Design Strategies for Efficient and Secure Memory

by

Tamara Silbergleit Lehman

Department of Electrical and Computer Engineering
Duke University

Date: _____

Approved:

_____
Benjamin C. Lee, Advisor

_____
Andrew D. Hilton

_____
Daniel J. Sorin

_____
Siddhartha Chhabra

_____
Ashwin Machanavajjhala

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the Graduate School of
Duke University

2019

Abstract

Design Strategies for Efficient and Secure Memory

by

Tamara Silbergleit Lehman

Department of Electrical and Computer Engineering
Duke University

Date: _____

Approved:

_____
Benjamin C. Lee, Advisor

_____
Andrew D. Hilton

_____
Daniel J. Sorin

_____
Siddhartha Chhabra

_____
Ashwin Machanavajjhala

An abstract of a dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the Graduate School of
Duke University

2019

Abstract

Recent computing trends force users to relinquish physical control to unknown parties, making the system vulnerable to physical attacks. Software alone is not well equipped to protect against physical attacks. Instead software and hardware have to enforce security in collaboration to defend against physical attacks. Many secure processor implementations have surfaced over the last two decades (*i.e.* Intel SGX, ARM Trustzone) but inefficiencies are hindering their adoption.

Secure processors use secure memory to detect and guard against physical attacks. Secure memory assumes that everything within the chip boundary is trusted and provides confidentiality and integrity verification for data in memory. Both of these features, confidentiality and integrity, require large metadata structures which are stored in memory. When a system equipped with secure memory misses at the last-level-cache (LLC), the memory controller has to issue additional memory requests to fetch the corresponding metadata from memory. These additional memory requests increase delay and energy. The main goal of this dissertation is to reduce overheads of secure memory in two dimensions: delay and energy.

First, to reduce the delay overhead we propose the first safe speculative integrity verification mechanism, PoisonIvy, that effectively hides the integrity verification latency while maintaining security guarantees. Secure memory has high delay overheads due to the long integrity verification latency. Speculation allows the system to return decrypted data back to the processor before the integrity verification completes, effectively removing the integrity verification latency from the critical path of a memory access. However, speculation without any other mechanism to safeguard security is unsafe. PoisonIvy safeguards security guarantees by preventing any effect of unverified data from leaving the trusted boundary. PoisonIvy is able to realize all

the benefits of speculative integrity verification while maintaining the same security guarantees as the non-speculative system.

Speculation is effective in reducing delay overhead but it has no effect on reducing the number of additional memory accesses, which cause large energy overhead. Secure memory metadata has unique memory access patterns that are not compatible with traditional cache designs. In the second part of this work, we provide the first in-depth study of metadata access patterns, MAPS, to help guide architects design more efficient cache architectures customized for secure memory metadata.

Based on the unique characteristics of secure memory metadata observed in the in-depth analysis, in the third part of this work we explore the design space of efficient cache designs. We describe one possible design, Metadata Cache eXtension (MCX), which exploits the bimodal reuse distance distribution of metadata blocks to improve the cache efficiency thereby reducing the number of additional memory accesses. We also explore an LLC eviction policy suitable to handle multiple types of blocks to improve the efficiency of caching metadata blocks on-chip further.

Acknowledgements

First and foremost I would like to acknowledge that the work in this dissertation is the product of a successful collaboration with my advisers Professor Benjamin C. Lee and Professor Andrew D. Hilton. I would like to thank my advisers, Professor Andrew D. Hilton and Professor Benjamin C. Lee, who from the beginning of my time at Duke, they have supported me and encouraged me to push myself to higher grounds. Regardless of my inexperience in Computer Engineering, Professor Hilton and Professor Lee have both welcomed me into the department and have shown me the path to success. Through classes and thoughtful discussions, they have both taken the time to help me improve my technical skills as well as my writing and public speaking skills. They have both showed me that I could succeed in this field as long as I was passionate and willing to learn. Words are not enough to express my gratitude to both of my advisers, Ben and Drew.

I would also like to thank the members if my committee, Professor Daniel Sorin, Dr. Siddhartha Chhabra and Professor Ashwin Machanavajjhala for their thoughtful input on my work. Each an every one of the members of my committee have supported me in many different ways and I would not be where I am today without their help.

I would also like to thank my friends and labmates whom have provided the critical support one needs as a PhD student: Dr. Qiuyun Llull, Dr Oppeoluwa Matthews, Dr. Songchun Fan, Dr. Ziqiang Huang, Atefeh Mehrabi and the many more labmates I have met since starting at Duke. My friends and labmates have been with me through my ups and downs, they have encouraged and supported me throughout these amazing years.

Finally, I would also like to thank my husband, Jon, who throughout my career has always been flexible and supportive. I would certainly not be here today if it

were not for him. He is the one that encouraged me to apply to the Masters program at Duke University, and it is thanks to his flexibility in life that we were able to move to Durham, NC. He has pushed me when I needed pushing, he has comforted me when I needed comforting, and most importantly, he has loved me unconditionally. Thank you for being with me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As more of our data becomes digital we become more vulnerable to cyber-attacks. The largest data breach seen so far was at Yahoo in 2013 [57], which affected a total of 3 billion people. A study done in 2018 [30] estimates that data breaches are only getting bigger, growing at a rate of 2% per attack. That same study estimates that data breaches costs companies an average of $3.9 million. Given the increasing impact of cyber-attacks, the computing industry has been producing new solutions to secure our data. Specifically, in the last decade, we have seen a surge of hardware security solutions.

In addition to the increasing impact of cyber-attacks, modern computing models provide physical access to untrusted entities, such as datacenter operators, who might examine sensitive code or data. Without physical control, users are vulnerable to attacks in which a malicious party inspects the physical connection of hardware components to either spy or modify the behavior to gain unauthorized access to the system. Indeed, such security concerns dampen enthusiasm for computing in the cloud [3, 61].

Software alone is not capable of protecting against physical attacks. Robust solutions against these types of attacks have to involve both software and hardware. Hardware mechanisms that guard against physical attacks must ensure two properties—privacy and integrity. For privacy, hardware must encrypt data whenever it crosses the processor chip boundary. The memory controller encrypts and decrypts data when it is written to and read from memory. The state-of-the-art in memory encryption combines one-time-pads (OTP) and counter-mode encryption [93].

For integrity, hardware must determine whether an adversary has altered the data since it was written to memory. A naïve mechanism computes hashes over the data when writing and verifies them when reading [46, 47, 78, 93–95]. However, hashes alone cannot guard against adversaries who record and replay old data and its hash [20]. Security against

replay attacks require Merkle Trees [49]. The tree leaves hash data, the next tree level hashes leaves, and so on. The root is on chip, which ensures its integrity.

Bonsai Merkle Trees (BMTs) reduce Merkle Tree overheads by differentiating between the encryption counters and data [63]. A BMT uses hierarchical hashes to protect counters employed in counter-mode encryption from replay attacks. And it uses simple hashes to protect data from tampering. BMTs ensure integrity with much less metadata than Merkle Trees.

1.1 Threat Model

We focus on a well studied threat model in which an attacker has physical access to the machine. When the processor is the trusted computing base, it must safeguard its interactions with off-chip memory. It ensures confidentiality with encryption and ensures integrity with keyed-hash message authentication codes (HMAC) [41].

Security solutions that involve both the hardware and the software are equipped to defend against both *physical attacks and malicious privilege software attacks*. Physical attacks are enabled when users relinquish physical control over the machine to untrusted parties [46, 47]. A malicious user in physical control over the machine can exploit characteristics of traditional systems to steal secrets or gain unauthorized access to the system. Traditional systems connect the processor and main memory by a simple data bus that sends information back and forth in the clear [23]. Attackers can intercept this connection to either steal secrets that travel through the bus or inject values to modify the system's behavior.

Malicious privilege software attacks exploit the higher capabilities of privilege software to steal or modify data in memory to gain unauthorized access to the system. Privilege software, such as operating systems (OS) or virtual machine monitors (VMM), run at higher privilege levels to be able to handle the metadata needed to run applications—such as the memory management unit and the system scheduler. However, malicious users have been

able to exploit the higher privilege capabilities to mount attacks. An example of this type of attack is a privilege escalation attack (*i.e.* jail-breaking), in which a malicious user exploits a bug in software to escalate their application to have same privileges as the OS [14].

Today's computing models make systems more vulnerable to these classes of attacks. Several new computing models—such as cloud, gaming and mobile computing—force users to both relinquish physical control over the hardware and also share resources with unknown parties. To defend against these attacks, the system requires a robust cryptographic platform that allows the system to attest to its own security features.

1.2 Remote Attestation

Without physical control, secure processors require a way to attest its identity. This challenge is also found in networking systems, in which two parties want to communicate securely and do not know how to guarantee their identities. How does Alice know that it is talking in fact with Bob if Alice is not able to see Bob? The answer is to use asymmetric cryptography [15].

Asymmetric cryptography provides a robust algorithm, based on number factorization, to verify the true identity of an unknown party [15]. The unknown party contains a combination of two cryptographic keys: the private key (known only to the owner) and a public key (distributed to authorized users). Data encrypted with the public key can only be decrypted with the private key, known only to the owner of the keys. In this way, if Alice wants only Bob to read her message, she can use Bob's public key to encrypt the message. When Bob receives the message, he can decrypt it with his private key and read the message that Alice sent. If a malicious user, Mallory, intercepts the connection, she would not be able to read Alice's message without the private key.

Secure processors use asymmetric cryptography to provide remote attestation capabilities. Secure processors need a hardware cryptographic environment that can attest to the security of the system. The remote attestation service allows users to validate

system's security mechanisms without physical access. The remote secure system has a machine-only readable private cryptographic key embedded on the processor at manufacturing time [2, 9, 79]. A public key is distributed to users, and they can use the public key to encrypt data before sending it to the secure processor.

An application's owner uses a tool chain to protect her binary before sending it to a vulnerable system. While industry-strength tools, such as those of Intel's SGX, are complex, they can be broken down into a simplified model that has three steps [2, 12]. First, the tool creates a random symmetric key and encrypts the binary image using Advanced Encryption Standard (AES) [53]. The encrypted image includes both code and data. Second, the tool encrypts the symmetric key with the public key of the target system using Rivest, Shamir, Adleman (RSA) [62]. Finally, the symmetric-encrypted image and the asymmetric-encrypted key are paired and sent to the system for computation.

The secure processor runs the application by unpacking the image in two steps. First, the processor uses its matching asymmetric private key to decrypt the symmetric key. With the symmetric key, the processor decrypts code and data as they are read from disk, and re-encrypts them with its own key, then writes them to memory. A separate symmetric key, known and used only by the processor, is used to read data from memory into caches and vice-versa. When code and data are swapped out, the processor decrypts it with its own key, re-encrypts it with the application symmetric key, and writes it to disk.

Once the remote system has attested to its security mechanisms, runtime data also has to be protected. Secure processors assume that the trusted boundary is the chip boundary, and protects data when it goes off-chip. To protect the data off-chip, the secure system has to use secure memory to provide both data confidentiality and integrity in memory.

1.3 Data Confidentiality

Keyed encryption is required for confidentiality, which prevents adversaries from observing data. The processor could use XTS-mode[1], which combines data with portions of its memory address before encrypting with AES. However, XTS serializes data access and decryption, lengthening the critical path to return data to the processor—AES latency ranges around 40 processor cycles. Skylake's AESDEC performs one round in 4 processor cycles and performs the 10-14 required rounds in 40-56 cycles, depending on key length [22].

Counter-Mode Encryption. Alternatively, counter-mode encryption (CME) allows the system to parallelize the encryption and the memory fetch. CME XORes plaintext with a one-time pad (OTP) to produce the ciphertext. The OTP is generated by encrypting a combination of a counter and the memory address with AES [78,93,94]. During decryption, the ciphertext is XORed with the same OTP to produce the plaintext. If counters are cached, the processor can generate the OTP for decryption and retrieve the ciphertext from memory in parallel. In this situation, the decryption latency overhead is only the XOR latency, which is one cycle long.

To preserve security of CME, the OTP, and by extension the counter, has to be used only once. The counter corresponding to a memory address is incremented after every write. If a counter were to overflow, the symmetric key used to encrypt counters and produce OTPs must be regenerated. Then, counters must be reset and memory contents must be re-encrypted. Because counter overflow is expensive, best practices increment counters at multiple data granularity.

Block and Page Counters. Large counters avoid overflow and expensive re-encryption, but small counters fit in cache more easily. Researchers resolve this tension with separate counters for memory blocks and pages [63,93]. Every block write increments the per-block counter. Overflow in the per-block counter increments the per-page counter corresponding to the block's page. Finally, each block's input to OTP generation is the concatenation of per-block and per-page counters.

[1]Xor Encrypt Xor (XEX) Tweakable Block Cipher with Ciphertext Stealing

Block and page counters reduce the frequency and cost of re-encryption. When the per-page counter increments, the memory controller loads each block from the page, decrypts it, and re-encrypts it with a new counter that concatenates the incremented per-page counter and a reset per-block counter. Although page re-encryption is moderately expensive, it happens infrequently and is much less expensive than re-encrypting all memory contents. The combined counter (7-bit per-block and 64-bit per-page counters) never overflows in practice. It would overflow in 75,000 years if the same block were written once per nanosecond.

Confidentiality prevents an attacker from stealing secrets when they leave the chip, which defines the boundary of the trusted computing base. The memory controller achieves confidentiality by encrypting data when it is stored off-chip. However, confidentiality on data alone is unable to provide authentication capabilities. An integrity verification mechanism is necessary for data authentication.

1.4 Data Integrity Verification

Integrity mechanisms prevent adversaries from modifying data without a user's knowledge. One approach uses hashes to verify memory integrity [17, 46, 47, 78, 94, 95]. For every write, the memory controller computes and stores a secure hash, a keyed-hash message authentication code (HMAC) of the data [41]. For every read, it loads and compares the previously computed hash against the current one. When hashes are equal, integrity is verified. When hashes differ, the system detects tampering and halts.

Hashes alone do not guard against replay attacks in which adversaries inject previously observed data and hashes onto the memory channel. As illustrated in Figure 1.1, Merkle Trees guard against replay attacks by building a hash tree over memory contents [49]. Leaves of the tree contain one secure hash per data block. Further up, each tree node hashes its children's hashes. The tree's root holds a secure hash that encompasses the whole protected memory space. The tree's root is held on chip, where it cannot be attacked, and

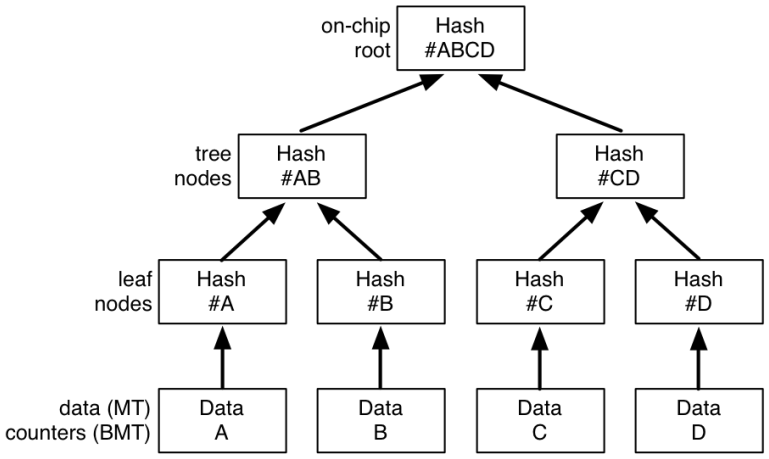

Figure 1.1: Merkle Tree Diagram

is always trusted. A data write propagates hash writes up the tree to the root.

Bonsai Merkle Trees (BMTs) guard against replay attacks at lower cost than Merkle Trees. BMTs construct a hash tree to protect only the counters used to generate OTPs instead of the entire memory space. This approach reduces space overheads from 50% to 14% [63], assuming 8-byte hashes and tree arity of eight. Moreover, BMTs perform better because counters are often cached on chip and need not be verified by the tree. Suppose the processor requests data and its corresponding counter is cached, the counter's integrity was verified when it was first loaded from memory and BMT (re-)traversal is unnecessary. Given BMTs to protect counters, simple keyed hashes are sufficient to protect the integrity of the data [2, 10, 19, 79].

1.5 Secure Memory Performance Optimizations

Security adds overheads, motivating performance optimizations that cache metadata and speculate around the integrity verification [20,64,65,73,74,78,94]. We survey recent progress to motivate our approaches, which builds atop best practices to address remaining performance challenges.

Secure memory adds large overheads on both latency and energy. For every memory

access, the memory controller must decrypt data and verify the integrity of both data and the counters. To decrypt data, the memory controller fetches the corresponding counter from memory. To verify the counter's integrity, the controller traverses the integrity tree up to the root by fetching one hash from memory at each level. And finally, to verify the data's integrity, the memory controller fetches the data hash from memory.

Despite recent advances, security mechanisms are expensive and harm performance as they manipulate several types of metadata, which are stored in memory along with data. When requesting data, the memory controller must also request counters for decryption and hashes for verification. In the worst case, a last-level cache miss could trigger up to ten memory transfers—counters for decryption and hashes from the tree, up to the root, for integrity verification—significantly increasing request latency and bandwidth demand.

1.5.1 Caching Metadata Blocks

The secure processor can cache metadata blocks on-chip to alleviate additional memory accesses. Metadata caches can have multiplicative effects, as they not only can save the memory access for the block resident in the cache but it can also shorten the tree traversal of the integrity verification. When a metadata block is found in the cache, the integrity of that block was verified when it was inserted into the cache, therefore the integrity tree traversal can stop right away.

Metadata caches not only mitigate performance overheads but they can also greatly diminish the energy overheads incurred by the additional memory accesses. A memory access consumes several orders of magnitude more energy than an on-chip SRAM access. Metadata caches play an important role in reducing secure memory overheads.

Prior work has explored different metadata caching designs. Some prior academic designs place metadata in the last-level-cache (LLC) along with the data [20, 68, 78, 79, 94], while industry and a few other academic designs use a dedicated cache in the memory controller [2, 10, 21, 29, 63, 64, 67, 73, 81, 92, 93].

Despite the promise that metadata caches seem to have, caching metadata is inefficient.

Even with caching metadata in a 128KB separate cache, we find that security harms performance by 9% on average and 65% in the worst case based on our measurements. On energy, security increases energy by 15% on average and 128% in the worst case. Surprisingly, no prior work, to the best of our knowledge, has done an extensive study of metadata access patterns to understand where these metadata cache inefficiencies come from.

1.5.2 Speculating Around Integrity Verification

On the other hand, speculative integrity verification effectively hides the verification latency, reducing delay overheads of secure memory significantly. On average a speculative system reduces delay overheads down to 2% on average by supplying data to the core for computation before retrieving its metadata and verifying its integrity [63]. Some academic mechanisms speculate quite eagerly, using data immediately and assuming integrity violations are detected before they produce any ill effects [63, 79]. Others, such as `authen-then-write`, speculate more carefully for smaller, 6% on average, yet still significant performance benefits [72]—note that in this work the performance overheads reported are higher (14%) because they assume a much smaller metadata cache, 8KB instead of 128KB.

However, speculation has a number of limitations despite academic progress and demonstrated potential. To date, none of the proposed mechanisms are completely safe and all of them open an avenue of attack that breaches the security guarantees. Even the conservative `authen-then-write` mechanism is vulnerable to side-channel attacks, which are identified in the same study [72]. Industrial designs, such as Intel's Software Guard eXtensions (SGX) [2], rely primarily on caches, not speculation, to balance security and performance. To encourage the adoption of speculative mechanisms, we must close its security vulnerabilities.

1.6 A Survey of Secure Hardware Solutions

To position and motivate our work, we explain the origins of secure hardware and survey the state-of-the-art implementations in both academia and industry. Most implementations described in this section benefit from the design strategies presented in this dissertation.

The original goal of secure hardware was to help protect and manage digital rights. Particularly, video-game developers found users breaking into gaming consoles to steal their intellectual property and cheat the system. Users were able to accomplish such attacks because they had physical control of the consoles running the software. Software alone is unable to protect against physical attacks, instead the hardware needs a to provide a robust security environment.

The eXecute Only Memory (XOM) machine [46] is the first academic work that introduced the idea of a secure execution environment. The XOM machine proposes new instructions to load and store data in secure memory. Several academic improvements followed the XOM machine, such as AEGIS [79] among others [20, 47, 78, 94]. We describe in detail the most relevant academic work for this dissertation:

AEGIS. [79] Inspired by the XOM machine, Suh *et.al.* improves the encryption and integrity guarantees of the tamper resistant environment. To protect the system against physical attacks, AEGIS uses AES-Cipher Block Chaining (CBC) for encryption and a Merkle tree with 128-bit MAC for integrity. AEGIS caches tree nodes in the last-level cache (LLC) along with data. In addition, it allows the processor to compute on unverified data and assumes that, if integrity fails, the system will halt before leaking information—an assumption we call unsafe speculation.

AISE. [63] Rogers *et.al.* protect a system from physical attacks with a modified encryption and integrity mechanisms. For encryption, they use AES-counter mode with two counters: a 64-bit counter per 4KB page and a 7-bit counter per 512-bit blocks. The counters are concatenated when producing the one-time pad. For integrity, they modify the traditional Merkle tree to produce the Bonsai Merkle Tree. BMTs protect counters, instead of the data, thereby shortening the tree height and decreasing its size. Counters

are cached in the LLC along with data. This scheme uses unsafe speculation as well.

New computing trends, such as cloud computing, made other industries vulnerable to physical attacks and more robust software attacks. These trends require users to relinquish physical control and share resources with untrusted parties. The increasing vulnerability of these types of attacks is why in recent years we have seen more hardware companies release their own secure hardware implementations. Below is a brief survey of the state-of-the-art in industrial secure processors.

IBM SecureBlue and SecureBlue++. [9, 86] SecureBlue is the first industrial implementation of a secure processor, released by IBM in 2004. SecureBlue provides both confidentiality and authentication of the data in memory. SecureBlue++, which is an updated implementation of the secure processor, separates the integrity tree over the virtual memory space, protecting each process's memory from other software running in the same machine. The process management metadata is tightly integrated with the security metadata. The particular details of the hardware implementation are a well guarded secret within the company and it is unclear what type of caching mechanism they use and they make no mention of any speculation mechanism. Today, SecureBlue is used in tens of millions of systems, although the specific details are confidential [8].

Intel SGX. [2, 21] Software Guard eXtensions (SGX) provides secure regions of memory (*i.e.* enclaves) for programmer managed secrets. Instruction set extensions allow programmers to create, load, and enter an enclave. The Memory Encryption Engine (MEE), a micro-architectural extension to the memory controller, encrypts and hashes enclave contents to ensure confidentiality and integrity. SGX also uses counter mode encryption but unlike the academic work AISE, it only uses one 8B counter per 64B block. For the integrity tree, SGX uses the Carter Wegman hashing algorithm, with 8B hashes. SGX only provides up to 128MB of secure memory to the system. With the reduced size of protected memory, the integrity tree only has four levels, reducing the overheads of the integrity verification mechanism but introducing large paging overheads instead.

ARM TrustZone. [27] TrustZone only provides ISA extensions to enable a secure

software environment, the secure world. ARM TrustZone describes a secure processor that is active only when the secure world is enabled. ARM recommends their customers combine TrustZone capabilities with secure memory to be able to protect the system against physical attacks as well.

AMD SME and SEV. [37] Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV) provide only confidentiality—no integrity verification—for data in memory. SME and SEV differ only on what structure is in charge of the cryptographic keys used for encrypting data in memory. SME provides the capability of creating a system-wide encryption key at boot time, which is used to encrypt all of the memory contents. SEV provides virtual machines the capability of selecting certain regions of memory to be encrypted by a virtual machine owned cryptographic key.

While all these solutions rely in some form of secure memory to protect the system against physical attacks, we find that both confidentiality and authentication are necessary to fully protect against all types of physical attacks. In this work we focus on a secure memory implementation that is most similar with Intel SGX and AISE. When applicable we will point out the differences and explain how they relate to the results presented.

1.7 Contributions

In this dissertation we focus on reducing overhead of secure processors by architecting systems that account for both security and efficiency. We revisit micro-architectural support for secure memory to provide new perspectives on performance and energy efficiency while maintaining security guarantees. We build on prior work [20, 63], adapting it to reflect modern software demands and hardware design priorities.

Computing speculatively on data that has not yet been verified hides latency. A memory load could trigger metadata transfers that require hundreds of cycles. Prior studies supply unverified data for computation, assuming that an adversary cannot exploit this window of vulnerability [10,63,65,92]. Unfortunately, an adversary could stream data from

memory and preempt metadata needed for verification indefinitely. Pipeline stalls for verification ensures security but harms performance [72]. In Chapter 2, we present the first speculation mechanism for integrity verification, PoisonIvy [42], that guarantee security while still enhancing performance.

Moreover, software often stresses memory, which increases both energy and delay overheads when adding an integrity mechanism in the critical path to return data from memory. We find that integrity increases the energy delay squared product (ED^2) by 10% (average) and 400% (worst case) for applications with large memory footprints, even when an integrity tree's metadata is held in a 256KB dedicated cache (data shown in Figure 3.2). Integrity mechanisms require several types of metadata and transferring data from main memory to the processor is expensive. Each additional DRAM access requires approximately 150 pJ per bit read [48]. Yet prior studies evaluate delay and neglect energy. Furthermore, prior work considers less memory-intensive applications and assume speculative integrity verification, which show penalties of 1.8% (average) and 13% (worst case) [63]. To better understand where overheads come from, in Chapter 3, we present the first in-depth analysis of metadata access patterns, MAPS [43]. This work is aimed to aid architects to address the inefficiencies of caching metadata and as a guideline for future metadata caching designs.

Finally, based on observations of our in-depth analysis, in Chapter 4, we explore the design space for new cache architectures that address inefficiencies. Prior work assumes that traditional cache architectures are sufficient to hold metadata blocks efficiently. Unfortunately, metadata access patterns present new challenges for cache design. Metadata blocks have bimodal reuse distances: either smaller than 8KB or larger than 2MB. This large discrepancy complicates metadata cache sizing decisions. We present a design space exploration, which addresses the limitations of traditional cache architectures by considering the particular characteristics of metadata access patterns.

1.8 Experimental Methodology

Throughout this dissertation work we used the following experimental methodology:

Security Architecture. We evaluate all pieces with a state-of-the-art security configuration. First, we assume that all of memory is protected. Second, we use counter-mode encryption with a 7-bit counter for each 64B-block and an 8-byte counter for each 4KB-page [2]. Third, we use an 8-arity Bonsai Merkle integrity tree, in which each node contains eight 8-byte hashes of its children, for verifying the integrity of encryption counters [63]. Because we assume that all 4GB of memory are protected, the integrity tree has a total of 8 levels. Finally, we use 8-byte hashes per 64B-block for verifying the integrity of data. All metadata is stored in memory except for the integrity tree's root, which is stored within the processor. Moreover, all types of metadata are eligible for caching except where noted. We assume a system that uses AES for encryption and HMAC with SHA-2 for all hashes and truncate the resulting hash to 8B.

Table 1.1: Simulation Configuration

Pipeline width	4 (dispatch+commit) 8 (issue)
ROB size	192
Issue Queue size	60
Store buffer size	42
Load queue size	72
Clock Frequency	3GHz
L1 I & D Cache	32KB 8-way
L2 Cache	256KB 8-way
L3 Cache	2MB 8-way
Memory Size	4GB
Hash Latency	40 processor cycles
Hash Throughput	1 per DRAM cycle

Simulators. We combine MARSSx86 and DRAMSim2 for cycle-level, full-system simulation of the processor and memory [54, 66]. Table 1.1 summarizes simulation parameters. We use a 2MB LLC to match the per-core LLC capacity seen in commercial muti-core processors (*e.g.*, 8MB for 4 cores [31]). We modify the memory controller model to implement metadata caches. We also implement a timing model to estimate costs of data en/decryp-

[2] Even though Intel SGX structures encryption counters differently, with 8B counters per 64B-block, our findings remain relevant [21].

tion and integrity verification. All experiments include stream prefetching at the LLC. We evaluate processor energy with McPAT [44], metadata cache energy with CACTI [75], and memory energy with DRAMSim2.

Workloads. We evaluate all benchmarks in PARSEC [7], SPLASH2 [90] and SPEC-CPU2006 [24]. We simulate regions of interest for PARSEC and SPLASH. We fast-forward 1B user instructions for SPEC. For all three suites, we run simulations for 500M user instructions, after warming up the simulated structures for 50M user instructions. We present results for memory-intensive benchmarks that have more than 5 misses per thousand of instructions (MPKI) in the LLC. This set of benchmarks contains diverse access patterns. Graphs show results with a 16KB metadata cache unless otherwise specified. The averages are labeled as follows: Parsec average (*parAvg*), SPEC average (*spcAvg*) and SPLASH2 average (*sphAvg*). We also present the geometric average for memory-intensive benchmarks (*miAvg*) and the geometric average for all benchmarks (*allAvg*). The working sets for memory-intensive benchmarks range from 16MB to 2GB.

Chapter 2

Reducing Delay Overhead with PoisonIvy

Encryption and integrity trees guard against physical attacks, but harm performance. Prior academic work has speculated around the latency of integrity verification, but has done so in an insecure manner. As far as we know, no industrial implementations of secure processors have included speculation. This work presents `PoisonIvy` [42], a mechanism which speculatively uses data before its integrity has been verified while preserving security and closing address-based side-channels. `PoisonIvy` reduces performance overheads from 40% to 20% for memory intensive workloads and down to 1.8%, on average when combined with a 32KB metadata cache.

2.1 Unsafe Speculative Integrity Verification

Security increases average memory access time. Data en route from memory to the last-level cache must wait for decryption plus integrity verification, which may need data hashes and tree nodes from memory. Speculating on integrity removes the second set of overheads from the critical path. The memory controller supplies data to the processor before verifying integrity, permitting computation on decrypted but unverified data.

Speculation exposes tensions between performance and security. For performance, speculation requires a permissive mechanism that computes far ahead of verification to hide its latency, which is tens of cycles when counters are cached but hundreds (to thousands) of cycles when counters and tree nodes are in memory. Unfortunately, instructions will compute on unverified data, posing risks to confidentiality and integrity.

For security, speculation must restrict the visibility of computation on unverified data. A conservative mechanism for speculation does not modify architected state [72], but it performs poorly as the instruction window fills quickly and exposes the long verification

latency. Furthermore, guarding architected state alone is insufficient because speculative computation may issue loads and stores that send unencrypted addresses across the memory channel.

Prior work delivers performance or security but rarely both. A less conservative approach allows the memory controller to return unverified data to the core, assuming that the window of vulnerability between data use and verification is too short for attackers to exploit [10, 63, 65, 79, 92]. Shi *et.al.*, close this vulnerability by stalling stores until all outstanding verifications complete ("`authen-then-write`") [72]. This approach prevents an application from being "tricked" into writing data to memory, but puts pressure on the store buffer, constrains speculation, and exposes verification latency. Although researchers have sought to scale store buffer capacity [69–71, 77], no modern processor has a large enough store buffer to tolerate thousands of cycles of latency.

Side-Channel Attacks. Shi *et.al.* present a series of side-channel attacks, in a system that speculates, that leak secrets before verification fails [72]. These attacks exploit the fact that, when using OTP encryption, each bit of ciphertext corresponds to exactly one bit of plaintext. If an attacker wishes to flip a specific bit of plaintext, she can flip the corresponding ciphertext bit without breaking any cryptographic system.

In one attack, Shi *et.al.*, assumes that the attacker knows the contents of instruction memory (*e.g.*, has a copy of the program). First, the attacker alters bits in instruction words to cause secret data to be used in a load's address, taking care to place the secret into bits unaffected by address translation but not within the block offset. Then, the attacker observes addresses on the memory bus to learn the secret. Other attacks include manipulating pointers in linked data structures, and searching for a secret value by repeatedly altering constants that are compared to the secret. These attacks show that safe speculation must safeguard both data and instructions.

Some might argue that changes to the encryption algorithm could close these side-channels and permit safe speculation. Although XTS-mode encryption avoids vulnerabilities from OTPs, it penalizes performance by serializing data access and decryption.

Diffusive ciphers avoid the one-to-one mapping of ciphertext and plaintext bits, ensuring a bit flip in ciphertext changes multiple bits in plaintext. However, diffusion only reduces odds of a successful attack. If an attacker targets N bits—and does not care about the remaining $(512 - N)$ bits—in a cache block, diffusion reduces the odds of success to 1 in 2^N. For example, 1 in 256 attacks are successful when 8 bits are targeted, which is far from secure.

2.2 Safe Speculation with PoisonIvy

`PoisonIvy` makes speculation safe with a few key principles. First, unverified data may be used for any purpose within the processor, but it cannot affect any information leaving the processor before its integrity is verified. Second, any instruction affected by computation on unverified data is poisoned by the speculation. Instructions could be affected via input register values, preceding instruction words, preceding control flow, or values used in address translation. Speculation built around these principles is not only safe but also efficient. The only operations blocked by integrity verification are those that require off-chip communication.

`PoisonIvy`'s fundamental mechanism is *poison*, which is inspired by the key mechanism of Continual Flow pipelines (CFP) [76]. CFP poison bits indicate a load's value is missing due to a long-latency cache miss. In contrast, `PoisonIvy`'s poison bits indicate an unverified value is being used speculatively. The microarchitecture uses poison to determine what information must be restricted to the processor.

Whereas CFP requires a mechanism to recover from misspeculation, such as checkpoints, `PoisonIvy` does not expect to recover from misspeculation. Computing on data for which integrity verification fails is a drastic problem, indicating a physical attack against the system. The only safe outcome is halting the system. Indeed, Intel SGX halts the system whenever verification fails even though it does not speculate.

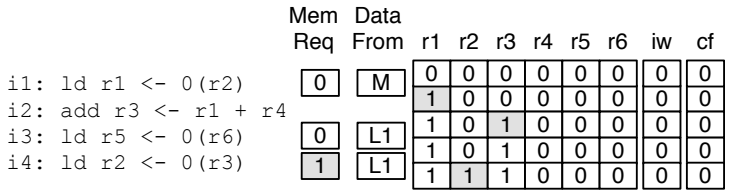

Figure 2.1: Propagation of the memory request poison bit

2.2.1 Propagating Poison

Whenever the memory controller has data to handle an LLC miss but cannot verify it immediately, it returns the data to the LLC and core speculatively. When a cache receives speculative data, it marks the line as poisoned. When the core receives speculative data, it poisons the output of the load that requested it. This poison propagates, marking instructions and data that are affected by the unverified value.

Registers. Poison bits propagate through registers as in CFP. When an instruction reads its input registers (or bypasses), it reads the corresponding poison bits. It then ORs these poison bits with the instruction's Instruction Word (IW) and Control Flow (CF) poison bits, which are discussed next, to determine whether its output register is poisoned. Then, the poison information for its output register is propagated on the bypass network and written into the register file along with the instruction's output value. In out-of-order processors with register renaming, we require one poison bit per physical register. The poison bits for all registers are cleared once all outstanding verifications are completed successfully.

Figure 2.1 shows an example in which the first instruction, `i1`, misses at the LLC and memory returns data speculatively, setting the poison bit on the output register, `r1`. When the second instruction, `i2`, reads its input registers, it finds that `r1` is poisoned, so it poisons its output register, `r3`. `r3`, the input register of the fourth instruction, `i4`, is poisoned, so `i4`'s request to the memory hierarchy is also marked as poisoned. If the request misses at the LLC, the memory controller stalls the request until verification completes.

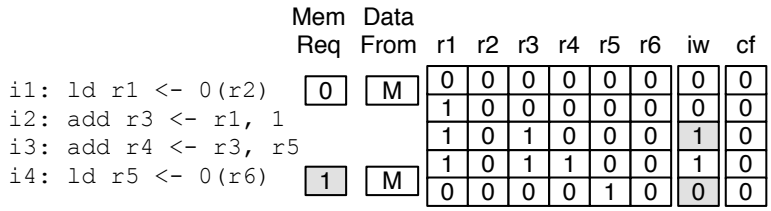

Figure 2.2: Popagation of the instruction word poison bit

Instruction Word. Whereas CFP cannot speculate around an instruction cache miss and has no notion of poisoning instruction words, `PoisonIvy` must track poisoned instructions with an instruction word (IW) poison bit. If a fetch misses all cache levels, memory speculatively supplies a poisoned instruction word. An attacker may have tampered with the memory blocks holding these words and computation with these instructions cannot be trusted until verification completes. The IW poison bit is set at the processor front-end and cleared only when outstanding speculations are verified.

Figure 2.2 shows an example in which the IW poison bit is set. Instruction `i2` is retrieved from memory and the instruction cache miss is resolved with speculative data. This instruction and all subsequent ones carry IW poison along with them through the pipeline. When a memory request misses at the LLC, like `i4`, the memory controller stalls the request until verification completes. At this point all poison bits, including IW, are cleared. Register `r5` is now poisoned by the newly speculative load returned from memory.

After fetching an instruction speculatively, all subsequent fetches (even those that hit in cache) must be poisoned, as the instruction stream may be corrupted, changing data and control dependencies. This requirement, combined with the rarity of instruction cache misses that are satisfied by memory, means there is little advantage to tracking poison bits for each line in the instruction cache. Instead, `PoisonIvy` uses one poison bit to track speculation on any instruction word.

Control Flow. Speculative data can affect the program's control flow by poisoning

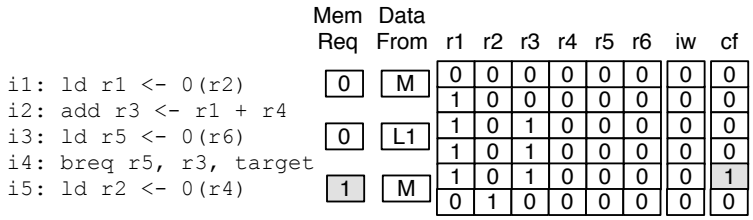

Figure 2.3: Propagation of the control flow poison bit

data values that dictate branches. `PoisonIvy` tracks poisoned control flow (CF) with one poison bit per thread. The CF poison bit is set in the register-read stage of the pipeline when any poisoned instruction might modify the program counter. A poisoned branch sets the bit whether it was correctly predicted or not. The CF poison bit is cleared when verification completes or instructions older than CF-poisoned instructions are squashed.

This method of setting the CF poison bit produces correct outcomes. First, any instruction that executes after a poisoned branch observes poisoned control flow. Second, an older instruction that executes after the branch, due to out-of-order scheduling, may observe a set CF bit even though its control flow was not actually poisoned. This is a safe and conservative outcome. Third, a younger instruction that executes before the branch observes a cleared CF bit. This is a correct outcome as anything the attacker did, absent other poison reaching this instruction, did not affect its computation. These rules greatly simplify implementation, eliminating the need to propagate poison through younger instructions that have already executed out-of-order.

Figure 2.3 shows an example with poisoned control flow. Suppose the value of `r3` is affected by speculation and instruction `i4` compares `r3` to `r5`. Unverified values could cause the program to execute different instructions. This vulnerability corresponds exactly to the binary search attack devised by Shi *et.al.* For this reason, `PoisonIvy` must track when control flow has been affected by poisoned values.

Although there is logically one CF poison bit per thread, an implementation can safely have multiple physical copies (*e.g.*, one per superscalar lane) if desired. These copies need

not be updated at exactly the same time. Rather, they can be updated with the same latency that is required for branch misprediction signals to propagate to that part of the execution core.

Address Translation. The processor may need to access memory when translating a virtual address to a physical address. For example, it may miss in the TLB and fail to find the page table entry in cache. In such situations, the translation itself may be returned speculatively from memory. Whenever `PoisonIvy` has a speculative translation, any memory request that uses the translation produces poisoned values and, in the event that it misses at all cache levels, cannot be allowed off the chip until speculation is verified.

When the DTLB receives a speculative translation, the poison bit in the entry is set. Memory instructions OR the poison bit of their translation with the poison bits of their other inputs to determine whether the instructions are poisoned. When the ITLB is filled with a speculative translation, the IW poison bit is set, poisoning all instruction words.

Memory Hierarchy. In `PoisonIvy`, the memory hierarchy must propagate poison and enforce invariants that ensure security. For poison propagation, every line in the L1 data cache and in lower-level caches has a poison bit added to the tags. When a line is filled speculatively, its poison bit is set. Additionally, whenever a poisoned store writes to a cache line, its poison bit is set. As previously mentioned, the instruction cache does not have per line poison bits. Instead, the IW poison bit is set whenever the instruction cache receives a speculative fill, whether directly from the memory controller or indirectly from a lower-level cache.

`PoisonIvy`'s memory hierarchy ensures that (i) no poisoned data is written off-chip and (ii) no memory requests are made with poisoned addresses. The first of these is enforced by the LLC, which may not evict a dirty and poisoned block. When the LLC must evict a block, it treats dirty, poisoned lines as locked and selects a non-poisoned victim. If no such line exists—*i.e.*, all lines in the set are poisoned and dirty—then eviction must stall until speculation completes. Note that such stalls are extremely rare.

Second, the memory controller ensures that no request with a poisoned address escapes

the chip. Memory requests carry their poison information with them down the memory hierarchy. If a poisoned request misses at the LLC and reaches the memory controller, the request stalls until speculation completes. Enforcing this rule at the memory controller, and not higher in the cache hierarchy, delays only off-chip requests. Those that hit in cache are satisfied with no performance penalty. Note that non-cacheable memory and IO operations must always stall until all poison bits are cleared.

`PoisonIvy` accounts for prefetchers, which are prevalent in memory systems. Each prefetcher maintains a poison bit, which is set whenever it is influenced by poisoned information. The prefetcher then includes this poison in each request that it generates. Poisoned prefetches may bring data through the cache hierarchy but stall at the memory controller until verification completes.

Timing Information. `PoisonIvy` accounts for one more subtle piece of state in the processor—timing information. Suppose a victim computes on speculative, unverified data and modifies the processor's shared resources such as the LLC. An attacker on another core observes some timing effect due to the victim's speculative computation, like higher average memory access time, as the attacker's data is evicted by the victim's writes to the LLC. Thus, the attacker has timing information derived from the victim's speculative execution that could be leaked from the processor. Closing this side-channel requires poisoning timing information.

`PoisonIvy` protects against this new timing side-channel attack with a Timing Poison (TP) bit. Whenever a program executes an instruction that can observe time (*e.g.*, `rdtsc`) or performance counters, the TP bit is ORed with the instruction's poison information. A `rdtsc` instruction poisoned by TP produces the correct time when executed, but its output register is poisoned. Time measurements or values computed from them cannot leave the chip until verification completes. Note that the TP bit does *not* close any existing timing side-channels that an application may be vulnerable to.

Setting TP chip-wide when any core speculates is simple and conservative. Normal programs do not read time often, if ever, so the performance impact is minimal. More

fine-grained management could set TP when (i) any thread on the same core has poisoned data, (ii) the core issues request to shared cache that holds poisoned data, or (iii) the core issues request to memory controller when it is stalling another request with a poisoned address.

Clearing Poison Bits. Once the memory controller has received all outstanding verifications, it will notify the respective structures—*i.e.* all levels of the cache hierarchy, the DTLB, the register file, issue queue, memory controller queue—to clear poison bits. In each cache, including the DTLB, the poison bits are flash cleared. Clearing the poison bits in the LLC implicitly unlocks the lines. The memory controller flash clears poison bits from its request queue, allowing memory requests that have been prevented from accessing memory to proceed.

While LLC cache misses do not happen frequently, waiting for all outstanding verifications to complete might unnecessarily introduce additional delays. If a program has a large memory footprint with many LLC misses, poison spreads and eventually halts processing until verification catches up. To reduce delays from verification, we clear poison bits in batches with an epoch-based policy.

2.2.2 Managing Poison with Epochs

Thus far, we have described `PoisonIvy` with one poison bit per structure (*e.g.*, one bit per register, one bit per cache line, etc). In such a scheme, all outstanding verifications must be completed before clearing the poison bit. This scheme works well if the program has small, natural bursts of LLC misses, followed by many cache hits—when the burst ends, all requests are verified and speculation is cleared. However, if the program accesses memory uniformly over a longer period, it may provide no natural break for verification to complete—more and more state is poisoned until structures fill, the processor stalls, and verification catches up.

Epochs. `PoisonIvy` resolves this difficulty by replacing each poison bit with a pair of poison bits—one per *verification epoch*. When the memory controller first returns specula-

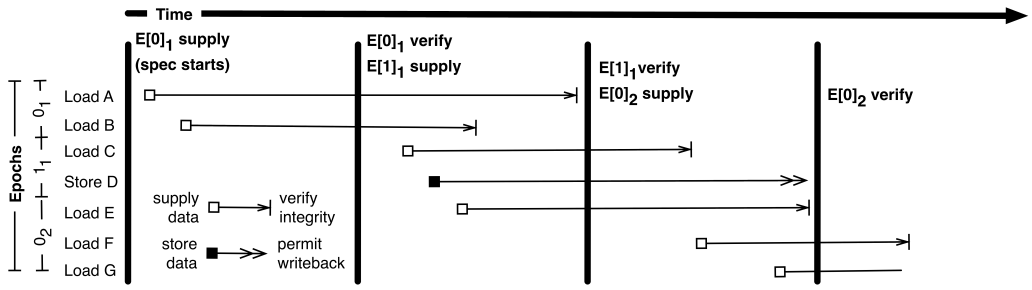

Figure 2.4: Clearing poison with epochs

tive data, it does so in epoch 0 and sets the first bit in the pair for each affected structure. As more requests are returned speculatively in epoch 0, they set the first bit in each appropriate poison pair. After some time—measured in number of cycles, speculative requests, or other conditions—the memory controller transitions to verification epoch 1, in which new speculative requests set the second poison bit for each affected structure.

While new requests arrive in epoch 1, epoch 0's requests are verified as memory cycles become available for hashes, and tree nodes. No new requests are added to epoch 0 and its number of outstanding verifications decrease over time. When all of epoch 0's requests are verified, the memory controller notifies the processor and clears epoch 0's poison bits in all structures. Note that epoch 1's bits are unaffected.

After epoch 0 completes verification, the memory controller closes epoch 1 to new requests and opens a new instance of epoch 0. To ensure proper semantics, the memory controller waits as long as needed to completely clear poison bits across the processor before initiating the new instance of epoch 0. This small delay does not affect the latency of requests to memory since the controller releases requests as soon as their poison bits are cleared. New memory requests may be performed speculatively in epoch 1 while the clear signals for epoch 0 propagate.

Figure 2.4 shows how the speculation mechanism pipelines data supply and verification across epochs. Our definition of epochs permits variable duration and allows memory accesses to dictate the cadence of data supply and verify. The first instance of epoch 0,

denoted $E[0]_1$, begins when loads A and B supply unverified data to the LLC. When $E[0]_1$'s length exceeds the minimum duration, the first instance of epoch 1, denoted $E[1]_1$, begins. Memory accesses C through E are attributed to $E[1]_1$ until its length exceeds the minimum duration and $E[0]_1$'s data is verified, which starts the second instance of epoch 0, denoted $E[0]_2$. Accesses F and G are attributed to $E[0]_2$. `PoisonIvy` could support more epochs with correspondingly more poison bits, but we find no significant advantage to having more than two epochs.

Loads can be verified out of order, which gives the memory controller flexibility when scheduling metadata requests. Because `PoisonIvy` can tolerate very high verification latency without stalling the pipeline, the memory controller should prioritize demand requests over verification requests. The controller should schedule verification over demand requests only when its verification queues fill. Verification requests affect performance when metadata transfers saturate memory bandwidth, which is rare, or when dependent requests have to wait for verification to complete to go out to memory.

The memory controller supplies unverified data to the cache hierarchy and performs two additional tasks. First, the controller increments a counter for the number of pending verifications in the current epoch. Second, the controller marks the outstanding requests as poisoned by setting the bit corresponding to the current epoch number.

Poisoned Cache Lines. The LLC controller confines poisoned cache lines to the chip. First, poisoned lines cannot be evicted since evictions release unverified data to memory. The eviction policy accommodates this constraint with little performance impact. The processor pipeline stalls only when an LLC set fills with dirty lines during an epoch. In practice, epochs clear within a few thousand cycles and such stalls never occur.

Second, poisoned lines cannot be shared by the cache or inter-socket coherence protocol. Stalling coherence requests typically risks deadlock, but our mechanism poses no such risk—coherence and verification are independent. The memory controller verifies integrity and clears poison bits even if the rest of the system stalls.

Verifying a Load. When the memory controller verifies a load's data integrity, it also

decrements the counter associated with the load's epoch. The memory controller consults the outstanding queue entry to identify the epoch.

An epoch ends when its last pending verification is resolved and its counter decrements to zero. The memory controller clears the poison bit for that epoch for all outstanding verifications in the queue. When a memory request's poison bits have been cleared, it is also released from the memory controller to proceed off-chip. The memory controller also sends a message up the memory hierarchy and to the core to clear their poison bits. When all poison bits in a LLC line are clear, the controller releases the cache line and permits eviction to memory.

Area Overhead. Adding poison bits for each epoch adds a small amount of area overhead throughout the processor. Table 2.1 shows the area breakdown of the different structures that require poison bits. The total additional area is ≈9.5KB.

Table 2.1: Poison Storage

Structure	**Size**	**Poison (bits)**
Registers	196	392
Issue queue	60	120
dTLB size	64	128
DL1 cache	32KB	1024
L2 cache	256KB	8192
L3 cache	2MB	65536
Mem controller queue	32	64
IW	–	2
CF	–	2
Pipeline latches, etc	–	≤4096
Poison bits	–	**79556**
		≈ 9.5KB

The logic and propagation overhead is very small. Nearly all poison bits are added to regular, dense SRAM structures (registers, caches) in which wire area is proportional to capacity [89]. Poison bits follow instruction/data through existing datapaths, avoiding irregular and area-intensive wires.

2.3 Evaluation

We evaluate `PoisonIvy` on a system with counter-mode encryption and BMTs [63]. We use HMAC with SHA-2 for all hashes and truncate the resulting hash to 8B. We use the 8B hash for both tree nodes, which ensure counter integrity, and data hashes, which ensure data integrity. A larger hash would only make speculation more important. The arity and height of the hash tree are eight and nine. We use 7b per-block counters and 64b per-page counters to cover 4KB of data in one 64B block. We assume a 32KB separate metadata cache that contains all metadata types unless otherwise specified.

2.3.1 Evaluating Performance from Speculation

Figure 2.5—and all other figures—shows performance overheads normalized to that of an unsecure system (*i.e.*, neither integrity nor encryption). We compare four security mechanisms:

- **No Speculation.** The memory controller waits for verification before supplying data to the LLC. This inherently safe mechanism is implemented in industrial designs (*i.e.*, Intel's SGX).
- `Authen-then-write.` Stores cannot write the L1 cache until all outstanding speculative loads are verified [72]. This mechanism guards against speculative data escaping the core, but does not guard against address-based side-channel attacks.
- **Unsafe Speculation.** The memory controller supplies data to the LLC assuming verification will succeed. No mechanism prevents results of unverified computation from leaving the chip [63, 79].
- `PoisonIvy.` Data is supplied unverified and speculative computation is enabled. Poison prevents results of unverified computation from leaving the chip. It also guards against address-based side-channel attacks.

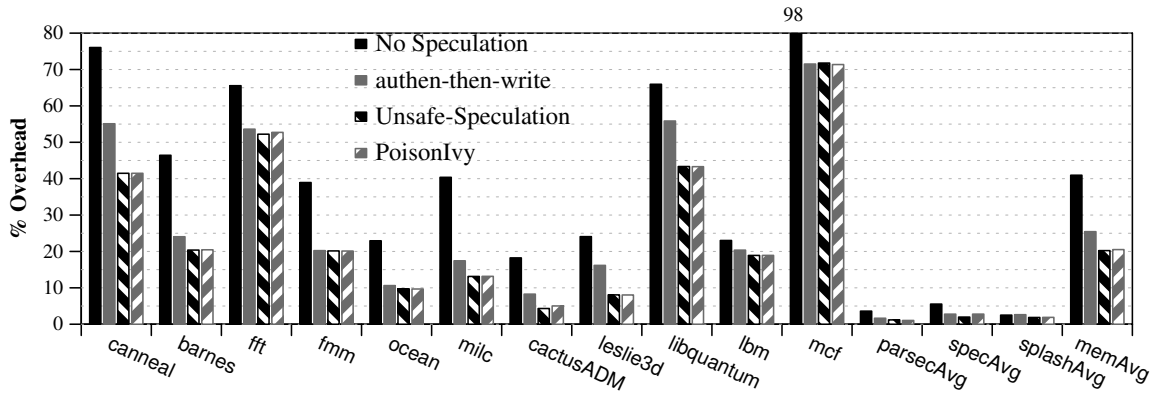

Figure 2.5: PoisonIvy Performance Results

In Figure 2.5, we show the average overhead for all benchmarks in each suite (parsecAvg, specAvg, and splashAvg). These averages are quite low as they include the many benchmarks in each suite that exhibit few LLC misses. Such benchmarks have inherently low overhead as schemes for memory integrity only add latency to off-chip accesses. To better show the overall trends, we also include the average of 11 memory-intensive benchmarks—having more than 10 misses per thousand of instructions (MPKI)—from the three suites (memAvg).

Comparison to No Speculation. `PoisonIvy` (and the other schemes) significantly outperform No Speculation—generally exhibiting about half as much overhead. Without speculation, verification latencies are exposed on the critical path when returning load data, which prevents dependent instructions from executing and filling the out-of-order window.

Comparison to Authen-then-write. Although `authen-then-write` performs much better than No Speculation, `PoisonIvy` outperforms or matches it. Benefits are most notable for *canneal* and *libquantum*. `PoisonIvy`'s performance advantage comes from the fact that `authen-then-write` is constrained by the store buffer capacity during speculation. After the memory controller responds to an LLC miss with unverified data, the core can execute and commit instructions. However, the next store cannot complete and modify the L1 cache until verification completes. Waiting stores quickly fill the store buffer and stall instruction dispatch.

By the time metadata returns and integrity is verified, the datapath has been waiting for hundreds of cycles and the out-of-order window has filled. Because of the limited capacity of the store buffer, the datapath has few opportunities to continue computation when verification latencies are long. Latencies are at least 80 processor cycles when hashing data to check integrity, assuming hash and counter are cached, and are much higher when loading metadata from main memory.

In contrast, `PoisonIvy` permits far more computation during integrity verification by allowing stores to commit data to the L1. Dirty blocks are poisoned and can escape to the L2 and LLC but cannot leave the secure chip. The pipeline only stalls when the LLC must stall eviction due to poisoned dirty lines. `PoisonIvy` not only outperforms `authen-then-write`, it also improves security by guarding against address-based side-channel attacks, which were presented in the same study that presented `authen-then-write` [72].

Comparison to Unsafe Speculation. Unsafe Speculation, which does nothing to restrict computation on unverified data, performs best but is least secure. Figure 2.5 shows that `PoisonIvy` generally matches this best-case performance. In effect, our system has the performance of unsafe speculation while guaranteeing security. A few benchmarks experience trivially higher overheads ($< 1\%$) when compared with unsafe speculation.

Surprisingly, `PoisonIvy` performs as well as a system with unsafe speculation for *mcf*, which is known for pointer chasing. This result arises from the fact that many metadata requests hit the cache, resulting in short verification latencies—shorter than the time required for a load to return data to the core, flow through dependent instructions to another dependent load and for that dependent load's miss to reach the memory controller.

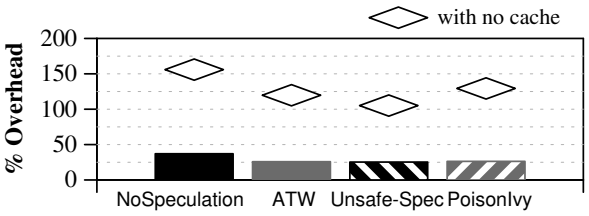

Figure 2.6: Performance of a pointer chasing microbenchmark

For further insight, we implement a pointer chasing microbenchmark—which does no computation with the output of a load beyond using it as the input to the next load—and

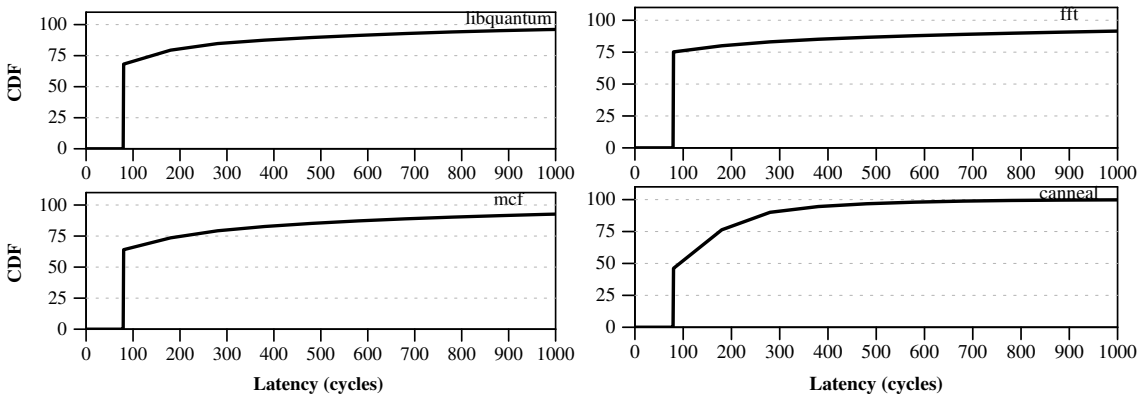

Figure 2.7: Verification latency analysis

evaluate all three systems without metadata caching. Figure 2.6 shows the result of this experiment with and without a metadata cache. With a metadata cache, the results are similar to what was observed for *mcf*. However, without a metadata cache, `PoisonIvy` performs 24% worse than unsafe speculation and 10% worse than `authen-then-write`. `authen-then-write` performs well because the microbenchmark is dominated by loads and has no stores, thereby avoiding pressure on its bottleneck, the store buffer. In contrast, `PoisonIvy` performs well for normal workloads that mix loads and stores. Thus, `PoisonIvy` performs well but pays the price for security when given challenging memory accesses and no metadata cache.

2.3.2 Examining Sensitivity to Verification Latency

To further understand the performance characteristics of these schemes, Figure 2.7 shows the cumulative distribution functions (CDFs) for verification latency when filling a cache line from main memory. The minimum value is the hash latency—80 processor cycles in our study. Many requests are verified in this short latency, indicating that the required metadata is in the memory controller's dedicated cache. For example, in *mcf*, 64% of requests require only this minimum latency due to cache hits, leading to the behavior described above.

These short latencies are quite easy to speculate around and rarely impact dependent

memory requests. When a memory request hits in the metadata cache, its verification latency is hidden by the time it takes to supply data to the core, execute dependent instructions, and issue another memory request through the cache hierarchy to the memory controller. By the time a dependent request reaches the memory controller, verification is almost finished and the dependent request stalls for a very short time, if at all.

Although the minimum latency is modest, the CDFs reveal large means and long tails. Average latency is approximately 273 cycles, much too long for a processor to cover in its store buffer (*e.g.*, with `authen-then-write`). Furthermore, the distributions have long tails with verifications that require as many as 3000 cycles, arising from multiple DRAM requests to retrieve metadata that are delayed behind higher priority data requests. These long tails motivate `PoisonIvy`'s epoch-based speculation.

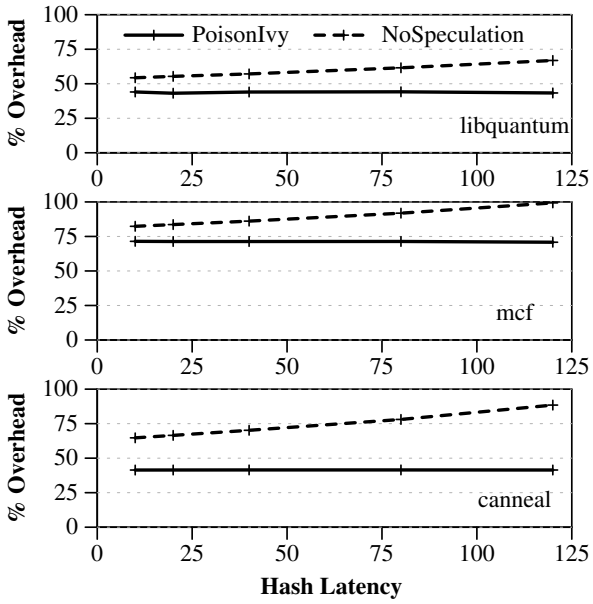

Figure 2.8: Hash latency sensitivity

2.3.3 Analyzing Sensitivity to Hash Latency

Figure 2.8 evaluates sensitivity to hashing latency. Without speculation, the hashing latency is on the critical path when returning data to the core, even when metadata is in the memory controller's cache; a non-speculative design is quite sensitive to hash latency. In contrast, `PoisonIvy` removes the hashing latency from the critical path to service a memory request. As hashing latency decreases, so do benefits from speculation. But even at the lowest latency that we examined, 10 processor cycles, speculation reduces performance overheads from 28% down to 20%, on average, for memory-intensive benchmarks.

2.3.4 Assessing Sensitivity to Metadata Cache Size

Metadata caches matter for three reasons. First, requests for counters are still on the critical path even with speculation. If counters are not cached on a metadata cache, the encryption process doubles the critical path, since counters have to also be fetched from memory. Second, with speculation, verification latency determines the time that structures are poisoned. Metadata caching reduces verification latency as requests for tree nodes and hashes might avoid main memory. Finally, metadata caching might be an alternative to speculation if area overheads do not matter.

Exploring the effects of metadata cache size allows us to consider an important question—how does metadata caching alone compare to `PoisonIvy` when poison storage is included? As poison bits are added to many large structures, they require a noticeable amount of storage—about 9.5KB in the system described by Table 2.1.

Figure 2.9 shows the metadata cache size required before NoSpeculation and `authen-then-write` match `PoisonIvy`'s performance with a 32KB metadata cache. The horizontal line shows `PoisonIvy`'s total storage requirements—41.5KB of which 32KB is metadata cache and 9.5KB is poison information.

A system with no speculation always requires much more storage to match `PoisonIvy`'s performance. For *canneal*, a system without speculation requires 2048KB of metadata caching to match `PoisonIvy`'s performance. In the best case, for *lbm*, twice as much storage is required. For five benchmarks, more than 4MB of storage is required to perform

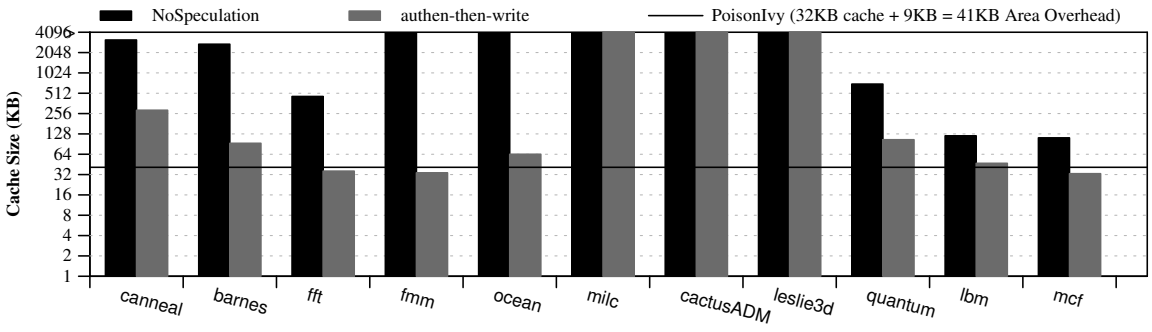

Figure 2.9: PoisonIvy storage overhead evaluation

as well as PoisonIvy's use of 41.5KB.

authen-then-write's performance comes much closer but still requires metadata caches that are much larger than PoisonIvy total storage overheads. For authen-then-write, three-quarters of the benchmarks require at least twice the storage to match PoisonIvy's performance. The remaining benchmarks—*fft, fmm, lbm and mcf*—can match PoisonIvy's performance with a 32KB metadata cache. It is important to note that while authen-then-write performs well for a few benchmarks, it does not protect against address-based side-channel attacks. PoisonIvy preserves all security guarantees of the non-speculative solution.

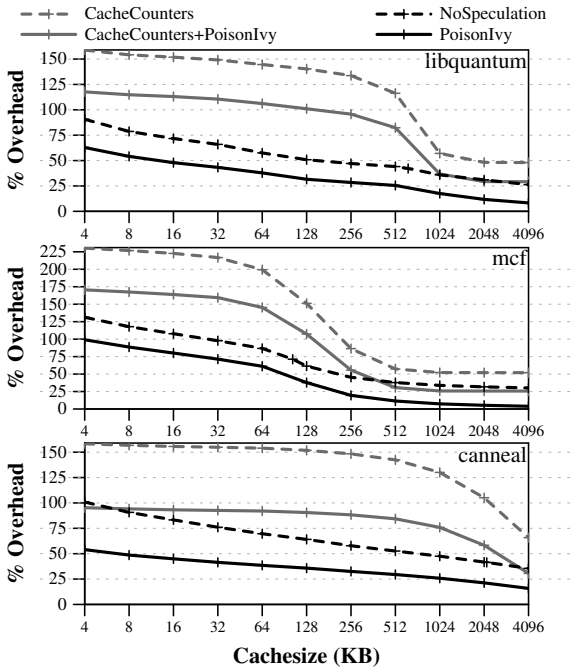

Figure 2.10: Metadata cache evaluation

Figure 2.10 shows sensitivity to metadata cache design, presenting performance overheads under No Speculation and PoisonIvy while varying the metadata cache size and its contents; caches may hold only counters or all metadata. Although prior work caches only counters [63], we find caching all metadata types is worthwhile. For an average of memory-intensive benchmarks, holding all types of metadata in a 32KB cache reduces PoisonIvy's overheads from 64% down to 20%.

As would be expected, performance overheads decrease with larger cache sizes. Overheads significantly and rapidly drop off when the cache accommodates the metadata working set. As the cache size increases and the latency of verifying data integrity decreases, speculation becomes less important. However, speculation with PoisonIvy provides significant benefits even with a 4MB metadata cache.

2.4 Related Work

In recent years there has been multiple pieces of work both in academia and industry that serve both as orthogonal solutions to reducing secure memory overheads and inspirations of our PoisonIvy design.

System Software. When multiple users share resources, one could exploit vulnerabilities in the hypervisor to uncover secret data [56]. HyperWall controls the hypervisor's memory accesses [80]. NoHype removes the hypervisor from the software stack and relies on existing processor features to perform virtualization tasks [38]. HyperCoffer [92] introduces a virtual machine shadow that implements encryption for privacy and hash trees for integrity. In contrast to these system solutions, SGX and ISO-x extend the instruction set to isolate an application from others [1, 19]. These mechanisms are orthogonal to `PoisonIvy`, which applies whenever encryption and trees are used for privacy and integrity.

Poison Bits. Poison bits have been used to accelerate pipelines during cache misses [25, 26, 76]. Poison allows the pipeline to continue while waiting for long-latency memory operations. Unlike prior schemes in which poison indicates missing data, `PoisonIvy` uses poison to mark data that cannot leave the chip. Accordingly, `PoisonIvy` does not need the re-execution mechanisms in prior work. Instead, it tracks poisoned data through the memory hierarchy.

Taint Tracking. Information flow security—also known as taint—tracks untrusted data whereas poison tracks microarchitectural effects of latency tolerance schemes (CFP, etc). Information flow security has been studied extensively at all levels of the abstraction stack: from applications [13, 18, 40, 87, 88], to OS/privileged software [51], and even to logic gates [28, 45, 52, 83–85]. Most of these systems track information flow to prevent code execution (*e.g.*, branches) that could leak secret information. Gate Level Information Flow Tracking (GLIFT) uses taint tags for each bit at the gate level to track the information flow through the system [28].

The main difference between taint and poison used in `PoisonIvy` is that the latter hides verification latency for memory integrity (*e.g.*, SGX). This difference matters. `PoisonIvy`

(i) halts the system when verification fails and cannot use software to handle security exceptions; (ii) guards against physical attacks (*e.g.*, memory tampering) instead of unsafe application data (*e.g.*, SQL injection); (iii) distrusts memory and cannot write poison/taint tags to DRAM since unprotected tags break security and protected tags negate performance from speculation; (iv) uses poison only to restrict unverified data to chip and cannot exploit programmable propagation/checks.

Secure Memory Architectures. We described in Section 1.6 representative architectures that combine the building blocks for secure memory—confidentiality, integrity, and performance. IBM SecureBlue, Intel SGX, ARM TrustZone and AMD SME and SEV are industrial design whereas XOM, AEGIS and AISE are academic designs. `PoisonIvy` provides safe speculation that could be used with any of these schemes. Being built atop state-of-the-art design decisions makes `PoisonIvy` immediately relevant.

2.5 Conclusions and Future Work

Today's computing platforms are moving to usage models in which the user and the party in physical control of a system are not the same—*e.g.*, mobile devices, game consoles, and cloud computing. In this model, the party in physical control of the system may be viewed as an adversary with the ability to carry out physical attacks by spying and/or tampering the off-chip connections of the system. Over the last decade computer architects have been looking for ways to provide protection from the hardware layer against physical attacks.

A trusted processor employs encryption and integrity trees to guard against physical attacks. We propose a new security architecture that ensures security yet significantly lowers performance overheads. We architect a speculation mechanism that uses poison to track speculative data and addresses throughout the processor and memory hierarchy. An epoch-based management policy allows the core to compute on speculative data while clearing speculation in batches. The last-level cache prevents data from leaving the processor before it is verified. Poison tracks speculative dependencies throughout the processor

(*i.e.*, core, register, IW, CF and DTLB) and ensures that no memory operations are sent across the memory's command and address buses before speculative data has been verified. `PoisonIvy` reduces performance overheads of security to 1.8%, on average.

The complexity of `PoisonIvy` has prevented industry partners from implementing `PoisonIvy` on real hardware. The complexity of `PoisonIvy` is mainly due to the fact that poison is tracked everywhere in the processor (*i.e.* registers, all caches). If poison bits were not tracked at such a fine granularity, and instead were only considered at the memory controller, the performance gains of the speculation mechanism would be reduced. We leave as future work the evaluation of the trade-off between complexity and performance.

Chapter 3

Understanding Energy Overhead with MAPS

While speculation is important to reduce the delay overheads, it does nothing to improve the energy overheads introduced by secure memory. When a system with secure memory encounters a last-level-cache (LLC) miss, the memory controller has to issue additional memory requests, on top of the data memory request, to fetch the encryption and integrity metadata. The number of additional memory requests can vary from 0, if both data hashes and encryption counters are cached on-chip, to 10, if no metadata block is found cached. The metadata cache plays an important role in reducing energy overhead.

Prior studies cache metadata in a variety of forms introducing different trade-offs. Several pieces of work propose to cache metadata in the LLC reducing overheads but also introducing competition between metadata and data for space in the cache [20, 68, 78, 79]. Other pieces of work cache only the counters used for decryption, which can shorten the critical path when speculation hides the latency of integrity verification [42, 63], but neglecting the data hashes required for every memory request. Without caching the data hashes or the tree nodes, means that dynamic energy costs are at least twice those of an insecure memory system.

Despite the importance of metadata caching, there is no prior work on understanding metadata access patterns. Architects have either adopted prior cache designs [20, 64, 65, 67, 73, 74, 81] or choose to cache only one type of metadata [10, 29, 63, 92–94]. We present the first detailed analysis of metadata access patterns.

In this chapter, we introduce MAPS [43], an in-depth analysis that results in observations to help guide more efficient metadata cache designs. First, we show that choosing the size of the metadata cache is a non-trivial exercise given that the three metadata types behave differently. We analyze reuse distance and find that metadata blocks have a bimodal reuse distance access pattern. Second, even though metadata types have drastically

different behaviors and complicate the cache sizing decision, we find that caching all metadata types results in a more efficient cache design nonetheless, reducing overheads in half. Third, we find that requests that update a piece of metadata (*i.e.* writes) tend to be followed by long reuse distances. Finally we show that the traditional optimal eviction policy, Belady's MIN algorithm [6] is not optimal for metadata caches. Unlike traditional designs, metadata caches have non-uniform miss costs opening up an opportunity for eviction policy exploration.

3.1 Secure Memory Metadata Structures

A secure memory system protects the user from physical attacks. A physical attack can involve snooping off-chip connections to steal secrets. It can also involve tampering with the values returned from or stored in memory. To prevent and detect physical attacks, secure memory uses encryption for confidentiality and hashing for integrity verification.

Confidentiality. To prevent an attacker from stealing secrets through the off-chip connections, secure memory provides confidentiality by encrypting all off-chip data. State-of-the-art secure memory uses counter-mode encryption. Counter-mode encryption allows the slow part of the encryption process to happen in the background while the encrypted data is fetched from memory.

Integrity. Secure memory detects data tampering by verifying data integrity with Bonsai Merkle trees (BMTs) [63]. A BMT consists of a hash tree over the counters used in counter-mode encryption. The root of the tree is stored on-chip to establish an origin of trust. The tree is composed of keyed Hash Message Authentication Codes (HMACs) for each child block; an 8B-HMAC is sufficient to track block integrity. When a counter is fetched from off-chip memory, the memory controller traverses the BMT, comparing hashes along the way, to verify that the counter's value is the same as when it was last written out to memory. The BMT is stored in main memory with the exception of the root, which is stored on-chip.

Overheads. Every miss in the last-level cache (LLC) requires several memory requests to transfer metadata in addition to data. The number of additional requests depends on the amount of memory protected and the length of the counters. For example, Intel SGX protects up to 128MB of memory with a single 8B counter per 64B-block. It stores three levels of the BMT in main memory in addition to the counters and the data HMACs. For each LLC miss, the SGX memory encryption engine (MEE) fetches five additional blocks from memory—one block for the counter, one block for the data hash, and three blocks tree traversal.

Metadata blocks can be cached to alleviate the overheads from additional memory requests. If a counter block is found in the metadata cache, the memory controller does not need to traverse the BMT because the counter was verified when it was brought into the cache. For Intel SGX, the metadata cache reduces the number of additional memory accesses from five to one, if the counter is cached, or to zero if both the hash and counter are cached. Although metadata caches promise efficiency, metadata access patterns are diverse and present new challenges for cache design.

3.1.1 Caching All Metadata Types

Including all metadata types within the cache is important for reducing overheads from secure memory. First, hashes are required to verify data integrity for every memory access. If the cache excludes hashes, every data access triggers at least one metadata access from memory. Second, counters are required to decrypt data for every memory access. If the cache excludes counters, every data access triggers another memory request for the corresponding counter and even more memory accesses for integrity verification.

Third, tree nodes are required to verify counter integrity. If the cache excludes the tree, a request for a counter requires many metadata accesses to memory to traverse the whole integrity tree. Caching the integrity tree provides a safety net for performance when counters cannot be contained in the cache due to long reuse distances or capacity constraints. Tree nodes have shorter reuse distances because of the tree's shape (*i.e.* fewer

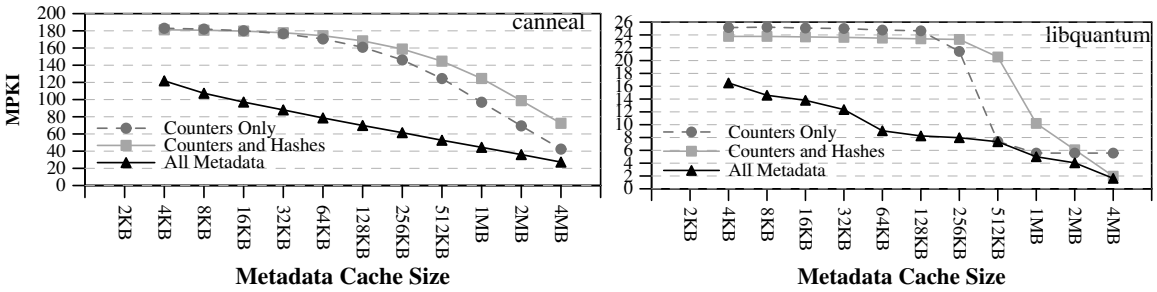

Figure 3.1: Metadata cache efficiency when varying metadata types held.

blocks higher in the tree).

The metadata cache reduces delay overheads for both decryption and verification. It also reduces energy costs, primarily by avoiding expensive memory transfers. DRAM and SRAM accesses require 150pJ [48] and 0.3 pJ [75] per bit, respectively. Energy overhead falls when the metadata cache services more metadata requests.

Figure 3.1 suggests that caching all metadata types has significant benefits. Measurements for *canneal* (left) show that the cache size needed for a given miss rate is smaller when including all metadata types. Achieving 73 metadata misses per thousand instructions (MPKI) requires a 2MB cache that holds only counters or a much smaller 128KB cache that holds any metadata type.

However, permitting the cache to hold hashes in addition to counters leads to subtle interactions between metadata types. Measurements for *libquantum* (right) indicate negligible returns when caching counters with less than 128KB of capacity and diminishing marginal returns when caching them with more than 1MB of capacity. Including hashes with counters harms performance because hashes compete with counters for capacity and counter misses require expensive traversals of the integrity tree. For a 1MB metadata cache, the competition between hashes and counters increases MPKI from six to ten.

Permitting the cache to hold tree nodes, along with hashes and counters, reduces MPKI significantly when the cache size is smaller than 512KB. Caching tree nodes has higher per-block benefits as they provide larger data coverage per 64B-block. Moreover, there are fewer tree node blocks, making it easier to cache them. Experiments with other metadata

cache configurations (hashes only, tree nodes only, hashes and tree nodes, and counters and tree nodes) produce trends similar to those in Figure 3.1.

3.1.2 Addressing Challenges of Caching All Metadata Types

Common caching strategies do not work well for metadata access patterns, which are diverse for three reasons. First, metadata consists of three different types: counters, hashes and tree nodes. Each type has different behaviors due to the varying amount of data each protects. When metadata is organized into 64B blocks, a hash block protects only 8 data blocks whereas a counter block protects 64 data blocks. Each leaf in the integrity tree protects eight counter blocks, which amount to $64 * 8 = 512$ data blocks. Nodes higher in the tree protect more data. The more data a block protects, the more often the block will be reused and the fewer blocks of that type are required, making it easier to cache them.

Second, metadata types exhibit different access patterns. Requests for hashes and counters are driven by workload behavior such as load misses in the LLC or dirty-line evictions from the LLC. In contrast, requests for tree nodes are driven by counter misses in the metadata cache. This distinction causes significant differences in reuse distances.

Finally, metadata types have different miss costs. A miss for a counter block might require traversing the integrity tree, increasing the number of memory accesses by the number of tree levels. In contrast, missing on a hash block requires accessing memory only once for that same block. Furthermore, the miss costs can vary not only between metadata types but also within types.

Traditional cache designs assume misses are independent and have a uniform miss cost. This assumption does not apply to metadata. The miss cost for a block depends on which other blocks and metadata types are cached. Suppose that counter block A has all of its parent tree nodes in the cache while counter block B has only its highest tree level in the cache. If an eviction must decide between block A and B, it might be better to evict A to avoid an expensive miss for B, even if B is reused further into the future. The precise performance outcome of this decision depends on the cache contents when these blocks are

reused.

3.2 Metadata Access Patterns

Mechanisms for confidentiality and integrity pose new challenges in managing multiple types of metadata. When requesting data, the memory controller must also request (i) the corresponding counter to decrypt the data, (ii) tree nodes to verify counter integrity, and (iii) simple hashes to verify data integrity. Links between types of metadata complicate locality analysis and caching policy. Should the cache prioritize counters because a cached counter is secure and avoids expensive tree traversal? Or should the cache prioritize trees because a cached tree node is often reused, especially when it is close to the root?

We consider a system with a per-page counter and a per-block counter for encryption [63, 93]. Although Intel SGX uses a single per-block counter, our per-page and per-block counters approach reduces the memory space overhead and makes caching counters easier. Our analysis applies to other system organizations and we note where differences apply.

3.2.1 Selecting the Metadata Cache Size

Although industry does not divulge specific sizes or design decisions for their metadata caches, intuition says that much more of the on-chip SRAM budget should be dedicated to the LLC than to the metadata cache. To verify this intuition, we simulate four LLC sizes (512KB, 1MB, 2MB, 4MB) and six metadata cache sizes (16KB, 64KB, 256KB, 512KB, 1MB, 2MB). We measure energy and delay relative to a system with a 2MB LLC and without secure memory.

Figure 3.2 plots efficiency for varied system configurations. Each line color shows results with the same LLC size. Each mark type shows results with the same metadata cache size (*i.e.*, squares indicate 256KB metadata caches). The x-axis reports the total capacity budget for both LLC and metadata caches. Results assume that the microarchitecture can

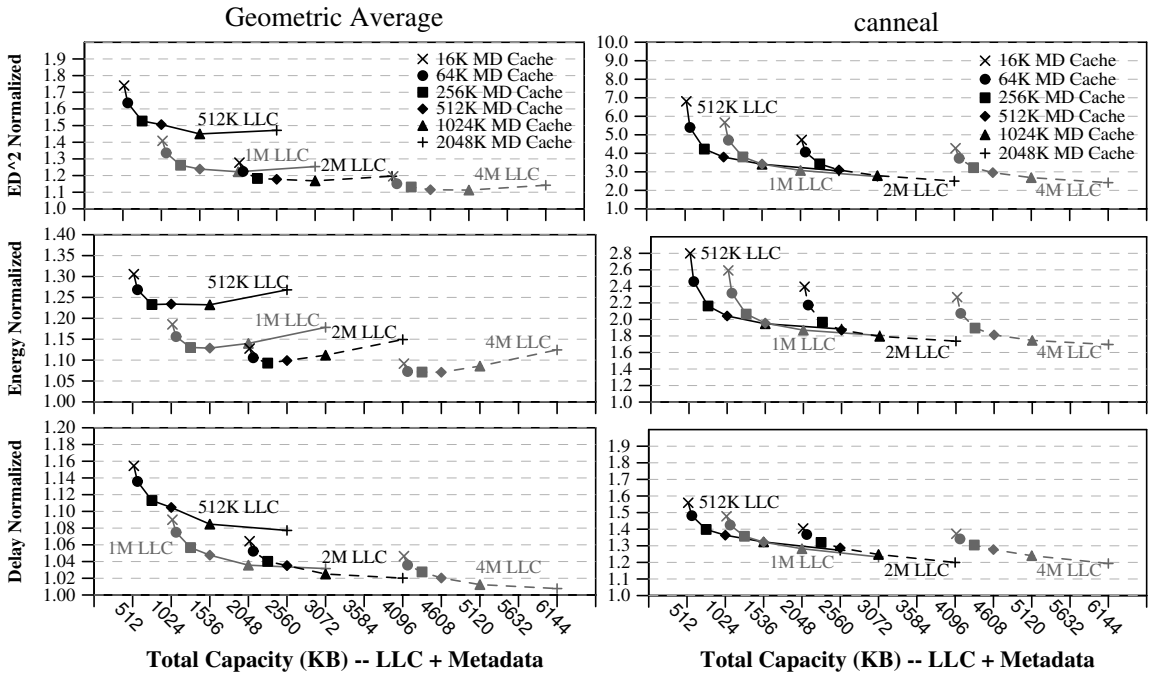

Figure 3.2: Comparison of LLC and metadata cache sizes.

speculate and hide verification latency [42]. Experiments without speculation produce the same general trend.

The results for the average (left) in Figure 3.2 align with our intuition. However, memory-intensive benchmarks, such as *canneal* (right), flip the trend and give us new insight. Given a bit more than a 1MB budget, the average benchmark would perform better with a 1MB LLC and a 16KB metadata cache (*i.e.*, gray x is lower than the black diamond). In contrast, *canneal* would perform better with a 512KB LLC and a 512KB metadata cache (*i.e.*, black diamond is lower than the gray x). Canneal has little spatial locality and requires many more metadata blocks than the average workload. With a smaller LLC, *canneal*'s metadata has smaller reuse distances and thus is able to cache them more efficiently.

The results for the average benchmark suggest that architects should design a cache hierarchy with a smaller metadata cache and a larger LLC for the common case. Results for memory intensive benchmarks indicate that mechanisms that adapt the system for different

Table 3.1: Metadata organization

Metadata Type	Organization		Data Protected	
	PoisonIvy [42]	**Intel SGX [2]**	**PoisonIvy [42]**	**Intel SGX [2]**
Counters	1, 8-byte per-page 64, 7-bit per-block	8, 8-byte per-block	4KB	512B
Integrity Tree	8, 8-byte hashes		$4 * 8^{\text{level}}$ KB	$512 * 8^{\text{level}}$
Hashes	8, 8-byte hashes		512B	

workload needs would benefit cache efficiency.

3.2.2 Understanding the Amount of Data Protected

Metadata access patterns can be explained, in part, by the difference in the amount of data each block protects. All metadata types are stored in memory. Metadata is grouped into 64B blocks, which is the granularity of transfers to the memory controller. A piece of metadata that protects more data will be reused more often. Table 3.1 quantifies the amount of data protected by each metadata type for two secure memory organizations, PoisonIvy (PI) and Intel Software Guard eXtensions (SGX).

A 64B block of counters includes sixty four 7b per-block counters and one 8B per-page counter. The block's sixty four per-block counters protect $64 \times 64\text{B} = 4\text{KB}$ of data. Every LLC miss that addresses the same page will require the same counter block to decrypt the data fetched from memory. The temporal locality of the counter blocks depends strictly on the temporal locality of the 4KB page in an application. Note that Intel SGX uses a larger 8B per-block counter, changing the behavior of counter blocks to match that of the hash blocks.

A 64B block of hashes from the integrity tree protects counter integrity. The 64B block includes eight 8B hashes, each of which protects a 64B block of counters or tree nodes. Because 64B of counters covers 4KB of data (see above), the eight hashes protect

$8 \times 4\text{KB} = 32\text{KB}$ of data at the leaves of the tree. The amount of protected data increases as the node approaches the root. Each tree node covers $8\times$ more data than its child. Since blocks cover 32KB of data at the leaves (or 4KB in the case of Intel SGX), its parent covers 256KB (32KB), its grandparent covers 2MB (256KB), and so on. The on-chip root covers all memory (or all of SGX's secure memory, which is up to 96MB when using SGX1 instructions [2]).

Finally, a 64B block of hashes protect data integrity. The 64B block includes eight 8B hashes. Because each hash protects a 64B block of data, the eight hashes protect $8 \times 64\text{B} = 0.5\text{KB}$ of data.

3.2.3 Analyzing Reuse Distance

We reason about metadata and their access patterns by looking at reuse distance. An effective cache holds blocks that are reused frequently and evicts blocks that are not reused or reused infrequently. Figure 3.3 presents the cumulative distribution function (CDF) of metadata reuse distance for 2MB-LLC with no metadata cache. Each CDF shows that $y\%$ of metadata exhibit reuse distance $\leq x$ bytes. CDFs that rise sharply on the left illustrate short reuse distances, which are easier to cache. Those that extend gradually to the right illustrate poor locality. The reuse analysis provides several insights that can guide metadata cache design.

First, the reuse distance of a metadata block is affected by the last-level cache (LLC) organization. When a data block is cached, its counter and hash blocks are unnecessary. Second, the reuse distance of metadata blocks are affected by the spatial locality of the application. Spatial locality for data translates into temporal locality for metadata. This relationship exists because one metadata block covers multiple data blocks. If a data block is within 512B of another data block, then the reuse distance of all metadata blocks (counter, hash and tree nodes) protecting those two blocks will be equal to that of the two blocks.

For every 4KB page of data, nine 64B-blocks of metadata are needed in ideal conditions, excluding tree nodes. The nine blocks include one block for the counters and eight blocks for

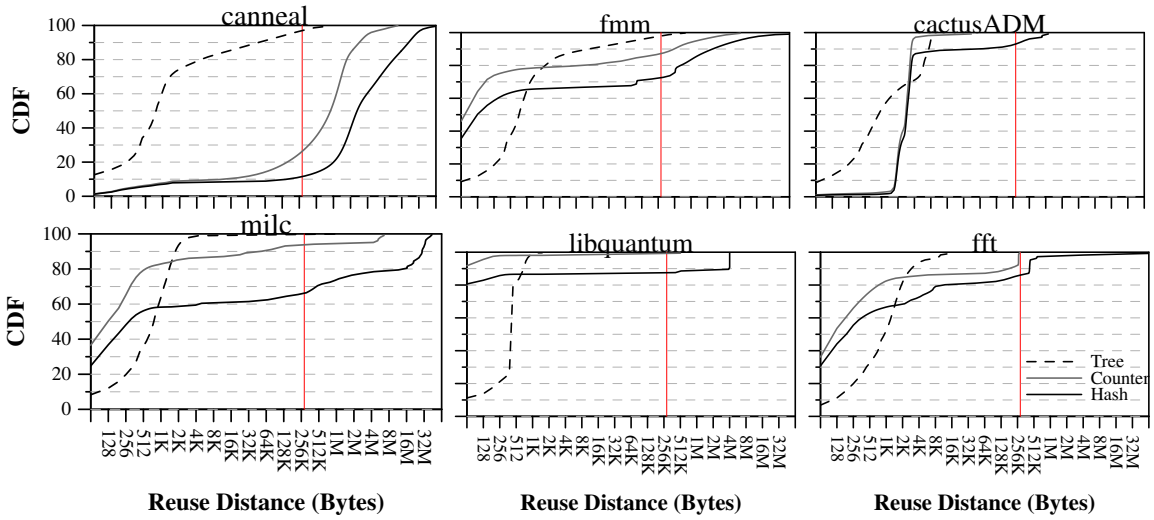

Figure 3.3: Cumulative distribution function (CDF) for reuse distance.

the data hashes. To cover a 2MB LLC, a minimum of $9*64B*(2048KB/4KB) = 288KB$ are needed. Figure 3.3 highlights this value with a vertical line through it. If the application has less than ideal spatial locality, even more metadata is needed. Most benchmarks in Figure 3.3 often show a slight increase for counters and hashes at 288KB and sometimes show a sharp increase (*e.g.*, *fft*).

Counters. Counter blocks depend on the page-level spatial locality of an application. In some cases, such as *canneal*, the reuse distance is large and almost 50% of the blocks have reuse distance larger than 1MB. In others, such as *libquantum*, smaller memory footprints produce correspondingly smaller reuse distances for counter blocks. More than 90% of *libquantum*'s counter blocks have reuse distance of less than 4KB.

Tree Nodes. Tree nodes have the smallest reuse distance because the tree blocks cover the most data (see Table 3.1). For most benchmarks, almost 90% of the tree blocks have reuse distances smaller than 4KB. A small (*e.g.*, 4KB) metadata cache could be sufficient to cache only the tree nodes. Even for benchmarks where the counters have long reuse distance, such as *canneal*, 80% of the tree blocks exhibit a reuse distance smaller than 4KB.

Note that the reuse distance analysis focuses on workload-driven accesses and assumes that no metadata cache is present. Because of the interdependencies between counters and

tree nodes, reuse distances for tree nodes might increase when a metadata cache is present.

Hashes. Hashes have the longest reuse distance, making them the most difficult metadata to cache effectively. Hashes typically experience a small burst of reuse as the eight blocks they cover are often used soon due to spatial locality in the data. However, once that data is held in the LLC, the hash is not needed until the data is evicted, at which point the hash must be updated immediately if the evicted block is dirty.

A good example of the small bursts with short reuse distances is shown in Figure 3.3 . *Libquantum*'s reuse characteristics for hash blocks come from the fact that it repeatedly streams through a 4MB array. As the accesses are sequential, the first access to a data block corresponding to a particular hash is quickly followed by accesses to the other data blocks in the same 64B-hash block. This means that 7 of 8 (87.5%) blocks have short reuse distances. Once all hashes have been used, the hash block is not touched again until the next iteration through the array, producing a 4MB reuse distance for the remaining 12.5% of hash accesses.

Caching hashes is just as important as caching the other metadata types for two reasons. First hashes are needed to verify the integrity of the data itself. Without a mechanism for speculation—the case in all industry implementations—the additional memory access for the hash makes the critical path to read a piece of data from memory at least a factor of two slower than a system without secure memory. Second, the energy required for an additional access to memory is several times larger than the energy required for an on-chip cache [48]. Because the hash block is needed for every memory access, energy costs are doubled when the hash is not cached on chip.

3.2.4 Discovering Bimodal Reuse Distances

An interesting observation regarding metadata access patterns is that reuse distance is bimodal for most benchmarks. Not many benchmarks report moderate reuse distances. Figure 3.4 shows reuse distance classified into four classes: (i) $\leq$128 blocks (8KB), (ii) 128-256 blocks (8KB-16KB), (iii) 256-512 blocks (16KB-32KB), and (iv) >512 blocks (32KB).

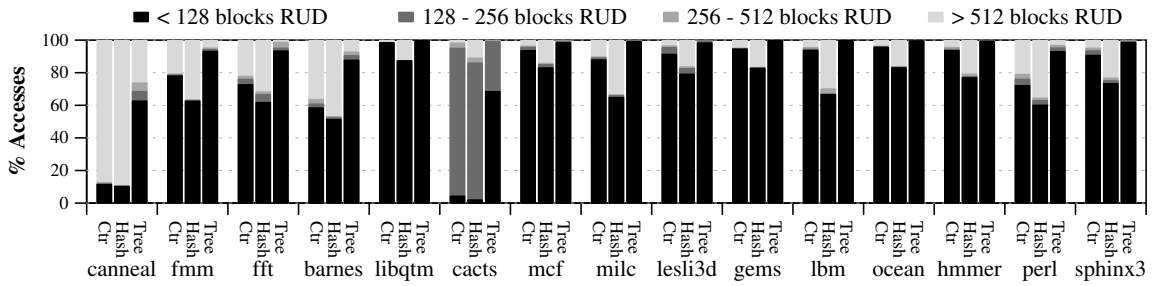

Figure 3.4: Classification of metadata accesses according to reuse distance.

The figure shows that all benchmarks have at least 50% of their accesses in the smallest class, except for *canneal* and *cactusADM*, while most of the remaining accesses reside in the largest class.

We often see a metadata block reused frequently over a short period of time and then not for a long time due to the application's spatial locality. When an application misses at the LLC, it is expected to miss for several blocks in the LLC for the same page and thus share the same counter block. When a counter block is brought into the metadata cache, it is often reused frequently for a short period of time. The same is true for hash blocks except that it happens less often since data blocks share hash blocks as long as they are within 512B.

The fact that we have either short or long reuse distances is important. A cache that holds blocks indiscriminately might evict a block that will soon be used and replace it with a block that will not be reused for a long time. Architects could build on the body of work in reuse prediction for traditional caches [5, 33, 39, 55, 82], adding information about the metadata type.

This bimodal behavior also impacts the cache size. The cache efficiency can only be affected if the cache size is large enough to capture the reuse distance of the blocks. In this case, the metadata cache size should either be smaller (less than 8KB) to capture the shorter reuse distances. Or it should be larger (greater than 32KB) to capture some of the blocks with larger reuse distances. Any cache size in the middle, between these sizes, would not capture the blocks with large reuse distances, regardless of any cache policy, and

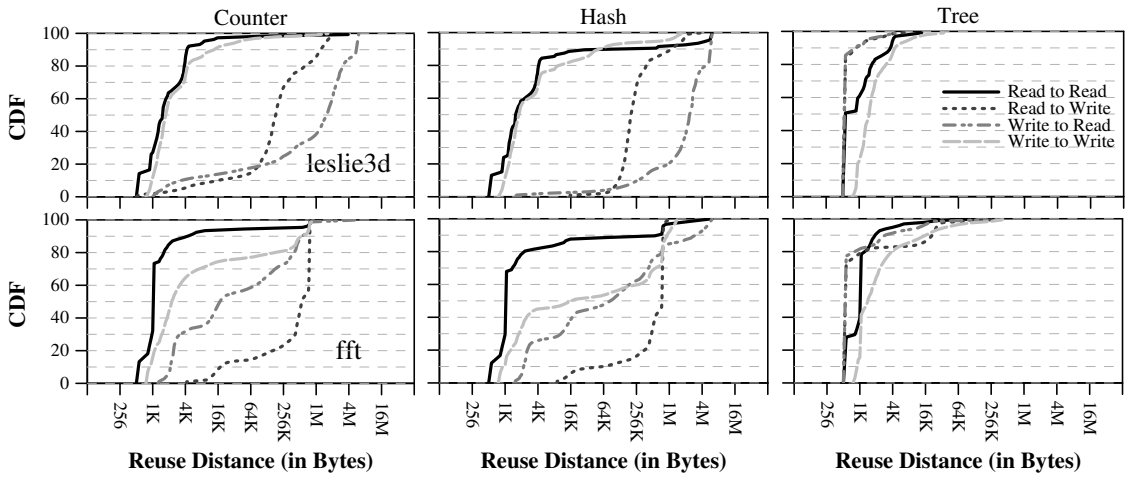

Figure 3.5: Reuse distance cumulative distribution function (CDF) by request type.

would be big enough to capture the blocks with shorter reuse distances, also regardless of any cache policy.

3.2.5 Examining Request Types

The memory request type is a strong indicator of the block's reuse distance. Figure 3.5 shows reuse distance CDF broken down by request and metadata type for the two memory-intensive benchmarks with the most write requests: *fft* with 20% writes and *leslie3d* with 5% writes. These benchmarks are representative of the rest. Reuse distances decrease when an access goes from one type of request to the same type. This pattern follows from the fact that metadata blocks experience the application's spatial locality as temporal locality. If an application reads block A, it is likely that the next block will be spatially nearby and thus share metadata blocks. With counters, this pattern is more pronounced because counters protect a whole 4KB page.

The metadata cache can use partial writes for hash and tree nodes to take advantage of the shorter reuse distance of write-after-write requests [20]. Partial writes are implemented by adding a valid bit per-hash and per-frame, a total of eight bits for each frame in the cache. If a hash needs to be updated and the block is not in the cache, an empty block can

be inserted in the cache as a placeholder for the whole hash block and store only the one updated hash, while the other hashes in the block are marked invalid.

Partial writes are helpful because two write requests that share the metadata block are likely to be evicted around the same time. Memory writes arise from evicting dirty lines from the LLC. Spatially local blocks have similar behaviors and tend to be replaced at approximately the same time under many eviction policies. Since write-to-write requests have shorter reuse distances, it is likely that the rest of the hash block will be updated soon. When the block needs to be evicted, any hash that has not been filled must be read from memory. The partial write mechanism saves one access to memory as long as the hash block is complete when evicted from the metadata cache. The benefits are modest, but the implementation is simple.

Note that the reuse CDFs assume that no metadata cache is present so the tree node reuse distances can actually be longer than shown. In the presence of a metadata cache, a write to a tree node would occur only when a counter block is evicted from the metadata cache. In the absence of a metadata cache, the write to a tree node would occur immediately following the write to a counter. Therefore, we can safely assume that metadata caches extend the tree nodes' reuse distances.

3.3 Metadata Interdependence

Metadata accesses are highly correlated. Tree block accesses depend on the cacheability of the counters. Counter blocks, in turn, can become more expensive to evict from the cache if their associated ancestors in the tree are not present in the cache. The relationship between counters and tree nodes can complicate eviction choices. Instead of all blocks having the same miss cost, as in a traditional cache, blocks can have different miss costs depending on its metadata type and which of its ancestors are already cached. Non-uniform miss costs complicate eviction policies.

3.3.1 Exploring Eviction Policies

For decades, researchers have sought better eviction policies for caches. A common policy is *pseudo-LRU*, which performs well generally but is far from the optimal Belady's MIN policy [6]. *Pseudo-LRU* also performs poorly for metadata caches because of metadata's bimodal reuse distances. To improve upon *pseudo-LRU*, we explore a recent policy that classifies blocks based on the metadata type.

The Economic Value Added (*EVA*) eviction policy uses the concept of value added to classify blocks according to their "age" [5]. The "age" of a block is dictated by the number of accesses to the cache since it was inserted. The value added is determined by the following formula:

$$\text{EVA}(\text{age}) = P(\text{age}) - C * L(\text{age})$$

where P is the hit probability as a function of the block's age, C is the average cache hit rate, and L is the expected remaining lifetime as a function of the block's age. The hit probability, P, is computed based on the number of hits at the block's current age. The remaining lifetime, L, is computed based on the expected number of hits and evictions at ages greater than the block's current age. The block that has the smallest "value added" is evicted from the cache.

We might expect *pseudo-LRU* to perform worse than *EVA* because it is unable to recognize those blocks that have longer reuse distances. Under *pseudo-LRU*, a block that is about to be reused might be evicted while keeping a block that was just added to the cache (most recently used) that will not be reused for a long time. Our experiments show that across benchmarks, there is no one eviction policy that worked for all. Figure 3.6 shows misses per thousand instructions (MPKI) in a 64KB metadata cache for different eviction policies: *pseudo-LRU, EVA, MIN* and *iterative MIN*, which we will discuss in the next section. We evaluate a 64KB metadata cache because it aligns with reuse distances in Section 3.2.3. The results were surprising, so we investigated further.

EVA does not perform as expected because metadata types have bimodal reuse distances. *EVA* uses one histogram for each type to predict reuse distance and hit probability.

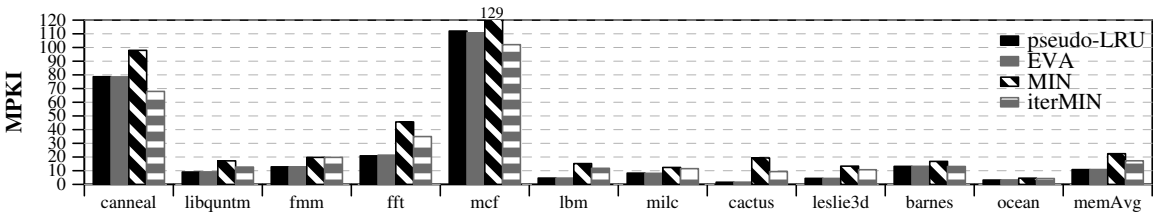

Figure 3.6: Varied eviction policies with 64KB metadata cache.

The bimodal characteristic of metadata reuse distances makes the one histogram approach ineffective for metadata caches.

3.3.2 Finding the Optimal Eviction Policy

Belady's *MIN* algorithm [6] is proven to be the optimal eviction policy for traditional caches. This algorithm relies on future knowledge of the cache accesses. The best eviction candidate is the one that is reused furthest into the future. For *MIN* to be optimal, the access trace must be independent of the cache design and the miss cost must be uniform.

Metadata accesses do not conform to Belady's assumptions. First, the access trace of metadata blocks depends on the cache design and its eviction policy. A tree node is only needed if the children it protects are not in the cache when needed. The access trace changes depending on the cache size and eviction decisions. For example, when a counter is evicted, its parent will be needed the next time that counter is requested. Second, the cost of missing different blocks in the metadata cache depends on how many parent blocks are in the cache. A miss for a counter that has its immediate parent in the cache is much less expensive than one for a counter that does not have any of its ancestors in the cache.

Ignoring these differences and naively applying *MIN* to metadata gives results that are not only sub-optimal but are generally worse than those from the other algorithms. We simulate the benchmark once using true-LRU, gather the cache access trace, and feed that trace back into the simulator to provide future knowledge for *MIN*. However, *MIN* not only fails to account for differences in miss costs, it also starts using incorrect future knowledge once it makes a replacement decision that deviates from true-LRU. In effect, changing the

contents of the cache changes future accesses in ways that deviate from the trace.

Non-uniform Miss Cost. Prior work has examined non-uniform miss costs in the context of NUMA systems. Jeong *et. al.* propose *CSOPT*, an algorithm that considers all possible eviction candidates with breadth-first search [34]. To prune the search space, *CSOPT* tracks the cost of each eviction candidate, eliminating the ones that have higher costs to reach the same state. This algorithm assumes a fixed trace and provides optimal replacements for a system with non-uniform memory access latencies. *CSOPT* accommodates only two costs (*e.g.*, local or remote latency).

Metadata caches can have large number of miss costs, from one additional memory access to accesses equal to the number of levels in the tree. This difference in miss costs increases analytical complexity. We find that running *CSOPT* once on a program's access trace is expensive. Costs range from 32 minutes for applications with small memory footprints, such as *perl*, to more than 6 days—the simulator does not finish—for memory-intensive benchmarks such as *canneal*. We run the algorithm for four-way associative caches and costs increase with associativity.

Varying Access Stream. The problem is even more complex when studying memory integrity. The decision to keep or evict a counter or tree node inherently changes the number and type of accesses required later in the stream. A miss on some types of metadata will trigger requests for other types of metadata. One way to address this problem is to borrow an idea common to compiler analyses: start with a solution that is "too good to be true" (in our case, a memory access trace with no tree nodes) and iterate to a fixed point. In particular, simulate the current trace making optimal replacement decisions and adjusting tree node requests as needed.

Iterating *CSOPT* to a fixed point for *perl* requires 32 minutes per iteration. Although the iterative procedure makes progress as the number of misses reported in each iteration decreases, it fails to complete after two days. We also iterate Belady's *MIN*, which we call *iterMIN*, to a fixed point. Even though the iterative variant finishes quickly, its results could be worse that those from *pseudo-LRU*. Figure 3.6 highlights the importance of considering

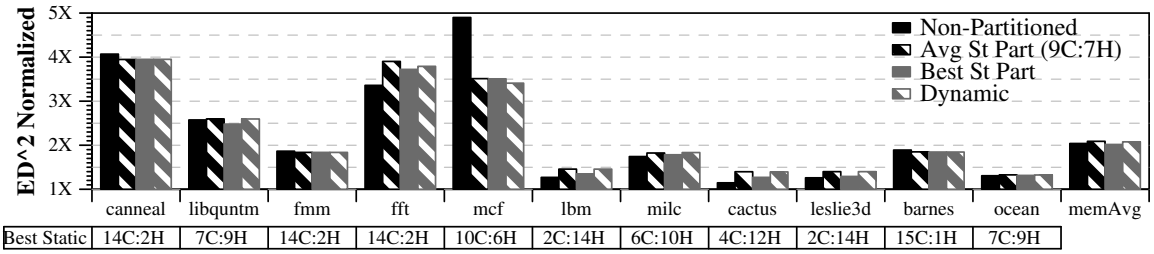

Figure 3.7: Cache partitioning schemes

the different miss costs to find an eviction policy. For most benchmarks, neither *MIN* nor *iterMin* perform better than *pseudo-LRU* and indeed do much worse.

Even though the trace used in *iterMin* is correct and makes decisions that accurately reflect the most distant reuse, these decisions do not incorporate the differences in cost. *IterMin* frequently chooses to keep a near, low-cost miss at the expense of a distant, high-cost miss. To find the optimal eviction policy, the different miss costs have to be taken into account. Designing a computationally tractable algorithm that finds optimal eviction decisions is future work.

3.3.3 Surveying Cache Partitioning

Partitioning the cache may help us manage multiple metadata types. If the cache were partitioned to give more capacity to metadata types that need it, cache efficiency might improve. Tree nodes need not to be included in the partitioning scheme because their reuse distances are either too short to be evicted by most reasonable policies or too long to be cached practically. Moreover, trees ensure counter integrity and are needed only after counter cache misses.

Figure 3.7 shows ED^2 overhead of four different cache organizations: (i) no partition, (ii) partitioned with the best split for the application, (iii) partitioned with the average best split across all applications, and (iv) dynamically partitioned. Partitioning the cache with the best split for the application only improves performance for a few benchmarks (see *barnes, canneal, libquantum and mcf*) and harms performance for others. Applications

requirements evolve throughout its execution and a static partition serves only to limit the cache capacity for each type. The most effective partition depends on the application's spatial and temporal locality. To mitigate the limitations of static partitions, the partitioning scheme needs to adapt to application behavior at run-time.

The dynamic partitioning scheme is inspired by set dueling [58, 59]. Leader sets assess cache effectiveness for counters and hashes. Two small, randomly selected collections of sets serve as competing leaders and the remaining sets as followers. Leaders define bit vectors that partition counters and hashes differently. Followers use hit statistics from leaders to guide run-time partitioning. To produce representative leaders, selected sets are distributed uniformly.

Figure 3.7 compares MPKI with varied partitioning schemes. Results were surprising as dynamically partitioning the cache does not help. In some cases, having the dynamic partition hurts the cache efficiency (see *fft*). Dynamic partitioning does not work well because sets are diverse and sampling them effectively is hard.

Metadata cache designs cannot rely on basic set sampling because sets in a metadata cache differ in three characteristics. First, blocks in each set have different metadata types and, as discussed in section 3.2.3, metadata types have different reuse distances. Second, the number of blocks for each type can also differ from set to set. This matters in an eviction or a partition policy because the block to evict depends on the content of the cache. Finally, the miss cost of blocks are different within and between sets. A miss to a hash block costs only one additional access, but a miss to a counter that does not have any parents in the cache cause as many accesses as levels in the tree.

3.4 Designing a Metadata Cache

Prior studies make assumptions about metadata cache design based on intuitions derived from traditional caches. Some of these assumptions hold under experimental scrutiny. For example, most of the on-chip budget for caches should be spent on the last-level cache and

not the metadata cache. Other assumptions are shown to be false. For example, caching only counters is insufficient to improve performance and traditional caching strategies do not apply to metadata caches.

We make several observations that highlight differences between data and metadata. First, metadata accesses have different types and exhibit different degrees of temporal locality. Second, temporal locality is usually short or long for hashes and counters. Even though tree nodes show short reuse distances, these can change when caching counters. Third, metadata read and write accesses exhibit different degrees of temporal locality. Finally, metadata reuse distances are interdependent across types, resulting in non-uniform miss cost.

Some differences between metadata and traditional caches provide direct guidance to architects:

- *Cache Contents*: Metadata caches should include all metadata types, enabling the cache to adapt dynamically to changing access patterns within and across benchmarks.

- *Cache Size*: The bimodal nature of metadata reuse distances indicate that the cache should be sized to match it. Choosing a size in the middle of the distribution wastes cache capacity.

- *Eviction Policy*: The metadata cache should have an eviction policy that accounts for multiple miss costs. The interdependent nature of the metadata structures affect the optimality of eviction decisions.

Other differences between metadata and traditional caches do not provide direct answers but instead seed questions for future research. Traditional replacement policies, even Belady's MIN, are ill-suited to metadata caches, metadata type and access type should figure into those replacement policies, and cache partitioning shows potential but needs new mechanisms to achieve that potential.

3.5 Conclusions

Secure memory incurs large energy and latency overheads due to the additional memory requests needed to verify and en/decrypt data. Metadata blocks can be cached to reduce overheads. Metadata access patterns vary according to their type—counters, hashes, trees—and differ from data access patterns. We perform a rigorous analysis of these access patterns, motivating computer architects to seek better solutions for caching metadata. Based on the analysis, we provide concrete guidelines and define possible avenues of research to design more holistic metadata caches.

Chapter 4

Exploring the Design Space for Caching Metadata

Secure memory incurs delay and energy costs due to the additional memory accesses required to fetch metadata, which is required to decrypt and verify a block's integrity. On-chip metadata caches can alleviate overheads, but traditional cache architectures are ineffective for metadata. In this part of the dissertation we present a design space exploration, in which the unique characteristics of metadata access patterns are used to improve caching effectiveness. In particular, we explore policies that extend the separate metadata cache size using additional capacity from the LLC. When blocks are cached in the LLC, they introduce heterogeneity among the blocks. We also explore eviction policies at the LLC to account for the heterogeneity of blocks at the LLC.

4.1 Metadata Caches

Caches within the processor alleviate overheads from additional memory accesses for secure memory metadata. Metadata cache design requires at least three decisions: what metadata types to cache, where to cache them and what are the best eviction candidates when holding multiple types of blocks. With regards to which types to cache, several studies propose caching only encryption counters [10, 29, 63, 92, 93]. Caching counters needed for decryption can shorten the critical path, especially when the micro-architecture speculatively supplies unverified data for computation. Speculation, as explained in Chapter 2, exposes only block decryption latency because it removes integrity verification from the critical path.

We argue in Chapter 3 that caching all metadata types improves cache efficiency. Including integrity tree nodes and data hashes, even at the expense of counters, improves performance. Counters tend to have long re-use distances and fully exploiting their temporal locality requires very large caches. Note that the re-use distance analysis described in Chapter 3 assumes no metadata cache is present. In practice, the re-use distances will get

larger once all metadata types are combined into the cache. Blocks that have short re-use distances might get evicted when held together in one unified metadata cache.

Several studies propose placing metadata in the processor's LLC, allowing the memory controller to exploit a large, existing resource [20, 78, 79]. However, this approach causes memory-intensive benchmarks, which are already pressed for LLC capacity, to suffer from cache pollution and additional misses.

Alternatively, on-chip memory controllers can include a dedicated metadata cache [10, 63–65, 73, 93]. Separating metadata from data simplifies processor design and couples the metadata cache to the memory controller, where metadata is used. To the best of our knowledge, industrial implementations, such as Intel SGX, implement a dedicated metadata cache because of the simpler design [21].

A dedicated metadata cache also enables partial writes [20], which addresses inefficient updates to hashes and integrity tree nodes. When a data block is updated in memory, both the encryption counter and the data hash must be updated. The counter block must be read from memory in order to increment it. The data hash block must be read from memory, retrieving a 64B block (the typical granularity of memory transfers [32]) that contains eight different hashes. Reading data hashes from memory is inefficient because updating a data block's hash requires no knowledge of its previous value and eight hashes are retrieved even though only one is updated.

Partial writes delay the read of the data hash block by storing the updated 8B hash in the cache and marking the rest of the block as invalid. The read of the full hash block happens only when the block is evicted from the cache. If all eight hashes in the block are updated before the block is evicted, then the read request is completely avoided.

Tree nodes are similarly organized and benefit from partial writes in the same way. In a dedicated metadata cache, benefits from partial writes are modest but the implementation is simple. Although partial writes could be implemented for the LLC, it would only benefit the metadata stored there, which is typically a small fraction of LLC contents.

Applications have diverse memory access patterns and hardware systems need to adapt

to them dynamically. Metadata caching can be efficient when memory intensity is low. We argue that metadata caching policies have to adapt to the needs of each application. As part of our design space exploration, we consider a mechanism that allows a small metadata cache to adapt its capacity by judiciously using additional cache frames from the LLC.

When storing metadata blocks in the LLC, along with data, the LLC has a new challenge: choosing the right block to evict when not all blocks are equal. As discussed in Chapter 3, both data and metadata blocks have non-uniform miss cost. The eviction policy has to not only take into account the reuse patterns of a particular block, but also the number of additional memory access a block would cost if evicted. In addition to LLC caching mechanisms, we explore eviction policies that consider the type of the block when making an eviction decision.

4.2 Exploiting Metadata Access Patterns

Metadata blocks have bimodal re-use distances, either short or long, as discussed in Chapter 3. Long re-use distances arise because metadata is only needed after misses or dirty LLC evictions. When data is brought on-chip, its metadata is not needed again until the data is evicted from the LLC and requested once again, which happens after many other data blocks are accessed. The expected lifetime of a cached block tends to match the cache's size [4]. If the block's re-use distance is longer than the cache size where it resides, it is unlikely that the block's next access will hit.

On the other hand, short re-use distances arise because each metadata block protects multiple data blocks (see Table 3.1). For example, in Intel SGX, a 64B metadata block holds eight 8B-hashes that together can verify the integrity of eight 64B data blocks (*i.e.*, 512B). Thus, spatial locality in the workload's data access patterns produces temporal locality in the metadata access patterns.

Bimodal re-use distances complicate cache sizing. In principle, the cache size should match the re-use distance of the blocks it holds. In practice, prior studies propose metadata

caches sized between 8KB and 64KB, which are too small to exploit long re-use distances [10, 29, 42, 63, 73, 92, 93]. Unfortunately, long re-use distances are typical and require larger caches.

Large metadata caches support longer re-use distances but may force architects to implement difficult trade-offs, such as a smaller LLC, due to area constraints. Although many workloads prefer larger LLCs over larger metadata caches, some prefer the opposite. For example, as shown in Chapter 3, *canneal* reports lower delay and energy overheads when using a 1MB LLC and 1MB metadata cache compared to using a 2MB LLC and a 16KB metadata cache [43]. Note that design decisions for last-level and metadata caches must be coordinated because they exhibit non-trivial interactions. When the LLC is smaller, it evicts data blocks more frequently and shortens the re-use distance of metadata blocks.

4.3 Metadata Cache and LLC Interaction

Based on the observation that metadata blocks have bimodal reuse distances and that applications have varied demands, we propose Metadata Cache eXtension (`MCX`). `MCX` is a mechanism that extends the metadata cache capacity by using additional capacity from the LLC. The goal of `MCX` is to reduce overheads of fetching metadata from memory. The mechanism extends metadata cache capacity by evaluating the trade-off of using the frames in the much larger LLC. Specifically, it transfers blocks evicted from the metadata cache to the LLC. Because evicted blocks are often selected from deeper in the LRU stack, they will naturally have longer re-use distances. Over time, the mechanism separates blocks with short and long re-use distances, placing the former in the small metadata cache and the latter in the large LLC.

`MCX`'s separation of metadata according to re-use distance creates a virtuous feedback loop for blocks with short re-use distances. As blocks with longer re-use are transferred to the LLC, the average re-use distance for blocks remaining in the metadata cache becomes shorter. Furthermore, already short re-use distances become shorter because the

intervening accesses are removed from the metadata cache's access stream. The progressive shortening of re-use distances for blocks in the metadata cache allows a very small structure to efficiently exploit amplified levels of temporal locality.

`MCX`'s mechanism also mitigates opportunity costs posed by blocks with long re-use distances. These blocks should reside in the LLC because they generate little benefit but inflict significant cost when residing in the metadata cache. Blocks with long re-use rarely benefit from a small cache because they are likely evicted before they are re-used. Yet these blocks can harm the cacheability of short re-use blocks by issuing intervening accesses, pushing those blocks deeper into the LRU stack, and causing inefficient evictions.

`MCX` use of the last-level cache (LLC) is inspired by victim buffering yet addresses its key limitation. Traditionally, caches use victim buffers to mitigate pathological access streams in which a block is evicted but re-used shortly afterwards [36]. The victim buffer is much smaller than the cache it supports to ensure fast access to the fully-associative structure. Unfortunately, a small victim buffer alone is insufficient support for the metadata cache. The majority of evicted metadata blocks have longer re-use distances than the victim buffer can accommodate. A victim buffer is either too small to effectively separate blocks by re-use distances or too large to access quickly. Instead, `MCX` combines both a small victim buffer and the LLC to support the metadata cache.

Because of the metadata blocks' bimodal re-use distances, using a combination of a small victim buffer and a large structure for a *retirement buffer* makes more sense. Using the LLC, the largest SRAM structure on the processor, as a *retirement buffer* for the metadata cache yields positive results. The LLC is large enough to capture the long re-use distances of metadata, while the small metadata cache is small enough to keep the blocks with shorter re-use distances.

First, we must manage metadata blocks in the LLC. When a metadata block is found in the LLC, `MCX` has several options regarding metadata movement. The management of metadata blocks affects the effectiveness of `MCX` and therefore it is not a trivial design choice.

Second, while effective, the use of the LLC as a *retirement buffer* for the metadata cache

introduces two new challenges. First, we must mitigate LLC pollution. As metadata blocks are inserted into the LLC, cache pressure increases and more blocks are evicted from the LLC. When more blocks are evicted from the LLC, more metadata blocks are needed and the application's memory intensity increases. We must understand the trade-off of sending a metadata block to the LLC and the consequences this action creates.

In the next two sections we evaluate the design trade-offs of these two challenges. First we explore designs that can help in reducing the separate cache evictions. Then we consider different mechanisms that minimize the number of blocks sent to the LLC to mitigate the LLC pollution.

4.3.1 Reducing Metadata Cache Evictions

Using the LLC as a *retirement buffer* helps reduce overheads from fetching metadata. If the LLC were used as a victim buffer, when a metadata access hits in the LLC, the entry is invalidated in the LLC and the block is forwarded to the metadata cache. The metadata cache must evict another block to make space. This eviction is beneficial only if the newly inserted block is re-used before it is evicted and if the evicted block is not needed. Unfortunately, metadata blocks rarely fit this behavior and most blocks that exhibit long re-use distances continue to exhibit the same behavior over time.

Sequential Re-use Distances. We reason about metadata cache dynamics by analyzing sequential re-use distances, which quantify the relationship between a block's first re-use distance and the next. Figure 4.1 uses a heat map to illustrate the sequential re-use distances of metadata blocks—the frequency at which each (x,y) pair of re-use distances is observed. The x-axis shows the first re-use distance found and the y-axis shows the second re-use distance—*e.g.* point (1KB, 2KB) shows the percentage of blocks that had a 1KB re-use distance followed by a 2KB re-use distance. The darker the color, the more instances of the pair of re-use distances found.

When the re-use distance of a block is short, the sequential re-use distance is short as well. This observation is consistent across all workloads and metadata types. Figure 4.1

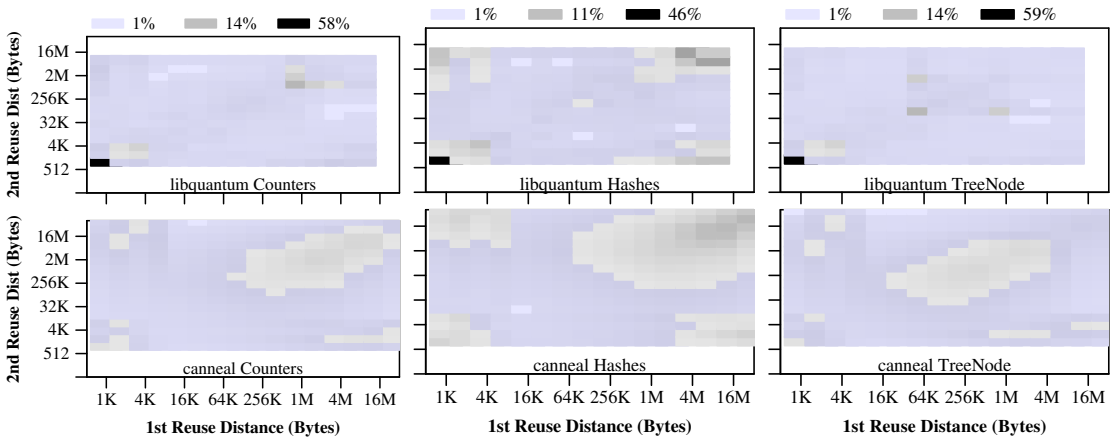

Figure 4.1: Sequential re-use distance analysis

illustrates this observation with the darkest points at (1KB, 1KB) in each graph. These blocks follow traditional expectations for temporal locality and are easiest to cache. Figure 4.2 presents results comparing the different management policies described below:

Victim Buffer Dynamics. When the re-use distance of a block is long, causing it to be evicted from the cache, the sequential re-use distance tends to be long. In Figure 4.1, benchmarks report more observations of blocks with long re-use distances followed by long re-use distances (top right of the graphs). This characteristic does not favor the victim buffer dynamics in which the requested block is removed from the buffer and re-inserted into the cache. The cache insertion causes another block to be evicted unnecessarily because the newly inserted block will most likely be evicted before it is re-used. Results for this designed are labeled *MCX-VB*.

In some cases, particularly for hashes, long re-use distances are followed by short re-use distances (bottom right region of graphs in Figure 4.1). Victim buffer dynamics might benefit these blocks by inserting them into the metadata cache. However, misplacing blocks with long re-use distances into the metadata cache can cause harmful interference whereas misplacing blocks with short re-use distances in the LLC is conservative and relatively harmless.

Semi-Inclusive Cache Dynamics. One might wonder whether the dynamics of a semi-inclusive, two-level cache hierarchy might perform better. Such a cache organization

would keep a block in both the LLC and the metadata cache. As with the victim buffer, this policy benefits blocks for which a long re-use distance is followed by a shorter one. However, transferring these blocks to the metadata cache while keeping them at the LLC degrades the capacity benefits from using both the LLC and the metadata cache. Results for this design are labeled *MCX-2L*.

`MCX` Dynamics. `MCX` keeps blocks at the LLC once they are sent there because blocks with long re-use distances are usually followed by long re-use distance accesses. This approach avoids unnecessary evictions, reducing data movement and metadata cache evictions. Although this approach may transfer blocks with short re-use to the LLC, the forgone performance is minimal. First, retrieving metadata from the LLC is still much faster than retrieving it from main memory. Second, even though the LLC latency is longer than the smaller separate cache, the latency to access the metadata is hidden.

The LLC latency is hidden by even longer latencies and speculative integrity verification [42]. For counters, efficient implementations of counter-mode encryption hide the latency of obtaining the counter behind the latency of retrieving the ciphertext from memory. Retrieving the counter from the LLC requires approximately 25-30 processor cycles [11] and encrypting the counter to produce the one-time-pad necessary for data decryption requires approximately 40 processor cycles [16]. In contrast, retrieving the data block from main memory requires hundreds of processor cycles [32]. For tree or data hashes, speculation hides the latency of verifying integrity by supplying unverified data for computation [42].

Figure 4.2 indicates that `MCX` outperforms both implementation alternatives significantly for both the average benchmark and a few benchmarks. For example, *barnes* performs poorly with both *MCX-2L* and *MCX-VB* because they cause harmful data movement between the metadata cache and the LLC. By keeping blocks in the LLC once they are retired, `MCX` reduces the number of unnecessary evictions at the metadata cache, which in turn reduces data movement between the two caches.

Cache Insertion Policy. To further reduce unnecessary evictions from the smaller separate cache, the cache controller needs to account for the heterogeneity among metadata

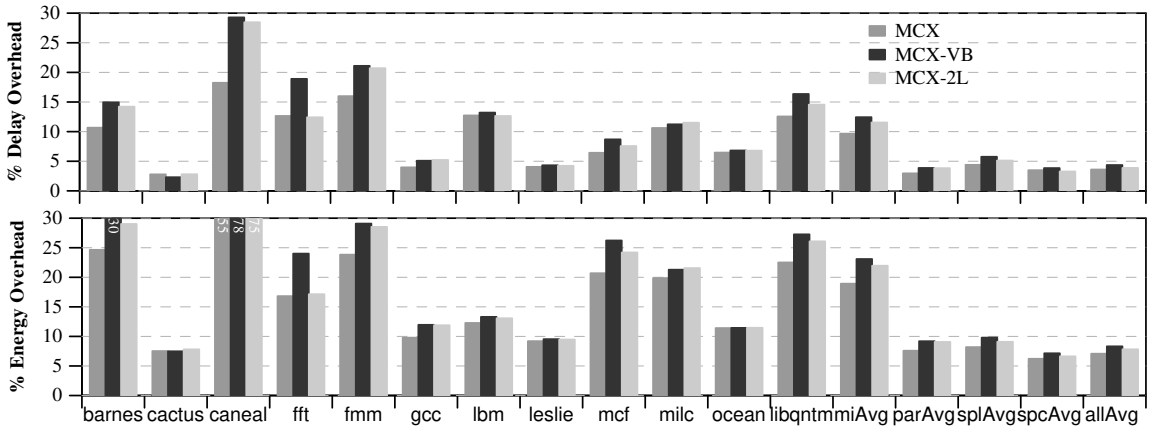

Figure 4.2: Management schemes performance results

types. In Chapter 3 we showed that metadata types have different reuse distance patterns. Both counters and hashes can have long reuse distances because their temporal locality depends on the spatial locality of the application. On the other hand, since integrity tree nodes protect much larger data regions (see Table 3.1), their temporal locality tends to be shorter than counters and hashes. To exploit this difference in reuse distances, counters and hashes need to avoid evicting integrity tree nodes from the smaller cache.

`MCX` uses a bypassing mechanism for counter and hash blocks to avoid evicting integrity tree nodes from the metadata cache. When counters and hash blocks are fetched from memory, they are inserted into the victim buffer, instead of the cache—note that `MCX` requires both a small victim buffer to serve the separate cache and the LLC as the *retirement buffer*. As it is the case in traditional cache victim buffers, when a block is re-used while in the victim buffer, the block is inserted into the cache [36]. If a block is not re-used before it is evicted from the victim buffer, it will essentially bypass the metadata cache. Bypassing the metadata cache for counters and hashes ensures that integrity tree nodes are not unnecessarily evicted from the metadata cache by longer re-use distance blocks. Once a block is evicted from the victim buffer, `MCX` decides if to send said block to the LLC or not.

4.3.2 Mitigating Last-Level Cache Pollution

Evicting metadata blocks to the LLC, an action we refer to as *retirement* does not always benefit performance. When a metadata block is retired, the LLC in turn evicts a data or metadata block to make space. If the newly added metadata is evicted from the LLC before it is re-used, the previous eviction could have been avoided by not retiring the metadata into the LLC. To manage LLC pollution, we need to understand the implications of deciding when a metadata block is sent to the LLC.

Using the LLC alone to hold metadata blocks improves the cache efficiency, because the larger structure is able to hold more metadata blocks. However, using only the LLC increases the LLC pressure, and more data blocks are evicted from the LLC unnecessarily. Some metadata blocks are only used once and never re-used, or re-used after a long period of time. Such blocks take up valuable space from the LLC. Deciding which blocks to send to the LLC becomes an important part of the `MCX` design. In Section 4.3.6 we describe one approach to this decision mechanism which considers the effectiveness of caching metadata blocks in the LLC.

4.3.3 Effects on the LLC and Memory Controller

`MCX` requires no additional modifications to the connection between the memory controller and the LLC. The memory controller and the LLC already interact to handle LLC cache fills from memory. `MCX` takes advantage of already existing connections to send metadata blocks to the LLC. The only difference is that he memory controller must access metadata blocks in the LLC. The memory controller can issue coherence messages to the LLC, as if it were another core, to obtain metadata blocks. The LLC controller can respond to requests from the memory controller as it does with core requests.

Since metadata blocks have physical addresses, no modifications are required to insert metadata blocks in the LLC. The physical address of metadata blocks is used to index into the LLC frames. The LLC controller does not need to differentiate between data and metadata when inserting blocks into the cache. The only difference between data and

metadata is the entity that is allowed to access them.

To preserve security guarantees, only the memory controller is allowed to access and modify metadata blocks. Once the metadata is verified and brought into the chip, it is assumed to be correct and trusted. If structures other than the memory controller could access metadata, a malicious user could force a core to request a metadata block, modify it, and corrupt the integrity tree.

`MCX` requires an address filter in the LLC controller. The LLC controller compares the address of each request against the range of physical addresses occupied by metadata blocks. If the requested address falls within the metadata address range, then the LLC controller must verify that the request came from the memory controller. It rejects requests to this address range from any other entity.

4.3.4 Multi-Core and Multi-Chip Environments

`MCX` extends to a multi-core setting. The LLC is a shared resource across all cores. `MCX` changes the role of the memory controller to act like another core. Since the memory controller is already part of the coherence protocol, no additional changes are required to the coherence protocol. The metadata blocks in the LLC do not change the coherence protocol either because the LLC controller ensures that only the memory controller accesses them. Since there is only one entity allowed to access the metadata blocks and they can only reside in the LLC and in no other cache.

A multi-chip environment requires no special handling of the metadata, which does not travel across chip boundaries. The metadata that protects values in a particular memory device is only valid within the chip's boundary that contains the memory controller. The memory controller is responsible for decrypting and verifying only the integrity of the data residing in its local memory. When another chip requests data, the corresponding memory controller decrypts and verifies the integrity of the data before transmission to its final destination. Thus, metadata is only present in the chip associated with its physical address.

Multi-chip packages can use symmetric key encryption to protect data that travels across chips. The bus that connects multiple processor chips is as unsafe as the connection between the processor and memory. A simple approach to protect the inter-chip connection is to communicate data using a symmetric key between the chips. Chips can agree on the key upon system startup using the Diffie-Hellman key exchange [15]. When servicing a remote chip's request for data, the memory controller encrypts the data with the symmetric key and sends the ciphertext to the other chip.

4.3.5 `MCX` Oracle Implementation

To fully understand the performance implications of the decision to send metadata blocks to the LLC, we start by evaluating an oracle implementation of `MCX`. The oracle mechanism, `MCX`O, assumes perfect knowledge of the benefit of storing blocks in the LLC. To avoid evicting blocks from the LLC unnecessarily, the mechanism needs to know when a metadata block will benefit from being cached in the LLC, which happens once a metadata block is requested from the LLC. In the oracle implementation we assume that all evicted metadata blocks are saved in the *oracle buffer* before being inserted into the LLC.

The *oracle buffer* is a structure that holds metadata blocks until needed. When a metadata block is not found in the metadata cache and is found in the *oracle buffer*, the LLC controller inserts the metadata block into the corresponding LLC set. This mechanism avoids evicting blocks in the LLC for metadata blocks that are not reused. The *oracle buffer* keeps track of how many insertions the corresponding set in the LLC has from the moment a metadata block is added to the buffer. To limit the age of metadata blocks in the *oracle buffer*, when the number of insertions exceed 150% of the number ways, the metadata block is kicked out of the *oracle buffer*—*i.e.* if the block was dirty it is written to memory otherwise discarded.

Delay and Energy Overheads. We compare `MCX`O results with three policies. First we look at a system that uses only a 16KB separate cache. Second, we evaluate a system that uses only the LLC to hold metadata blocks (shown in Figure 4.3). Finally, we compare `MCX`O

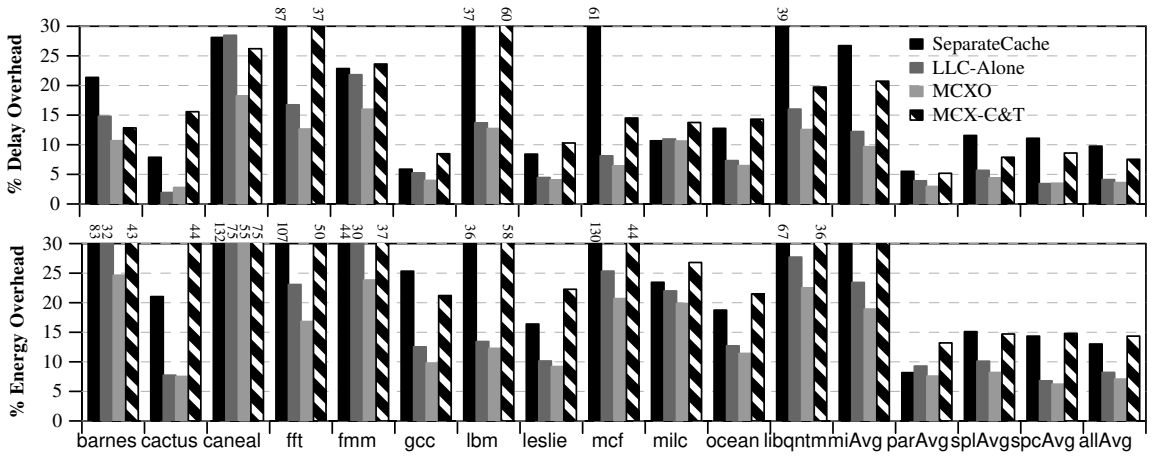

Figure 4.3: `MCX` oracle implementation energy and delay overhead

against an implementation that uses a static scheme to choose the best mix of counters, hashes, and tree nodes to send to the LLC; there are eight schemes possible for the three metadata types. The best static scheme sends both counters and tree nodes to the LLC (results for the best static scheme are shown in Figure 4.3 and they are labeled *MCX C&T*).

Figure 4.3 indicates that `MCX`'s dynamic approach outperforms the best static policy for most benchmarks. For *mcf*, *MCX-C&T*, the best static policy, reports 44% energy overhead while `MCX` reports 16% overhead. Other benchmarks report similar benefits.

On average, `MCXO` and using the LLC alone do not have drastic differences, they both improve delay and energy overheads compared to using a separate cache. They reduce delay from 10% down to 4% and energy from 16% down to 8% for LLC-Alone and 7% for `MCXO`. The reason why there is a small difference between these two mechanisms is that using the LLC alone is only harmful when there is already enough pressure in the LLC to hold data blocks. Most benchmarks across the three benchmark suites are not very memory intensive, and perform well with either mechanism. However, when we look at the memory intensive benchmarks alone, `MCXO` does outperform LLC-Alone. In particular, `MCXO` has larger benefits on energy, since we are considering a system that uses PoisonIvy as described in Chapter 2, which already hides most of the delay overheads. `MCXO` reduces delay from 11% down to 9% and energy from 24% down to 19% when compared against

caching metadata in the LLC alone. A deeper look into each individual benchmark gives use better insight.

MCXO significantly reduces delay and energy overheads for workloads that experience LLC pressure and also struggle to keep metadata blocks in the small, dedicated structure. For *fft*, MCXO reduces energy and delay overheads from 24% down to 16% and from 16% down to 13%, respectively, compared to using the LLC alone. This workload streams through three 8MB matrices, producing abundant spatial locality on data pages that translates into abundant temporal locality for metadata blocks holding encryption counters [1].

For `fft`, MCXO discovers the short re-use distances for counters, placing them in the dedicated metadata cache. MCXO also discovers the long re-use distances for hashes. Since blocks containing hashes protect a total of 512B of data (8B per 64B data block), hash blocks for *fft* have limited temporal locality, and thus MCXO retires them to the LLC. Tree nodes are rarely requested to verify counter integrity because counters are often cached. MCXO avoids wasting metadata cache capacity for hashes and tree nodes, keeping more counters cached in the dedicated metadata cache.

MCXO's advantage is derived primarily from caching counters. Although caching tree nodes and data hashes reduces memory requests and energy costs significantly, it does not improve performance because delays from integrity verification are already hidden by integrity speculation. Performance improves with the number of counters that are cached profitably. When workloads access memory randomly, poor spatial locality for data translates into poor temporal locality for counters. When counter re-use distances are long, architects can do little beyond increasing the LLC size. For example, MCXO has difficulty improving `lbm` and `leslie3d`'s performance.

Another interesting benchmark is *canneal*. *canneal* has a good mix of reads and writes and has a random access pattern, which means there is little spatial locality. Because of the little spatial locality, metadata blocks have little temporal locality. When using the LLC

[1]Data spatial locality translates into metadata temporal locality because one 64B metadata block holds multiple pieces of metadata protecting hundreds of bytes of contiguous data.

alone, *canneal* wastes LLC capacity for metadata blocks that are never reused, increasing the LLC pollution. While *canneal* does not benefit delay overhead from using the LLC alone, this mechanism does reduce energy overhead significantly, from 132% down to 75%. This significant reduction comes from the fact that the LLC is a much larger structure than the separate metadata cache and it is able to hold more metadata blocks on-chip. On the other hand, MCXO is able to lower both delay and energy overheads significantly, from 26% down to 16% and from 75% down to 55% respectively. MCXO avoids evicting blocks in the LLC unnecessarily by holding blocks in the *oracle buffer* until they are reused.

While using the LLC-Alone mechanism improves both delay and energy overheads, with *fmm* MCXO reduces overheads even more, getting delay from 21% down to 15% and energy from 30% down to 24%. This behavior is because *fmm* is sensitive to the LLC miss rate, and when using the LLC alone, the miss rate increases, worsening the number of metadata blocks required. On the other hand, MCXO is able to hold metadata blocks in the *oracle buffer* minimizing the LLC pollution.

4.3.6 MCX with LLC Effectiveness

Given the optimistic results of the MCXO policy implementation, we propose a realistic implementation of MCX using the LLC effectiveness. The benefit of sending a metadata block to the LLC is only realized once the metadata block is re-used when resident in the LLC. If a metadata requests hits at the LLC, the system avoids that additional memory access. On the other hand, if a metadata block is evicted from the LLC without being re-used, the eviction caused by the metadata block when it was inserted into the LLC could have been avoided. We call this trade-off, the LLC effectiveness. The LLC effectiveness is better when more LLC metadata hits are observed and it is worse when more metadata blocks are evicted from the LLC.

We consider a decision mechanism that decides on which blocks to send to the LLC based on the effectiveness of caching metadata at the LLC, a mechanism we call MCX1. The mechanism relies on existing eviction policies and makes no prior assumption about

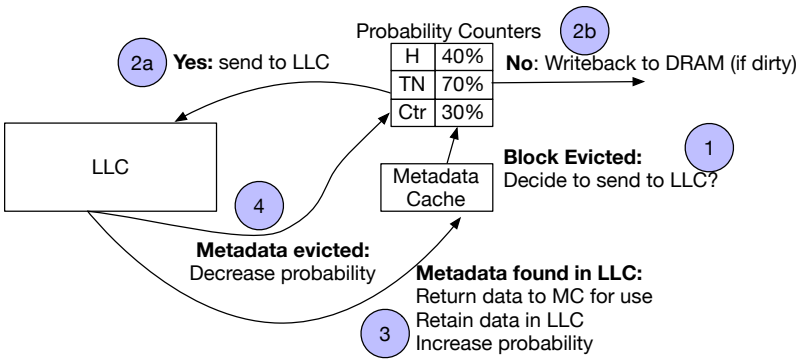

Figure 4.4: Overview of `MCX1` operation.

re-use distances. Metadata blocks are inserted into the metadata cache as they arrive from main memory. As computation proceeds, `MCX1` infers re-use distances from natural cache dynamics. Evicted blocks are more likely to exhibit longer re-use and retained blocks are more likely to exhibit shorter re-use distances.

`MCX1` tracks the effectiveness of retirements by keeping a value at the memory controller, which we call the *probability counter*, that sets the probability of sending blocks to the LLC. When a block is evicted from the dedicated metadata cache, step 1 in Figure 4.4, `MCX1` determines the eviction destination by generating a uniform random number in $[0, 100)$ and comparing it to the probability counter. If the random number is smaller than the counter, the block is sent to the LLC—step 2a in Figure 4.4. Otherwise, the block is evicted normally—step 2b in Figure 4.4—and written back to DRAM if it is dirty.

The probability counter increases when metadata accesses hit in the LLC—step 3 in Figure 4.4. Hits indicate that retirements to the LLC are effective and metadata should be retired more aggressively. Metadata accesses hit in the LLC when the block's re-use distance is too long for the metadata cache, because it was previously evicted, but short enough for the LLC because it produced a hit there.

The probability counter decreases when metadata are evicted from the LLC—step 4 in Figure 4.4. Misses indicate that retirements produce poor outcomes and may arise for two reasons. Either the block's re-use distance is too long to be captured by the LLC or the retirement of metadata blocks increase LLC pressure by competing with data blocks.

Under these conditions, MCX1 reduces metadata retirement to alleviate LLC pressure and avoid inefficient LLC evictions.

With MCX1 mechanism, the memory controller requires information from the LLC to update the probability counters. The LLC notifies the memory controller when a metadata block is evicted, even in a clean state, so that the controller can adjust its counters accordingly. The messages sent between the LLC and the memory controller need to include the type of metadata evicted from the LLC. The messages can be buffered and sent when the bus is idle.

MCX1 must continue to learn new patterns even when retirements to the LLC produce poor outcomes. When the LLC is under pressure or metadata blocks have long re-use distances, increased evictions could cause the probability counter to reach zero. But if no metadata are retired to the LLC, it is impossible to learn the impact of retirement. To ensure it continues adapting, MCX1 imposes a lower bound on the probability counters. This bound must be low enough to reduce the number of retired blocks when outcomes are poor but high enough to obtain feedback regarding the policy's effectiveness. We find that ten is an effective lower bound.

To increase the accuracy of the probability counters, MCX1 instantiates separate counters for each metadata type—counters, tree nodes, and hashes. As shown in Figure 4.1, the three metadata types exhibit different re-use distance distributions and tracking retirement effectiveness per type is important. For example, benchmarks like *libquantum* benefit from retiring counters to the LLC but not hashes as the latter have longer re-use distances than the LLC can capture; see the darker gray region in the upper-right part of libquantum's graph in Figure 4.1. With separate probability counters, MCX1 reduces the number of hashes retired to the LLC, because it sees little benefit of doing so, while still retiring most counters and tree nodes.

4.3.7 Handling Pathological Access Patterns in MCX1

Some workloads may generate pathological access streams and produce erratic behaviors. MCX increments probability counters when metadata blocks hit in the LLC, thereby increasing the frequency of blocks retired to the LLC. As blocks are more frequently retired, LLC pressure increases and blocks are more likely to be evicted. MCX decrements probability counters when the LLC evicts metadata blocks, thereby decreasing the frequency of blocks retired. With a pathological access stream, the probability counters never reach a stationary point and MCX oscillates between aggressive and conservative retirement policies. Workloads with many random accesses, such as *mcf*, are particularly problematic.

MCX uses an epoch based approach to determine whether the system is in an unsteady state. At the end of an epoch (a fixed number of memory requests) the memory controller compares the current value of the probability counters with the values from the last epoch. If the difference between them is higher than a threshold, then the system assumes an unsteady phase is present. In our experiments an epoch is made of 50,000 memory requests and the threshold is 70.

We smooth MCX's retirement policy by drawing inspiration from the exponential back-off mechanism in networks [50]. MCX slows down the reaction of the mechanism to metadata block evictions from the LLC. In the common case, MCX decreases the probability counters when only one metadata block is evicted from the LLC. When MCX detects a phase of unsteady behavior—*i.e.* the difference between the probability counter's current value and the prior epoch's value is higher than the threshold—it exponentially increases the number of evictions required for updating the probability counters.

Once the system reaches a steady phase, MCX exponentially decrements the number of evictions required to update the probability counters across epochs to return to its default value of one. Decrementing the number of evictions after discovering a steady state allows MCX to react quickly to changing patterns.

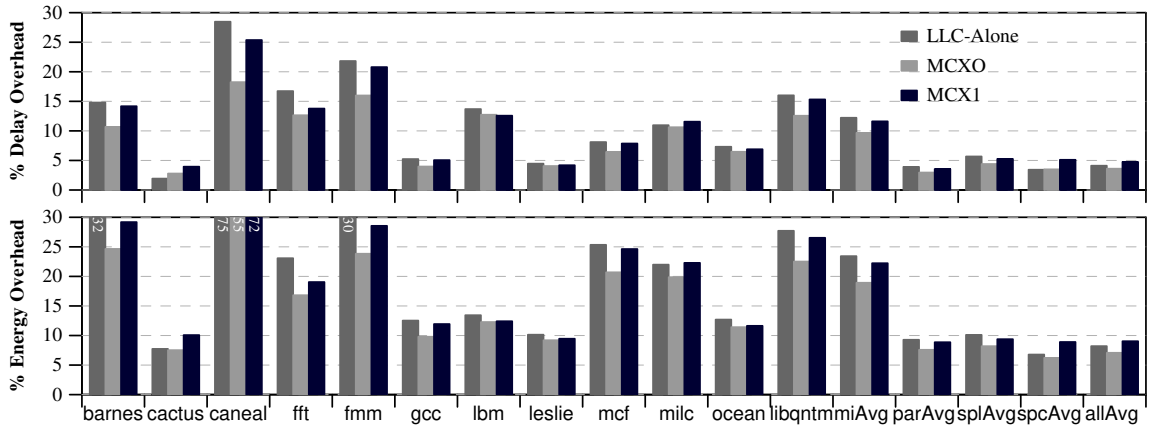

Figure 4.5: MCX oracle implementation energy and delay overhead

4.3.8 Evaluating MCX1

Figure 4.5 shows results of using MCX1 compared to MCXO and using the LLC alone. While MCX1 reduces overheads slightly, it does not perform as well as the oracle implementation. For memory intensive benchmarks, MCX1 average energy overhead is at 22% compared to MCXO at 18%. Once again looking at individual benchmarks helps us understand the results.

As expected, most applications results with MCX1 end up in between caching at the LLC alone and MCXO, except for *cactusADM*. *cactusADM* with MCX1 performs worse than using the LLC alone. By looking at the detailed results we notice that MCX1 keeps more hashes on-chip than the LLC alone mechanism. We also note that MCX1 has more LLC data misses than using the LLC alone. This combination of results can only be explained by the fact that since MCX1 uses a probabilistic approach to choose when to send a block to the LLC, it is often choosing the wrong set of metadata blocks to send to the LLC, increasing the number of LLC evictions unnecessarily. When MCX1 decides not to send a counter block to the LLC, even though using the LLC alone should in theory result in more LLC data misses, fetching the counter from memory requires integrity tree traversal which in turn causes more LLC evictions than only using the LLC. We suspect this behavior is present in most applications, explaining why MCX1 does not perform as good as the oracle implementation.

4.4 Heterogeneity-Aware Cache Eviction

Eviction policies in traditional cache designs treat all blocks in a cache as equals. When considering metadata caching mechanisms that combine different types of blocks, it is important that the eviction policy takes this heterogeneity into account when evicting blocks. In a system with secure memory there exists four different types of blocks: data, counters, hashes and tree nodes. Each type has different access patterns, dependencies and corresponding miss costs. In this section we consider a heterogeneity-aware cache eviction policy.

The caching mechanisms considered in this part of the work, involves the LLC holding all four types of metadata in one way or another. Since the LLC is large enough to capture the reuse distances of most types, the eviction policy becomes a crucial part of this design exploration. The LLC eviction policy needs to be adapted to account for the four types of blocks it may hold.

4.4.1 Dynamic Re-Reference Interval Prediction

After carefully evaluating the state of the art in LLC eviction policies we decided that Dynamic Re-Reference Interval Prediction (DRRIP) is a good candidate for metadata caching. DRRIP's key insight is that some blocks are added to a cache without ever being re-used [33], which is exactly the challenge we are addressing with in this work. When a least-recently-used (LRU) based eviction policy is used, these blocks end up being the last blocks evicted, as when blocks are added they are marked as the most recently used (MRU) block. To account for this distinct access pattern, DRRIP assumes that all blocks inserted into the cache have this behavior, and updates this assumption accordingly. Initially every block is inserted into the cache with long re-reference interval prediction. As the cache accesses proceed when a block is accessed the block is promoted to having a shorter re-reference interval prediction, as it was reused while in the cache.

To track the re-reference interval prediction, DRRIP requires a value for each cache

line, the re-reference prediction value ($RRPV$) [33]. $RRPV$ represents the expected reuse distance magnitude. When a new block is added to the cache, this value is set to indicate a *long re-reference interval*. When an eviction decision has to be made, the cache controller inspects each cache line in the set looking for the first block that has the $RRPV$ set to *distant re-reference interval*. If the controller does not find any blocks with this $RRPV$, the controller increments the $RRPV$ for all cache lines in the set and searches every way once again.

While effective for most applications, not all applications behave well when all blocks are considered to have a *long re-reference interval* when inserted. Jaleel *et.al.* acknowledge that some application have some blocks that have *medium re-reference interval*. When these blocks are marked with the *long re-reference interval*, they are evicted from the cache before they are able to be re-used. To account for these set of applications, DRRIP has a dynamic component that considers adding blocks to the cache with a *medium re-reference interval RRPV*. At runtime, DRRIP adjusts the number of blocks that are added with the two values. If an application favors one over the other, DRRIP adjusts the number of blocks for each category accordingly. We use the set-dueling implementation described in that same work, which is inspired by utility cache partitioning [60].

4.4.2 Evaluating the LLC Eviction Policy

Figure 4.6 presents results when the LLC eviction policy is switched from pseudo-lru to an adapted DRRIP policy. Using DRRIP in combination with `MCX1` is effective. DRRIP prioritizes counter and tree node blocks in the LLC and keeps those on-chip for a longer period of time compared to hash and data blocks. The memory intensive benchmark delay overhead goes from 10% down to 8% with `MCX1` and from 18% down to 16% when compared to the `MCX` oracle implementation with pseudo-lru LLC eviction policy.

cactusADM benefits from either using `MCX1` or using the LLC alone. The eviction policy switch aids this application in keeping more metadata on-chip, reducing both the additional memory accesses and the additional evictions from the LLC.

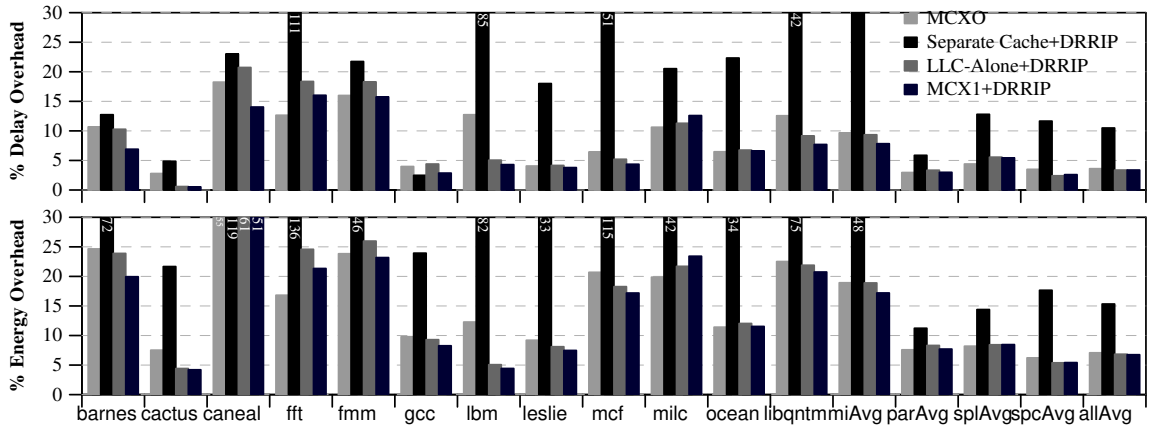

Figure 4.6: LLC Eviction Policy Comparison, pseudo-lru vs. DRRIP

The combination of MCX1 and DRRIP for *canneal* is successful, as it reduces delay overheads from 18% down to 14% when compared with MCXO. Energy overheads also improve, MCX1 with DRRIP reduces them down from 55% down to 51%. The MCX1 mechanism helps in filtering out the hash blocks, which have too long re-use distance to benefit from being cached in the LLC. The DRRIP eviction policy helps in keeping more counters cached in the LLC, reducing the number of integrity tree traversals. Both of these policies, reduce the LLC data misses by 7%. *canneal* is sensitive to LLC misses as it has a working set that does not fit in the higher level caches, using primarily the LLC to serve the processor.

The biggest improvement is on *barnes*, where using MCX1 with DRRIP reduces delay overhead down to 6% and energy overhead down to 20%. This significant improvement of MCX1 is because once metadata is in the LLC, if the eviction policy evicts the "correct" blocks, more blocks are able to stay cached on-chip. In the case of *barnes*, even though the LLC alone serves more counters and hashes from the LLC, the number of additional data LLC misses is larger. These means that the probabilistic approach of MCX1 is effectively filtering out blocks that do not benefit from the LLC.

4.4.3 Heterogeneity-Aware DRRIP

DRRIP can be easily extended to account for heterogeneous blocks by simply choosing a different value for the RRPV based on the type of block at insertion. Data blocks follow the same patterns as the ones described in the original DRRIP work, therefore we do not change that particular behavior. On the other hand, metadata blocks tend to have shorter re-reference intervals than data. Specifically, tree nodes tend to have the shortest re-reference intervals of all block types. Thus a simple extension to the DRRIP eviction policy is to decide on a larger variety of values to assign to the RRPV based on the type of block being inserted. The eviction candidate is chosen using the same mechanism as the in the original work.

The dynamic portion can be further extended to separate the characterization of blocks by type. In the original work, DRRIP uses set-dueling to select between the long and the medium RRPV values. In our extended implementation we consider the same implementation, except that the decision to use one value or the other is localized to the block types. We keep one selector indicator for each type, four in total for each set. The selector adjusts which policy to use at runtime depending on the performance for each type of block.

DRRIP can also be further extended to account for the non-uniform miss costs, an eviction policy which we call *DRRIP+*. When choosing an eviction candidate with multiple miss costs, the controller has to be minimize the cost to evict a particular block in addition to considering the re-reference interval. Choosing the block with the smallest miss cost, might not be the best option because if the block with a larger miss cost has a longer re-reference interval, the additional cost might be amortized over a longer period. Therefore, we increment the RRPV value at a slower pace for blocks that have larger miss costs. The specific value of the miss cost does not have to be known, just as the particular re-reference interval value does not have to be known either. The only information that matters when evicting a block is the *relative* miss cost and re-reference interval among the eviction candidates.

To evaluate the relative difference, the original DRRIP policy increments the RRPV by

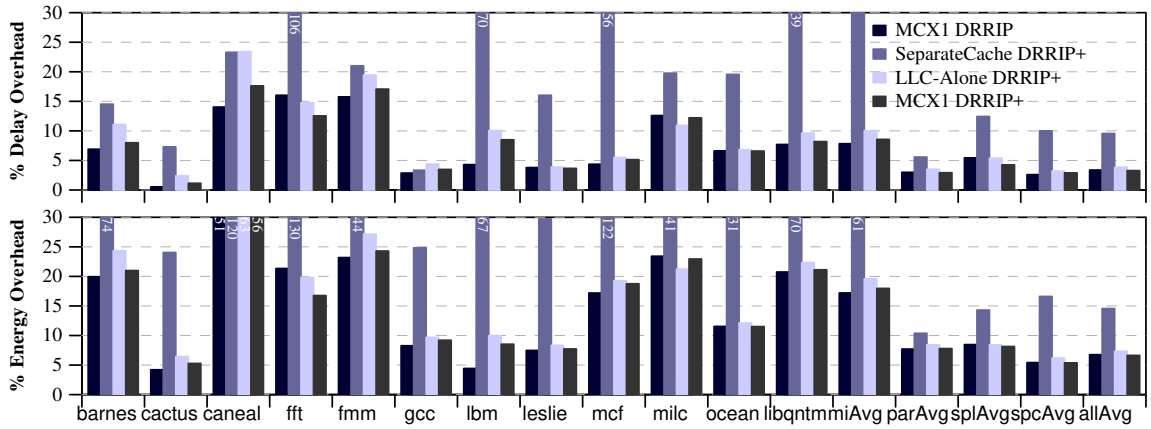

Figure 4.7: LLC Eviction Policy Comparison, DRRIP vs. *DRRIP+*

one for all blocks. When caching all four types of blocks in one cache, counters and tree nodes are expected to have both shorter re-use distances and higher miss costs than data blocks and hash blocks. *DRRIP+* increments the RRPV by 2 for hash and data blocks and by 1 for counters and tree nodes. The reason for this difference is that counters and tree nodes tend to have both shorter re-use distances and higher miss costs than the other types. In Section 3.2 we discussed the differences in reuse distance patterns in metadata types. Because of the way metadata is structure, we showed that tree nodes have the shortest reuse distances. We also noted that the temporal locality of metadata blocks is mainly guided by the spatial locality of the data. Thus, counters tend to have shorter re-use distances than both hash and data blocks. Furthermore, the miss cost of eviction hashes is always one, as hashes are not protected by the integrity tree. Counters and tree nodes do have miss cost that can be larger than one depending on how many antecedents are cached on-chip. Data block miss cost depend on the counters and tree nodes being cached on-chip. However, since data blocks tend to have much longer re-reference intervals we keep the increment of the RRPV the same as hash blocks.

We compare results with DRRIP and the extended version of DRRIP, *DRRIP+*. We present results in Figure 4.7. Unfortunately, *DRRIP+* benefits only one benchmark. To better understand why, we examine this benchmark, *fft* in detail. *fft*' overheads are reduced down to 16% in energy and 13% in delay when using *DRRIP+* in combination with `MCX1`.

When comparing the detailed results for *fft* with the two eviction policies, the main difference is that *DRRIP+* is more likely to evict a hash over a counter or a tree node. As a result, `MCX1` sees more evictions more hashes, and thereby reducing the number of hash blocks sent to the LLC. This relation ends up causing less LLC data misses, about 6% lower, and also ends up causing more LLC hits for counters and tree nodes, reducing the total number of memory access down by 6%.

Unfortunately, other benchmarks actually suffer from evicting hashes over tree nodes. For example, even though *l*bm has more counter blocks serviced from the LLC, it has more hash blocks serviced from the memory. This behavior is because the LLC evicts hashes over tree nodes, reducing the number of hashes that can hit at the LLC. On average, DRRIP+ and DRRIP perform equally, but several benchmarks perform better with DRRIP.

4.5 Further Analysis

We evaluate results of using `MCX1` with DRRIP as the LLC eviction policy in further detail. We analyze the number of blocks serviced from the separate cache, the LLC and memory. We also look at the separate cache and LLC size sensitivities. Finally we evaluate the impact of the speculation mechanism when improving the metadata caching mechanism.

4.5.1 Reducing Memory Requests for Metadata

Figure 4.8 presents the ratio of metadata to data requests issued to main memory. The figure differentiates between metadata types and compares the caching mechanisms. Without caching (not shown in graph), memory serves ten metadata requests for every data access. These metadata requests include encryption counters, tree nodes, and data hashes. Both `MCX1` and using the LLC alone with DRRIP reduce the ratio to 0.2 such that memory serves 2 metadata accesses for every ten data accesses, on average. `MCX1`'s ratio is lower than those from using a dedicated metadata cache. Much of its advantage comes from reducing the number of counters requested from memory, which in turn reduces the number

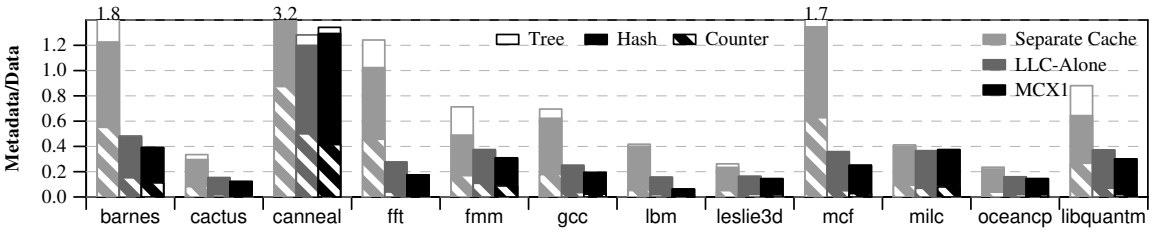

Figure 4.8: Metadata per data memory accesses breakdown

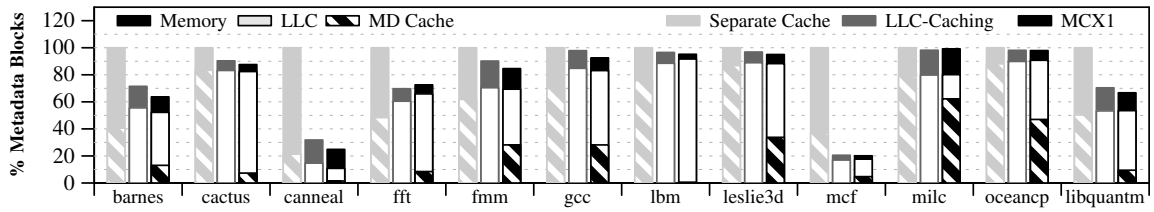

Figure 4.9: Metadata block source breakdown

of tree nodes required. Memory serves 0.002 requests for tree nodes. `MCX1` achieves these reduction by caching metadata blocks with long re-use distances in the LLC, enabling more "useful" counter blocks to stay on-chip, thereby reducing the number of times the integrity tree must be traversed.

`MCX` keeps more metadata blocks cached by separating them by re-use distance and retiring those with long re-use distances to the LLC. Figure 4.9 indicates where metadata requests are satisfied. The dedicated metadata cache alone is too small to be effective and, on average, causes 53% of metadata requests to go to main memory. The LLC alone reduces metadata requests serviced from memory down to 19%. `MCX` reduces total memory requests down to 16%. `MCX` achieves these results by keeping both more counters and more tree nodes at the lower levels on-chip, reducing the number of levels in the integrity tree visited. On average, `MCX` generates 14% fewer memory requests for counters and 62% fewer memory requests for tree nodes when compared against using the LLC alone (see Figure 4.8). `MCX` also reduces memory requests for data hashes, by 16% compared against using the LLC alone, even though they tend to have longer re-use distances. `MCX` reduces the LLC miss rates, thereby reducing the number of hashes required overall.

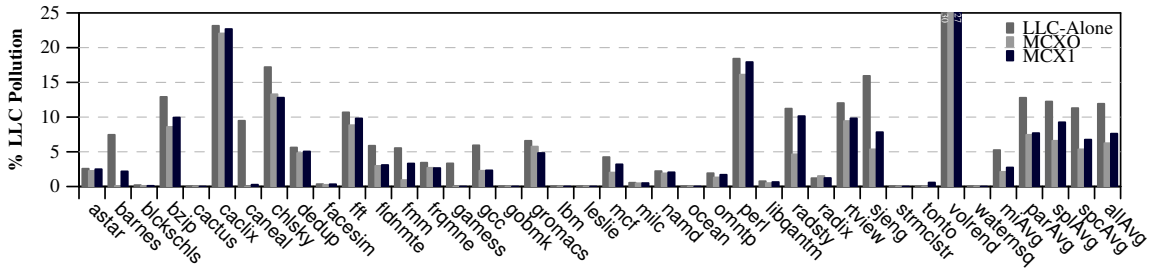

Figure 4.10: LLC pollution evaluation

4.5.2 Mitigating Last-Level-Cache Pollution

`MCX` reduces LLC pollution primarily with a probabilistic approach to determine a block's eviction destination at runtime. Figure 4.10 compares LLC pollution across mechanisms. `MCX` reduces pollution compared to caching at the LLC alone, primarily for workloads that are memory-intensive. These benchmarks' value the LLC capacity more and are therefore more sensitive to the LLC miss rates. Not surprisingly, benchmarks that see larger reductions in LLC pollution, such as *barnes* and *canneal*, correspond to benchmarks that have larger improvements with `MCX`.

Among memory-intensive workloads, *canneal* is most affected by using the LLC. Even without memory security, *canneal* suffers from high cache miss rates and little spatial locality. Secure memory magnifies the effects of this memory intensity and *canneal* data misses increase by 17% when using the LLC alone to cache metadata. `MCX` reduces the pollution to less than 1% by filtering the number of metadata blocks sent to the LLC. Such high LLC pollution is caused by *canneal*'s poor spatial locality, which translates into poor temporal locality for encryption counters and data hashes.

Only one benchmark reports higher LLC pollution with `MCX`, *fft*. Improvements of this benchmark are modest when compared to using only the LLC. `MCX` is trading off more data LLC misses for keeping more metadata blocks cached on chip, as it experiences hits when metadata blocks are held in the LLC. This tradeoff results in a wash when looking at overall performance.

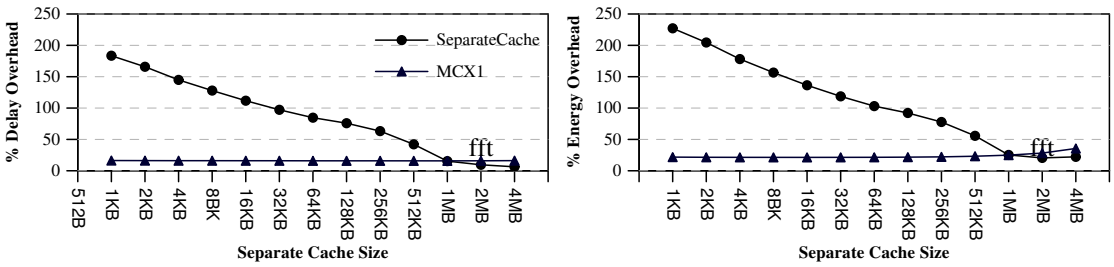

Figure 4.11: Metadata cache size sensitivity

4.5.3 Assessing Sensitivity to Cache Size

Metadata Cache. Figure 4.11 shows that `MCX1` outperforms alternatives regardless of the dedicated metadata cache's capacity unless the cache is large enough to hold all the metadata. Indeed, `MCX1` permits the use of a much smaller dedicated metadata cache without subjecting the system to prohibitively high overheads. The dedicated metadata cache could shrink to a few kilobytes—we use 16KB in our main results—and the system would still perform much better than the alternatives. Even though we only show results for one benchmark (*fft*) in Figure 4.11, all other benchmarks have similar trends.

`MCX1` sustains its performance even as the metadata cache shrinks by drawing on additional capacity from the LLC on demand. Viewed from another perspective, shrinking the metadata cache by hundreds of kilobytes could permit larger LLCs that could benefit both data and metadata, depending on the workload's dynamic memory access patterns. If the system uses the dedicated metadata cache alone, overheads increase drastically as cache size decreases.

The small dedicated metadata cache serves three purposes. First, it caches blocks with short re-use distances. Second, it selectively excludes blocks from the LLC. With such filtering, `MCX1` keeps the more "useful" blocks cached reducing the number of times the integrity tree must be traversed. Third, the dedicated metadata cache employs the "partial writes" optimization [20], in which some write misses avoid an extra read if all portions of the metadata block are written before it is evicted

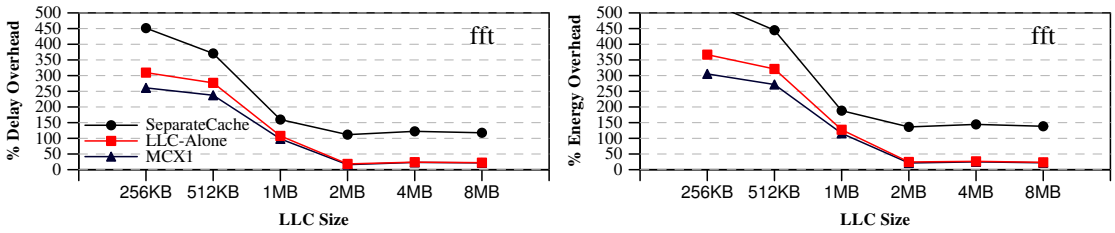

Figure 4.12: LLC size sensitivity

Last-Level Cache. The LLC size can also affect secure memory overheads. First, the LLC size changes the number of data blocks fetched from memory, and therefore the number metadata blocks required. Second, the LLC size determines the metadata block's re-use distances—smaller LLC sizes produce smaller metadata re-use distances and vice-versa—because a block's expected cache lifetime is equal to the cache size that holds it [5]. When LLC pressure is high, blocks are evicted more quickly, therefore metadata blocks have better temporal locality, making them easier to cache.

Figure 4.12 shows results when the metadata cache size is fixed at 16KB and the LLC size is varied for a representative benchmark, *fft*. `MCX1` outperforms other mechanisms, even at small LLC sizes thanks to its probabilistic approach to determining an evicted block's destination. At 256KB and 512KB LLCs, `MCX1` detects LLC pressure and reduces the number of metadata blocks retired to the LLC. Doing so mitigates LLC pollution and performs better than mixing data and metadata indiscriminately when using the LLC alone. Using a dedicated metadata cache alone is also ineffective. Even though metadata blocks tend to have short re-use distances, these distances are still too long for a small metadata cache to exploit.

The performance of *fft* plateaus at 2MB because of its working set sizes (1MB, 64MB and 512MB). At 2MB, both `MCX1` and caching at the LLC alone can keep both data and metadata cached profitably. Increasing LLC capacity beyond 2MB, does not add any benefit because of the long re-use distances of both data and metadata.

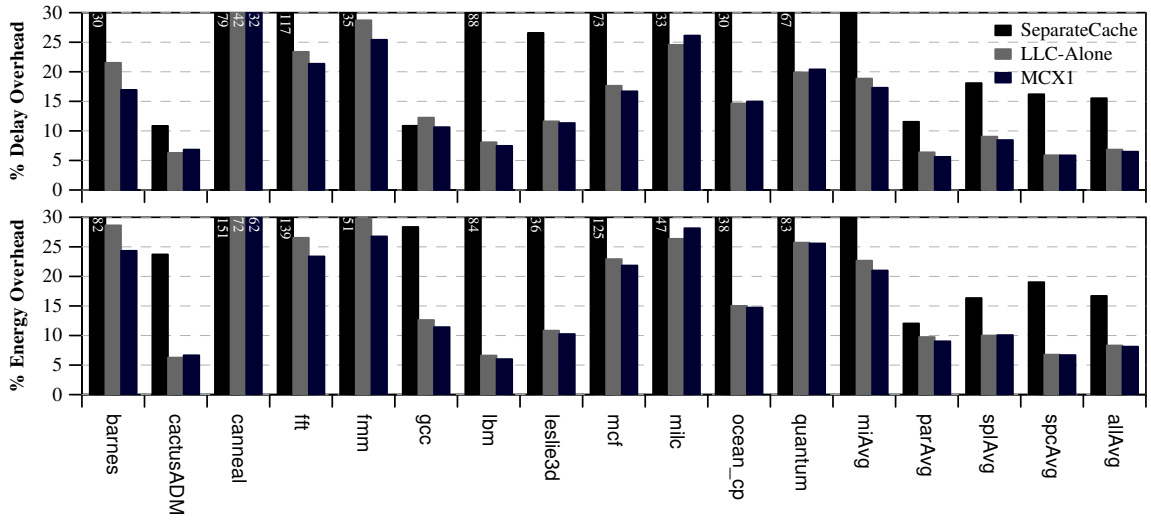

Figure 4.13: Evaluation without speculative integrity verification

4.5.4 Assessing Sensitivity to Speculation

Thus far, the evaluation assumes a speculative micro-architecture for integrity verification, as described in Chapter 2. Speculation hides the latency to verify integrity, which in turn hides the even longer latency of fetching tree nodes and data hashes from cache or main memory. If the micro-architecture did not speculate, the latency to fetch metadata required for verification may extend the critical path for instructions that request data from main memory. `MCX1`'s approach of retiring blocks may expose the system to the LLC's latency, which is higher than the dedicated metadata cache's but still an order of magnitude lower than main memory's.

Figure 4.13 shows results without a speculative mechanism. Both `MCX1` and using the LLC alone suffer from the LLC's longer latency, but `MCX1` uses the LLC capacity more profitably for a few benchmarks. Without speculation, the gap between `MCX1` and using the LLC alone gets much smaller. The difference between the two is no more than 2% for both delay and energy. A few benchmarks, such as *barnes* and *canneal* do perform much better with `MCX1`.

4.6 Related Work

Countless prior works have explored ways to improve the effectiveness of caching. The most closely related of these works are those that examine (and attempt to exploit) data re-use distance patterns.

Re-Use Prediction: Many works explore how to explicitly predict the re-use characteristics of blocks [39, 55, 82, 91], giving the cache concrete information to work with when making replacement decisions. These schemes use the program counter in their predictions. While a program counter can generally be tied to a normal data access easily, it is less clear how to tie the PC to metadata accesses. Furthermore, many of these schemes use roughly 10KB of storage to make their predictions, which is an acceptable overhead when compared to a 2MB cache (*i.e.*, 0.5% overhead), but unreasonable for a 4KB metadata cache.

Replacement Policies: Significant work has gone into improving replacement policies. One recent work by Beckmann *et.al.* propose Economic Value Added (EVA), an eviction mechanism that valuates blocks based on the value added of keeping them in the cache [5]. This mechanism requires expensive computation, thus it is slow to adapt to changing behavior. Metadata caches require a mechanism that adapts quicker to account for diversity in metadata blocks.

Non-uniform memory access (NUMA) systems pose similar cache design challenges as metadata caches [34, 35]. In NUMA systems, some memory accesses can take longer time to access than others. Some eviction policies were proposed but the shortcoming of those is that they only consider two miss costs. Metadata caches can have multiple different miss costs. The optimal solution for the eviction policy in NUMA systems does not scale to multiple miss costs, and ends up being computationally intractable. Computing the optimal eviction policy when considering multiple miss costs and multiple reuse distances is an NP-Complete problem, and we leave this area of research as open avenue for new solutions.

Cache Partitioning: Another way to improve a cache's efficiency is to partition it to account for variable re-use distance distributions. Utility-based cache partitioning [60] is

a mechanism that dynamically partition caches based on their re-use behavior. However, this technique is unable to capture the longer re-use distances that metadata blocks exhibit because of the cache's capacity limitation.

4.7 Conclusion

Secure memory incurs both delay and energy overheads due to the additional memory accesses required to fetch metadata. Metadata caching has been proposed to lower the additional memory accesses. In Chapter 3, we showed that traditional cache designs are not suitable for metadata. In this work we explore the design space for new cache architectures. We propose `MCX`, that takes advantage of the particular characteristics of metadata access patterns.

`MCX` reduces delay and energy overheads by caching blocks more efficiently on-chip. `MCX` keeps more metadata blocks on-chip because it separates them according to their re-use distances. Short re-use distance blocks are cached in a separate cache while long re-use distance blocks are cached at the LLC along with data. Caching metadata blocks in the LLC produces LLC pollution. We explore several implementations of `MCX`. In one approach we present `MCX1` which minimizes LLC pollution with a probabilistic approach. `MCX` when combined with the DRRIP eviction policy in the LLC reduces energy and delay overheads down to 8% and 4% on average, and down to 16% and 8% respectively for memory intensive benchmarks.

Chapter 5

Conclusion

Modern computing trends expose systems to physical attacks and robust software attacks. The software alone is not capable of defending against such attacks. Hardware and software collaborations are the only solution against physical and robust software attacks.

In the last two decades we have seen a shift in both academia and industry working towards secure hardware solutions. Most prior work either imposes expensive overheads in both energy and delay or sacrifices some security guarantees to achieve better performance.

In this dissertation work we focus on the micro-architectural approaches of secure memory to improve the performance and energy overhead. First we propose the first safe speculation mechanism that allows the system to hide the verification latency while still maintaining the security guarantees.

Then we perform the first in-depth analysis of secure memory metadata to guide architects to design more efficient metadata caches. We provide several observations that we believe are important in designing metadata caches and disproof common misconceptions.

Finally, based on the observations from the second piece of work, we do a design space exploration for a new metadata caching mechanism. By exploiting the particular characteristics of metadata blocks, we explore metadata cache designs that can adapt based on the application behavior. In one design point, we evaluate `MCX1`, which reduces energy overhead down to 8% and delay overhead down to 4%. `MCX1` reduces delay and energy overheads by caching blocks more efficiently on-chip. `MCX1` keeps more metadata blocks on-chip because it separates them according to their re-use distances. Short re-use distance blocks are cached in a separate cache while long re-use distance blocks are cached at the LLC along with data. Caching metadata blocks in the LLC produces LLC pollution, which `MCX` minimizes with customized approach.

5.1 Future Work

While both energy and delay overheads are important, a secure system has other aspects to consider. On one hand, systems like Intel SGX that require a programmer to rewrite their application to use the secure region require non-trivial expertise. Intel SGX requires programmers to be able to reason about the particular security requirements for their application and then to be able to decide how to safely partition it. Certain partitions can lead to expensive paging mechanics and inefficiencies of having to get in and out of a secure region. We leave as future work the analysis and the framework to help aid programmers make decisions on how to partition their application.

Another dimension which we did not address in this work is the memory space overheads. Secure memory requires in some implementations as high as 25% of all protected memory to be reserved for metadata. The space overhead do not scale well in a datacenter environment. If a particular system has 4TB of protected memory, 25% of that (a whole 1TB of memory) has to be reserved for metadata regardless of how much secure memory is used. Mechanisms that adapt dynamically to the secure memory usage are needed to enable large memory systems to be protected as well. We leave as future work, the design of a dynamic mechanism that make the metadata space overhead proportional to the secure memory usage.

Bibliography

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proc. International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[2] I. Anati, F. Mckeen, S. Gueron, H. Haitao, S. Johnson, R. Leslie-Hurd, H. Patil, C. Rozas, and H. Shafi, "Intel software guard extensions (Intel SGX)," in *Tutorial at International Symposium on Computer Architecture (ISCA)*, 2015.

[3] D. Bartholomew, "On demand computing–IT on tap," 2005.

[4] N. Beckmann, "Design and analysis of spatially-partitioned shared caches." Ph.D. dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2015.

[5] N. Beckmann and D. Sanchez, "Maximizing cache performance under uncertainty," in *Proc. High Performance Computer Architecture (HPCA)*, 2017.

[6] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, 1966.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[8] R. Boivie, E. Hall, C. Jutla, and M. Zohar, "Secureblue: Secure cpu technology," *Technical report*, 2006.

[9] R. Boivie and P. Williams, "Secureblue++: Cpu support for secure execution," *Technical report*, 2012.

[10] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, "SecureME: a hardware-software approach to full system security," in *Proc. International Conference on Supercomputing (ICS)*, 2011.

[11] I. Coorporation, "Intel 64 and IA-32 architectures optimization reference manual," 2016.

[12] V. Costan and S. Devadas, "Intel SGX explained," Cryptology ePrint Archive, Report 086, Tech. Rep., 2016.

[13] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2007.

[14] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Proc. International Conference on Information Security (CIS)*, 2010.

[15] W. Diffie and M. Hellman, "New directions in cryptography," *Transactions on Information Theory*, 1976.

[16] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *Hot Chips Symposium*, 2016.

[17] R. Elbaz, D. Champagne, R. Lee, L. Torres, G. Sassatelli, and P. Guillemin, "TEC-Tree: A low cost, parallelizable tree for efficient defense against memory replay attacks," in *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2007.

[18] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *Transactions on Computer Systems (TOCS)*, 2014.

[19] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-x: A flexible architecture for hardware-managed isolated execution," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2014.

[20] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2003.

[21] S. Gueron, "A memory encryption engine suitable for general purpose processors," *Proc. International Association for Cryptologic Research (IACR)*, 2016.

[22] S. Gulley and P. Simon, "Intel xeon scalable processor cryptographic performance," *Intel Corporation*, 2017.

[23] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[24] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, 2006.

[25] A. Hilton and A. Roth, "BOLT: Energy-efficient out-of-order latency-tolerant execution," in *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[26] A. Hilton, S. Nagarakatte, and A. Roth, "iCFP: Tolerating all-level cache misses in in-order processors," in *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[27] A. Holdings, "Arm1176jzf-s technical reference manual," *Revision r0p7*, 2009.

[28] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "Theoretical fundamentals of gate level information flow tracking," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2011.

[29] R. Huang and G. E. Suh, "Ivec: off-chip memory integrity protection for both security and reliability," *Proc. International Symposium on Computer Architecture (ISCA)*, 2010.

[30] IBM Security and Ponemon Institute Report, "2018 cost of a data breach study: Global overview," *IBM Report*, 2018.

[31] *Intel Core i7-7500U Processor*, Intel Product Specification Sheet, 2017.

[32] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[33] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2010.

[34] J. Jeong and M. Dubois, "Optimal replacements in caches with two miss costs," in *Proc. Symposium on Parallel Algorithms and Architectures (SPAA)*, 1999.

[35] ——, "Cache replacement algorithms with nonuniform miss costs," *Transactions on Computers*, 2006.

[36] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. International Symposium on Computer Architecture (ISCA)*, 1990.

[37] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," *White paper*, 2016.

[38] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "NoHype: virtualized cloud infrastructure without the virtualization," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2010.

[39] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," in *Proc. International Conference on Computer Design (ICCD)*, 2007.

[40] C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz, "Crowdflow: Efficient information flow security," in *Information Security*, 2015.

[41] H. Krawczyk, R. Canetti, and M. Bellare, "Hmac: Keyed-hashing for message authentication," *RFC2104*, 1997.

[42] T. S. Lehman, A. D. Hilton, and B. C. Lee, "PoisonIvy: Safe speculation for secure memory," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2016.

[43] ——, "MAPS: Understanding metadata access patterns in secure memory," in *Proc. International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018.

[44] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2009.

[45] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," in *Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[46] D. Lie, T. Chandramohan, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *SIGPLAN Notices*, 2000.

[47] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *Proc. Symposium on Operating Systems Principles (SOSP)*, 2003.

[48] K. Malladi, I. Shaeffer, L. Gopalakrishnan, D. Lo, B. Lee, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile DRAM," in *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[49] R. C. Merkle, "Protocols for public key cryptosystems." in *Proc. Symposium on Security and Privacy (SP)*, 1980.

[50] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Communications of the ACM*, 1976.

[51] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: from general purpose to a proof of information flow enforcement," in *Proc. Symposium on Security and Privacy (S&P)*, 2013.

[52] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in I2C and USB," in *Design Automation Conference (DAC)*, 2011.

[53] N. I. of Standards and Technology, "197: Advanced encryption standard (AES)," *Federal Information Processing Standards Publication (FIPS)*, 2001.

[54] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A full system simulator for x86 CPUs," in *Proc. Design Automation Conference (DAC)*, 2011.

[55] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Exploiting compressed block size as an indicator of future reuse," in *Proc. High Performance Computer Architecture (HPCA)*, 2015.

[56] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing hypervisor vulnerabilities in cloud computing servers," in *Proc. International Workshop on Security in Cloud Computing (WSCC)*, 2013.

[57] N. Perlroth, "All 3 billion yahoo accounts were affected by 2013 attack," *The New York Times*, 2017. [Online]. Available: https://www.nytimes.com/2017/10/03/technology/yahoo-hack-3-billion-users.html

[58] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2007.

[59] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," *Proc. International Symposium on Computer Architecture (ISCA)*, 2006.

[60] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2006.

[61] S. Ramgovind, M. M. Eloff, and E. Smith, "The management of security in cloud computing," in *Information Security for South Africa (ISSA)*, 2010.

[62] R. L. Rivest, A. Shamir, and L. M. Adleman, "Cryptographic communications system and method," 1983, uS Patent 4,405,829.

[63] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and Bonsai Merkle trees to make secure processors OS- and performance-friendly," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2007.

[64] B. Rogers, M. Prvulovic, and Y. Solihin, "Efficient data protection for distributed shared memory multiprocessors," in *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006.

[65] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin, "Single-level integrity and confidentiality protection for distributed shared memory multiprocessors," in *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[66] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *Computer Architecture Letters (CAL)*, 2011.

[67] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2018.

[68] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[69] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler, "Late-binding: enabling unordered load-store queues," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2007.

[70] T. Sha, M. M. Martin, and A. Roth, "Scalable store-load forwarding via store queue index prediction," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2005.

[71] ——, "Nosq: Store-load communication without a store queue," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2006.

[72] W. Shi and H.-H. S. Lee, "Authentication control point and its implications for secure processor design," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2006.

[73] W. Shi, H.-H. S. Lee, M. Ghosh, and C. Lu, "Architecture support for high speed protection of memory integrity and confidentiality in multiprocessor systems," in *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004.

[74] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, "High efficiency counter mode security architecture via prediction and precomputation," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2005.

[75] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," Compaq Computer Corporation, Tech. Rep., 2001.

[76] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[77] S. Subramaniam and G. H. Loh, "Fire-and-forget: Load/store scheduling with no store queue at all," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2006.

[78] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2003.

[79] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *Proc. International Conference on Supercomputing (ICS)*, 2003.

[80] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[81] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with effcient integrity verification structures," *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[82] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2016.

[83] M. Tiwari, X. Li, H. M. Wassel, F. T. Chong, and T. Sherwood, "Execution leases: A hardware-supported mechanism for enforcing strong non-interference," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2009.

[84] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, "Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2011.

[85] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[86] G. Vasse, "IBM extends enhanced data security to consumer electronics products," *IBM News Release*, April 2004.

[87] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[88] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "Memtracker: Efficient and programmable support for memory access monitoring and debugging," in *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[89] N. Weste, D. Harris, and A. Banerjee, "CMOS VLSI design," *A circuits and systems perspective*, 2005.

[90] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. International Symposium on Computer Architecture (ISCA)*, 1995.

[91] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer, "Ship: Signature-based hit predictor for high performance caching," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2011.

[92] Y. Xia, Y. Liu, and H. Chen, "Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks," in *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[93] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," *Proc. International Symposium on Computer Architecture (ISCA)*, 2006.

[94] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2003.

[95] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta, "SENSS: Security enhancement to symmetric shared memory multiprocessors," in *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2005.

Biography

Tamara Silbergleit Lehman is a PhD candidate at Duke University advised by Andrew D. Hilton and Dr Benjamin C. Lee. Her thesis work focuses on reducing overheads of secure memory. Her research interests lie on the intersection of computer architecture and security. She is also interested in memory systems, simulation methodologies and emerging technologies. Tamara has a Masters of Engineering degree in Computer Engineering from Duke University. Her latest publication on understanding metadata access patterns in secure memory at ISPASS 2018, MAPS, won the best paper award. Her earlier work on developing a safe speculation mechanism for secure memory, PoisonIvy, published in MICRO 2016 got an honorable mention in Micro Top Picks. Tamara also worked at Intel Labs for two summers, in 2015 and 2016, with the group that developed Intel SGX and published a patent with that same team at Intel Labs in 2018.