MAPS: Understanding Metadata Access Patterns in Secure Memory

Tamara Silbergleit Lehman, Andrew D. Hilton, Benjamin C. Lee
Electrical and Computer Engineering
Duke University
{*tamara.silbergleit, andrew.hilton, benjamin.c.lee*}@*duke.edu*

***Abstract*—Secure memory increases both the latency and energy required for memory accesses. To reduce these overheads, computer architects have sought to cache metadata on the processor chip, but placing metadata in a simple cache has not been as effective as expected. With a detailed analysis of metadata access patterns, we clarify myths in metadata caching and provide insight into more efficient caching strategies. We provide three observations that can help architects design future metadata caches. First, caching all metadata types improves efficiency. Second, the size of the metadata cache should match the reuse distance of the metadata. Third, when designing a better eviction policy, the traditional Belady's MIN algorithm cannot be used as the optimal replacement policy.**

***Keywords*-Secure Memory, Computer Architecture, Cache Architecture, Memory Characterization, Memory Hierarchy**

I. Introduction

Several new computing models—cloud, gaming, and mobile systems—force users to relinquish physical control over the hardware. Losing physical control introduces a new threat model. An attacker can probe the memory channel to steal secrets traversing the memory bus. Protecting the user against these attacks requires a memory controller that ensures confidentiality and integrity.

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. Recent industry implementations, such as Intel's Software Guard eXtensions (SGX) [1], use counter-mode encryption, which uses a counter to generate a one-time-pad and encrypt data [6], [30]. However, confidentiality on data alone is insufficient to prevent physical attacks.

Integrity prevents an attacker from modifying data coming into the processor. A secure processor must verify the integrity of data stored off-chip. Intel SGX uses a small hash for each data block and an integrity tree over the counters used by the encryption mechanism; this structure is also known as a Bonsai Merkle Tree [15], [20]. To verify data integrity, the memory controller checks the data hash and traverses the integrity tree up to the root, which resides securely on chip, and checks the hashes at each level. Complementing hashes for the data, the hash tree for the counters used during encryption protect the system from replay attacks.

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 the 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. Finally, to verify the data's integrity, the memory controller fetches the data hash from memory.

Prior studies cache metadata to reduce overheads. Caching metadata in the processor's last-level cache (LLC) reduces overheads but introduces competition between metadata and data for space [5]. Caching only the counters used for decryption can shorten the critical path when speculation hides the latency of integrity verification [12], [20], but neglecting the data hashes required for every memory request increases dynamic energy costs to at least twice that of a system without secure memory.

Metadata caches complement another strategy to reduce overheads—speculatively using unverified data. Prior studies safely speculate around integrity verification latency by supplying data immediately for computation and restricting the effects of that computation until verification completes [12]. This mechanism is effective only if the verification latency is not too long. Verification may become a bottleneck if neither hashes nor tree nodes are cached.

Despite the importance of metadata caching, there is no prior work on understanding metadata access patterns. Architects have either adopted prior cache designs [5], [21], [22], [24], [25], [27] or cached only one type of metadata [20], [31]. We are the first to perform a detailed analysis of metadata access patterns. We make the following observations:

- Caching all metadata types increases cache efficiency
- Reuse distances are bimodal (*i.e.*, short or long)
- Request type is a strong indicator of reuse patterns.
- Belady's MIN [3] is not optimal for metadata caches.
- Traditional eviction policies do not work well when the metadata cache services all metadata types.

II. Background and Motivation

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.

A. Security Mechanisms

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.

Counter-mode encryption parallelizes the encryption and the memory access. While the counter is encrypted to produce a one-time-pad (OTP), the data is fetched from memory. When data is brought on-chip, the memory controller XORs the data with the OTP in one cycle. The XOR operation produces both the data's ciphertext and plaintext. The counter used to produce the OTP is used only once in the system's lifetime.

The memory controller guarantees unique one-time-pads for counter-mode encryption by maintaining one counter for each 64B-block. The counter is incremented every time it is used to encrypt data, ensuring its value is only used once. Keeping the per-block counters on-chip is too expensive—4GB of secure memory with 8B per-block counters would require 512MB of counters. Instead, current implementations of secure memory store counters in main memory [1].

To reduce the space overhead of counters, prior work uses two counters: one per-page counter and a smaller per-block counter [20], [30]. Per-block and per-page counters increment when the corresponding blocks are written and when per-block counters overflow, respectively. When the per-page counter is incremented, all blocks in the page are loaded on-chip and re-encrypted with the new page counter. Re-encrypting a whole page is expensive, but it happens infrequently and can be done off the critical path. Using per-page and per-block counters significantly reduces overheads. An 8B counter for every 4KB-page combined with a 7b counter for every 64B-block reduces space requirements from 512MB down to 64MB.

Integrity. Secure memory detects data tampering by verifying data integrity with Bonsai Merkle trees (BMTs) [20]. 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 within the chip.

The BMT enables verification of counter integrity but not of the data itself. An HMAC for each data block is also required; an 8B-HMAC is sufficient to track data integrity. The memory controller compares these hashes when fetching data from off-chip memory.

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 for the tree.

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.

B. Case for 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 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 [14] and 0.3 pJ [26] per bit, respectively. Energy overhead falls when the metadata cache services more metadata requests.

Figure 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

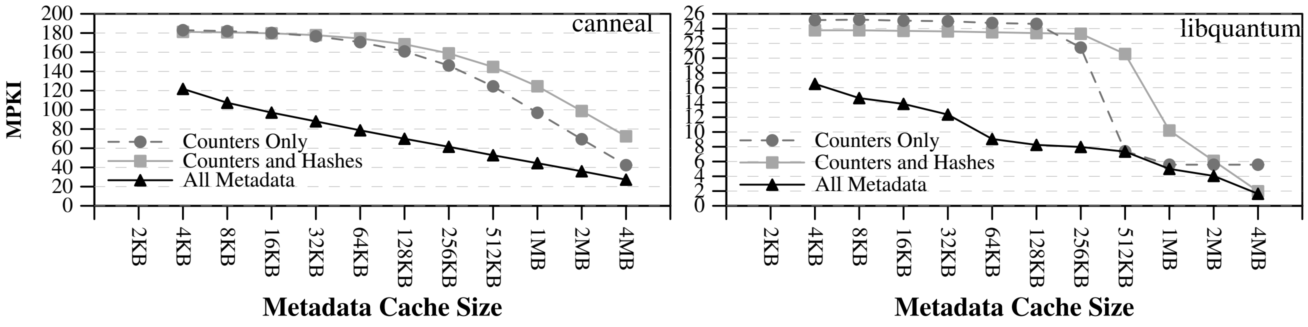


Figure 1: Metadata MPKI when caching (i) only counters, (ii) counters and hashes, and (iii) all metadata types.

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 1.

C. 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 design assumes 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.

III. Simulation Methodologies

We assume that all of memory is secure and that the memory controller speculatively supplies data for computation before its integrity has been verified [12]. We evaluate metadata cache designs using MARSSX86 for full system simulation [16] and DRAMSim2 to model memory [23]. We extend DRAMSim2 with timing models for Bonsai Merkle Trees and counter-mode encryption. We also add a metadata cache to the simulator. We use McPAT [13] to estimate processor energy, CACTI [26] to estimate metadata cache

Processor	out-of-order core
Clock Frequency	3GHz
L1 I & D Cache	32KB 8-way
L2 Cache	256KB 8-way
L3 Cache	2MB 8-way
Memory Size	4GB
Memory Latency	from DRAMSim2
Hash Latency	40 processor cycles [7]
Hash Throughput	1 per DRAM cycle

Table I: Simulation Configuration

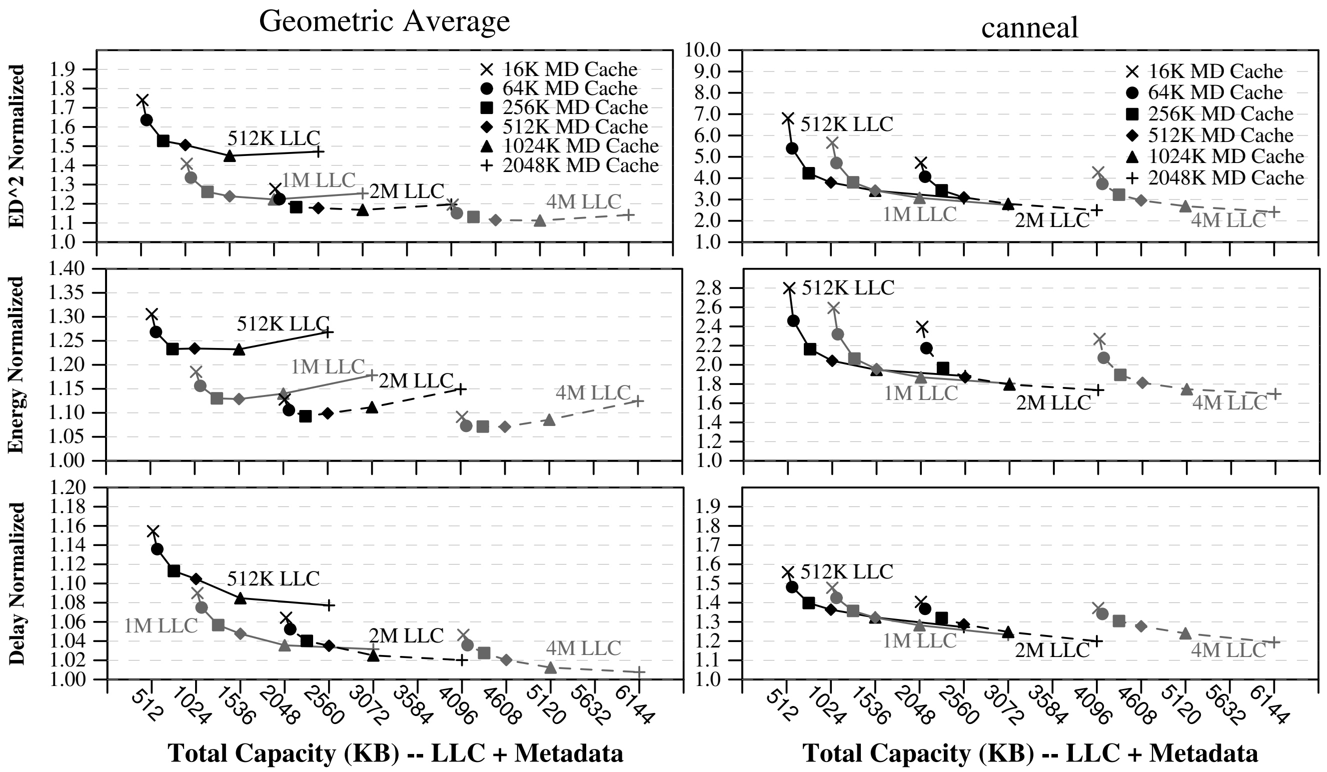


Figure 2: Comparison of LLC and metadata cache sizes. Normalized to system with 2MB LLC and without secure memory.

energy, and DRAMSim2 to estimate memory energy. Table I summarizes our experimental configuration.

We run three benchmarks suites: PARSEC [4], SPLASH2 [29] and SPEC 2006 [8]. For PARSEC and SPLASH2 benchmarks, we simulate 500 million user instructions from the region of interest. For SPEC benchmarks, we fast-forward by 1 billion user instructions and simulate 500 million user instructions. Caches are warmed up with 50 million instructions for all benchmarks. We focus our attention on benchmarks that have LLC MPKI greater than 10 because they are most affected by the security mechanism. For completeness, we also present geometric averages for all benchmarks.

IV. 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 [20], [30]. 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.

A. 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 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 speculate and hide verification latency [12]. Experiments without speculation produce the same general trend.

The results for the average (left) in Figure 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 workload needs would benefit cache efficiency.

B. 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 II 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

Metadata Type	Organization		Data Protected	
	PI [12]	SGX [1]	PI [12]	SGX [1]
Counters	1, 8B / page 64, 7b / blk	8, 8B / blk	4KB	512B
Integrity Tree	8, 8B hashes		$4 * 8^{\text{lev}}$ KB	$512 * 8^{\text{lev}}$ B
Hashes	8, 8B hashes		512B	

Table II: Metadata organization

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 [1]).

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.

C. 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 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 the data hashes. To cover a 2MB LLC, a minimum of $9 * 64\text{B} * (2048\text{KB}/4\text{KB}) = 288\text{KB}$ are needed. Figure 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 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 II). For most benchmarks, almost 90% of the tree blocks have reuse distances smaller than 4KB. A small (*e.g.*, 4KB)

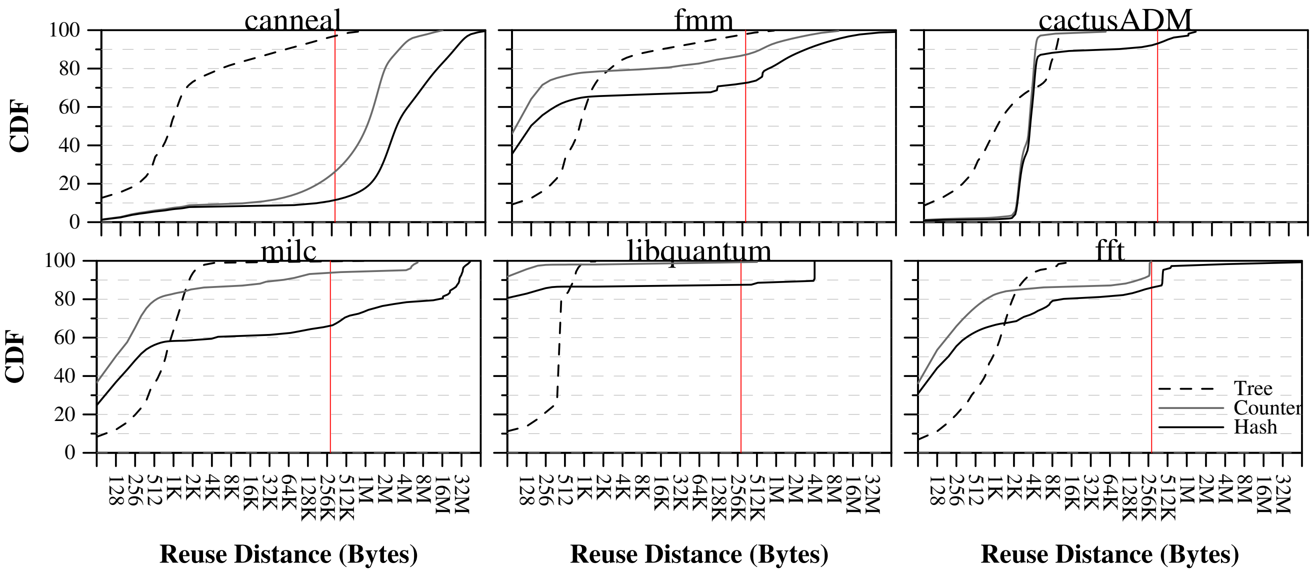


Figure 3: Cumulative distribution function (CDF) for reuse distance, split by metadata type for six representative benchmarks.

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 . *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 [14]. Because the hash block is needed for every memory access, energy costs are doubled when the hash is not cached on chip.

D. *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 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). 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 tra-

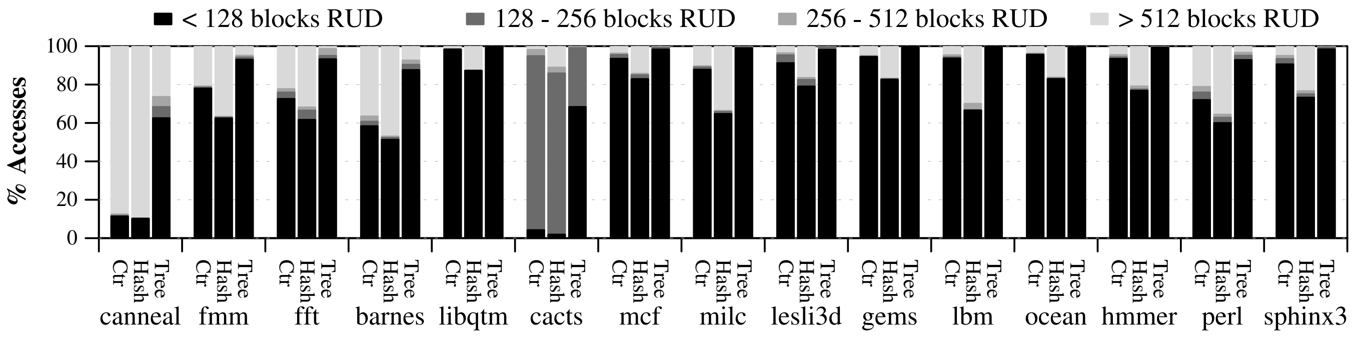


Figure 4: Classification of metadata accesses according to reuse distances.

ditional caches [2], [9], [11], [17], [28], 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 would be big enough to capture the blocks with shorter reuse distances, also regardless of any cache policy.

E. Request Types

The memory request type is a strong indicator of the block's reuse distance. Figure 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 [5]. 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.

V. 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.

A. 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 [3]. *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.

EVA uses the concept of economic value added to classify blocks according to their "age" [2]. The "age" of a block is

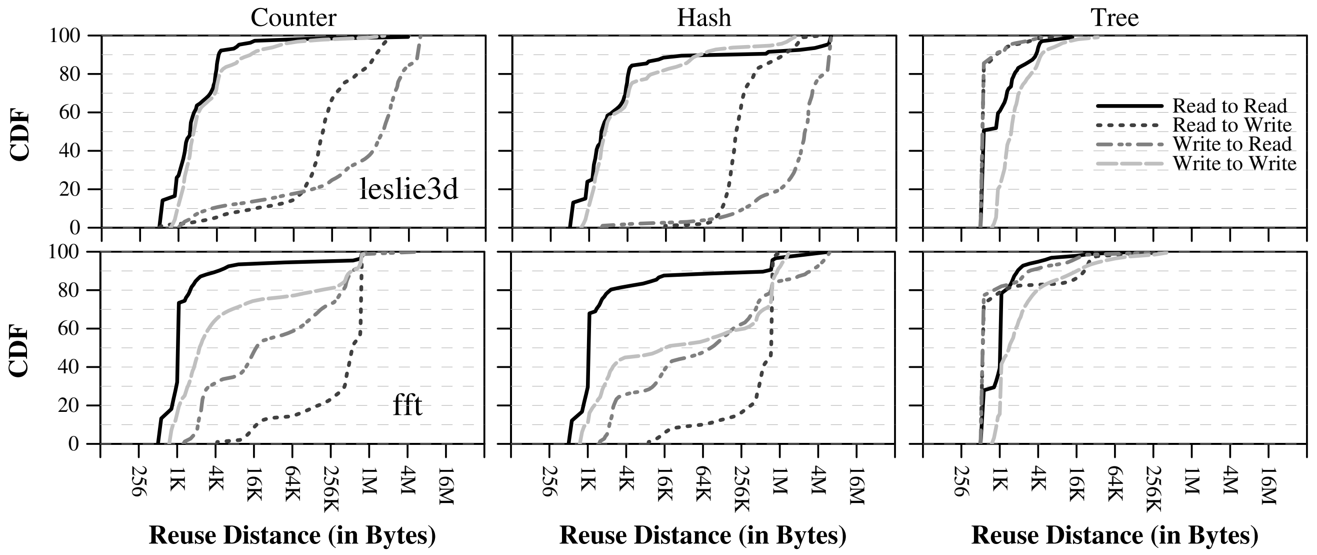


Figure 5: Cumulative distribution function (CDF) for reuse distance, split into request and metadata types.

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

B. The Optimal Eviction Policy

Belady's *MIN* algorithm [3] 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

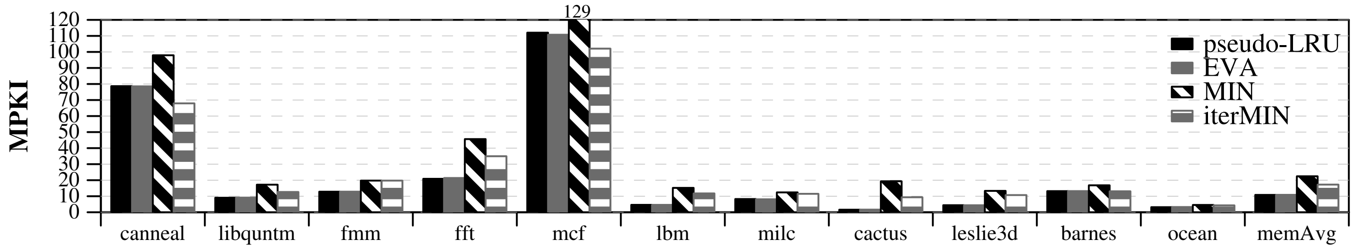


Figure 6: Metadata misses with varied eviction policies and 64KB metadata cache.

possible eviction candidates with breadth-first search [10]. 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 6 highlights the importance of considering 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.

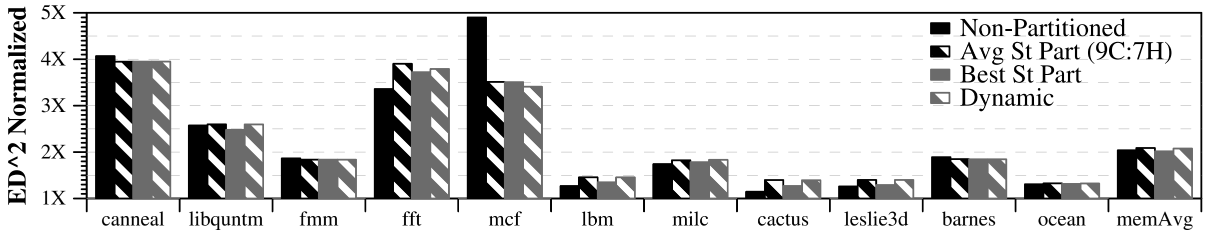
C. Cache Partitioning

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 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 [18], [19]. 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 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.



Best Static	14C:2H	7C:9H	14C:2H	14C:2H	10C:6H	2C:14H	6C:10H	4C:12H	2C:14H	15C:1H	7C:9H

Figure 7: Cache partitioning schemes and overheads from secure memory. Best static partition shown below the x-axis.

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 IV-C, 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.

VI. 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.

VII. 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.

Acknowledgments

This work is supported by the National Science Foundation under grants CCF-1149252 (CAREER), CCF-1337215 (XPS-CLCCA), SHF-1527610, and AF-1408784. This work is also supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these sponsors.

References

[1] 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.

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

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

[4] 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.

[5] 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.

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

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

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

[9] 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.

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

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

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

[13] 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 *International Symposium on Microarchitecture (MICRO)*, 2009.

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] W. Shi, H.-H. S. Lee, M. Ghosh, and C. Lu, "Architectural 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.

[25] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, "High efficiency counter mode security architecture via prediction and precomputation," in *32nd International Symposium on Computer Architecture (ISCA)*, 2005.

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

[27] 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.

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

[29] 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.

[30] 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.

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