within a 64-core CMP at both nominal and NT V_{dd}. We observe that at a nominal V_{dd} of 1.0V, 14% of the total CMP power is attributed to cache leakage and another 14% to cache dynamic power. Overall, dynamic power represents 60% of the total CMP power consumption. However, when the same CMP operates in the near-threshold range, with a core V_{dd} of 400mV and a cache V_{dd} of 650mV, leakage power dominates, accounting for 75% of the total CMP power consumption. Close to half that leakage power is consumed by caches. While these numbers vary as a function of cache size, voltage and other factors, we find that reducing cache leakage will result in significant power savings at near-threshold voltages.

SRAM-based caches are generally the most vulnerable structure within the chip and are especially sensitive to low voltage. They are optimized for density and therefore rely on the smallest transistor design available for a given technology. While this approach enables larger cache capacities, it has the adverse effect of making such units particularly vulnerable when operating at low voltages [82]. Process variation effects become increasingly pronounced as a function of V_{dd} reduction [50]. Consequently, this creates imbalances in the SRAM cells where a variety of failures can occur including timing and data retention errors. Such error rates are exacerbated in the near-threshold range, significantly compromising the ability of caches to reliably

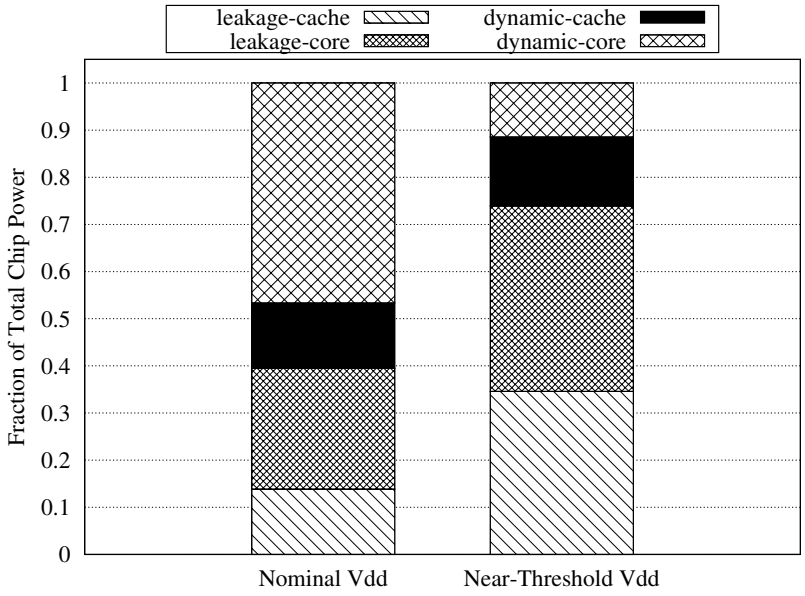


Figure 4.1: Dynamic and leakage power breakdown for a 64-core CMP at nominal and near-threshold voltages.

store data. Although a large body of error correction techniques have been proposed to deal with the high error rates in SRAM at low voltages [11, 51], the overhead associated with such approaches in the near-threshold range makes them inefficient.

This work proposes a near-threshold chip multiprocessor design that uses Spin-Transfer Torque Random Access Memory (STT-RAM) to consolidate on-chip caches. We find STT-RAM to be an attractive candidate for implementing near-threshold systems for several reasons including low leakage, high density, and non-volatility [26, 38, 60, 72]. At one eighth the leakage of SRAM designs, STT-RAM based caches can operate at higher voltages and still save energy as a result of non-volatility. Raising the supply voltage has the advantage of alleviating the reliability concerns typically associated with low V_{dd}. Moreover, the inherently high write latencies of STT-RAM cells can be efficiently tolerated due to the low clock speeds at which

near-threshold cores execute. This obviates the need for large SRAM buffers to mitigate performance bottlenecks caused by slow STT-RAM write speed [26, 74]. Moreover, unlike phase-change memory (PCM) and NAND flash memories [78], STT-RAM enjoys near-unlimited write endurance [72].

Based on these insights, we design a near-threshold chip multiprocessor (CMP) that utilizes dual voltage rails that can power processor cores and caches separately. We allocate a V_{dd} rail in the near-threshold range to the processing cores since they can operate at low frequencies. A second high V_{dd} supply rail is dedicated to the STT-RAM cache. This improves cache write latency relative to the cores. Furthermore, with this approach cache read latencies are substantially faster than the cycle time of the NT cores. This allows L1 caches to be shared by clusters of multiple cores, eliminating the need for cache coherence within the cluster. We show that this improves both latency and energy relative to traditional private cache designs. We redesign the shared cache controller to time-multiplex requests from different cores. The cluster size is chosen such that the vast majority of the read requests are serviced within a single core cycle to ensure no degradation in cache access latency.

The shared L1 cache enables another key feature of our CMP design. Since the L1 is shared by all cores within a cluster, migrating threads from one core to another has very low overhead compared to private cache designs because cached data is not lost in the migration. We take advantage of this feature to further reduce energy consumption with a dynamic core consolidation mechanism. The technique dynamically co-locates threads on the most energy efficient cores shutting down the less efficient ones depending on the characteristics of the workload. A runtime mechanism uses a greedy optimization that dynamically chooses the active core count which minimizes

energy consumption. The motivation behind core consolidation is two-fold: (1) NTV cores have high leakage and powering some of them off can sometimes lead to net energy gains and (2) applications have low-IPC phases during which multiple threads can be consolidated on a single core with small impact on performance.

Evaluation using SPLASH2 and PARSEC benchmarks shows 11% performance improvement with the shared cache design and 33% combined energy savings with the dynamic core consolidation optimization enabled.

Overall, this work makes the following contributions:

- Proposes STT-RAM as a great candidate for saving leakage and improving performance in near-threshold chips. To the best of our knowledge, this is the first work to use non-volatile caches in near-threshold chip multiprocessors.
- Introduces a novel process variation aware shared cache controller design that efficiently accommodates requests from cores running at different frequencies.
- Presents a low overhead dynamic core consolidation system that transparently virtualizes hardware resources to save energy.

4.2 Near-Threshold CMP With STT-RAM Caches

We design an NT CMP that uses STT-RAM for all on-chip caches. Figure 4.2 illustrates the chip's floorplan. The CMP is organized in clusters within which all cores share single L1 and L2 caches. The clusters themselves share the last-level cache (L3). The CMP makes use of two externally regulated voltage domains. One domain, which contains the core logic is set to low NT V_{dd}. The second, which encompasses the entire STT-RAM cache hierarchy and a few logic units, runs at high nominal V_{dd}.

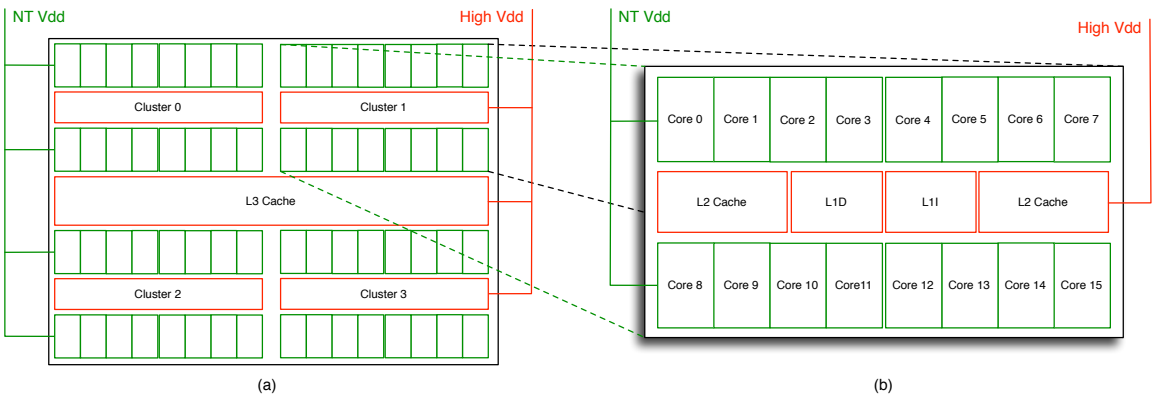


Figure 4.2: Floorplan of the clustered 64-core CMP design (a) with details of the cluster layout (b).

Note that two voltage domains are generally needed for SRAM-based NTV systems also because SRAM requires a higher voltage to operate reliably.

Running the STT-RAM caches at nominal V_{dd} dramatically improves write speed relative to the NT cores, reducing write latency from 10 cycles to about 3 cycles for a core running at 500MHz. Level-shifters [20] are needed for all cross voltage domain up-shift transitions (from low to high voltage domain). The delay overheads introduced by these circuits are compensated by the speed gain in the units running at higher voltages. We account for the level-shifting delay and power overhead in our evaluation.

An additional benefit of the high voltage cache is that read accesses are very fast relative to the core speeds. For example, a 256KB STT-RAM L1 cache has a read speed around 0.4ns (in line with data reported by recent work [26, 38, 72]). The level shifters needed to access the high-V_{dd} shared cache add some delay overhead (0.75ns according to [20]). This overhead is incurred only when the voltage is upshifted

from the NT V_{dd} of the cores to the high V_{dd} of the cache. Even with the level-shifting overhead (which can be pipelined at the cache side), the cache response time is significantly faster than the cycle time of the NT cores (ranging between 1.6ns and 2.4ns).

We exploit the fast read speeds and share a single L1 instruction, L1 data, and L2 cache among all the cores for each cluster. This is accomplished by running the shared L1 cache at a high frequency (2.5GHz in our experiments to match the 0.4ns access time) and time-multiplexing requests from the cores in each cluster. The main advantage of the shared cache design is that coherence is no longer necessary within each cluster. This greatly reduces the latency cost of sharing data between threads that are executing on cores in the same cluster. It also reduces coherence traffic, design complexity, and energy cost.

The large core-to-core variation associated with NT operation [50, 53] makes the approach of limiting the entire CMP to match the frequency of the slowest core very inefficient. Since fast cores are almost twice as fast as slow ones, we allow the respective cores across the CMP to run at the highest frequencies they can achieve. To keep the design cost effective, each cluster uses a single PLL for generating its base clock. The reference clock that feeds this PLL is based on the maximum frequency of the cache (e.g. 2.5GHz corresponding to 0.4ns). The cores run at integer multiples of the reference clock (e.g. 1.6ns, 2.0ns, 2.4ns) generated through clock multipliers. As a result, all cache access requests will align at cycle boundaries with the cache's reference clock, enabling the cache controller to efficiently arbitrate between requests from different cores.

4.2.1 Time-Multiplexing Cache Accesses

The shared cache controller handles multiple parallel requests from different cores using a form of time multiplexing. The primary goal of the cache controller is to return read hit requests to individual cores within a single core cycle. Since cores have cycle times, slower cores have more time slacks to have their requests serviced compared to faster cores. As a result, requests arriving at the same time are ordered based on the frequency of the requesting cores. Higher frequency cores are serviced first, while requests from slower cores receive lower priority. If the cache bandwidth is exceeded and a hit request cannot be serviced in time, a "half-miss" response is sent to the core and the request is serviced in the following cycle. In our evaluation, only about 4% of cache acesses result in half-misses.

Figure 4.3 shows an example of how multiple access requests from cores that are using different clock periods (1.6ns-2.4ns) are handled by a shared cache operating at 2.5GHz (0.4ns clock period). A cycle-by-cycle timeline of such requests is outlined in Figure 4.3 (a). In this figure, each request is associated with a line segment that represents the cycle time of the original core that issued it. This is a multiple of the reference clock that is used by the cache. For instance, Core 0 is running at 625MHz, which means its cycle time of 1.6ns is equal to 4 cache cycles. Since Core 0's request is received in cycle 0, to ensure that the cache responds within a single core clock cycle, the cache must send the data (or miss signal) by the end of cycle 3. Each core's request takes 2 fast cache cycles (0.8ns) to arrive at the cache due to wire and level-shifting overhead.

To keep track of all requests, the cache controller maintains a request register and a priority register for each core in the cluster. The request register stores the

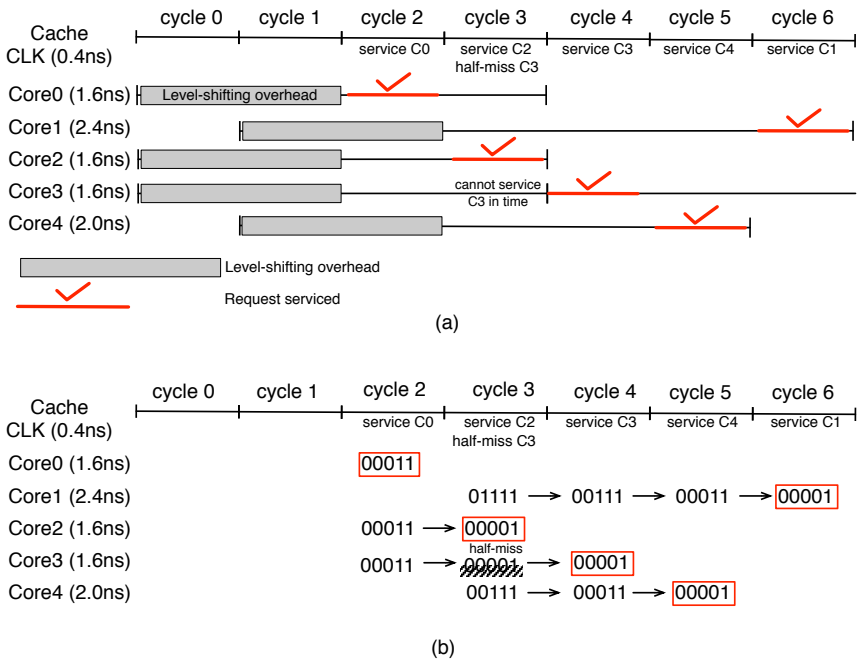


Figure 4.3: Example timeline of access requests from cores running at different frequencies to a shared cache running at high frequency.

requested address, type, and the data for the read requests. The priority registers are shift registers preloaded with a representation of the number of fast cache cycles available for each request. Figure 4.3 (b) shows a view of the priority registers for the same example. For instance, for Core 0, which needs to be serviced in two cache cycles, the request register is preloaded with "00011". Note that the cycles required to service each request account for the level shifting overhead. In other words, even though Core 0's request needs to be serviced in 1.6ns or four cache cycles, two of those are spent in the level shifters and wires. The remaining two are recorded in the priority register. For Core 1's request, which needs to be serviced in four cache cycles, the register is preloaded with "01111". All priority registers are right-shifted

by one position each cache cycle to indicate a reduction in the available time for all unserviced requests.

At the end of cycle 2, the cache has three requests from Core 0, Core 2, and Core 3, out of which the cache can only service one. The cache controller picks the request that expires the soonest (i.e. the one with the fewer "1" bits) using simple selection logic. In this example all three requests have equal priority so the cache randomly chooses to service Core 0. This is indicated by the red "checkmarks" in Figure 4.3 (a) and the red rectangles in Figure 4.3 (b). The priority register corresponding to Core 0 is cleared and becomes available for a new request in the following cycle.

In cycle 3, the requests from Core 2 and Core 3 are both critical, meaning they have to be serviced in the current cycle (priority register is "00001"). Since the cache can only service one request it will choose Core 2's. A "half-miss" event will be sent to Core 3 to indicate that the request could not be fulfilled in a single cycle, but this is not necessarily an L1 miss. Core 3's request will be rescheduled through a reinitialization of the priority register. To increase its priority the register will be initialized to a lower value (in this example "00001"). Core 3's request will be serviced in cycle 4, which corresponds to a 2-cycle total hit latency. Requests from Core 4 and Core 1, issued in cycle 1 will be serviced in their priority order in cycles 5 and 6 respectively.

4.3 Dynamic Core Management

The shared L1 cache design significantly reduces the performance overhead of migrating threads within the same cluster. This is because no cache data is lost

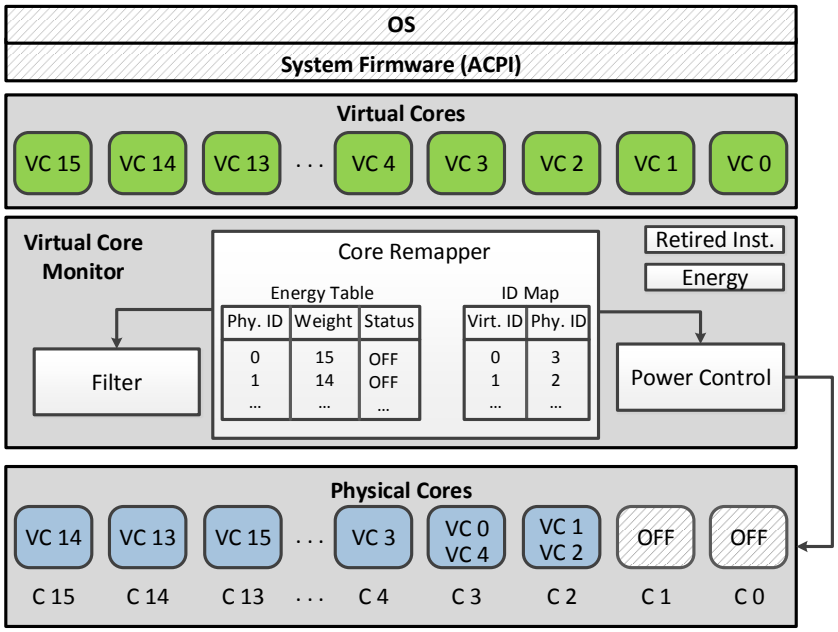


Figure 4.4: Overview of the virtual core management system integrated in one cluster.

after the migration. We take advantage of this feature to further reduce energy consumption with a dynamic core consolidation mechanism. The motivation behind core consolidation stems from the fact that NT cores exhibit high variability in maximum operating frequency. They also have a high ratio of leakage to dynamic power. As a result, cores that achieve a higher frequency at the same voltage are more energy efficient that then the low-frequency ones. In some situations it is therefore more energy efficient to power off the least efficient cores and consolidate their threads to the more efficient ones. This is generally true in low-IPC phases.

4.3.1 Core Virtualization

We find that low-IPC execution phases are relatively short and therefore taking advantage of them requires a low overhead, fast reacting mechanism for migrating

threads and shutting down cores. We present a new hardware management mechanism that dynamically consolidates cores through a virtualization extension. The proposed system takes advantage of shared resources to transparently remap running applications across a set of heterogeneous cores.

Implementing this management system in hardware as opposed to the OS enables faster response times and lower performance overhead. In addition, the hardware-based core consolidation system is transparent to the OS and does not require OS intervention or support. This makes the solution easily deployable and backward compatible irrespective of the underlying hardware differences. Figure 4.4 depicts an overview of how our core management system would be integrated into a chip multiprocessor.

A key feature of our design is the ability to autonomously and transparently migrate threads to different physical cores without OS intervention. To that end our system makes use of virtual cores that provide a homogeneous view of processor resources to the OS. The virtual resources are made visible to the OS via the Advanced Configuration and Power Interface (ACPI) available within system firmware.

The core consolidation mechanism dynamically shuts down physical cores following an energy optimization algorithm. However, from the OS point of view, all virtual cores are always available. If some physical cores are off, a core mapping mechanism assigns multiple virtual cores to a single physical core.

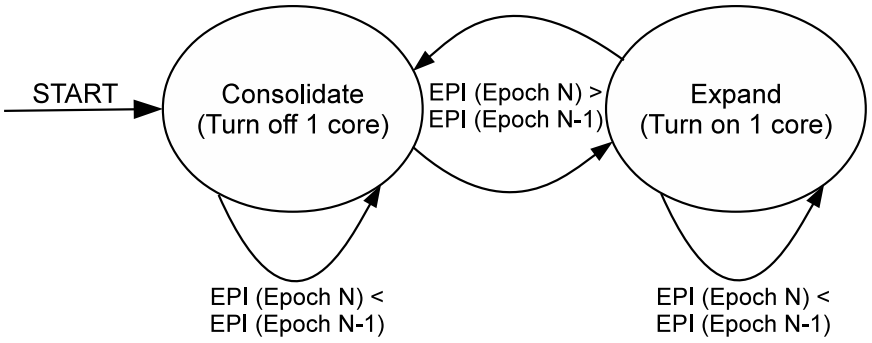


Figure 4.5: Greedy selection for dynamic core consolidation.

4.3.2 Energy Optimization Algorithm

An energy monitoring and optimization system is implemented in firmware running on a dedicated on-chip microcontroller that is deployed in many of today's processors [33] for energy management.

A virtual core monitor (VCM) block, shown in Figure 4.4 is responsible for monitoring the energy per instruction (EPI) for each virtual core using hardware performance counters. The VCM also runs the energy optimization algorithm designed to dynamically search for the optimal number of active cores.

A simple greedy search algorithm (illustrated in Figure 4.5) guides the energy optimization. Execution is broken down into multiple epochs. At the end of each epoch the algorithm decides whether a physical core should be shut down, turned on, or if nothing needs to change. The EPI of the current epoch is compared to that of the previous one. If the difference exceeds a predefined threshold, then physical cores are either turned off or on.

The system starts with all physical cores on for an entire epoch. At the end of the first epoch, one physical core is shut down and its virtual core migrated to another

core. The new EPI is measured at the end of the epoch. If energy is lower, the greedy search continues by progressively shutting down additional cores. If energy is higher, the search reverses direction. If EPI difference between the current and previous epoch is lower than the threshold, the current state is maintained for the next epoch. This is done to avoid excessive state changes for minor energy benefits.

In addition, the algorithm applies an exponential back-off to eliminate unnecessary oscillations between neighboring states. The history of recent state changes within each cluster is recorded. If the system detects an oscillating pattern, it exponentially increases the number of epochs during which it will hold the current state before attempting a state change (e.g. 2, 4, 8, 16, and 32 epochs).

4.3.3 Virtual Core Consolidation

Core consolidation within a cluster is handled by the core remapper module depicted in Figure 4.4. Whenever a power down/up event is required, the remapper examines the pool of active physical cores. An energy efficiency score is precomputed and recorded in a table. The score is determined based on the frequency of the core. Faster cores are more energy efficient because they can achieve a lower energy per instruction at the same voltage than lower frequency cores. The primary reason is that the high leakage power that dominates NT cores is a fixed cost independent of frequency. Using the energy profile, the system will turn off the least efficient active core, or turn on the most efficient inactive core as dictated by the greedy search.

Once a core is marked for deconfiguration, the remapper assigns one of the remaining active physical cores as a host for the unassigned virtual core. To keep the design simple, allocations to active physical cores are performed in a round robin

fashion. We start allocations with the most efficient core (fastest) and move down to the least efficient one (slowest). This means that multiple virtual cores are more likely to be consolidated on the faster physical cores, thus alleviating the performance impact of consolidation.

The migration of the virtual core follows two main phases. In the first phase, the deconfigured core stops fetching new instructions and saves the next PC into a consolidation register. The core continues to execute instructions until all in-flight instructions are committed. The register file content is then saved. In the second phase, the target physical core is interrupted and the register file image and the PC are transferred. Execution resumes on the new core. Once the remapping is complete, the virtual-to-physical ID map is updated accordingly to reflect the new association. A request is then issued to the power control module to power gate the deconfigured core. A similar migration process is followed when a new physical core is activated and a virtual core is migrated to it.

If multiple virtual cores are mapped to a single physical resource, hardware-based context switches are performed at regular intervals that are much smaller than the typical OS context-switch interval. This ensures fairness and uniform progress of the virtual cores such that they all appear to be running simultaneously.

4.3.4 Mitigating Core Consolidation Overhead

There are a few sources of potential performance overhead associated with our core consolidation mechanism. The biggest potential cost for our remapping scheme is the loss of data stored in local caches. If remapping is frequent, "cold-cache" effects can severely degrade performance. In our CMP design, we restrict the remapping of

virtual cores to occur only within clusters. This means that application level threads that are associated with virtual cores don't lose any data locality since the entire cache hierarchy is shared at the cluster level.

Another source of overhead is the loss of architectural state associated with each individual thread including branch prediction history and on-chip data stored in register files or reorder buffers. After every consolidation the architectural information of each newly remapped thread is lost. It takes tens of cycles to rebuild those states before the thread can perform any useful work. Therefore if remapping occurs too frequently, the overall performance can suffer. We address this issue by carefully choosing a reasonable consolidation interval. With experiments we find that remapping performed every 160K instructions carries only a small performance penalty and returns optimal energy savings.

Finally, another potential source of overhead is related to the action of powering on cores. After a core is turned on from a power-gated state, voltage noise can cause timing errors [52]. To prevent that, the core is stalled for a brief period of time. However, because the cores run at NT voltage and their power is relatively low compared to their available capacitance the noise is small. As a result the penalty for voltage stabilization is only about 10-30ns [50] or 5-15 cycles for a core running at 500MHz.

All types of overheads discussed above are properly reflected in our design and included in the evaluation results shown in Section 4.5.

Hierarchy	Size	Block Size	Assoc.	Rd/Wr Ports
L1I (Private/ Shared w/i Cluster)	16KB (Private)/ 256KB (Shared w/i Cluster)	32B	2-way	1/1
L1D (Private/ Shared w/i Cluster)	16KB (Private)/ 256KB (Shared w/i Cluster)	32B	4-way	1/1
L2 (Shared w/i Cluster)	8MB (Small)/ 16MB (Medium)/ 32MB (Large)	64B	8-way	1/1
L3 (Shared w/i Chip)	24MB (Small)/ 48MB (Medium)/ 96MB (Large)	128B	16-way	1/1

Table 4.1: Summary of cache configurations.

4.4 Evaluation Methodology

We modeled a 64-core CMP with a range of cluster sizes from 4 to 32 cores. Most experiments were conducted with a cluster size of 16 cores, which we found to be optimal. We also experimented with three cache configurations: small, medium, and large. The size of the caches were chosen to provide between 1MB (small) and 4MB (large) of cache for each core, in line with existing commercial designs [33, 64]. Also in line with existing designs, our medium cache configuration accounts for approximately 25% of the total chip area. In the large configuration the total cache area represents 50% of the chip area. Most of our results are reported for the medium cache configuration. The small and large are included for reference and trend analysis. Table 4.1 summarizes our cache configurations at different levels.

In our experiments each core has a dual-issue out-of-order architecture. We used SESC [63] to perform all of our simulations. We collected runtime, power, and energy

CMP Architecture	
Cores	64 out-of-order
Fetch/Issue/Commit Width	2/2/2
Register File Size	76 int, 56 fp
Instruction Window Size	56 int, 24 fp
Reorder Buffer Size	80 entries
Load/Store Queue Size	38 entries
NoC Interconnect	2D Torus
Coherence Protocol	MESI
Consistency Model	Release Consistency
Technology	22nm
NT-V_{dd}	0.4V (Core), 0.65V (Cache)
Nominal-V_{dd}	1.0V
Core Frequency Range	375MHz-725MHz
Median Core Frequency	500MHz
Variation Parameters	
V_{th} std. dev./mean (σ/μ)	12% (chip), 10% (cluster)

Table 4.2: CMP architecture parameters.

information. Table 4.2 summarizes the baseline architecture configuration parameters. NVSim [16] combined with CACTI [57] was used to obtain STT-RAM latency, energy, and area. Similarly, per access energy for all SRAM memory structures including register file, reorder buffer, load/store queue, and instruction window were extracted through CACTI. McPAT [43] was used to model energy per access for all CMOS logic units such as ALUs and FPUs. We included a model for leakage power based on estimated unit area and technology (CMOS vs. MTJ). This information was inserted into SESC's activity model in order to obtain total power and energy consumption. Table 4.3 lists the technology parameters we obtained from NVSim and CACTI for various types of L1 data caches. The cache areas reported take into account the higher density of STT-RAM compared to SRAM. We rounded STT-RAM

	V_{dd} Rail	Area (mm^2)	Rd/Wr Lat. (ps)	Rd/Wr Eng. (pJ)	Leakage (mW)
SRAM (16KB×16)	Low (0.65V)	0.9176	1337	2.578	573
SRAM (16KB×16)	High (1.0V)	0.9176	211.90	6.102	881
SRAM (256KB)	High (1.0V)	0.9176	533.60	42.41	881
STT-RAM (256KB)	High (1.0V)	0.2451	388.20/ 5208	29.32/ 209.30	114

Table 4.3: L1 data cache technology parameters.

cache read latency up to 0.4ns to align clock edges between the shared cache and cores. Parameters of other cache hierarchies are similarly simulated and properly fed into our architecture simulations.

Two benchmark suites were adopted in the evaluation: SPLASH2 and PARSEC. SPLASH2 (*barnes, cholesky, fft, lu, ocean, radiosity, radix, raytrace,* and *water-nsquared*) was configured to run with reference input sets. PARSEC (*blackscholes, bodytrack, streamcluster,* and *swaptions*), on the other hand, was launched with sim-small input sets. We used VARIUS [68] to model variation effects on threshold voltages (V_{th}) across the CMP. We generated distributions of core frequencies that were used in the simulations.

4.5 Evaluation

In this section we show performance and energy benefits of the proposed architecture. We also include sensitivity studies on optimal cluster size, shared cache behavior, and dynamic core consolidation mechanism. For easy reference, Table 4.4 summarizes all the architecture configurations used in our evaluation.

Configuration & Description	
PR-SRAM-NT (baseline)	NT chip with SRAM private L1(I/D) cache and shared L2/L3 cache
HP-SRAM-CMP (alt. baseline)	Traditional high-performance CMP with cores and caches at nominal V_{dd}
SH-SRAM-Nom	NT core with nominal V_{dd} SRAM shared L1(I/D) cache and shared L2/L3 cache
SH-STT	SH-SRAM-Nom with all caches built in STT-RAM
SH-STT-CC	SH-STT that performs hardware-managed dynamic core consolidation
SH-STT-CC-Oracle	SH-STT-CC with oracle knowledge for dynamic core consolidation
PR-STT-CC	SH-STT-CC with private L1(I/D) cache
SH-STT-CC-OS	SH-STT-CC with OS-managed dynamic core consolidation

Table 4.4: Architecture configurations used in the evaluation.

4.5.1 Power Analysis

Figure 4.6 shows the reduction in power consumption from the proposed STT-RAM-based CMP architecture without dynamic core consolidation (SH-STT). Since the power savings we obtain are dependent on the size of the cache, we show results for three cache configurations (Table 4.1): small, medium, and large. The medium size cache is the most typical one, with about 2MB/core of total cache capacity. In this configuration, the cache accounts for approximately 25% of the chip area.

We compare to a baseline that uses SRAM caches running at a low voltage rail (0.65V) in a traditional private cache hierarchy (PR-SRAM-NT). This is the most typical near-threshold CMP design. The reason why SRAM caches run at a higher voltage rail is to ensure acceptable reliability since SRAM caches running at NT V_{dd}

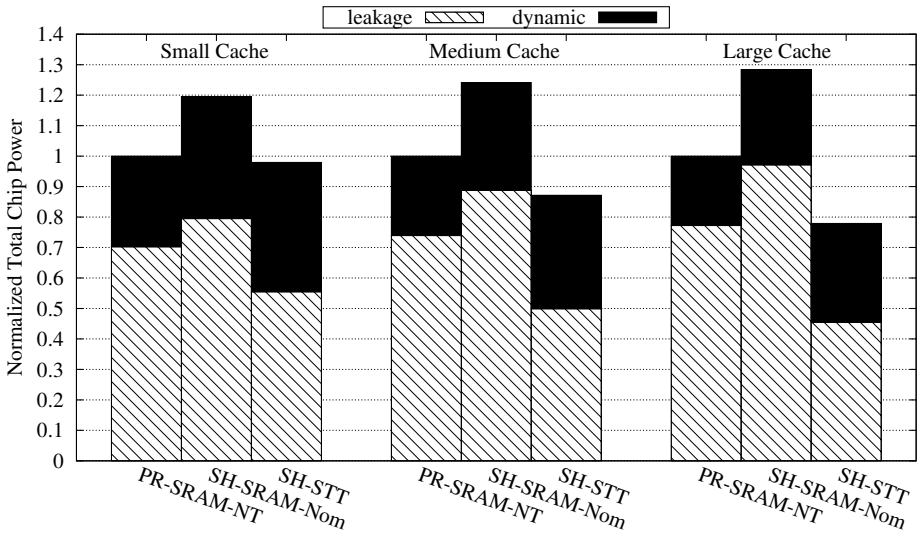


Figure 4.6: Power reduction of proposed design for three L2/L3 cache sizes: small, medium, and large.

would be unusable without cell resizing or strong error correction [21, 51] – both of which carry significant overheads.

We can see that power is lower for SH-STT compared to the baseline in all configurations. The reduction in total power comes from lower leakage power at the cost of slightly increased dynamic power (due to nominal voltage STT-RAM reads and the high cost of STT-RAM writes). For the small cache configuration the power is only about 2.1% lower. For the medium and large configurations the power savings are significant, at 12.9% and 22.1% respectively.

Figure 4.6 also shows a breakdown of leakage and dynamic power for each configuration. We can see that for STT-RAM, even though dynamic power is higher due to the nominal voltage cache operations, the reduction in leakage power compensates for it in all three cache size configurations.

For reference, we also compare to an SRAM design in which the cache is shared and also running at nominal voltage (SH-SRAM-Nom), the same configuration used by our proposed design but with SRAM caches. This ensures reliable operation but is costly in terms of power. SH-SRAM-Nom uses between 22% and 65% more power than SH-STT for the three cache sizes. This is due to the much higher leakage power consumed by SRAM running at nominal voltage.

4.5.2 Performance Analysis

The shared cache design brings significant performance improvements compared to the baseline system. Figure 4.7 shows the execution time of the proposed STT-RAM design (SH-STT) with medium-sized cache. The results are normalized to the PR-SRAM-NT baseline. Process variation effects (core frequency distributions) are modeled in all configurations. We can see that the SH-STT configuration reduces execution time by an average of 11%. This performance improvement is due to the benefits of within-cluster cache sharing. Applications that benefit the most are those in which there is significant data sharing and reuse such as *raytrace*. Applications such as *ocean* also benefit significantly because they make heavy use of synchronization (*ocean* has hundreds of barriers). Synchronization is much faster in the shared cache design because it involves much less coherence traffic.

We also compare SH-STT to SH-SRAM-Nom (as before) and we add another baseline, HP-SRAM-CMP. HP-SRAM-CMP represents a conventional high-performance design in which the entire CMP (cores plus caches) run at nominal voltage. Figure 4.7 shows that compared to SH-SRAM-Nom our proposed SH-STT design achieves marginally better performance (1.2% on average) because of slightly faster read speed

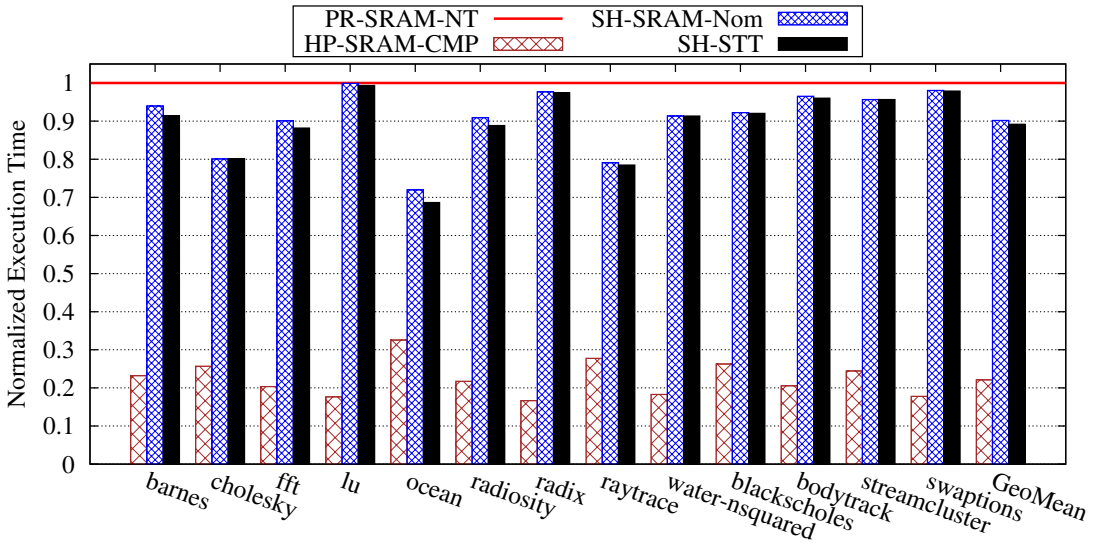


Figure 4.7: Relative runtime of SPLASH2 and PARSEC benchmarks for various designs with medium-sized cache.

of STT-RAM compared to SRAM. The high-performance HP-SRAM-CMP achieves the lowest execution time because it runs at high voltage and high frequency. This performance, however, comes at a much higher energy cost.

4.5.3 Energy Analysis

Our design reduces both power consumption and execution time resulting in important energy savings. Figure 4.8 shows that SH-STT has between 13% and 31% lower energy than PR-SRAM-NT baseline depending on cache sizes. As expected we see larger energy savings for larger cache sizes. We also show that the SH-SRAM-Nom configuration which uses shared SRAM caches at nominal V_{dd} uses 8-16% more energy than the NT SRAM baseline (PR-SRAM-NT).

Figure 4.9 shows the energy breakdown by benchmark for our designs with the medium-sized cache relative to the PR-SRAM-NT baseline. The shared STT-RAM

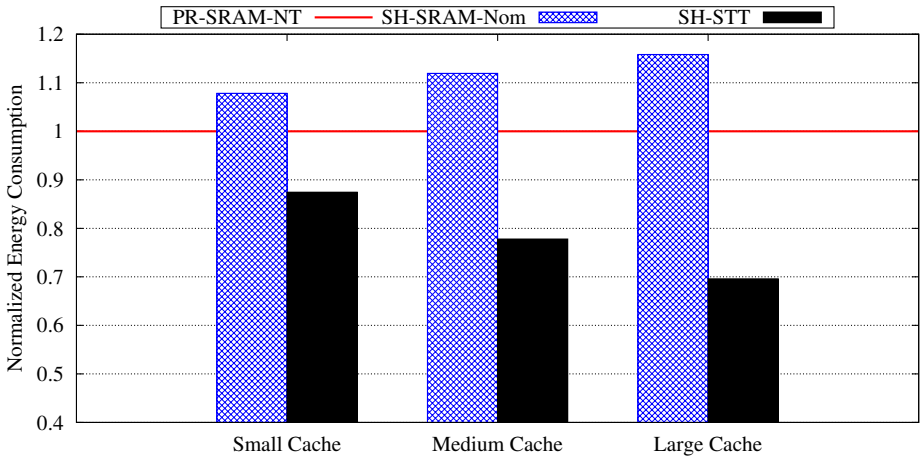


Figure 4.8: Energy consumption for small, medium, and large L2 and L3 cache configurations.

cache design (SH-STT) reduces energy by an average of 23%. This is in stark contrast with a similar shared cache configuration that uses SRAM at nominal V_{dd} (SH-SRAM-Nom), which increases energy by 12%. The high-performance baseline HP-SRAM-CMP consumes 40% more energy on average than the PR-SRAM-NT baseline. Relative to HP-SRAM-CMP, our SH-STT design has an average of 45% lower energy consumption. When we add dynamic core consolidation (SH-STT-CC), we reduce energy by an additional 10% for a combined 33% reduction relative to PR-SRAM-NT (51% reduction relative to HP-SRAM-CMP).

We also include an oracle version of the dynamic core consolidation solution (SH-STT-CC-Oracle) to show the limits of our greedy-search-based energy optimization. We obtained SH-STT-CC-Oracle by choosing the optimal number of cores to consolidate at each evaluation interval. SH-STT-CC-Oracle reduces energy consumption by 36%. The small 3% difference between the Oracle and our implementation is due

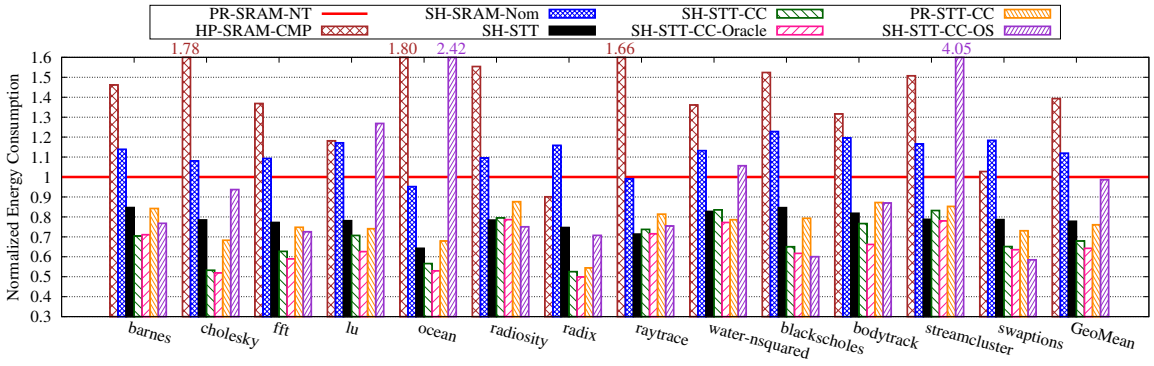


Figure 4.9: Energy consumption for SPLASH2 and PARSEC benchmarks with a core consolidation interval of 160K instructions and a medium-sized L2 and L3 cache.

to the slight sub-optimality of the greedy search we perform. Overall it is a small penalty to pay for a fast optimization that can be deployed in production systems.

Figure 4.9 also compares the energy reduction of SH-STT-CC relative to other possible alternatives for implementing core consolidation. PR-STT-CC shows the energy of a solution that attempts core consolidation with private STT-RAM caches. Because of the overhead of consolidating cores with private caches (which results in loss of cache locality after consolidation), PR-STT-CC reduces energy by only 24% compared to 33% for SH-STT-CC.

We also compare with an approach in which core consolidation is handled by the OS at coarser time intervals (1ms). SH-STT-CC-OS does not require any hardware support since consolidation is controlled by the OS. However, because consolidated threads are context-switched at coarser intervals, critical threads can easily bottleneck the entire application when they are not running. This hurts performance significantly to the point where energy actually increases by 27% compared to SH-STT.

Cluster Size (#cores)	Shared Cache Size (KB)	Performance Gain (%)
4	64	4.82
8	128	6.29
16	256	10.81
32	512	2.50

Table 4.5: Cluster size impact on performance.

4.5.4 Optimal Cluster Size

A key parameter for our design is the cluster size. We run simulations with cluster sizes of 4, 8, 16, and 32 cores. Table 4.5 summarizes the results. Note that, as we increase the cluster size we also proportionally increase the shared L1 cache size. For the entire CMP, the total core count and the sum of all L1 cache capacities remain constant.

Performance improves in SH-STT when going from 4 to 16 cores per cluster by 5% to 11% compared to PR-SRAM-NT baseline. This is due to the increased opportunity for data sharing and reduced coherence traffic. The downside is increased bandwidth pressure on the shared cache. When the cluster size is increased to 32 cores, performance improvement drops to only 2.5%. The larger cache size (512KB for 32 cores vs. 256KB for 16 cores) has higher access latency and lower bandwidth. At the same time the number of cores goes from 16 to 32, generating a lot more requests and overwhelming the reduced bandwidth. The optimal cluster size for this design is therefore 16 cores.

4.5.5 Shared Cache Impact on Access Latency

The shared cache design cannot guarantee single cycle access to all cache read hits. If requests cannot be serviced in the equivalent of a core cycle, a "half-miss" response is returned to the core. In order to better understand the impact of the shared cache contention on access latency, we conducted two sets of experiments.

The first experiment measures cache utilization by looking at the number of requests arriving at the shared cache each cycle. Figure 4.10 shows percentage of the total cache cycles in which a given number of requests arrive at the shared cache. We count all requests handled by the cache including reads, writes, line fills, etc. We show numbers for five different benchmarks and the arithmetic mean *of all* our benchmarks.

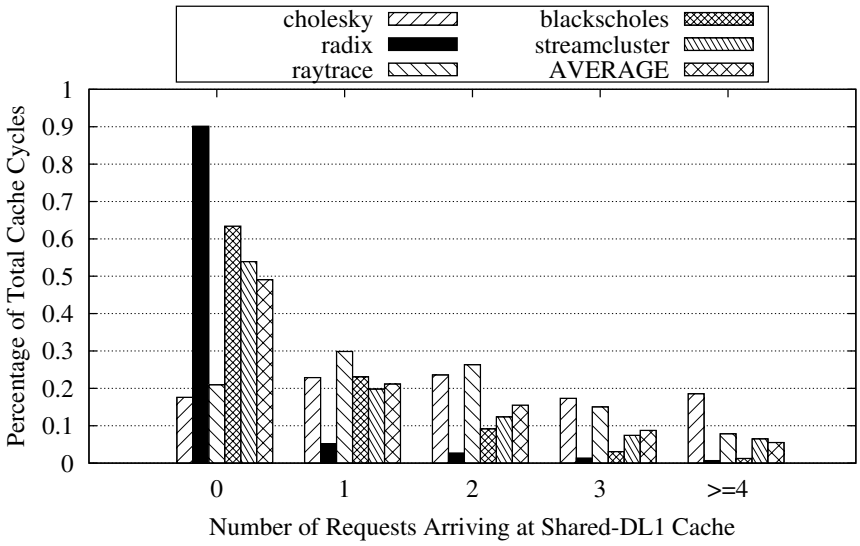


Figure 4.10: Shared DL1 cache utilization rate in one cluster.

We can see that, on average, almost half of the cache cycles (49%) have no incoming requests, 21% with one request, 15% with two requests, 9% with three requests, and 6% with more than four requests. This shows that requests exceeding the number of available ports (1 read/1 write) occur in about 30% of the cache cycles. However, these are fast cache cycles and each requesting core has considerable time slacks in which to receive a response. As a result, most of these requests will not receive a delayed response.

Figure 4.11 shows a histogram of the percentages of read hit requests serviced in 1, 2, or more core cycles. We can see that the vast majority of requests are handled in 1 cycle (95.8%). About 4% of requests result in half-misses and over 99% of those are handled in 2 cycles. As a result, the performance impact of the cache contention is small, and more than compensated by the benefits of the shared cache.

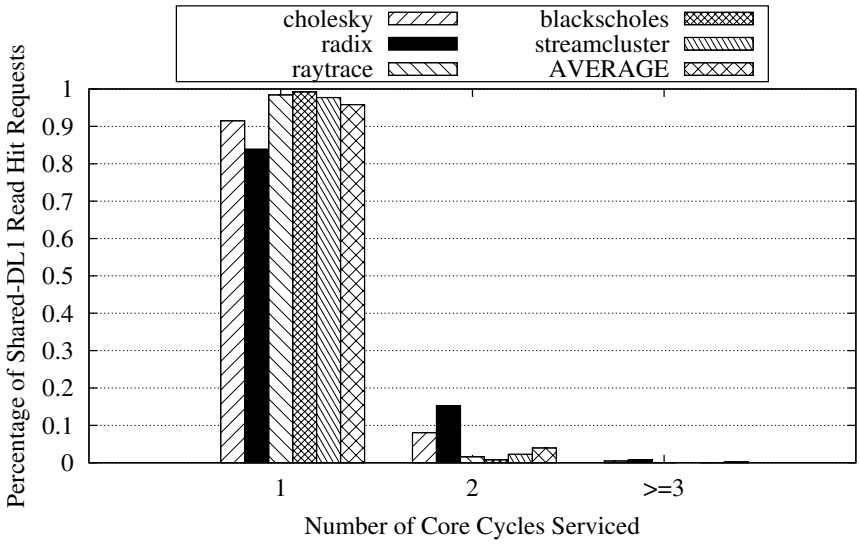


Figure 4.11: Fraction of read hit requests serviced by the shared DL1 cache in 1, 2, or more core cycles.

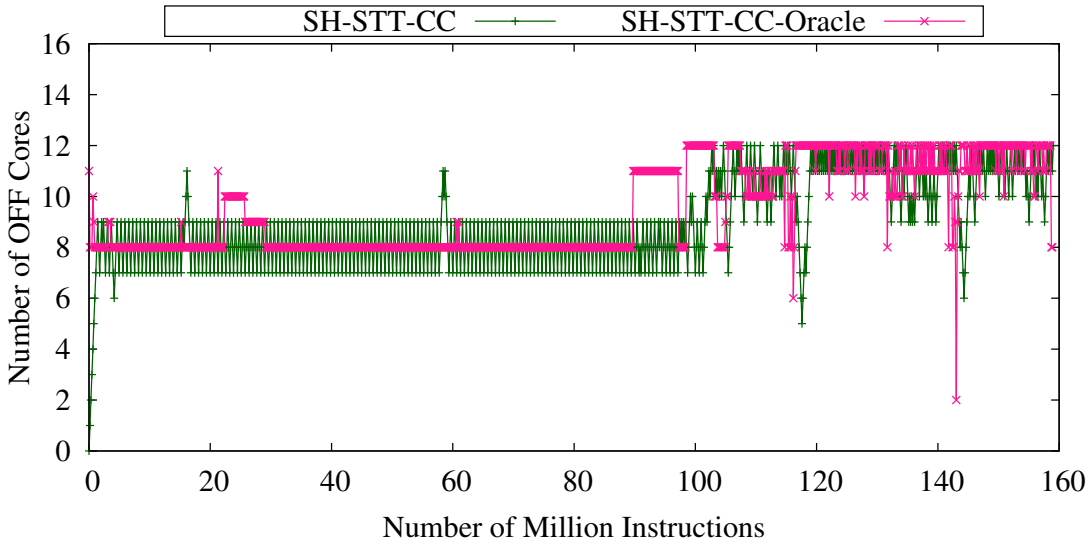


Figure 4.12: Core consolidation trace of *radix*.

4.5.6 Dynamic Core Consolidation

Figure 4.12 shows a detailed runtime trace of *radix* when performing dynamic core consolidation. We show traces for both SH-STT-CC and SH-STT-CC-Oracle to compare the effectiveness of our consolidation mechanism. We can see that except for a few data points, our consolidation trace matches very well with the oracle trace. This leads to very close energy savings for SH-STT-CC (48%) and SH-STT-CC-Oracle (50%) compared to PR-SRAM-NT baseline.

Occasionally the greedy search does not respond sufficiently fast to keep up with workload changes, whereas the oracle adapts immediately. This can be observed in benchmarks such as *lu*, shown in Figure 4.13. The greedy search gradually searches for the optimal energy point, resulting in some temporary sub-optimal behavior. As a result, for the *lu* benchmark, our proposed SH-STT-CC design saves 29% energy while SH-STT-CC-Oracle saves 38%.

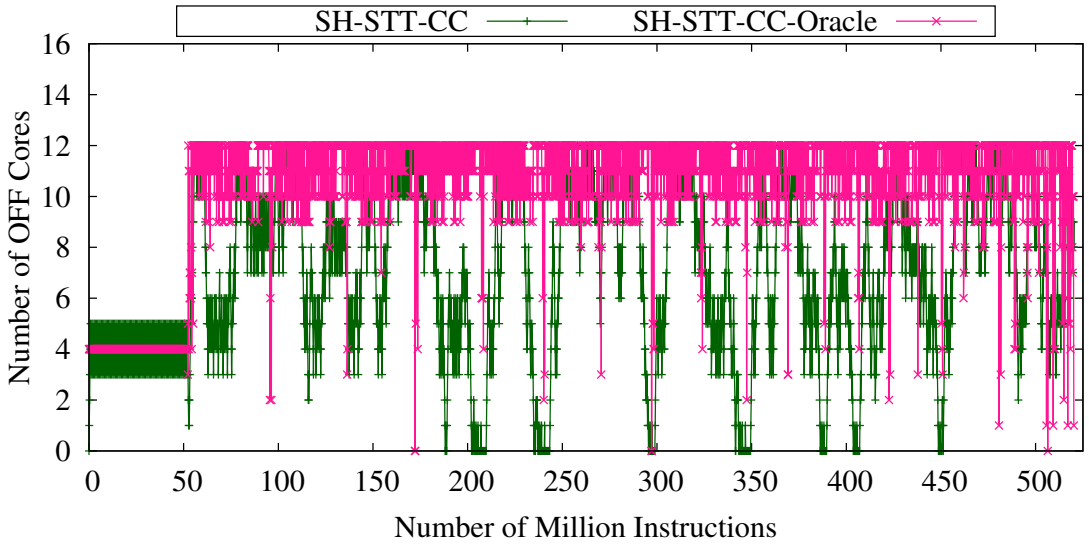


Figure 4.13: Core consolidation trace of *lu*.

Dynamic core consolidation takes advantage of the large variability in application behavior both within and across workloads. To illustrate this, Figure 4.14 shows the average number of active cores in a cluster for each benchmark. We can see that on average only 10 out of 16 cores in a cluster are used. Note that, however, there is high variability in the number of active cores both across and within benchmarks. The markers on each bar indicate the range of active cores throughout the execution. The startup phase of each benchmark is excluded. We can see that for most benchmarks, core consolidation takes advantage of the full dynamic range from 16 to 4 active cores per cluster. Some exceptions include *radix* which only activates 11 cores per cluster at the most and *blackscholes* which never uses fewer than 6 physical cores.

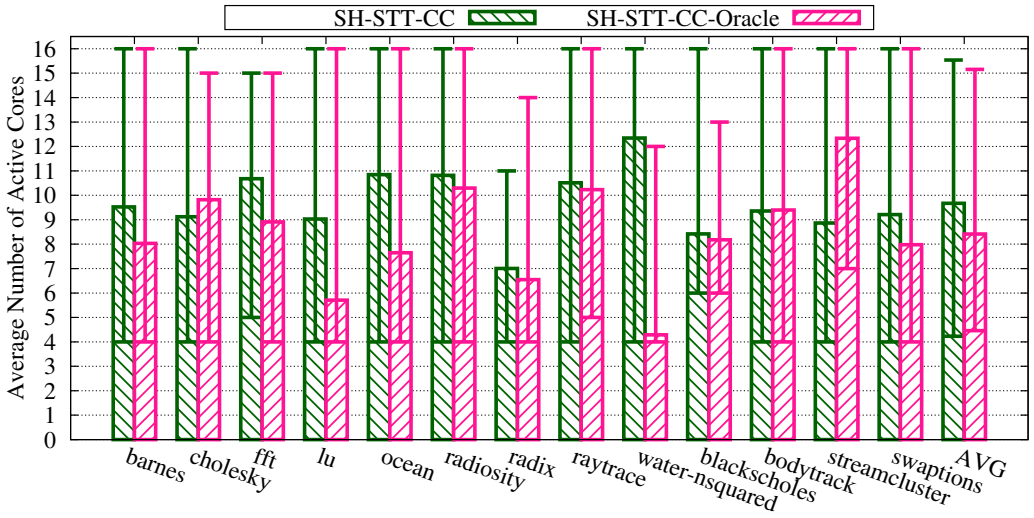


Figure 4.14: Average number of active cores (and min and max values) using core consolidation for SPLASH2 and PARSEC benchmarks.

4.6 Conclusion

This is the first work to explore the use of STT-RAM in near-threshold processors. We find STT-RAM to be an ideal SRAM replacement at near-threshold for two reasons: first, it has very low leakage, which dominates near-threshold designs; second, it can efficiently run at nominal voltages, avoiding the reliability problems of low-V_{dd} SRAM. We show that an architecture designed to exploit STT-RAM properties by sharing the cache hierarchy and implementing dynamic core consolidation mechanism can help lower energy consumption by up to 33% and improve performance by 11%.

CHAPTER 5

NVInsecure: When Non-Volatile Caches Meet Cold Boot Attacks

5.1 Introduction

The characteristics of non-volatile memory including performance, energy, and reliability have already been examined in extensive prior work [22,26,35,38,41,60–62, 72,74,75,77–79,85]. Despite all their expected benefits, non-volatile memories also introduce new security vulnerabilities as data stored in these memories will persist even after being powered-off. In particular, non-volatile memory is especially vulnerable to "cold boot" attacks. Cold boot attacks, as first proposed by Halderman et al. [27], use a cooling agent to lower the temperature of DRAM chips before physically removing them from the targeted system. Electrical characteristics of DRAM capacitors at very low temperatures cause data to persist in the chips for a few minutes even in the absence of power. This allows the attacker to plug the chips into a different machine and scan the memory image in an attempt to extract secret information. When the memory is implemented using NVM, the cold boot attacks become much simpler and more likely to succeed, because of its non-volatile data storage.

Cold boot attacks were primarily demonstrated against DRAM memory chips. To protect against cold boot attacks on main memory, one approach is to apply memory encryption techniques [10, 29, 42, 44, 65, 73, 80, 81] to guarantee sensitive data is always encrypted when stored in the main memory. Another approach is to keep secret keys stored in SRAM-based CPU registers, caches, and other internal storage during system execution [12, 23–25, 54, 55, 59, 71, 84]. The rationale behind this design philosophy is that cold boot attacks against on-chip SRAM structures are deemed to be extremely difficult. This is because SRAM data persistence at cold temperatures is limited to a few milliseconds [4].

However, the security implications of implementing the main memory and microprocessor caches with NVM have received little attention. While memory encryption schemes are feasible, cache encryption is not practical due to low access latency requirements. Cold boot attacks on *unencrypted NVM caches* will be a serious concern in practice, especially in the future with the prevalence of smart mobile devices and Internet of Things (IoT) that are more likely to be exposed to physical tampering—a typical setup for cold boot attacks.

This work examines the security vulnerabilities of microprocessors with NVM caches. In particular, we show that encryption keys can be retrieved from NVM caches if an attacker gains physical access to the device. Since removing the processor from a system no longer erases on-chip memory content, sensitive information can be leaked. This work demonstrates that AES disk encryption keys can be identified in the NVM caches of a ARM-based system running the Ubuntu Linux OS. Since no microprocessors with NVM caches are currently commercially available we rely on a full system simulator [9] for our experiments. This gives us full visibility into

the cache content at any point during the execution and allows us to evaluate the vulnerability of systems with different cache sizes and configurations.

We have examined multiple attack scenarios to evaluate the probability of a successful attack depending on the system activity, attack timing and methodology, etc. In order to search for AES keys in cache images, we adopted the key search algorithm presented in [27] for main memories, and made the necessary modifications to target caches which cover only non-contiguous subsets of the memory space. We find that the probability of identifying an intact AES key if the processor is stopped at any random point during execution ranges between 5% and 100% on average, depending on the workload and cache size. We also demonstrate a reproducible attack with 100% probability of success for the system we study.

To counter such threats, this work proposes an effective software-based countermeasure. We patch the Linux kernel to allocate sensitive information into a designated memory page that we mark as uncacheable in its page table entry (PTE). This way secret information will never be loaded into vulnerable NVM caches but only stored in main memory and/or hard disk which can be encrypted with a reasonable performance cost. The performance overhead of this countermeasure ranges between 2% and 45% on average depending on the hardware configuration.

Overall, this work makes the following main contributions:

- The first work to examine cold boot attacks on non-volatile caches.
- A comprehensive algorithm of searching AES keys in cache images has been developed.
- Two types of cold boot attacks have been performed and shown to be effective on non-volatile caches.

- A software-based countermeasure has been developed and proven to be effective.

5.2 Threat Model

The threat model assumed in this study is consistent with prior work on "cold boot" attacks. In particular, we assume the attacker gains physical access to the target device (e.g. an IoT device, a smartphone, a laptop, or a desktop computer). Further, the attacker is assumed to have the ability to extract the microprocessor or system motherboard from the device and install them into a debugging platform, on which it is possible to extract sensitive information from the non-volatile caches. In practice, such a platform is not hard to obtain. Many microprocessor manufacturers offer debugging and development platforms that allow a variety of access functions, including functionality to retrieve the cache content. For example, for the ARM platform, the manufacturer offers the DS-5 development software [5] and associated hardware DSTREAM Debug and Trace unit [6]. These tools enable debugging and introspection into ARM processor-based hardware. The attacked microprocessor can be plugged into a development board such as the Juno ARM Development Platform [7] either directly or through the JTAG debugging interface. In the DS-5 software, the *Cache Data View* can be used to examine the contents of all levels of caches and TLBs. Particularly for caches, information such as cache tags, flags and data associated with each cache line, as well as the index of each cache set, can be read and then exported to a file for further processing.

5.3 Cache Aware AES Key Search

In this work, we study the security of NVM-based microprocessor caches specifically by demonstrating AES key extraction attacks under the aforementioned threat model.

5.3.1 Technical Challenges

An AES key search algorithm for the main memory has been presented by Halderman et al. in their seminal work on cold boot attacks [27]. However, doing so in caches is not as straightforward. Particularly, the AES key search algorithm in Halderman et al. [27] assumes that a *complete* AES key schedule is stored in a *physically-contiguous* memory region. This is a relatively safe assumption in their case since the size of memory pages on modern computers are at least 4KB and a complete AES key schedule is 176 bytes (128-bit key/AES-128) to 240 bytes (256-bit key/AES-256). The algorithm proposed by Halderman et al. can, therefore, simply scan the entire memory image sequentially to search for potential AES keys [27]. However, neither the *completeness* of the key schedule nor the *contiguity* of the memory spaces can be assumed in the case of caches.

Non-contiguous memory space. Caches only capture a small non-contiguous subset of the memory space. Since cache lines are typically only 32-128 bytes, data that was originally stored in physically-contiguous memory is not necessarily stored in contiguous cache regions. Therefore, the logically sequential AES key schedules, typically 176 to 240 bytes, can be separated into multiple physically disjoint cache lines as shown in Figure 5.1(a).

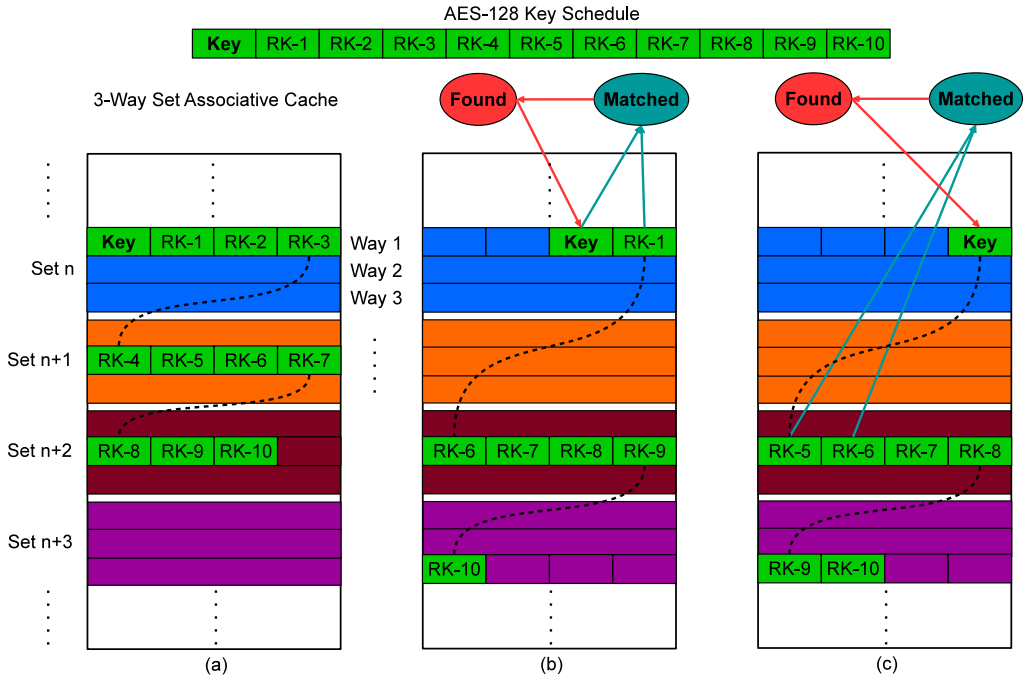


Figure 5.1: Cache view of AES key schedule with 64-byte cache lines.

Incomplete key schedules. Another relevant cache property is that data stored in the cache is subject to frequent replacements. Parts of a complete AES key schedule can be missing from the cache, which makes our key search more difficult to conduct. Examples of these situations are shown in Figure 5.1(b) and 5.1(c). Particularly, in Figure 5.1(b), the cache line that holds the `RK-2`, `RK-3`, `RK-4`, and `RK-5` has been evicted from the cache.

5.3.2 Search Algorithm Design

To address these issues our algorithm will first reconstruct the cache image by sorting cache lines by their physical addresses that we extract from the cache tags and indexes, and then feed the reconstructed cache image to the AES key search

function. In this way the logically contiguous data will still be contiguous in our reconstructed cache image regardless of the cache architecture. Our algorithm then performs either a *QuickSearch* or *DeepSearch* to search for the key in the reconstructed cache image. Note that in this work we assume an inclusive cache model so that the algorithm only examines the shared last-level cache (LLC) to simplify and accelerate the search. However, our algorithm can be easily expanded to work on other cache inclusiveness designs. Our AES key search algorithm is summarized in Algorithm 1.

Algorithm 1: AES key search algorithm.

```
Input: Original cache image
Output: List of keys found
begin
    Sort cache image by cache line address
    for each key_schedule candidate in sorted image do
        enc_key ← first 16 bytes in key_schedule
        dec_key_schedule ← reconstruct(key_schedule)
        dec_key ← first 16 bytes in dec_key_schedule
        if defined(QuickSearch) then
            Check relation between enc_/dec_key and firstRoundKey in
            key_schedule/dec_key_schedule
            if relation satisfied then
                Output enc_/dec_key
            end
        end
        if defined(DeepSearch) then
            Check relation between any two consecutive round keys in
            key_schedule/dec_key_schedule
            if any relation satisfied then
                Output enc_/dec_key
            end
        end
    end
end
```

QuickSearch. The DRAM cold boot attack was designed to deal with bit decay errors that occur due to instability in volatile DRAM cells at low temperatures after they are disconnected from power. To handle these errors, each candidate key is expanded through all 10 rounds of the AES key schedule. The hamming distance between the result of the expansion and the data that follows the candidate key in memory is computed. If the hamming distance is below a predefined error threshold, the candidate key is considered valid.

In contrast, the data from an NVM cache is expected to be error free, since most non-volatile memory has a data retention time as long as 10 years and is robust against soft errors [47]. As a result, our key search algorithm can be simplified and made much faster by only generating and attempting to validate the first round of the key expansion (16 bytes). If there is a match between the first round expansion of the candidate key and the data stored in the cache, the candidate key is validated. As long as the key itself followed by one round (16 bytes) of the expanded key exists in the cache, our algorithm can successfully detect the key as shown in Figure 5.1(b). We call this variant of the key search algorithm, *QuickSearch*. In our evaluation we find this algorithm to be very effective at identifying the encryption key, if one is present.

DeepSearch. The AES key schedule is stored on multiple cache lines since it is larger (at least 176 bytes for AES-128 mode) than the typical cache block (32-128 bytes). Cache evictions can displace parts of the AES key schedule from the cache, including the first round of the key expansion, which our *QuickSearch* algorithm uses to validate the key. These cases are rare since they require the memory alignment to be such that the encryption key falls at the end of a cache line and the first round of

the key expansion is on a different line. To deal with these cases we develop a more in-depth algorithm (which we call *DeepSearch*) that considers multiple rounds of the key expansion. In this implementation, as long as the key itself is inside the cache and there exist two consecutive rounds of expanded keys, our algorithm can find the key as shown in Figure 5.1(c).

The downsides of *DeepSearch* is that it runs considerably slower than *QuickSearch* and the search has some false positives. However since the attacker will only perform this *DeepSearch* once on the cache image, the slower runtime won't have any effects. As for the false positives, they are also in a manageable range. Our experiments involve searching millions of cache snapshots with a number of benchmarks and cache configurations for AES keys. To keep the runtimes manageable, the majority of our experiments are conducted using *QuickSearch*.

5.3.3 Implementation-Specific Considerations

We demonstrate the AES key extraction attack against `dm-crypt`, a cryptographic module that is used in mainstream Linux kernels. As will be explained in Section 5.4, the specific target of our demonstrated attacks is the disk encryption/decryption application, LUKS (Linux Unified Key Setup), of the Ubuntu OS, which by default invokes the `dm-crypt` kernel module for disk encryption/decryption using AES-XTS mode [2].

In the AES implementation of `dm-crypt` the decryption key schedule is different from the encryption one. We illustrate the key schedule for the decryption process in Figure 5.2(a). The decryption key schedule is first reversed in rounds from the encryption key schedule. An inverse mix column operation is then applied to rounds 1

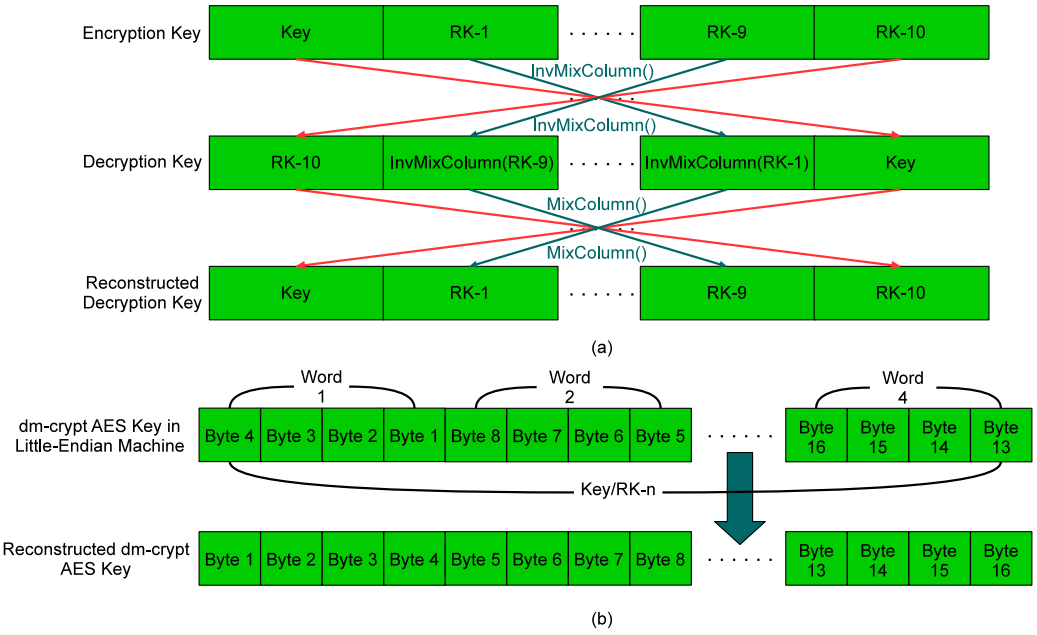


Figure 5.2: Details on AES implementation dependent modifications to the key search algorithm.

through 9 of the key schedule. As a result, we need to perform searches for encryption keys and decryption keys separately. Specifically, before searching for decryption keys we first convert the candidate schedules back to the encryption key schedule format.

Another artifact that affects our key search algorithm is specific to little-endian machines, which store the least significant byte in the lowest address. `dm-crypt` adopts an implementation which stores the AES key schedules as an array of words (e.g. 4 bytes) instead of bytes. This leads to a mismatch in the representation on little-endian architectures, as shown in the first line of Figure 5.2(b). Our search algorithm takes into account this artifact and convert the little-endian representation to the big-endian one before conducting the key search.

5.3.4 Discussion on Scalability

Our key search algorithm is designed for AES on inclusive cache models. However it can also be applied to other cache models (exclusive and non-inclusive) and modified to work with other encryption algorithms.

Other cache models. There are two ways to apply our key search algorithm when the assumption of cache models changes to exclusive or non-inclusive. The first way is to directly apply the algorithm sequentially to each cache at all levels. Since it is highly likely that the AES key and the first round of the key schedule exist in the same cache line, our *QuickSearch* algorithm will have a high probability of finding the key if it is actually in any of the caches. In this context our algorithm will just be slower since it will search all caches for the key instead of only examining the LLC. To deal with rare cases that AES key falls at the end of a cache line in one cache while the other parts of the key schedule are in cache lines of other caches, we can use the second way to apply our algorithm by reconstructing the cache image with contents of all levels of caches. In this reconstruction process we sort cache lines from all the caches by their physical addresses and ignore the ones with the same physical address (in non-inclusive model). Then we can either apply *QuickSearch* or *DeepSearch* on this reconstructed cache image to search for keys.

Other encryption algorithms. Our cache-aware key search algorithm can also be modified to work with other encryption algorithms such as RSA [66], DES [1], etc. One of the key modifications is to find data relationships inside the encryption key structures of those algorithms and develop mathematical models to validate expected data with the data stored in caches, similarly as we show the AES example in Figure

5.1. To reconstruct cache data for search we can simply rely on our current algorithm with modifications as needed. Halderman et al. [27] has already explored this topic in the main memory context. We leave this work on caches as future work.

5.4 Evaluation Methodology

Because microprocessors with NVM caches are not currently available in commercial computers, we therefore relied on a full system simulator to conduct our experiments. Specifically, we modeled an 8-core ARMv8-based processor using the gem5 simulator [9]. Cold boot attacks have been demonstrated on both x86 and ARM architectures in the past. Our simulation selected ARMv8 architecture for its broader adoption in mobile devices which are particularly vulnerable to the physical access required by cold boot attacks. Our results should be generally applicable to other microprocessors.

We simulated a traditional 2-level cache hierarchy with private L1 instruction and data caches for each core and a shared inclusive L2 cache as the last level cache. Our cache configuration parameters are in line with the ones used in many modern computer systems. Since our key search algorithm focuses on the LLC, we experimented with different LLC sizes from 2MB to 128MB (8MB as the default size of LLC if not explicitly stated). We examine a broad range of LLC sizes from small (2MB) to very large (128MB). The hardware configurations of our simulated system are summarized in Table 5.1.

The system is configured to run Ubuntu 14.04 Trusty 64-bit operating system. We installed the cryptsetup application - LUKS (Linux Unified Key Setup) in the Ubuntu OS and used it with the `dm-crypt` module in Linux kernel to encrypt a 4GB

Hardware Configuration	
Cores	8 (out-of-order)
ISA	ARMv8 (64-bit)
Frequency	3GHz
IL1/DL1 Size	32KB
IL1/DL1 Block Size	64B
IL1/DL1 Associativity	8-way
IL1/DL1 Latency	2 cycles
Coherence Protocol	MESI
L2 Size	2, 4, 8 (default), and 128MB
L2 Block Size	64B
L2 Associativity	16-way
L2 Latency	20 cycles
Memory Type	DDR3-1600 SDRAM [48]
Memory Size	2GB
Memory Page Size	4KB
Memory Latency	300 cycles
Disk Type	Solid-State Disk (SSD)
Disk Latency	150us

Table 5.1: Summary of hardware configurations.

partition of a simulated hard drive. The disk encryption/decryption algorithm we configured for LUKS was AES-XTS [2] with 128-bit keys.

We ran SPEC CPU2006 benchmark suite [28] stored in the encrypted hard drive to simulate applications that run on the target system. SPEC CPU2006 benchmark suite includes integer and floating-point single-threaded benchmarks among which some are more computation bound and others are more memory bound [37]. To keep the simulation time reasonable, we use the checkpoint functionality provided by gem5 [9] to bypass the OS boot-up phase and ran each benchmark with up to 1 billion instructions. For experiments which require periodically taking LLC image snapshots

Mixed Benchmark Group	
mixC	*calculix, dealII, gamess, gromacs, h264ref, namd, perlbench, povray*
mixM	*astar, cactusADM, GemsFDTD, lbm, mcf, milc, omnetpp, soplex*
mixCM	*dealII, gamess, namd, perlbench, astar, cactusADM, lbm, milc*

Table 5.2: Overview of mixed benchmark groups.

we use a sampling interval of 1 million instructions. To further test our attack scenario and countermeasure approach in a multi-programmed/multi-threaded environment, we also ran several groups of mixed benchmarks from SPEC CPU2006 - *mixC*, *mixM*, and *mixCM*. As detailed in Table 5.2, *mixC* contains 8 computation-bound benchmarks, *mixM* contains 8 memory-bound benchmarks, and *mixCM* contains 4 benchmarks from *mixC* and another 4 benchmarks from *mixM*.

5.5 Attack Analysis

To analyze the severity of the vulnerabilities, we examine the probability of successfully retrieving disk encryption keys from a processor's last level cache under two attack scenarios.

5.5.1 Random Information Harvesting

We first investigate an attack scenario in which the attacker gains access to the target machine and disconnects the processor from the power supply at an arbitrary point during the execution. We make no assumptions that the attacker has a way to force a certain code sequence to execute. This is typical, for instance, when a defective device that stores sensitive data is discarded without proper security measures.

Experiment Label & Description	
NoNEON	System without ARM's cryptographic acceleration support
NEON	System with ARM's cryptographic acceleration support
mixC	SPEC CPU2006 computation bound benchmark mix
mixM	SPEC CPU2006 memory bound benchmark mix
mixCM	SPEC CPU2006 computation bound plus memory bound benchmark mix
STAvg	Geometric mean of single-threaded benchmarks from SPEC CPU2006

Table 5.3: Summary of experiment and workload labels.

Another example is when an attacker steals a device and physically disconnects its power supply before removing the processor. Since the system could have failed at any point, we would like to examine the probability of successfully identifying and retrieving the disk encryption key as a function of workload, cache size, etc.

To study the probability of success for such an attack we take periodic snapshots of the LLC, at intervals of roughly 1 million instructions. We then run the *QuickSearch* key search algorithm on each cache snapshot. Figure 5.3 shows the probability of finding the AES keys in the 8MB last level cache for different benchmarks. We examine both systems with and without ARM's cryptographic acceleration (NEON) support. For easy reference, Table 5.3 summarizes the labels we use for different experiments.

To better analyze results we classify the SPEC CPU2006 benchmarks into two categories - compute-intensive and memory-intensive [37]. We can see from the results in Figure 5.3 that when running computation bound benchmarks the probability of

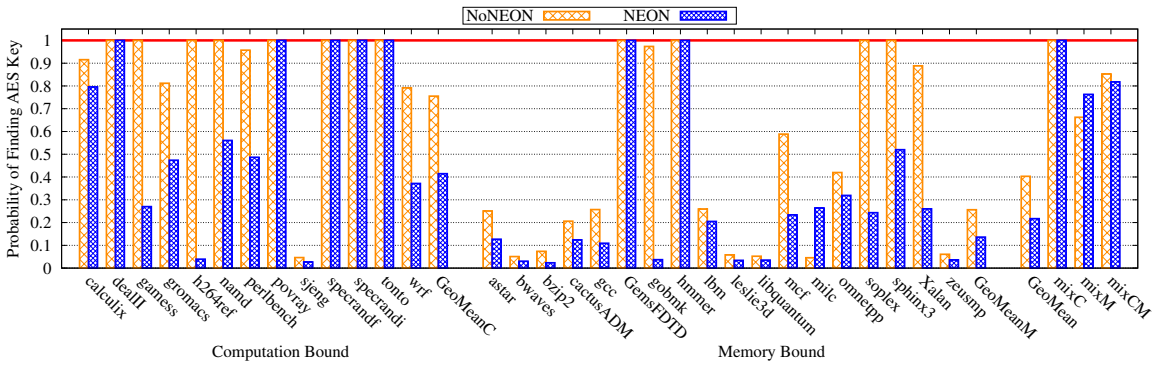


Figure 5.3: Probability of finding AES keys in 8MB LLC with various types of benchmarks.

finding AES keys in the cache is higher than running memory bound benchmarks. On average, there is a 76% probability of finding AES keys in the system without NEON support when running computation bound benchmarks and a 26% probability when running memory bound benchmarks.

When the system is configured with NEON support, the probability of finding the key in the cache drops for both classes of benchmarks to 41% and 14% respectively. This is because the NEON technology stores encryption keys in vector registers that are large enough to hold the key schedule. These registers are also infrequently used for other functions which means they don't have to be spilled to cache (and memory). As a result, in processors with NEON support the encryption key is read from memory much less frequently, leading to better resilience to this type of attack. A typical case is seen in *perlbench* with 96% for NoNEON and 49% for NEON. However there are also exceptions as seen in *povray* (100% for both systems) and *gobmk* (97% for NoNEON and 4% for NEON). On average, the probability of finding the key in the random

attack is 40% for the system without NEON support and 22% for the system with NEON support.

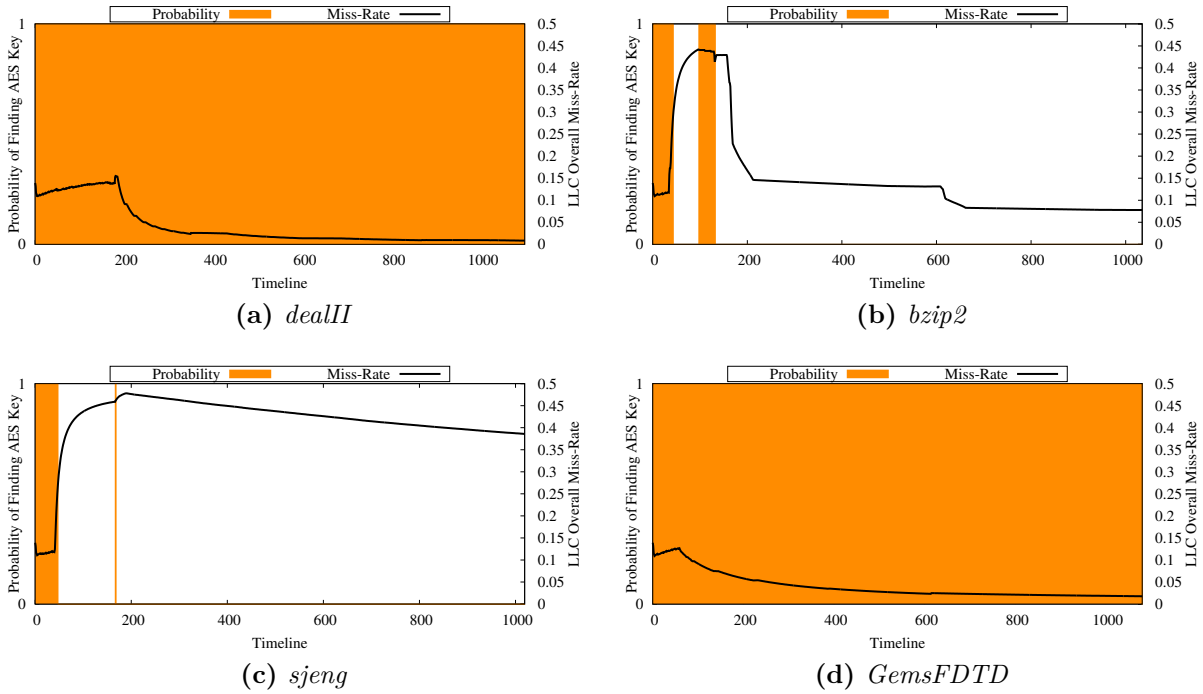
There are two principal factors that affect the probability the encryption key is found in the cache at random points during execution. The first is how recent the last disk transaction was, since encrypted disk access requires the AES key to be brought into the cache. The second factor is the cache miss rate, since the higher the churn of the data stored in the cache the sooner an unused key will be evicted. Computation bound benchmarks in general have a smaller memory footprint so that their cache miss rates are lower, allowing keys to reside in the cache for longer. Memory bound benchmarks, on the other hand, have a larger memory footprint associated with a higher cache miss rate therefore evicting keys more frequently. Figure 5.4 illustrates these effects for selected benchmarks running on systems without NEON support, showing for each cache snapshot over time whether the key was found or not (1 or 0). The figure also shows the cumulative miss rate of the LLC over the same time interval.

Benchmark *dealII* shown in Figure 5.4a is a good illustration of the behavior of a compute-bound application. The disk encryption key is brought into the cache early in the execution as the application accesses the disk to read input data. The miss rate is low throughout the execution of the application which means the key is never evicted and the probability of finding the key while running this application is 100%.

Figure 5.4b shows the behavior of a memory bound application, *bzip2*. Keys are brought into the cache early in the execution and remain in the cache for a period of time while the miss rate is low. The miss rate, however, spikes as the application begins processing large data blocks for compression. This evicts the key from the

cache. A disk operation causes the key to be brought into the cache again, but the consistently high miss rate causes the key to be evicted again shortly. Even though later in the execution cache miss rate drops, the lack of disk accesses keep the key away from the cache for the rest of this run. Note that for clarity we only show a fraction of the total execution.

While most benchmarks behave as their broader category, there are a couple of exceptions as shown in Figure 5.4c and 5.4d. *sjeng* is a computation bound benchmark with high cache miss rate therefore AES key remains in the cache for a small fraction of the execution. *GemsFDTD* is a memory bound benchmark but it has an overall low miss rate and therefore the key persists in the cache throughout the execution.



(a) *dealII* **(b)** *bzip2*

(c) *sjeng* **(d)** *GemsFDTD*

Figure 5.4: AES key search trace with cumulative LLC miss rate information of (a) *dealII*, (b) *bzip2*, (c) *sjeng*, and (d) *GemsFDTD* benchmarks running on systems without ARM's cryptographic acceleration (NEON) support.

We also collect results for multi-program mixed workloads to increase system and disk utilization and cache contention. The results are included in Figure 5.3 as *mixC*, *mixM*, and *mixCM*. The benchmark applications included in each mix are listed in Table 5.2. As expected, when the system is fully utilized, with one application running on each core (for a total of 8), the probability of finding the key increases. This is because each application accesses the disk and those accesses occur at different times, causing the encryption key to be read more frequently. The compute bound *mixC* shows 100% probability of finding the key for both systems (with and without NEON support).

While a system with high utilization is clearly more vulnerable, a mitigating factor is that cache contention is also higher when many threads are running. As a result, cache miss rates are also higher and the key may be evicted more often. This is apparent when we examine the memory-bound *mixM* workload which shows 66% probability without NEON support and 76% with NEON support. Even with the higher miss rate, the fully-loaded system is clearly more vulnerable than lightly-loaded system, as seen in the single-threaded experiments. When a mix of both compute and memory bound applications is used (*mixCM*) the probability of finding the key is 85% for NoNEON and 82% for NEON.

We also note that the NEON-accelerated system is almost as vulnerable as the system without hardware acceleration when the system is running the mix workloads. This is likely caused by the more frequent spills of the vector registers when context switching between the kernel thread running the encryption/decryption process and the user threads. Spilling the vector registers holding the encryption key increases

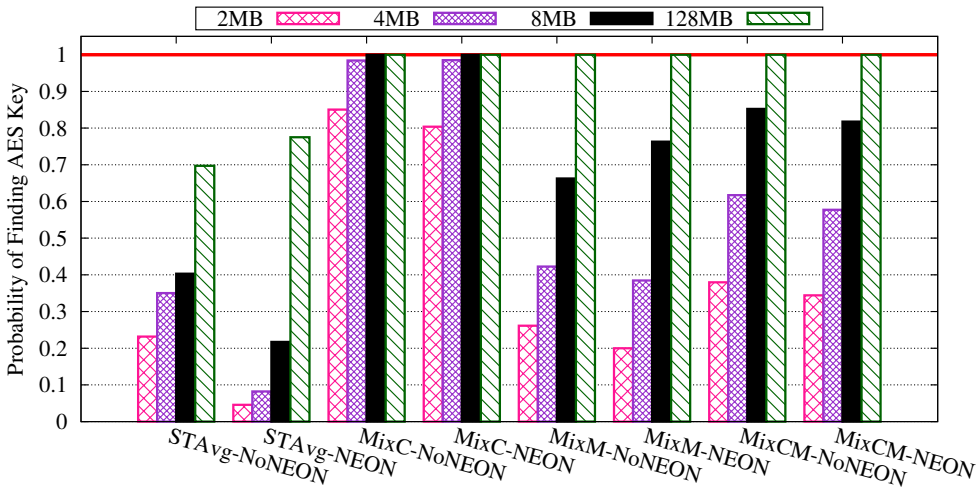


Figure 5.5: Overall probability of finding AES keys in LLCs with various sizes.

reads and writes of the key from and to memory, exposing it to the cache more frequently.

Figure 5.5 shows the overall probability of finding AES keys in systems with different sized LLCs. As expected, larger caches increase the system vulnerability to this type of attack. We can see that as cache size increases the probability of finding keys also increase across all the benchmarks. With 2MB cache the average probability of finding AES keys is from 4.6% to 85% depending on the system and application; for a 128MB LLC the probability of a successful attack ranges from 70% to 100%. Increasing the cache size reduces capacity misses therefore increasing the fraction of time the key spends in the cache.

Vulnerability in disk encryption vs. decryption. In our experiments we find that disk encryption/decryption operations yield different time sensitivity of keys appearing in the caches. For decryption operations (disk reads) keys immediately appear in the cache while for encryption operations (disk writes) keys will appear in

the cache with a substantial delay following the execution of the disk write operation. This is because for disk write operations Linux uses a page cache to buffer writes to the disk in order to improve performance. Only when the page cache is full, or when the encrypted disk is unmounted, will the contents of the page cache be written back to the encrypted disk. The AES keys will be retrieved and will appear in the cache at this time. Running the `sync` command after a write operation will force a write to disk and encryption keys will then immediately appear in the cache.

5.5.2 Targeted Power-off Attack

The second attack scenario we consider is one in which the attacker is able to trigger a graceful or forced power-off sequence before physically removing the processor. In this attack scenario, the attacker aims to use the power-off sequence to deterministically ensure the disk encryption keys are found in the cache. Since the power-off sequence involves unmounting the disk, this likely results in a series of encryption/decryption transactions that will bring the encryption key into the cache. During the system shutdown process, the attacker can stop the execution at any time to search for secret keys or simply wait until the device is completely powered off to examine cache images for secret keys. The goal of this attack scenario is to find a reproducible and reliable way to obtain the encryption key from a compromised system.

Figure 5.6 shows the sequence of operations executed after running the `poweroff` command. There are two operations (highlighted in green) in the power-off sequence which will bring disk encryption keys to the cache. The first operation (operation 2) is when operating system asks all remaining processes to terminate. In this operation

```
root@aarch64-gem5:/# poweroff
Session terminated, terminating shell...exit
 ...terminated.
 * Stopping rsync daemon rsync                              [ OK ]  // 1
 * Asking all remaining processes to terminate...           [ OK ]  // 2
 * All processes ended within 1 seconds...                  [ OK ]  // 3
 * Deactivating swap...                                     [ OK ]  // 4
 * Unmounting local filesystems...                          [ OK ]  // 5
 * Stopping early crypto disks...                           [ OK ]  // 6
 * Will now halt                                                    // 7
[  604.955626] reboot: System halted
```

Figure 5.6: `poweroff` command triggered operation sequence.

the process in charge of disk encryption will be terminated. This will invoke the `sync` system call to flush data from the page cache to the encrypted disk which requires reading the AES keys. Before the system is actually powered off, all filesystems must be unmounted as shown in operation 5. The encryption keys are again used in unmounting the encrypted disk drive and they will again appear in the cache.

We experimented with two power-off methods in the evaluation - normal and forced. We examine the probability of successfully identifying the key under the two scenarios. Table 5.4 summarizes the results of our power-off attacks on various LLC sizes. The command associated with each power-off method is also listed in Table 5.4.

We can see from the results that starting from LLC size of 8MB keys will remain in the cache no matter what power-off methods we use. For 2MB and 4MB LLC sizes, keys will exist in the cache when the forced power-off method is used but not for the normal one. For the smaller cache sizes like 2MB or 4MB, after keys are brought into

Mode	Command	Keys exist in cache after power-off?			
		2MB	4MB	8MB	128MB
Normal Power-off	poweroff (-p)	N	N	Y	Y
Forced Power-off	poweroff -f	Y	Y	Y	Y

Table 5.4: Summary of targeted power-off attack results.

the cache other operations which don't involve encryption disk accesses may bring new data in and get AES keys evicted. Therefore we won't see the keys after system is powered off. However for larger caches with 8MB or 128MB the keys will stay in the cache after system shutdown regardless of the method.

Forced power-off is different from normal power-off in that it doesn't power-off the system in a graceful way. This means forced power-off will only perform the action of powering off the system. However in order to power off the system the local filesystems are still going to be unmounted to prevent data loss. In this process the keys will be brought into the cache and stay inside the cache after system is powered off in all cache sizes we tried in the experiments. Forced power-off attacks virtually guarantee that the system we investigate, in all configurations, will expose the secret keys in the NVM cache. This shows that a potential attacker has a reliable and reproducible mechanism to compromise disk encryption keys.

Figure 5.7 shows the presence of the disk encryption keys in the cache throughout the normal power-off sequence for the NoNEON and NEON systems, for different LLC sizes. We can see that the encryption key appears in the cache at roughly the same time following operation no. 2 in the power-off sequence (Figure 5.6). It is then quickly evicted in the 2MB LLC system, but persists for increasingly longer

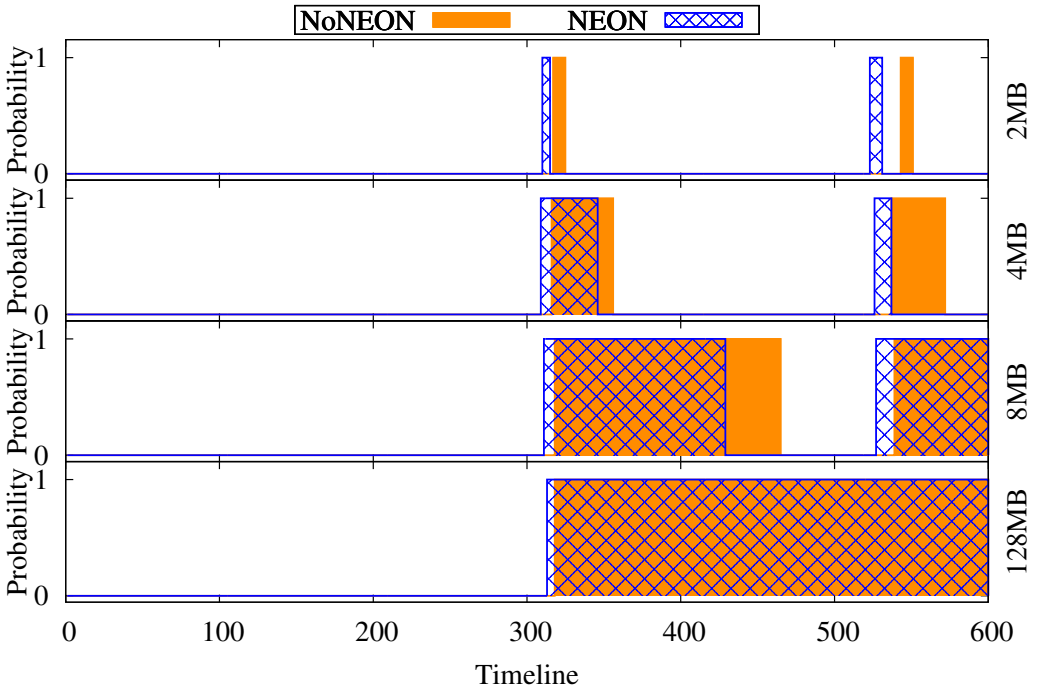


Figure 5.7: AES key search sequence of normal power-off from start to completion with various sizes of LLCs.

time intervals as the size of the LLC increases. For the 128MB cache, the key is never evicted before the system halts. The key is again read into the cache following operation no. 5 (unmounting the filesystem). In the 2MB and 4MB cases the key is again evicted before the system halts. Even for these systems an attacker could force the key to remain in the cache in a predictable way. The attacker would simply have to trigger the power-off sequence and then disconnect the processor from the power supply after a predetermined time period before the key is evicted. Since the power-off sequence is fairly deterministic, this approach has a high probability of success.

5.6 Countermeasures

In order to mitigate threats of cold boot attacks against NVM caches, we propose a simple and effective software-based countermeasure. Our countermeasure is designed to force secret keys to be only stored in encrypted main memory and bypass the NVM cache throughout the execution. We develop a software API for declaring memory blocks as secrets to inform the system that their storage in the cache is not allowed. While our countermeasure applies to any secret information stored by the system, we use the disk encryption example as a case study to illustrate the concept. It is worthwhile noting that our solution only protects NVM caches from cold boot attacks. We assume the main memory is encrypted using existing techniques so that cold boot attacks no longer work against the main memory [10, 29, 42, 44, 65, 73, 80, 81]. We emphasize that encryption of caches is infeasible given the requirements of low access latency of caches.

5.6.1 Countermeasure Design

The process of decrypting an encrypted storage device in a system typically involves using the `cryptsetup` command-line utility in user space which calls the `dm-crypt` kernel module. This process is illustrated in Figure 5.8. The kernel establishes within its internal cryptographic structures the key to be used for accessing the encrypted device that has been selected via the `cryptsetup` utility. Although the process of establishing the key inside the kernel entails generating multiple copies of the key, the relevant block cipher routines in the `crypto` module dutifully use `memset()` to wipe the key after a new copy is created. As such, we only focus on the final memory location where the key is stored which is tracked by the `crypto_aes_ctx`

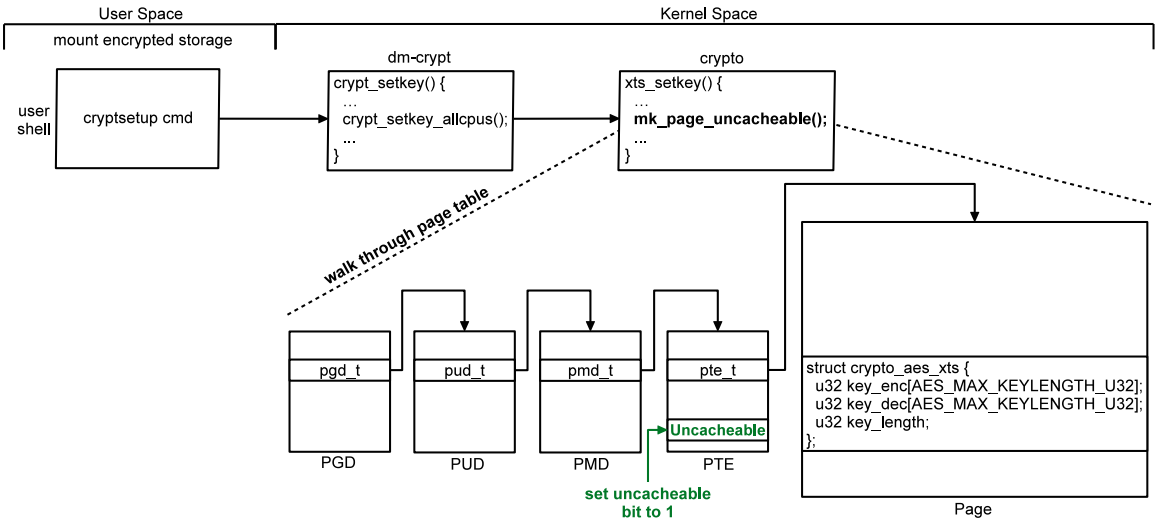


Figure 5.8: Countermeasure deployment in a system with encrypted storage.

structure. In our solution, we devise a countermeasure that is applicable to kernels that are configured to utilize hardware acceleration, as well as default kernels configured for environments where such acceleration support is unavailable.

Systems with hardware cryptographic support. Modern systems usually make use of hardware acceleration for cryptographic operations. We assume the kernel is built with the AArch64 accelerated cryptographic algorithms that make use of NEON and AES cryptographic extensions defined in the ARMv8 instruction set. This is done by including the `CONFIG_CRYPTO_AES_ARM64_*`, `CONFIG_ARM64_CRYPTO`, and `KERNEL_MODE_NEON` kernel parameters as part of the build. This translates to using architecture specific cryptographic libraries defined in `/arch/arm64/crypto` of the Linux source. In order to eliminate the presence of cryptographic keys in the cache, our solution involves marking the page associated with the address of the `crypto_aes_ctx` structure as uncacheable. We implement the necessary changes for

this approach within `xts_set_key()` routine located in `aes-glue.c` where we walk through the page table in search of the appropriate page table entry that maps to the designed page. Once we locate the correct PTE, we set the `L_PTE_MT_UNCACHED` flag to label the page as uncacheable.

A similar approach can be applied to x86 systems by inserting an entry into the Memory Type Range Register (MTRR) that corresponds to the region of interest. For newer x86 systems, the page attribute tables (PAT) can be used instead of the MTRR. Although the solution operates at a page size granularity—that is, the smallest size of memory blocks to be marked as uncacheable is a page—the performance impact is minimal for small page sizes. We also note that our experiments show that this approach performs better than a solution that periodically evicts the keys from the cache. This is because a periodic eviction approach requires adding changes to the `ce_aes_xts_encrypt()` and `ce_aes_xts_decrypt()` routines which would incur overhead for every block of data that is encrypted or decrypted. It also introduces a window of vulnerability where the keys could be recovered in cases where an attacker initiates a forced power-off (ungraceful shutdown) of the system while storage is being accessed.

Systems without hardware cryptographic support. If the kernel lacks support of accelerated cryptographic hardware, we use the default cryptographic library defined in the `/crypto` directory of the Linux source. This boils down to modifying the `crypto_aes_set_key()` in `aes_generic.c`. However, we use a similar approach to the one described previously by marking the page which contains the `crypto_aes_ctx` structure to be uncacheable. The primary difference is that encryption and decryption that are used in `aes_encrypt()` and `aes_decrypt()` respectively do not make

	NoNEON	NEON	Countermeasure
Single-threaded Benchmark	23 - 70%	5 - 77%	**0%**
mixC	85 - 100%	80 - 100%	**0%**
mixM	26 - 100%	20 - 100%	**0%**
mixCM	38 - 100%	34 - 100%	**0%**
Normal Power-off	0 - 100%	0 - 100%	**0%**
Forced Power-off	100%	100%	**0%**

Table 5.5: Probability of finding AES keys with and without the countermeasure.

use of the 128-bit NEON registers. As such, the performance impact with this approach is higher since multiple fetches of the expanded key from memory are needed for each round of encryption or decryption.

5.6.2 Countermeasure Effectiveness

Table 5.5 summarizes the effectiveness of our countermeasure. We can see that by marking the AES key structure uncacheable our countermeasure completely eliminated the security vulnerability of NVM caches for system with and without processor cryptographic acceleration support. All attack scenarios we examined are now unable to find the disk encryption keys in the cache, regardless of the benchmarks running on the system. The targeted power-off attacks also fail to identify any AES keys in the cache once the countermeasure is applied.

5.6.3 Performance Overhead

The effectiveness of our countermeasure comes with the cost of some performance overhead. Figure 5.9 shows the performance overhead for different types of benchmarks executed in the context of our random attack. In general, the overhead for the

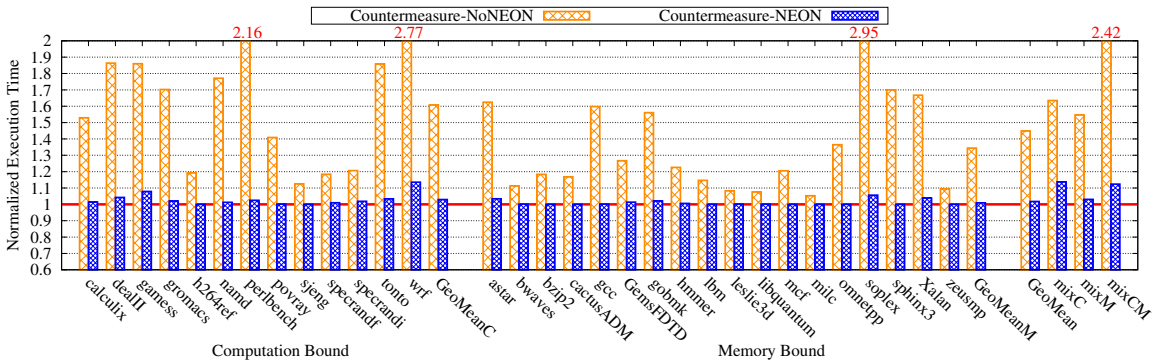


Figure 5.9: Average countermeasure performance overhead of all sizes of LLCs with different benchmarks.

system with NEON support is very low, averaging 2% for the single-threaded benchmarks. The overhead increases substantially if the system has no NEON acceleration – up to 45% for single-threaded benchmarks. Performance overhead of the countermeasure correlates directly with the number of encryption/decryption transactions. Since the encryption key is uncacheable, every access to the key will result in a slow memory transaction (300 cycles vs. 20 cycles for the LLC as shown in Table 5.1). The NEON hardware support helps alleviate this overhead substantially by storing the key in vector registers and bypassing slow memory accesses for most transactions.

The performance overheads are higher as expected for multi-programmed workloads because they perform more encryption/decryption transactions overall. However in this case systems with NEON support is again better at absorbing these overheads than systems without NEON support. Overheads for the three workload mixes are 64% in NoNEON and 14% in NEON for *mixC*, 55% in NoNEON and 3% in NEON for *mixM*, and 142% in NoNEON and 12% in NEON for *mixCM*.

When comparing between compute and memory bound benchmarks, we can see that there is no significant difference for the NEON group (computation bound 3% vs. memory bound 1%). However this difference increases to 27% in the NoNEON group due to much more cache hits now are all translated into memory accesses in the computation bound benchmarks than those in the memory bound benchmarks.

5.6.4 Discussion

The performance overhead due to our countermeasure can be reduced with software optimization or hardware support.

Optimization on software-based countermeasure. Our current countermeasure strictly mark secret information as uncacheable throughout system execution. However one observation here is that when the authenticated user is currently using the system, it is unnecessary to keep the sensitive data uncacheable since physical cold boot attacks are unlikely to happen when the user is in possession of the device (or at least it is not necessary as cache/memory data can be dumped directly using software approaches). One possible performance optimization is to enable two modes of handling secret information—cacheable and uncacheable. When user is logged in, cacheable mode on secrets is enabled so that user won't experience any performance degradation of the system. Only when user is logged off or the system is locked, secret information inside caches will be erased and then uncacheable mode will be turned on to protect from cold boot attacks.

Hardware-based countermeasure. Another approach for reducing performance overhead is to rely on a hardware-based countermeasure. One possible solution is to leverage existing write buffers associated with non-volatile caches [26, 74, 78]. These

buffers are implemented with volatile memory (usually SRAM) which are necessary because latency associated with write operations in NVM is too high. Storing secrets in such volatile memory modules will bypass the cache. We leave exploration of the feasibility of this method as future work.

5.7 Conclusion

This work demonstrates that non-volatile caches are extremely vulnerable to cold boot attacks. We successfully conducted two attacks on disk encryption keys—random attacks and targeted power-off attacks. Our experiment results show that the probability of finding the secret AES keys in NVM caches ranges from 5% to 100% with varying workloads and cache configurations in random attacks and always reaches 100% in targeted power-off attacks. To defend computer systems against these attacks we developed a software-based countermeasure that allocates sensitive information into uncacheable memory pages. Our proposed countermeasure completely mitigates cold boot attacks against NVM caches. We hope this work will serve as a starting point for future studies on the security vulnerabilities of NVM caches and their countermeasures.

CHAPTER 6

Related Work

The work conducted in this dissertation relies on an extensive body of research that mainly falls into four categories: STT-RAM, near-threshold architectures, variation-aware thread mapping, and cold boot attacks and defenses.

6.1 STT-RAM

As STT-RAM has gained more and more attention recently, many researchers have focused on solving the long-latency and high-energy write issues associated with this technology in order to make it a feasible SRAM replacement. For instance, Zhou et al. [86] proposed Early Write Termination to terminate redundant bit writes at their early stages to reduce write energy. Other work [72, 75] has explored factors which could affect data retention time of STT-RAM cells and found that there is a trade-off between non-volatility and write performance and energy of those cells. By relaxing the non-volatility requirement they observe that they can improve energy by using shorter write times.

Guo et al. [26] explored replacing large, wire-delay dominated SRAM arrays, such as caches, TLBs, and register files with STT-RAM. Some of the latency-critical units

as well as pipeline registers are implemented using SRAM. They have built an in-order 8-core processor with this hybrid design. Their goal is to save energy by replacing high-leakage CMOS with low-leakage STT-RAM. Their design is similar to our NVNoSleep system except that we use out-of-order cores and hybrid SRAM/STT-RAM structures for memory units that are updated frequently. We show that significant energy reductions can be achieved by aggressively turning cores off when idle, in addition to simply replacing SRAM with STT-RAM.

Previous work [69] has proposed building the last level cache or main memory with non-volatile memory like STT-RAM or PCRAM (Phase Change RAM) to provide checkpointing capabilities for reliability and power reduction. Their solution is targeted at coarse-grained server/system level checkpoints that can tolerate much higher checkpointing/restore overheads. Our checkpointing has much lower performance overhead.

Kvatinsky et al. [40] proposed a memristor-based multistate pipeline register design to provide low penalty switch-on-event multithreading capabilities.

6.2 Near-Threshold Architectures

Previous work by Zhai et al. [83] has proposed grouping several slower near-threshold cores into a cluster that shares a faster L1 cache in order to eliminate cache coherence traffic. This can speed up system performance and also reduce coherence energy. In their design they applied a relatively higher voltage to the shared SRAM L1 cache and found the optimal energy efficiency configuration is 2 cores per cluster with 2 clusters. They did not explicitly consider variation effects or heterogeneous core frequencies in their design. We use nominal voltage STT-RAM to build the

shared L1 cache. In our design the STT-RAM shared cache is much faster than the cores, making much larger clusters (16 cores) become optimal. In addition, our work takes advantage of this shared cache design to perform dynamic core consolidation to further optimize energy efficiency.

Work by Karpuzcu et al. [39] proposes a near-threshold clustered manycore design which uses a single V_{dd} domain with dynamic clustering of cores running at the same frequency. Frequency variation is only allowed between these clusters. They developed a core assignment algorithm to perform core-to-job mapping to make sure jobs are run on selected cores with the same frequency and at the same time deliver high performance per watt. Our approach is transparent to the OS and uses low-cost dynamic core consolidation to exploit the optimal hardware resources for running benchmarks at different phases.

6.3 Variation-Aware Thread Mapping

Variation-aware thread mapping has been explored by prior work [14,32,76], sometimes in conjunction with active power management [30]. They optimize thread allocation in variation-induced heterogeneous CMPs to improve performance and energy. In general, they rely on exposing heterogeneity to the system/application and using optimization algorithms - that run either in hardware or software - to find an optimized mapping of application threads to cores.

Miller et al. [50] proposed to provide dual voltage rails to each individual core in a process variation aware near-threshold microprocessor chip. This allows cores to frequently switch voltage rails to speed up and slow down execution and ensure equal progress across cores. While effective, their approach is costly to implement. It

requires two power distribution networks to be routed to each core and large power gates to enable fast rail switching.

Other work has addressed heterogeneity-by-design [45,58], examining how heterogeneous cores can be used to form big-little core pairs or leader-follower core groups to save energy or improve performance. Shelepov et al. [70] developed an OS-based thread scheduler for heterogeneous systems that could also be adapted to variation-induced heterogeneity.

6.4 Cold Boot Attacks and Defenses

The idea of cold boot attacks in modern systems was first explored by Halderman et al. [27]. Their work consisted of extracting disk encryption keys using information present in main memory (DRAM) images preserved from a laptop. The idea of this type of attack builds on the premise that under low temperature conditions, DRAM chips preserve their content for extended time durations. The attack also relies on the fact that AES keys can be inferred by examining relationships between subkeys that involve computing the hamming distance information. The AES key search algorithm has been proposed in [27] as well. Muller et al. [56] later expanded cold boot attacks to mobile devices. When volatile memories such as SRAM and DRAM are replaced by non-volatile ones (e.g. STT-RAM, PCM, and ReRAM) in future computers, cold boot attacks will become much easier to perform since data will be indefinitely preserved after cutting off power supply for several years without the need for any special techniques such as cooling. Our work is the first work to study cold boot attacks in the context of non-volatile caches.

Prior work can be classified into two types of countermeasures for addressing the issue of cold boot attacks. The first class of countermeasures involves handling cold boot attacks by securing the destination side of data. Several researchers have proposed encrypting memory data in order to prevent attackers from easily extracting secret information from main memory [10, 29, 42, 44, 65, 73, 80, 81]. Although this approach is effective for main memory, encryption techniques are challenging to apply to caches because of their large performance overhead. Other researchers, on the other hand, have examined a second class of countermeasures that involves handling cold boot attacks from the source side that generates the data. As such, this body of work proposes keeping secret keys away from main memory during system execution [12, 23–25, 54, 55, 59, 71, 84]. The idea behind this approach is to keep secret keys stored in CPU registers, caches, and other internal storage during system execution. This way secret keys wouldn't be found in main memory altogether, and as a result, render cold boot attacks ineffective against main memory subsystems. However these proposed approaches haven't considered the vulnerability of data stored in CPU caches since keys stored in CPU registers and other internal storage can still be fetched into caches during execution [23, 24, 54, 55, 71]. Moreover, a subset of this work even suggests storing secret keys inside CPU caches [12, 25, 59, 84]. This motivates our work for studying cold boot attacks in the context of non-volatile caches.

CHAPTER 7

Conclusion

The work presented in this dissertation has explored using STT-RAM to build low-power and secure processors. NVSleep demonstrates a low-power microprocessor framework that leverages STT-RAM to implement rapid checkpoint/wakeup of idle cores to save power. Respin presents an architecture that consolidates the private caches of near-threshold cores into unified L1 instruction/data caches that use STT-RAM to save leakage power and improve performance. In addition, a novel hardware virtualization core management mechanism is developed to increase resource efficiency and further save energy. Vulnerabilities of non-volatile memory as processor caches when exposed to "cold boot" attacks have been studied in depth in NVInsecure. Several types of proof-of-concept attacks have been successfully performed and an effective software-based countermeasure has been implemented to help address the security vulnerabilities of NVM caches.

BIBLIOGRAPHY

[1] Data Encryption Standard. *National Institute of Standards and Technology, Federal Information Processing Standards Publication 46-3* (October 1999).

[2] The XTS-AES Tweakable Block Cipher. *IEEE Std 1619-2007* (May 2007).

[3] Advanced Encryption Standard. *National Institute of Standards and Technology, Federal Information Processing Standards Publication 197* (November 2011).

[4] Anagnostopoulos, N. A., Katzenbeisser, S., Rosenstihl, M., Schaller, A., Gabmeyer, S., and Arul, T. Low-Temperature Data Remanence Attacks Against Intrinsic SRAM PUFs. Cryptology ePrint Archive, Report 2016/769, 2016.

[5] ARM. DS-5 Development Studio. https://developer.arm.com/products/software-development-tools/ds-5-development-studio/ds-5-debugger.

[6] ARM. DSTREAM High-Performance Debug and Trace. https://developer.arm.com/products/software-development-tools/debug-probes-and-adapters/dstream.

[7] ARM. Juno ARM Development Platform. https://developer.arm.com/products/system-design/versatile-express-family/juno-development-board.

[8] Berent, A. Advanced Encryption Standard by Example. http://www.infosecwriters.com/Papers/ABerent_AESbyExample.pdf.

[9] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. The gem5 Simulator. *ACM SIGARCH Computer Architecture News 39*, 2 (May 2011), 1–7.

[10] Chhabra, S., and Solihin, Y. i-NVMM: A Secure Non-Volatile Main Memory System with Incremental Encryption. In *International Symposium on Computer Architecture (ISCA)* (June 2011), pp. 177–188.

[11] Chishti, Z., Alameldeen, A. R., Wilkerson, C., Wu, W., and Lu, S.-L. Improving Cache Lifetime Reliability at Ultra-Low Voltages. In *International Symposium on Microarchitecture (MICRO)* (December 2009), pp. 89–99.

[12] Colp, P., Zhang, J., Gleeson, J., Suneja, S., Lara, E. D., Raj, H., Saroiu, S., and Wolman, A. Protecting Data on Smartphones and Tablets from Memory Attacks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2015), pp. 177–189.

[13] Crossbar. http://www.crossbar-inc.com/.

[14] Ding, Y., Kandemir, M., Irwin, M. J., and Raghavan, P. Adapting Application Mapping to Systematic Within-Die Process Variations on Chip Multiprocessors. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2009), vol. 5409 LNCS, pp. 231–247.

[15] Dong, X., Wu, X., Sun, G., Xie, Y., Li, H., and Chen, Y. Circuit and Microarchitecture Evaluation of 3D Stacking Magnetic RAM (MRAM) as a Universal Memory Replacement. In *Design Automation Conference (DAC)* (June 2008), pp. 554–559.

[16] Dong, X., Xu, C., Xie, Y., and Jouppi, N. P. NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 31*, 7 (July 2012), 994–1007.

[17] Dreslinski, R., Wieckowski, M., Blaauw, D., Sylvester, D., and Mudge, T. Near-Threshold Computing: Reclaiming Moore's Law Through Energy Efficient Integrated Circuits. *Proceedings of the IEEE 98*, 2 (February 2010), 253–266.

[18] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. Dark Silicon and the End of Multicore Scaling. In *International Symposium on Computer Architecture (ISCA)* (June 2011), pp. 365–376.

[19] Everspin. https://www.everspin.com/.

[20] Garcia, J., Montiel-Nelson, J., Sosa, J., and Nooshabadi, S. High Performance Single Supply CMOS Inverter Level Up Shifter for Multi-Supply Voltages Domains. In *Design Automation and Test in Europe (DATE)* (March 2015), pp. 1273–1276.

[21] Ghasemi, H. R., Draper, S., and Kim, N. S. Low-Voltage On-Chip Cache Architecture Using Heterogeneous Cell Sizes for High-Performance Processors. In *International Symposium on High Performance Computer Architecture (HPCA)* (February 2011), pp. 38–49.

[22] Goswami, N., Cao, B., and Li, T. Power-Performance Co-Optimization of Throughput Core Architecture Using Resistive Memory. In *International Symposium on High Performance Computer Architecture (HPCA)* (February 2013), pp. 342–353.

[23] Gotzfried, J., and Muller, T. ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices. In *International Conference on Availability, Reliability and Security (ARES)* (September 2013), pp. 161–168.

[24] Gotzfried, J., Muller, T., Nurnberger, S., and Backes, M. RamCrypt: Kernel-Based Address Space Encryption for User-Mode Processes. In *Asia Conference on Computer and Communications Security* (May 2016), pp. 919–924.

[25] Guan, L., Lin, J., Luo, B., and Jing, J. Copker: Computing with Private Keys without RAM. In *Annual Network and Distributed System Security Symposium (NDSS)* (February 2014), pp. 1–15.

[26] Guo, X., Ipek, E., and Soyata, T. Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing. In *International Symposium on Computer Architecture (ISCA)* (June 2010), pp. 371–382.

[27] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX Security Symposium* (July 2008), pp. 45–60.

[28] Henning, J. L. Performance Counters and Development of SPEC CPU2006. *ACM SIGARCH Computer Architecture News 35*, 1 (March 2007), 118–121.

[29] Henson, M., and Taylor, S. Memory Encryption: A Survey of Existing Techniques. *ACM Computing Surveys (CSUR) 46*, 4 (April 2014).

[30] Herbert, S., and Marculescu, D. Variation-Aware Dynamic Voltage/Frequency Scaling. In *International Symposium on High Performance Computer Architecture (HPCA)* (February 2009), pp. 301–312.

[31] Hewlett-Packard-Enterprise. The Machine. https://www.bloomberg.com/news/articles/2014-06-11/with-the-machine-hp-may-have-invented-a-new-kind-of-computer.

[32] Hong, S., Narayanan, S., Kandemir, M., and Ozturk, O. Process Variation Aware Thread Mapping for Chip Multiprocessors. In *Design Automation and Test in Europe (DATE)* (April 2009), pp. 821–826.

[33] Intel Core™ i7 Processor. `http://www.intel.com`.

[34] Intel Core i7-800 Processor Series and the Intel Core i5-700 Processor Series Based on Intel Microarchitecture (Nehalem). Intel White Paper, 2009. http://download.intel.com/products/processor/corei7/319724.pdf.

[35] Ipek, E., Condit, J., Nightingale, E. B., Burger, D., and Moscibroda, T. Dynamically Replicated Memory: Building Reliable Systems from Nanoscale Resistive Memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2010), pp. 3–14.

[36] International Technology Roadmap for Semiconductors (2009). http://www.itrs.net.

[37] Jaleel, A. A Pin-Based Memory Characterization of the SPEC CPU2000 and SPEC CPU2006 Benchmark Suites. http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf.

[38] Jog, A., Mishra, A. K., Xu, C., Xie, Y., Narayanan, V., Iyer, R., and Das, C. R. Cache Revive: Architecting Volatile STT-RAM Caches for Enhanced Performance in CMPs. In *Design Automation Conference (DAC)* (June 2012), pp. 243–252.

[39] Karpuzcu, U. R., Sinkar, A., Kim, N. S., and Torrellas, J. Toward Energy-Efficient Manycores for Near-Threshold Computing. In *International Symposium on High Performance Computer Architecture (HPCA)* (February 2013), pp. 542–553.

[40] Kvatinsky, S., Etsion, Y. H., Friedman, E. G., Kolodny, A., and Weiser, U. C. Memristor-Based Multithreading. *IEEE Computer Architecture Letters 13*, 1 (2013).

[41] Lee, B. C., Ipek, E., Mutlu, O., and Burger, D. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *International Symposium on Computer Architecture (ISCA)* (June 2009), pp. 2–13.

[42] Lee, R., Kwan, P., McGregor, J., Dwoskin, J., and Wang, Z. Architecture for Protecting Critical Secrets in Microprocessors. In *International Symposium on Computer Architecture (ISCA)* (June 2005), pp. 2–13.

[43] Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., and Jouppi, N. P. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *International Symposium on Microarchitecture (MICRO)* (December 2009), pp. 469–480.

[44] Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., and Horowitz, M. Architectural Support for Copy and Tamper Resistant Software. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (November 2000), pp. 168–177.

[45] Lukefahr, A., Padmanabha, S., Das, R., Sleiman, F. M., Dreslinski, R., Wenisch, T. F., and Mahlke, S. Composite Cores: Pushing Heterogeneity into a Core. In *International Symposium on Microarchitecture (MICRO)* (December 2012), pp. 317–328.

[46] Markovic, D., Wang, C., Alarcon, L., Liu, T.-T., and Rabaey, J. Ultralow-Power Design in Near-Threshold Region. *Proceedings of the IEEE 98*, 2 (February 2010), 237–252.

[47] Meena, J. S., Sze, S. M., Chand, U., and Tseng, T.-Y. Overview of Emerging Non-Volatile Memory Technologies. *Nanoscale Research Letters 9*, 526 (September 2014), 1–33.

[48] Micron. 2GB DDR3 SDRAM. https://www.micron.com/media/documents/products/data-sheet/dram/ddr3/2gb_ddr3_sdram.pdf.

[49] Micron-Intel. 3D XPoint™ Technology. https://www.micron.com/products/emerging-technologies/3d-xpoint-technology.

[50] Miller, T. N., Pan, X., Thomas, R., Sedaghati, N., and Teodorescu, R. Booster: Reactive Core Acceleration for Mitigating the Effects of Process Variation and Application Imbalance in Low-Voltage Chips. In *International Symposium on High Performance Computer Architecture (HPCA)* (February 2012), pp. 27–38.

[51] Miller, T. N., Thomas, R., Dinan, J., Adcock, B., and Teodorescu, R. Parichute: Generalized Turbocode-Based Error Correction for Near-Threshold Caches. In *International Symposium on Microarchitecture (MICRO)* (December 2010), pp. 351–362.

[52] Miller, T. N., Thomas, R., Pan, X., and Teodorescu, R. VRSync: Characterizing and Eliminating Synchronization-Induced Voltage Emergencies

in Many-Core Processors. In *International Symposium on Computer Architecture (ISCA)* (June 2012), pp. 249–260.

[53] Miller, T. N., Thomas, R., and Teodorescu, R. Mitigating the Effects of Process Variation in Ultra-low Voltage Chip Multiprocessors using Dual Supply Voltages and Half-Speed Stages. *IEEE Computer Architecture Letters 11*, 2 (2012).

[54] Muller, T., Dewald, A., and Freiling, F. C. AESSE: A Cold-Boot Resistant Implementation of AES. In *European Workshop on System Security* (April 2010), pp. 42–47.

[55] Muller, T., Freiling, F. C., and Dewald, A. TRESOR Runs Encryption Securely Outside RAM. In *USENIX Security Symposium* (August 2011), pp. 17–32.

[56] Muller, T., and Spreitzenbarth, M. FROST: Forensic Recovery of Scrambled Telephones. In *International Conference on Applied Cryptography and Network Security* (June 2013), pp. 373–388.

[57] Muralimanohar, N., Balasubramonian, R., and Jouppi, N. P. CACTI 6.0: A Tool to Model Large Caches. Tech. Rep. HPL-2009-85, HP Labs, 2009.

[58] Najaf-abadi, H. H., and Rotenberg, E. Architectural Contesting. In *International Symposium on High Performance Computer Architecture (HPCA)* (February 2009), pp. 189–200.

[59] Pabel, J. Frozen Cache. http://frozencache.blogspot.com.

[60] Pan, X., and Teodorescu, R. NVSleep: Using Non-Volatile Memory to Enable Fast Sleep/Wakeup of Idle Cores. In *International Conference on Computer Design (ICCD)* (October 2014), pp. 400–407.

[61] Pan, X., and Teodorescu, R. Using STT-RAM to Enable Energy-Efficient Near-Threshold Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)* (August 2014), pp. 485–486.

[62] Qureshi, M. K., Srinivasan, V., and Rivers, J. A. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *International Symposium on Computer Architecture (ISCA)* (June 2009), pp. 24–33.

[63] Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Strauss, K., Sarangi, S., Sack, P., and Montesinos, P. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[64] Riedlinger, R., Bhatia, R., Biro, L., Bowhill, B., Fetzer, E., Gronowski, P., and Grutkowski, T. A 32nm 3.1 Billion Transistor 12-Wide-Issue Itanium Processor for Mission-Critical Servers. In *International Solid-State Circuits Conference (ISSCC)* (February 2011), pp. 84–86.

[65] Rogers, B., Chhabra, S., Prvulovic, M., and Solihin, Y. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In *International Symposium on Microarchitecture (MICRO)* (December 2007), pp. 183–196.

[66] RSA-Laboratories. PKCS #1 v2.1: RSA Cryptography Standard. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf.

[67] Sakurai, T., and Newton, R. Alpha-Power Law MOSFET Model and its Applications to CMOS Inverter Delay and Other Formulas. *IEEE Journal of Solid-State Circuits 25*, 2 (April 1990), 584–594.

[68] Sarangi, S. R., Greskamp, B., Teodorescu, R., Nakano, J., Tiwari, A., and Torrellas, J. VARIUS: A Model of Parameter Variation and Resulting Timing Errors for Microarchitects. *IEEE Transactions on Semiconductor Manufacturing 21*, 1 (February 2008), 3–13.

[69] Sardashti, S., and Wood, D. A. UniFI: Leveraging Non-Volatile Memories for a Unified Fault Tolerence and Idle Power Management Technique. In *International Conference on Supercomputing (ICS)* (June 2012), pp. 59–68.

[70] Shelepov, D., Alcaide, J. C. S., Jeffery, S., Fedorova, A., Perez, N., Huang, Z. F., Blagodurov, S., and Kumar, V. HASS: A Scheduler for Heterogeneous Multicore Systems. *SIGOPS Operating Systems Review 43*, 2 (April 2009), 66–75.

[71] Simmons, P. Security through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption. In *Annual Computer Security Applications Conference* (December 2011), pp. 73–82.

[72] Smullen, C. W., Mohan, V., Nigam, A., Gurumurthi, S., and Stan, M. R. Relaxing Non-Volatility for Fast and Energy-Efficient STT-RAM Caches. In *International Symposium on High Performance Computer Architecture (HPCA)* (February 2011), pp. 50–61.

[73] Suh, G. E., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *International Symposium on Microarchitecture (MICRO)* (December 2003), pp. 339–350.

[74] Sun, G., Dong, X., Xie, Y., Li, J., and Chen, Y. A Novel Architecture of the 3D Stacked MRAM L2 Cache for CMPs. In *International Symposium on High Performance Computer Architecture (HPCA)* (February 2009), pp. 239–249.

[75] Sun, Z., Bi, X., Li, H., Wong, W.-F., Ong, Z.-L., Zhu, X., and Wu, W. Multi Retention Level STT-RAM Cache Designs with a Dynamic Refresh Scheme. In *International Symposium on Microarchitecture (MICRO)* (December 2011), pp. 329–338.

[76] Teodorescu, R., and Torrellas, J. Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors. In *International Symposium on Computer Architecture (ISCA)* (June 2008), pp. 363–374.

[77] Wang, Z., Jimenez, D. A., Xu, C., Sun, G., and Xie, Y. Adaptive Placement and Migration Policy for an STT-RAM-Based Hybrid Cache. In *International Symposium on High Performance Computer Architecture (HPCA)* (February 2014), pp. 13–24.

[78] Wu, X., Li, J., Zhang, L., Speight, E., Rajamony, R., and Xie, Y. Hybrid Cache Architecture with Disparate Memory Technologies. In *International Symposium on Computer Architecture (ISCA)* (June 2009), pp. 34–45.

[79] Xu, C., Niu, D., Muralimanohar, N., Balasubramonian, R., Zhang, T., Yu, S., and Xie, Y. Overcoming the Challenges of Crossbar Resistive Memory Architectures. In *International Symposium on High Performance Computer Architecture (HPCA)* (February 2015), pp. 476–488.

[80] Yitbarek, S. F., Aga, M. T., Das, R., and Austin, T. Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors. In *International Symposium on High Performance Computer Architecture (HPCA)* (February 2017), pp. 313–324.

[81] Young, V., Nair, P. J., and Qureshi, M. K. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2015), pp. 33–44.

[82] Zhai, B., Blaauw, D., Sylvester, D., and Hanson, S. A Sub-200mV 6T SRAM in 0.13μm CMOS. In *International Solid-State Circuits Conference (ISSCC)* (February 2007), pp. 332–606.

[83] Zhai, B., Dreslinski, R. G., Blaauw, D., Mudge, T., and Sylvester, D. Energy Efficient Near-Threshold Chip Multi-Processing. In *International Symposium on Low Power Electronics and Design (ISLPED)* (August 2007), pp. 32–37.

[84] Zhang, N., Sun, K., Lou, W., and Hou, Y. T. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *IEEE Symposium on Security and Privacy (SP)* (May 2016), pp. 72–90.

[85] Zhou, P., Zhao, B., Yang, J., and Zhang, Y. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *International Symposium on Computer Architecture (ISCA)* (June 2009), pp. 14–23.

[86] Zhou, P., Zhao, B., Yang, J., and Zhang, Y. Energy Reduction for STT-RAM Using Early Write Termination. In *International Conference on Computer-aided Design (ICCAD)* (November 2009), pp. 264–268.