Bonsai Merkle Forests: Efficiently Achieving Crash Consistency in Secure Persistent Memory

Alexander Freij
North Carolina State University
North Carolina, USA
atfreij@ncsu.edu

Huiyang Zhou
North Carolina State University
North Carolina, USA
hzhou@ncsu.edu

Yan Solihin
University of Central Florida
Florida, USA
Yan.Solihin@ucf.edu

ABSTRACT

Due to its durability, the security of persistent memory (PM) needs to be ensured. Recent works have identified the requirements for correctly architecting secure PM to achieve crash recoverability. A key performance bottleneck, however, lies in the integrity tree update, which needs to be consistent with the memory persistency model and incurs a very high performance overhead. In this paper, we aim to drastically reduce this performance overhead.

First, we propose to leverage a small on-chip non-volatile metadata cache (nvMC) for keeping a small portion of the integrity tree. We show that nvMC cannot be managed like a regular cache due to violating crash recoverability, and hence derive a set of invariants to be satisfied for the nvMC to work properly. Then, we propose the idea of *Bonsai Merkle Forests (BMF)*, which splits an integrity tree into multiple trees, leading to a forest, with the tree roots maintained in the nvMC. We propose and analyze different ways of BMF management. Our experimental results show that our proposed BMF schemes drastically reduce the performance overhead of BMT root updates, from 426% to just 3.5%.

KEYWORDS

persistency, security, integrity tree update, non-volatile cache, integrity forest

ACM Reference Format:
Alexander Freij, Huiyang Zhou, and Yan Solihin. 2021. Bonsai Merkle Forests: Efficiently Achieving Crash Consistency in Secure Persistent Memory. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3466752.3480067

1 INTRODUCTION

Recent non-volatile memory (NVM) or *persistent memory* (PM) products, such as DIMM-compatible Intel Optane DC Persistency Memory [18], provides a promising alternative to DRAM as main memory substrate, providing much higher density and better scaling potentials than DRAM, while providing non-volatility and byte addressability. Due to its non-volatility, data may remain in PM for long periods of time without power, exposing it to data leakage (if stored in plaintext) and unauthorized modifications (if no integrity protection) by potential attackers [10]. Therefore, the use of PM as main memory requires data to be encrypted and its integrity protected.

Furthermore, PM provides an intriguing possibility for *persistent enclaves*, that retain state across crashes or boots. In both of these uses, a *secure PM* with memory encryption and integrity verification would typically rely on *security metadata*, such as counters, MACs, and integrity trees [12, 30]. Secure PM is useless, however, if data cannot be recovered after a crash. In particular, researchers have pointed out that data must be crash-consistent with its security metadata according to the persistency model, otherwise after a crash the plaintext of data may not be recoverable, or crash recovery may trigger integrity check failures [14].

The work in [14] specified invariants that govern atomicity and ordering required in order for secure PM to support crash consistency and recovery. In essence, the invariants require that updates to security metadata (including persisting counters, persisting MACs, and updating of Bonsai Merkle Tree (BMT) root) follow the same ordering as specified by the persistency model for data. The paper proposed persist-level parallelism (PLP) optimizations that improved performance while adhering to the invariants. Unfortunately, while PLP optimizations were effective, the remaining performance overheads are still very high: 20% for epoch persistency and 109% for strict persistency were reported by the authors. The reason for the high overheads is primarily the latency of updating the BMT root which must be sequentially performed from leaf to root. For a large PM, the BMT may have a height of 8+ levels, and if each level takes 40 cycles (assuming no cache miss), it takes at least 320 cycles to update the BMT root.

There are no simple solutions to the performance bottleneck of updating the BMT from leaf to root. One may consider only protecting a portion of PM with BMT to reduce BMT size, however that comes with the need to partition memory between secure and insecure portions, placing the burden on the programmer, compiler, or OS to make accurate partitioning. One may also think of reducing BMT height by increasing the tree arity [44], but that requires smaller MACs that reduce security strength.

To arrive at a solution, we note that recent trends indicate expansion of the persistence domain to include on-chip structures. For example, Intel's enhanced Asynchronous DRAM Refresh (eADR)[34, 35] adds the cache hierarchy to the persistence domain, while Alshboul et. al [1] adds small battery-backed persist buffers. Considering such an expansion, the fundamental question we ask in this work is: *can the performance of secure PM be improved if a small on-chip non-volatile or battery-backed metadata cache is available?*

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480067

We will use the term *non-volatile Metadata Cache* (nvMC) to denote such a cache but the term is also applicable to battery-backed cache.

The possibility of having a small nvMC, coupled with the challenge of high BMT update overheads, prompted us to fundamentally rethink the BMT organization. In particular, we propose a new approach of splitting the integrity tree into many trees that collectively protect the PM. We call this approach *Bonsai Merkle Forests* (BMF), to indicate a forest topology instead of a tree. With BMF, we maintain a collection of integrity trees, each with its own root acting as the convergence point for its respective portion of memory. These roots are persisted in the nvMC, expanding the persistence domain to include a subset of the integrity tree. By doing this, tree or update path heights can be reduced, resulting in lower root update ordering overheads. We explore two variants: *Static BMF* provides a static forest configuration comprised of persisted BMT nodes, while *Dynamic BMF* dynamically reconfigures the forest topology based on access patterns. Dynamic BMF detects hot data and uses shallow trees for it, and using tall trees for cold data, thereby making the common case fast. Our Dynamic BMF efficiently provides integrity protection while protecting the entire PM and without reducing MAC sizes. Dynamic BMF reduces 79.1% of the performance overheads of BMT updates, resulting in nearly negligible overheads of 3.5% for epoch persistency, while only relying on a 4KB nvMC.

To summarize, the contributions of this work are:

- We propose a novel approach called *Bonsai Merkle Forest* (BMF), and formally specify its definition and validity criteria for integrity protection. BMF takes advantage of a small non-volatile (or battery-backed) on-chip metadata cache.
- We derive invariants for crash recoverability of PM protected by a BMF that guarantees crash consistency according to the persistency model used in the system.
- We propose a Static BMF and Dynamic BMF design; the latter reconfigures its topology in response to the detection of hot vs. cold data, that dynamically reduce the tree heights for hot data.
- We propose a design of architecture support that utilizes non-volatile Metadata Cache (nvMC) to accommodate a BMF topology efficiently.
- We evaluate and compare Static and Dynamic BMFs against BMT and found that secure PM overheads can be reduced to just 3.5% with Dynamic BMF.

The rest of the paper is organized as follows: Section 2 presents the background and related work. Section 3 derives the invariants needed to ensure correct crash recoverability and data integrity coverage. Section 4 details the analysis of our proposed Static and Dynamic Bonsai Merkle Forest mechanisms. Section 5 discusses hardware architecture and algorithm design. Section 6 presents our evaluation methodology. Section 7 evaluates our proposed mechanisms, and Section 8 concludes our work.

2 BACKGROUND AND RELATED WORK

2.1 Threat Model

Just like in prior work [10, 29, 40, 49], we assume that adversaries can physically access the memory system (bus, PM media, etc.) to perform passive snooping [45] or active tampering of values in PM [38], including replay of old valid values. Attackers may gain physical access through ownership, theft, or even acquisition after disposal [5, 29, 40, 49]. Due to non-volatility, data remanence is a bigger concern for PM than for DRAM [16, 27]. Similar to prior work [14, 22, 25, 50], we assume that attackers cannot access on-chip resources such as caches and registers, hence the physical boundary of the processor chip forms the trusted computing base (TCB).

2.2 Memory Encryption

Memory encryption conceals plaintext values written in off-chip main memory [4, 9, 20, 21, 23] or sent to other processing units [32, 47]. ECB mode [45], XTS mode [19], and counter mode [48] have all been proposed. Counter mode is commonly employed, including in Intel SGX [12] and in many recent works [3, 49, 51]. In counter mode, a pseudo one-time pad (OTP) is generated by encrypting counters which is then XORed with data plaintext to get data ciphertext, and vice versa. Reusing a counter value leads to reusing OTPs which compromise the encryption, hence counters must be incremented after each writeback to provide temporal uniqueness and concatenated with address to provide spatial uniqueness. Encryption counters can be split [47] or monolithic (e.g. in Intel SGX). Split counters co-locate a major per-page counter and many per-block minor counters in a single cache line, and each encryption counter is represented by the concatenation of a major and minor counter. Split counters offer significantly lower storage overheads and improve counter cache performance [47], and so we assume a split-counter scheme in this paper.

2.3 Memory Integrity Verification

Memory encrypted with counter-mode encryption is vulnerable to *counter replay attacks* which allow an attacker to break encryption and may reveal secret encryption keys [47]. Thus, integrity verification is required to ensure data integrity [39, 47]. Data fetched from off-chip memory or other processor chips have its integrity verified when brought on-chip [31, 32]. Memory integrity protection may rely on Merkle Tree [15], Bonsai Merkle Tree [30], or SGX Counter Tree [12]. BMT is the most space efficient of them all (hence the lowest height) as it covers only counters, hence we assume split counters protected by BMT (Figure 1).

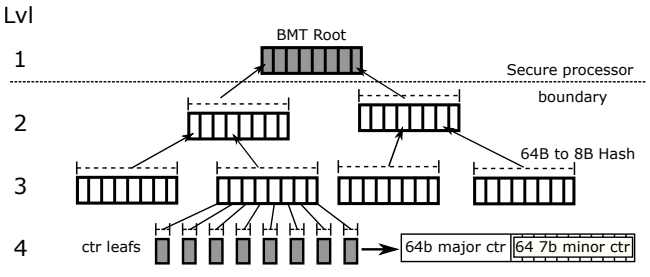

Figure 1: Example 8-ary Bonsai Merkle Tree with split counters.

2.4 Memory Persistency

Memory persistency models are defined to provide some guarantee to stores with regard to when they become durable with respect to other stores [2, 7, 8, 11, 13, 28]. These models provide a way to reason about the correctness of persistent memory state visible to a crash recovery observer. The most conservative is strict persistency (SP) which requires that durable stores follow sequential store order defined in the program. SP is intuitive for programmers, but restrictive in performance as stores cannot be overlapped or reordered. Epoch persistency (EP) and buffered epoch persistency (BEP) allow the programmer to define regions of code called epochs where stores within an epoch can be reordered and overlapped but their durability is strictly ordered across epochs [28]. In addition to memory persistency models, programmers must also define atomic durable code regions to support crash recovery [26, 33, 36, 37].

2.5 BMT Root Update Ordering

Supporting crash consistency in secure PM requires data and security metadata to be updated in a crash consistent manner. It was pointed out that data and counters must be atomically persisted [5, 51], and similarly data and MACs [40, 41]. While non-root BMT nodes do not need to persist (they can be rebuilt during crash recovery), the BMT root must be updated atomically with respect to the durable update of data and other security metadata such as counters and MACs (collectively called a *memory tuple*), and the update must be ordered following the persistency model [14]. The latter requirement incurs a serious performance bottleneck in secure PM. For example, if there are two stores that are ordered in the persistency model of choice, the BMT root update must be ordered (updated by the first store prior to updated by the second store) in an atomic manner with respect to the durable update to data and other metadata (BMT update must complete prior to durable update to data/metadata of the second store). Therefore, ordering root updates incurs high performance overhead, particularly in SP, as updates must traverse and update each level of the integrity tree. Due to this bottleneck, the performance overheads reported after various optimizations [14] are still very high: 20% for epoch persistency and 109% for strict persistency.

For this paper, we explore BMF for both strict and epoch persistency models, and compare our work against PLP [14].

2.6 Related Work

Prior work has explored skewing integrity trees [42, 46] to reduce integrity tree heights, but did not consider the impact of root update ordering for crash recoverable NVM. Taassori et. al [43] discussed per-application integrity trees and partitioned metadata caches to eliminate covert side channels, but did not consider applicability to NVM-base systems.

3 NON-VOLATILE METADATA CACHES

Intel's enhanced Asynchronous DRAM Refresh (eADR)[35] and recent work [1] point out the feasibility of battery-backed cache hierarchy or a portion thereof. Given that in a secure PM, the performance bottleneck is the atomicity and ordering requirements imposed by BMT root update, we explore the question of whether a small non-volatile or battery-backed metadata cache (nvMC) can alleviate the bottleneck. This section presents our first contribution that analyzes invariants that ensure crash recovery when using nvMC in conjunction with the BMT.

3.1 Intuition

First, let us intuit how nvMC can speed up BMT root update. In a traditional BMT, a data store that must persist to PM must also atomically persist its corresponding counter and update the BMT root. Figure 2(a) illustrates this, showing BMT root to be kept persistent on chip at all time, and counters (BMT leaves) are kept updated in PM. Since counters and BMT root are always crash-consistent, but intermediate BMT nodes are not guaranteed to be crash consistent, when a crash occurs, intermediate BMT nodes in PM are ignored/discarded. Instead, they are recomputed starting from counter leaves to root, and the computed root is validated against the persistent BMT root. Any mismatch triggers an integrity failure exception.

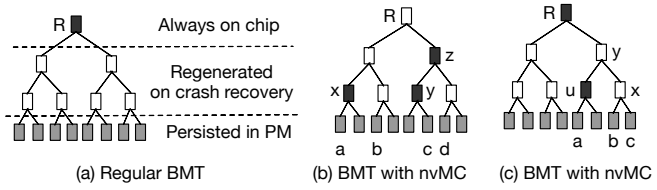

Figure 2: Illustration of BMT crash recovery, without nvMC (a) versus with nvMC (b and c).

Now consider that we have a nvMC that can keep some BMT nodes on chip persistently. Suppose that we manage nvMC just like a regular cache, where blocks are brought in on demand, and evicted after unused for a while. Hence, some BMT nodes will be present in the nvMC while others are not, including the BMT root. How can nvMC improve BMT? Figure 2(b) illustrates an example BMT with blocks containing nodes x, y, and z cached in nvMC. nvMC can provide two benefits. First, suppose that data block A is updated, resulting in counter a being updated as well. Since its parent node x is in the nvMC, *potentially* (we will analyze this further) only block x needs to be updated, without updating the path further up to the root R. This accelerates the persist of block A because of the shorter update path and that no cache miss is incurred in the update. Now suppose a crash occurs. nvMC retains its content due to its non-volatility, but everything else is lost. During crash recovery, counter block a validation can be performed by comparing the hash of a against x, which is trusted as it never left the processor chip. This accelerates the rebuilding of the BMT. Hence, potentially, nvMC can accelerate both the operation of persisting stores in normal execution, as well as the recovery process after a crash occurs.

Unfortunately, however, the scheme presented above does not work correctly. First, some nodes cannot be verified anymore during crash recovery. For example, node b cannot be verified because none of its ancestors were cached in the nvMC. Second, the ordering between persists required by the persistency model is hard to ensure anymore. For example, suppose that data blocks C and then D are modified and persisted in that order according to the persistency model. Thus, counter blocks c and d are modified as well, and the

modification propagates upward in the tree. If the update stops at the first ancestor that is cached in the nvMC, e.g. c stops at y while d stops at z, the updates cannot be ordered anymore as they stop at different BMT nodes and their update paths do not overlap. In fact, if y is evicted, it has to update node z. However, in this case, node z was updated in the opposite order of data persists, potentially violating the crash recovery correctness requirement.

Therefore, an nvMC has the potential to improve performance, but cannot be managed like a regular cache. It must satisfy some invariants that ensure correct crash recovery. In the next section, we will discuss our derivation of such invariants.

3.2 Crash Recovery Correctness Invariants

In Section 3.1, we have discussed the potential benefits and pitfalls of using nvMC. In this section, we analyze and formalize the requirements for correct crash recovery for a BMT with nvMC. We discuss exclusively nvMC for caching BMT nodes as they present the performance bottleneck of the system. Other metadata types (counters and MACs) are assumed to be cached in volatile (metadata) caches. We will start with the definition of *persistent root set*:

Definition 3.1. **Persistent root set** (PRS) is the set of BMT nodes that are stored in the nvMC and included in the persistence domain.

We refer to BMT nodes in nvMC as persistent roots because we wish that BMT updates can stop at the first ancestor node found in the nvMC instead of all the way to the real BMT root. Thus, cached nodes act like substitute roots, hence the term PRS.

Let us assume several notations. Suppose that there are N nodes in the PRS and they are referred to roots $R_1, R_2, \ldots R_N$. Let us denote $L(R_i)$ as all nodes that are leaf descendants of root R_i. We start with the first invariant that is needed for correct crash recovery:

Invariant 1. ***Covering Invariant:*** *The union of PRS leaves, $L(R)$, must be equal to the set of all counters, γ:*

$$\bigcup_{i=1}^{n} L(R_i) = \{\gamma_0, \gamma_1, \gamma_2 ...\} \tag{1}$$

Invariant 1 (covering) requires that at any time, the nvMC must contain nodes that collectively cover all counters (BMT leaves). This invariant is intuitive as non-covered counters cannot be verified during crash recovery; an example of invariant violation was earlier presented in Figure 2(b)-(c).

The next (no-leaf overlap) invariant is less intuitive. It states that two roots cannot cover the same leaf nodes:

Invariant 2. ***No-leaf Overlap Invariant****: No two roots may cover a common leaf:*

$$\forall \{i, j\} \mid i \neq j : L(R_i) \cap L(R_j) = \emptyset \tag{2}$$

We have already discussed with Figure 2(b) that when two roots cover the same leaves, persist ordering is difficult (but not necessarily impossible) to enforce. However, there is another scenario in which the overlap leads to failed crash recovery. A scenario is illustrated in Figure 2(c). In the figure, nodes u and R are in the nvMC. Suppose that node a was modified and the update stopped at root node u. Now suppose that a crash occurs. During crash recovery, node b is accessed. In order to validate b, node x is regenerated using b and c as input. Next, node y is also regenerated using nodes u and x as input. Then, the hash of y is regenerated and compared against R, which mismatches because y was regenerated using the updated u but R was never updated to reflect the change in u. This mismatch causes an integrity verification failure.

Fundamentally, the reason for the integrity failure during crash recovery is because a node serves both as a root (where updates stop) and as a descendant (to regenerate ancestor nodes). This is avoided if roots do not overlap in their leaf descendants, hence the invariant.

Invariant 2 imposes several constraints on using nvMC to persist BMT nodes. First, it constraints the placement policy of nvMC as some nodes cannot be cached if they violate the no-leaf overlap invariant. Second, it also constraints the replacement policy of nvMC as some nodes cannot be evicted if they result in some leaves uncovered. Furthermore, fetching a node block into the nvMC or evicting a node block from the nvMC may require actions to evict or fetch other blocks. In the next section, we present our second contribution of the concept of Bonsai Merkle Forest that allows the use of nvMC without violating both correctness invariants.

4 BONSAI MERKLE FORESTS

In this section, we present our second contribution: Bonsai Merkle Forest (BMF). We will show BMF designs that comply with Invariant 1 and Invariant 2. We derive two approaches that offer valid integrity forests, static and dynamic. Our initial approach explores conforming to these invariants by statically persisting BMT nodes, then explore dynamically changing the forest topology to efficiently utilize the nvMC and reduce tree heights while ensuring crash recoverability.

4.1 Splitting a BMT into Multiple Trees

The challenges discussed in Section 3 of persisting BMT nodes led to the observation that hierarchical dependencies in the BMT increases the chance of violating the no-leaf overlap invariant. To reduce inter-level dependencies and satisfy Invariant 2, we consider splitting a BMT into smaller sub-trees, with sub-tree roots acting as the convergence point for all updates from their descendants. It is reasonable to persist these sub-tree roots in the nvMC and include them in the PRS, since modifications to these roots will not be used for recomputing ancestral nodes in the BMT. The invariants discussed in [14] are still valid, but the crash recovery tuple must be reconfigured to include the persisted sub-tree root instead of the original BMT root. This means that a memory tuple is persisted once the persisted sub-tree root has been updated. The memory tuple redefinition also applies to the persist order invariant, since updates to the PRS must be ordered.

To not violate Invariant 2, when a sub-tree root is persisted, the edge between the root and its parent node is removed, causing the BMT to split into sub-trees. By splitting the BMT into multiple subtrees, each root covers a subset of integrity tree leaves, and the roots should collectively adhere to Invariant 1. With this observation, we first explore a static root set that ensures both invariants are followed and effectively reduces integrity tree update overheads.

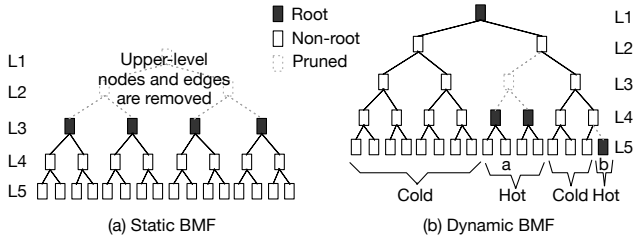

Figure 3: Examples illustrating two valid BMFs fitting in a 4-entry nvMC: Static BMF with Level 3 persisted (a), and Dynamic BMF (b).

4.2 Static Bonsai Merkle Forests

Recall that the sub-tree roots act as convergence points for their respective leaf updates and the BMT root acted as the convergence point for all leaf updates. A set of roots must be found that provide a convergence point for any leaf update while not overlapping with any other root. A prime candidate to fulfill these requirements would be a set of sub-tree roots that make up a single level in the BMT.

Figure 3 shows an example of two BMFs, showing PRS that must fit into a 4-entry nvMC. Counter leaves are not shown but would form level 6 if shown. *Static BMF* has a fixed forest topology that has a pre-selected PRS at boot time and does not change with time. *Dynamic BMF* has a forest topology that changes with time based on identification of hot vs. cold data. Figure 3(a) shows a Static BMF with all level 3 (L3) nodes selected as the PRS. If an update occurs at one leaf, it would only traverse the tree up to one of the PRS and updates it, before persisting memory tuples in main memory. Intermediate nodes in L4 and L5 are not cached in the nvMC; they may be cached separately, either in a volatile BMT cache, or kept in the same nvMC cache but without battery backing.

It is straightforward to see that the example Static BMF in Figure 3(a) satisfies the covering invariant as all leaves are covered by one of the PRS nodes. The no-leaf overlap invariant is also met because all members of the PRS are in a single level (L3), and all upper level (L1 and L2) nodes are removed. All updates to the leaves will traverse up to the persisted level L3 but not beyond. The L3 nodes form permanent PRS and therefore their cache blocks are pinned (i.e., never evicted) in the nvMC. In the example in Figure 3(a), the tree height has decreased from 5 levels in the original BMT to only 3 levels in the Static BMF, resulting in a tree height reduction of 40%.

4.3 Dynamic Bonsai Merkle Forests

A drawback of the Static BMFs is the possible nvMC under-utilization, as roots that are rarely updated occupy space in the nvMC just as roots that are frequently updated. It is well known that at any given time, most programs frequently access a small fraction of the total memory allocated to it. This active working set exhibits strong temporal and spatial locality while it is "hot". Ideally, we want to prioritize keeping roots for hot data, even reducing the tree heights for such data. By keeping tree heights low for hot data and high for cold data, we can use utilize the nvMC more effectively. To achieve that, we propose a Dynamic BMF scheme.

Figure 3(b) shows an example of Dynamic BMF with four nodes selected as roots (PRS). These roots were designated based on node access frequency, and results in a PRS consisting of one L5 node, two L4 nodes, and one L1 node, collectively covering all leaves. Mixing roots from different levels would normally result in violation of the no-leaf overlap invariant, since one root may be a descendant of another. However, we apply a novel observation that if some nodes and edges are *selectively pruned*, the invariant can be met, as can be observed in Figure 3(b), where all roots cover different sets of leaves. By pruning a node, all updates to the node's descendants only traverse up to that persisted node instead of the BMT root.

The selected PRS significantly reduced the tree traversal heights for hot data, allowing updates to only compute one to two levels of the BMF before persisting. For example, an update to node *a* updates only two levels of a tree, while an update to node *b* updates only one level of a tree. Suppose that 95% of all updates are to hot data spread equally over the five L5 nodes covered by roots, and only 5% of updates are to cold data. In this example, the average height is $0.95(0.2 \times 1 + 0.8 \times 2) + 0.05 \times 5 = 1.96$. This would be a reduction of 35% in average tree heights compared to Static BMF, and 61% reduction compared to the original BMT.

To dynamically change elements in the PRS, the requirements in both invariants need to be guaranteed at all times. Inserting and removing elements from the PRS requires new placement and replacement policies to be introduced: *prunes* removes edges between BMT nodes and inserts designated nodes into the PRS, and *merges* remove nodes from the PRS by re-inserting edges. Prune logic determines valid nodes to write to the nvMC and added to the PRS, while the merge logic determines which nodes can safely be evicted from the nvMC. Both of these mechanisms are required to satisfy Invariant 1 and Invariant 2, and will be discussed further in Section 5.

4.4 Using BMF for Crash Recovery

BMF roots are persistent and considered trusted because they do not leave the chip. If a new node is brought in as a new root, it is verified prior to use. Hence, on a crash, any valid nodes in the nvMC are considered trusted roots. These roots carry sufficient information to reconstruct the entire BMF topology, hence topology information needs no redundant storage. For crash recovery, the topology is first recovered starting with the lowest level roots. After that, counter leaves are either validated if their parent is a root, or its non-root parent regenerated. All non-root nodes are regenerated.

5 ARCHITECTURE DESIGN

In this section, we discuss architecture designs to enable persisting integrity tree roots. Our baseline architecture assumes a separate counter cache [47], BMT cache [5, 14], MAC cache [49], and ADR-backed write pending queue (WPQ) [24]. These structures enable supporting baseline strict and epoch persistency models for security metadata updates. Our architecture introduces a separate non-volatile cache located in the memory controller (MC) to store sets of roots on-chip, with per-cacheline non-volatile tag and data entries and volatile access counters. We also introduce per-cacheline

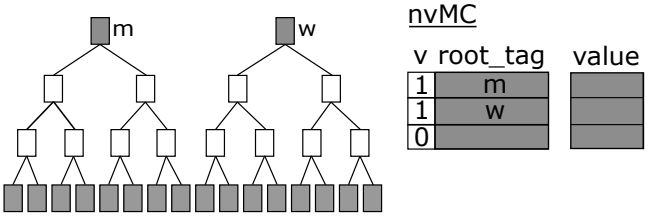

Figure 4: Static BMF with two subtrees. Regions in the persistent domain are shown in grey

access counters for BMT cache, and additional prune and merge logic to determine prune and merge targets respectively.

5.1 Static BMF Root Selection

To support our Static BMF policy, we introduce an nvMC located in the MC that interacts with the WPQ. Memory tuple components are gathered in the WPQ and persisted to main memory when all ciphertext, counter, and MAC updates have been written and an acknowledgment from the nvMC for root update completion has been received. This guarantees that memory tuples are treated as atomic persists and that the system is crash recoverable.

Figure 4 shows an example BMF with two persisted roots and a three-entry nvMC. Just as a cache, nvMC keeps a block with a tag, data, and state. The tag, state (e.g. valid bit), and data fields of the nvMC cachelines are either non-volatile or battery-backed to support integrity verification during crash recovery. As discussed in Section 4.2, one full level of the BMT nodes can be persisted in the nvMC. This requires the nvMC capacity to be sufficiently sized to hold them. Figure 4 illustrates an example of a three-entry nvMC which is was large enough to persist L2 nodes of the BMT, but is under-utilized. To keep L3 nodes of the BMT in the nvMC, it must have four entries. L4 nodes requires eight entries, L5 nodes requires 16 entries, etc. In general, for a k-ary BMT, in order to keep the next level nodes as roots, the nvMC size must increase by a factor of k. This is a key drawback of Static BMF; it takes exponentially larger nvMC sizes in order to reduce the tree height linearly.

5.2 Dynamic BMF Root Selection

A Dynamic BMF requires identifying hot and cold data, and for the forest topology to adapt by reducing the tree heights for hot data. This identification needs to be performed periodically. We define root evaluation interval (REI) as the interval (measured in the number of writes to PM) after which the BMF topology is re-evaluated and nodes are inserted into (pruned) or removed from (merged) the persistent root set (PRS). Only PRS elements are written in the nvMC, and so a prune consumes a cacheline in the nvMC, while a merge frees a cacheline. During an REI, candidate roots to be pruned/merged are identified based on the frequency of roots in nvMC being updated, hence we add a small saturating counter to each nvMC cacheline. The counters are decayed (e.g. by bit shifting) at the beginning of each REI. In addition, non-root nodes in the BMT cache are also augmented with counters.

Selecting an optimal BMF given an access pattern is a complex problem that cannot be solved quickly in hardware. To avoid the complexity, we constrain our Dynamic BMF design in the following ways. First, at each REI, only one node can be selected as a prune target, and only one node can be selected as a merge target. Hence, the prune and merge can be performed quickly in a time-bounded manner, as the forest topology is only altered incrementally at each REI. Second, a prune target may be replaced in the PRS by one of its children nodes. Hence, non-child descendants or the replacement by more than one child are avoided. Third, the original BMT root node is pinned to the nvMC even though it may not be a member of the PRS. The reason for this is to avoid cascading placement and replacement (to be discussed in more details later). Finally, we assume that nvMC is a fully associative cache. This assumption ensures that a BMF node can be replaced by a different node in the same cacheline, without violating the cache indexing function. With these constraints, we will now discuss the prune and merge operations in more details.

5.2.1 BMF Prune. Algorithm 1 shows how BMF prune is implemented. At each REI, all current root counter values that exceed threshold T are considered (lines 1-3). Among them, select the root with the highest counter value as the *prune target* P (line 4). Next we have two cases to handle. Suppose that P is not the original BMT root (line 5), then P is added back into its ancestor's tree by restoring its path to its ancestor A and updating all nodes along the path (lines 6-7). Note that because we pin the original BMT root node in the nvMC, an ancestor A is guaranteed to be found. Now, we are ready to create a new root to replace P. For that, a child C is selected among P's children that has the largest counter value, indicating the hottest child node (line 8). The child is then added into the nvMC as a new root by removing the link between P and C (line 9). In a special case where P turns out to be BMT root, P does not have an ancestor. Hence, in this case, to ensure covering, we add all P's children as new nvMC roots (line 11). Finally, P is removed from the nvMC (line 12). In hardware implementation, adding a child C and removing P from the nvMC can be performed by just replacing the tag and data of P with the tag and data of C, and this can be done easily because of nvMC's full associativity.

Algorithm 1 BMF Prune algorithm executed at each REI.

```
1:  Select roots from nvMC whose counters exceed T
2:  if None found then
3:      return
4:  Select a root with the largest counter value as prune target P
5:  if P is not BMT root then
6:      Find P's first ancestor A along the original update path
7:      Restore edges from P to A and update nodes from P to A
8:      Select a child C with the largest counter value
9:      Add C into the nvMC as a new root
10: else
11:     Add all P's children into nvMC as new nvMC roots
12: Remove P from the nvMC
```

Figure 5 shows the architecture design and an example of a prune operation. First, the roots in the nvMC are accessed to determine the most frequently accessed root in the PRS ①. If a root's counter does not exceed the preset threshold T, the next root is read in. This determines that node w is the prune target in our example, since its access counter value of 15 is the largest. As nvMC roots are accessed, their access counters are shifted and updated in the nvMC ②. The prune target's children addresses are generated and the BMT cache is accessed to determine the persist candidate ③.

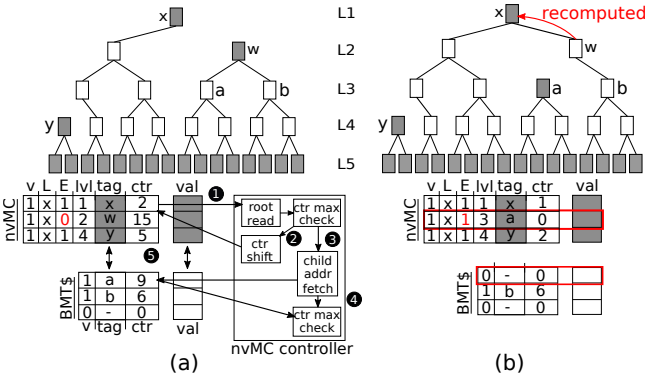

Figure 5: Illustration of a BMF prune operation. The forest topology, nvMC, and BMT cache states before and after prune are shown in part (a) and (b), respectively. *x*'s denote don't cares.

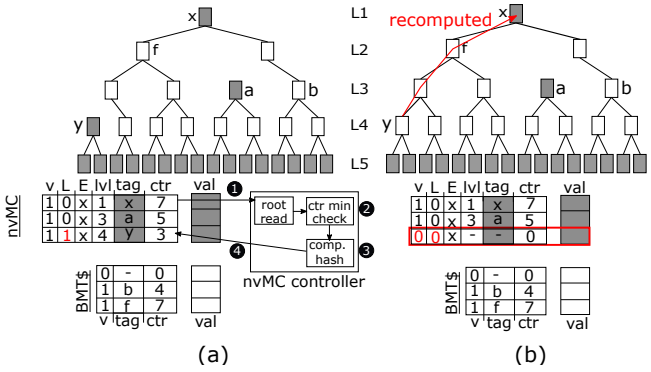

Figure 6: Illustration of a BMF merge execution. The forest topology, nvMC, and BMT cache states before and after the merge operation are shown in part (a) and (b), respectively. *x*'s denote don't cares.

Next, the prune target's child with the largest counter value will be selected as a new root ④. Since a has the larger counter value (9) of w's children, it is selected to be persisted, since this indicates that a is hotter than b (6). The prune proceeds by restoring w's edge to its ancestor x and updating it, which re-establishes w as a descendant of x, thus requiring future updates to descendants of w to traverse up to x. In the mean time, a is added as a new root by removing its edge with w. Without the edge between a and w, future updates do not need to traverse up to w, as the updates have been persisted once a is recomputed. The resulting new BMF topology is shown in Figure 5(b). The tree height for leaves converging at a has been reduced by one, which accelerates future updates to hot data covered by a. After pruning, a starts with a zero counter value in the nvMC and is removed from the BMT cache ⑤.

5.2.2 BMF Merge. Algorithm 2 shows how a BMF merge operation is implemented. A merge is triggered for two reasons: to consolidate trees covering cold data into a larger one, or to free up some space in the nvMC. In either case, the operation works the same. First, all current root counter values are read and roots with the smallest counter value, indicating that they cover the coldest data, are identified (line 1). If more than one such roots are identified, we break the tie by choosing the first lowest level root in the tree (lines 2-5) as the *merge target* M. The lowest level is chosen to break the tie because it is the least impactful root in the current PRS, since it provides minimal counter coverage. Next, we find M's first persisted ancestor A (line 6), restore the edges along the original path between them, and update A along the original path (line 7). M's cacheline in the nvMC is then invalidated (line 8), removing M from the PRS and freeing an entry in the nvMC.

Figure 6 shows the architecture design and an example of the merge operation with node y as the merge target. This situation may occur if node x was selected as the prune target, but no free entries were found in the nvMC to insert its child f, and so a merge is required. After iterating through the nvMC counter values ①, y is selected as the merge target since it has the lowest access counter (3) of the current root set, indicating that it is colder relative to other roots ②. The merge proceeds by restoring edges and updating

Algorithm 2 BMF Merge algorithm.

```
1: Select a root with the lowest counter in nvMC
2: if Multiple roots have equal counters then
3:     Select lowest root in tree
4:     if Multiple roots meet criteria then
5:         Select first root as merge target M
6: Find M's first ancestor A along the original update path
7: Restore edges from M to A and update all nodes in the path
8: Invalidate M's cacheline in nvMC
```

all nodes in the path from y to x since it is the first persistent ancestor found ③. Once x is updated, y's cacheline in the nvMC is invalidated, freeing the entry for the prune to proceed ④. The updated forest topology is shown in Figure 6(b). The access counters in the nvMC and BMT are not updated after a merge is performed, as the access state should not be changed for hot nodes when deciding which cold nodes to evict.

Earlier, we discussed that the original BMT root must be pinned in the nvMC. Pinning the BMT root in the nvMC guarantees that a persistent ancestor is available for merge targets to converge to. This simplifies the design of the merge operation by preventing cascading prune/merge cycles. Consider a situation where a nvMC is fully occupied by roots, and at an REI, the prune target selected does not have an ancestor. To ensure covering, the prune target must bring in all its children as new roots. This triggers a new merge operation to free up space for new roots. The merge, in turn, requires an ancestor to be found in the nvMC. If an ancestor is not found, it must be brought on chip, which then triggers another merge operation. This process can involve many prune/merge operations in a single REI, with the possibility of leading to a deadlock. To avoid this situation, we pin the original BMT root in the nvMC.

5.3 PRS Modification Atomicity

A PRS modification, i.e., a prune or a merge, is executed in multiple stages. These stages do not need to be atomically executed. It is achieved with two state bits, which are appended to each nvMC cacheline: an evictable (E) bit and a locked (L) bit, used for a prune and merge, respectively. These two bits ensure that the invariants

are not violated if a crash happens in the middle of a merge or prune.

Consider a situation where a crash occurs during a prune operation. If the prune target was evicted from the nvMC before the new root was written and a crash occurs, Invariant 1 would be violated because not all leaf nodes would be protected. To guarantee that leaf coverage is not compromised, our prune mechanism uses the state bit, *E* (Figure 5), which indicates if the corresponding nvMC cacheline can be evicted. The *E* bits are controlled in the following steps:

(1) After the prune target and new root have been found, the new root is fetched from the BMT cache and written to a free entry in the nvMC. The *E* bits for both prune target and new root are cleared, meaning that they cannot be evicted from the cache at this time. This occurs after ④ in Figure 5.
(2) Once the new root write operation has completed, the *E* bit of the prune target cacheline is set.
(3) The prune target cacheline is invalidated and evicted from the nvMC.
(4) The *E* bit of the new root cacheline is set. This completes the transaction and makes the new root an eligible prune target similar to other PRS elements.

These E bits ensure that the nodes involved in a prune operation are locked in the cache until the operation has completed. If a failure occurs after Step 1, the prune target and new root would reside in the nvMC with both cacheline's *E* bits set. This transient state does not violate Invariant 2, as a subset of the prune target's leaves now correspond to the new root and prevents leaf overlap between the prune target and new root. Similarly, the transient state does not violate Invariant 2 if a crash occurs between Step 2 and 3. Setting the *E* bit of the prune target in Step 2 is completed first to allow the prune target to be evicted in Step 3. Once the prune target is removed from the nvMC, setting the *E* bit of the new root in Step 4 establishes it as a PRS element and valid root. If a crash occurs between Steps 3 and 4, the new root has already been written to the nvMC and designated as a root, therefore integrity coverage is not compromised and Invariant 1 is obeyed.

A race condition may arise when while executing a merge. If updates to the merge target execute concurrently while updating the next persistent root during a merge, the merge target's new value won't be reflected in the next persistent root. Once the merge target is invalidated, this results in the merge target's descendants being unverifiable, as updates only traversed to the merge target and not the next persistent root. This violates Invariant 1 as not all leaves are verifiable and protected by the PRS. To prevent this, we use a cacheline lock bit, *L* (Figure 6), to ensure that the integrity coverage of the PRS is not compromised in the event of a crash during a merge operation. *L* indicates that the the designated nvMC cacheline is locked, preventing cacheline eviction and updates to the node when *L* is set. The *L* bits are controlled in the following steps:

(1) After the merge target has been found, updates to the merge target must be stalled. This is achieved by setting *L* for the merge target cacheline, which prevents eviction and updates to the node. This occurs after ② in Figure 6.
(2) Once the next persistent root has been updated, the merge target's *L* bit is cleared.
(3) The merge target cacheline is freed by invalidating the cacheline (Figure 6). This unblocks all updates to the merge target and all updates traverse to the next persistent root.

The transient state created by Step 1 forces all updates to the merge target to stall. This preserves the PRS and the root value in the event of a crash during a merge operation. Since the merge target is the least frequently accessed root in the PRS, stalling only affects a small number of updates which introduce negligible overheads. Concurrent updates to other segments of the tree are still allowed to complete, as they do not impact the value of the merge target and can be persisted once they reach their respective root. If a crash occurs between Step 1 and 2, the PRS elements and values remain unchanged since the merge target was not removed and leaf coverage is uncompromised. Similarly, integrity coverage is not compromised if a crash occurs between Step 2 and 3 because the merge target was not removed from the PRS, thus adhering to Invariant 1. Once Step 3 completes, the merge target is no longer a valid root, and all the stalled updates are now unblocked and proceed to update the BMT.

5.4 Persisting Intel SGX Roots

Intel SGX integrity trees (SITs) employ a tree of counters to verify memory integrity. Similar to BMTs, Intel SIT leafs cover encryption counter pages with stateful MACs that detect tampering efforts. However, an update to a SIT node requires the parent counter value to compute a child's MAC, whereas BMT node computation only requires its children values. In order to support crash recoverability, the correct parent counter value must be available to verify memory integrity. This requires all nodes from leaf to root to be persisted for integrity verification, instead of just the SIT root.

Persisting SIT nodes as roots requires several changes compared to BMF. First, when a prune occurs between a persisted node and its child, the child MAC is no longer updated with counters from the parent. Since the node is persisted and on chip, there is no need to recompute the MAC except if a merge occurs. However, when a persisted root is selected as the merge target and the path edges must be reinserted, the entire path from the merge target to the first persisted ancestor must be fetched from memory and updated, incurring significant memory traffic. We leave the full exploration of forest topologies in Intel SITs for future work.

6 EVALUATION METHODOLOGY

To evaluate our scheme, we built a cycle-accurate simulation model with Gem5[6]. The environment parameters that we assume are listed in Table 1. For all schemes, integrity verification of a newly fetched blocked is overlapped with decryption and data use, similar to [14, 22, 50]. An exception is raised if integrity verification fails. We assume separate metadata caches for counters, MACs, BMT nodes, and BMF roots.

We evaluate two persistency models: strict persistency (SP) and epoch persistency (EP). For SP, we implement write-through caches to issue stores to the MC in program order. The architecture assumed for EP includes all optimizations mentioned in the state-of-the-art PLP [14] that includes epoch tracking tables (ETT) and

Table 1: Simulation Configuration

Processor Configuration	
CPU	1 core, OOO, x86_64, 4.00GHz
L1 Cache	8-way, 64KB, 64B block, Access latency: 2 cycles
L2 Cache	512KB, 16-way, 64B block, Access latency: 20 cycles
L3 Cache	4MB, 32-way, 64B block, Access latency: 30 cycles
WPQ	32 entries
Volatile Metadata Cache Configuration	
Counter Cache	128KB, 8-way, 64B block
MAC Cache	128KB, 8-way, 64B block
BMT Cache	128KB, 8-way, 64B block
Non-Volatile Root Cache Configuration	
nvMC	battery-backed SRAM {512b, 4KB, 32KB, 256KB, 16MB}(default 4KB) Fully associative, 64B block Access latency: 2 cycles
REI	{2, 4, 8, 16, 32, 64} (default 32)
BMT	8 levels
MAC Latency	40 processor cycles[22, 38]
NVM Parameters	
Memory	8 GB PCM, 1200MHz Write queue: 128 entries, read queue: 64 entries tRCD/tXAW/tBUSRT/tWR/tRFC/tCL: 55/50/5/150/5/12.5ns[14, 24]

persist tracking tables (PTT), allowing two concurrent epochs while enforcing ordering between them. An *sfence* instruction is emulated to enforce persist ordering between stores (for SP) or epochs (for EP). We model 8GB NVMM with an 8-ary BMT for integrity verification, resulting in an 8-level BMT.

For the nvMC, we assume a battery-backed SRAM technology with parameters shown in Table 1. nvMC is fully associative and the sizes are selected to match the total size of a particular BMT level in order to cater to the Static BMFs(*sbmf*) scheme. For (*sbmf*), the system was initialized with that BMT level allocated in the nvMC. For Dynamic BMFs (*dbmf*), the nvMC is originally populated with just the the original BMT root, and subsequent prunes and merges change the topology at each REI.

To study the source of performance improvement, we analyze the average tree heights observed with our *dbmf* implementation. We also conduct a sensitivity study that varies the nvMC capacity to study the maximum achievable speedup with *sbmf* and *dbmf*. We also vary the REI frequency to study the impact of varying the impact of REI. Static and Dynamic BMF nvMC configurations require a persistent 56b tag store (assuming 64b addresses) and 64B data store, with the addition of a volatile 6-bit access counter co-located in cachelines for Dynamic BMFs. BMT caches require the addition of an access counter per cacheline to enable prune logic, and we assume a 6b counter per cacheline. All access counters in the nvMC are shifted to prevent counter overflow and reduce counter inertia between REIs.

Benchmarks We use 20 representative benchmarks from SPEC2006 [17] to evaluate our proposed BMF update models. All models are fast forwarded to representative regions and the next 100M instructions are simulated.

Evaluated Schemes The schemes used for evaluation are listed in Table 2. The baseline *secure_wb* applies uses secure memory but without persistency. *sp* and *o3* represent schemes utilizing BMT for strict persistency and epoch persistency similar to the state-of-the-art PLP [14]. *sp* includes sequential BMT root updates, while *o3* includes out-of-order BMT root update and update coalescing optimizations, which are the most aggressive ones from PLP. Our BMF schemes are labeled as *sbmf* (static) and *dbmf* (dynamic) with two sets of results for each persistency model. We also evaluate two memory configurations for each model: *'_full'* indicates protection of the entire PM, while the other excludes the stack from the protection, as stack holds temporary data that likely will not need to be persistent. The latter is the default.

Table 2: Evaluated Schemes

Name	Persistency Models
secure_wb (baseline)	Secure processor scheme with write-back caches and NVMM, which does not support any persistency model
sp	Strict persistency with sequential updates to full-height BMT
o3	Epoch persistency with out-of-order updates to full-height BMT within an epoch, but in order across epochs
Name	**BMF Configurations**
sbmf	Static Bonsai Merkle Forest with predetermined BMT height applied to sp/o3 models
dbmf	Dynamic Bonsai Merkle Forest with dynamic BMT heights and root designations applied to sp/o3 models

7 EVALUATION RESULTS

7.1 Summary

The overall performance results are shown in Table 3, which show the slowdown ratios over baseline, caused by our Static and Dynamic BMF schemes compared to the state-of-the-art BMT schemes with PLP optimizations [14].

Table 3: Performance overheads over baseline caused by our Static and Dynamic BMF schemes versus those caused by the state-of-the-art BMT schemes with PLP.

Epoch Persistency			
Memory coverage	o3	o3-sbmf	o3-dbmf
Non-stack	8.5%	5.8%	3.5%
Full	59.1%	33.5%	19.3%
Strict Persistency Model			
Memory coverage	sp	sp-sbmf	sp-dbmf
Non-stack	426%	345%	89%
Full	1,864%	1,409%	567%

Consistently across all cases, our Dynamic BMF reduces the performance overheads of BMT by 50-75%. The best performing configuration for our scheme is with epoch persistency, showing a

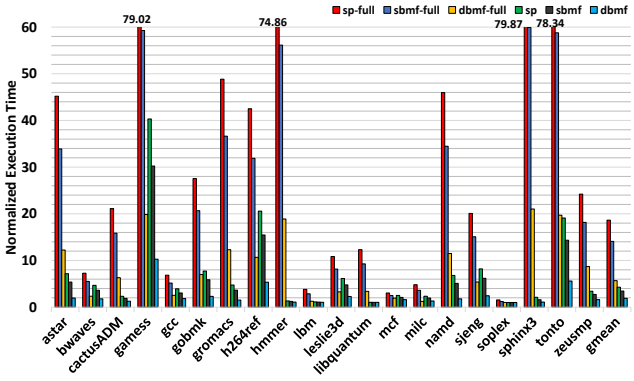

Figure 7: Execution time of strict persistency models with Static and Dynamic BMF schemes normalized to secure_wb

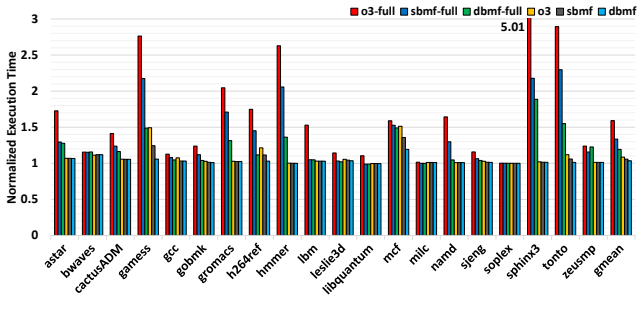

Figure 8: Execution time of epoch persistency models with Static and Dynamic BMF schemes normalized to secure_wb

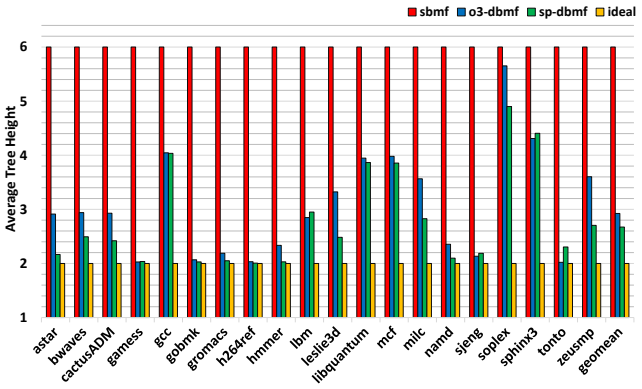

Figure 9: Comparison of average update path height for Static and Dynamic BMFs with strict and epoch persistency models.

performance overhead of only 3.5% (19.3% for full memory) compared to the *secure_wb* baseline without any persistency model. For strict persistency, *dbmf* shows 89% overhead (567% for full memory), reducing overheads by 79.1% (69.6% for full memory) compared to SP without *dbmf*. Our *sbmf* model with EP demonstrated 5.7% overheads (33.5% for full memory), while SP showed 3.45× slowdown (14.1× for full memory). Now we will analyze the performance of our schemes in more detail, followed by analyzing tree height reduction in *dbmf*s and performance by varying key design parameters.

7.2 Results for Strict Persistency

Strict persistency has not been recognized as necessary due to its performance overheads, but we evaluate it because it represents the worst-case scenario in terms of persist ordering as each store is atomic durable and ordered w.r.t others. Figure 7 shows the execution time of conventional BMTs (*sp_full*) and compared to our Static BMF (*sbmf*) and Dynamic BMF (*dbmf*), normalized to the *secure_wb* baseline, for full memory coverage (first three bars) and non-stack memory only (last three bars). Across all benchmarks, *sbmf* strictly outperforms *sp* due to the BMF height reduction over BMT, while *dbmf* strictly outperforms *sbmf* very substantially, demonstrating Dynamic BMF effectiveness.

To better understand the source of performance overheads, we measured the persist rate in terms of persists per kilo instruction (PPKI), and found strong correlation between PPKI and the amount of slowdowns with BMT. For example, *gamess* suffers 79× slowdown with sp-full because its PPKI is very high (100.72 for full memory, 51.38 PPKI for non-stack). With 8-level BMT, updating the BMT root from leaf to root takes $8 \times 40 = 320$ cycles. The estimated IPC is $\frac{1000}{320 \times 52} = 0.06$, close to the actual IPC of 0.061, indicating that the overhead is dominated by BMT root update ordering. The result is consistent with what is reported in [14]. With SBMF, 4KB nvMC allows the reduction of tree height by 2 levels (or 25%). The resulting back-of-envelop IPC estimation is $\frac{1000}{240 \times 52} = 0.08$, again very close with the observed IPC of 0.081. This results in a slowdown of 30.2× which is a reduction of $\frac{40.3-30.2}{40.3} = 25\%$, confirming our results. In general, memory-intensive benchmarks exhibit high overall performance improvement. Among them, *mcf* is predominantly read intensive and the impact of BMT updates is more limited than others. On the other hand, DBMF performance is harder to analyze as the update tree heights depend on data and changes over time.

7.3 Results for Epoch Persistency

Figure 8 shows the execution time of various schemes for epoch persistency, normalized to the non-persistent baseline for full memory protection (first three bars) and non-stack only (last three bars). We find that the *o3* model incurs an average overhead of 8.5% (or 59.1% for full memory), which is much lower than strict persistency due to the overlapping of BMT updates within an epoch and persist coalescing that reduces the total number of persists to PM. We observed a reduction in PPKI from 27.27 (113.75 for full memory) to 11.18 (33.97 for full memory) when comparing *sp* to *o3*.

Despite having less room to improve, our *sbmf* and *dbmf* schemes substantially reduce the performance overheads: compared to *o3*, *sbmf* cuts the overhead by 31.7%, reaching only 5.8% (33.5% for full memory). However, *dbmf* cuts the overhead even more by 58.8% to 3.5% (19.3% for full memory protection). The performance improvements of our schemes can again be attributed to the significant reduction in update path heights, which we will discuss next.

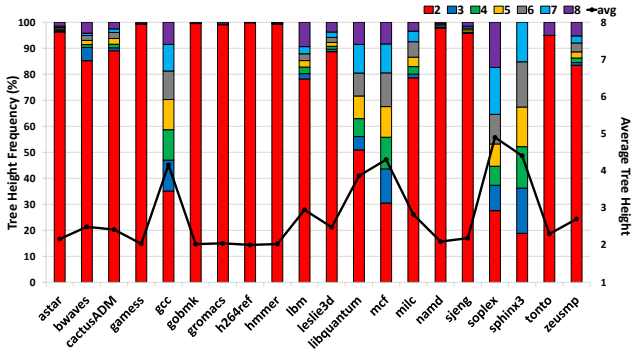

Figure 10: Update path height distribution and average tree heights for SP with our DBMF scheme.

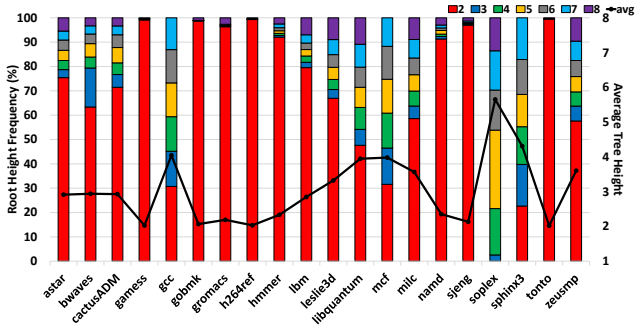

Figure 11: Update path height distribution and average tree heights for EP with our DBMF scheme.

7.4 Tree Height Reduction Analysis

To understand why Static and Dynamic BMF are effective in reducing performance overheads, we plot the average update path heights in Figure 9 for *sbmf*, *dbmf* for epoch persistency (*o3-dbmf*) and strict persistency (*sp-dbmf*), and for the ideal case where nvMC size is unlimited (all counters' parent nodes are roots). The tree height is always 6 for *sbmf* due to the 4KB nvMC holding all level 2 nodes as BMF roots. *dbmf* in most cases reduces the average tree heights to near ideal of 2. Compared to a conventional BMT, the average tree height is reduced by 87.4%. Only in one benchmark (*soplex*) the tree height only goes down to 4.6. However, *soplex*'s performance overhead is already very low. Upon closer examination, we found that the number of REIs triggered was several orders of magnitude lower compared to other benchmarks due to a very low PPKI, hence Dynamic BMF has not had much chance to adjust the BMF topology. When we reran *soplex* with a longer simulation window (500M instructions), the PPKI increases and the average tree height goes down from 4.6 to just 2.7, indicating *dbmf*'s effectiveness in adjusting the BMF topology.

Figures 10 and 11 show the breakdown of update/tree heights for SP and EP with our Dynamic BMF, respectively. The figure shows that on average, about 80% of the time, the tree heights are only 2, meaning that a change to counter value updates only its parent, which demonstrates that Dynamic BMF is extremely effective in reducing tree heights. Dynamic BMF reduces tree heights slightly more effectively in SP than in EP; the reason is that in SP, writes to the same counter encryption page are not coalesced, thus tracking access frequency at a finer granularity than *o3*.

7.5 Impact of REI Frequency

Figure 12 shows the impact of changing REI sizes from 2 to 64 persists on execution time with epoch persistency. The figure shows that there is not much impact on execution time. However, there is a slight increase in execution time on some applications (*gamess* and *h264ref*) as REI increases. This is because the Dynamic BMF implementation we choose only alters the BMF slightly on each REI in order to limit the complexity of BMF prune and merge operations. Hence, a large REI may result in the BMF taking a long time to adapt to the access pattern exhibited by the applications.

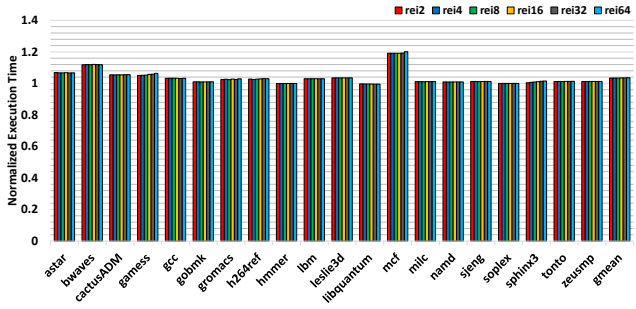

Figure 12: Execution time of *o3-dbmf* with different REI frequencies

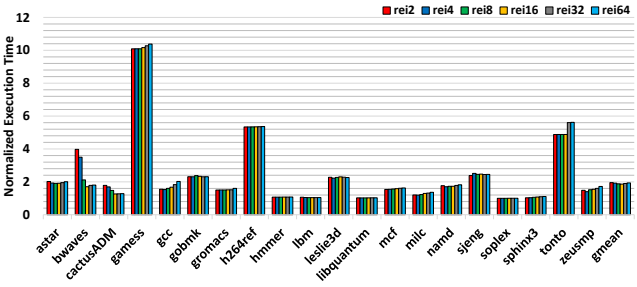

Figure 13: Execution time of *sp-dbmf* with different REI frequencies

Figure 13 shows the impact of changing REI sizes from 2 to 64 persists assuming strict persistency. Overall, there is not much impact on execution time. This is due to strict persistency not allowing persist coalescing to memory addresses and leaf access frequency is tracked at a finer granularity, allowing concurrent access to the same memory address to be the deciding factor in the BMF root evaluation. With lower REIs, some benchmarks, like *bwaves*, observed much higher performance overheads. This is because the REI was not large enough to encompass the reuse distance of memory addresses, and thus performance improves as the REI increases.

7.6 Cache Capacity Sensitivity

Impact of nvMC Capacity In this experiment, we vary the nvMC size to study its impact on performance overheads. Figures 14 and

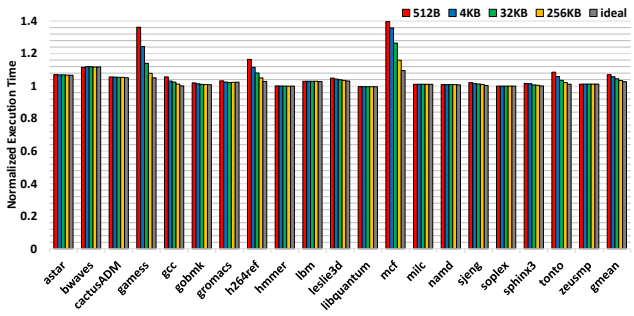

Figure 14: Execution time of *o3-sbmf* with different nvMC sizes.

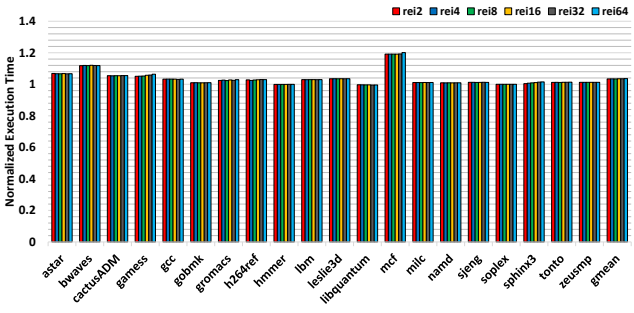

Figure 15: Execution time of *o3-dbmf* with different nvMC sizes.

15 shows execution time for Static BMF vs. Dynamic SBMF for different nvMC sizes, assuming epoch persistency. Figure 15 shows that for *dbmf*, the execution decreases going from 512 bytes to 4KB and increasing it beyond 4KB does not improve performance. In contrast, *sbmf* keeps on reducing the execution time even until 256KB. This shows that *dbmf* is much more space efficient than *sbmf*; *dbmf* with 4KB nvMC has a 0.2% overhead compared *sbmf* with 256KB nvMC. This demonstrates that Dynamic BMF provides significant performance improvement and minimal die area costs.

Figures 16 and 17 shows execution time for Static BMFs and Dynamic BMFs for different nvMC sizes assuming strict persistency. for Static BMFs, the figure shows that performance improves as the nvMC capacity increases, as lower levels of the BMT can be fully persisted, with the *ideal* capacity persisting all counter leaf parents. For Dynamic BMFs, the increase in capacity does not improve performance significantly beyond 4KB, showing the effectiveness of Dynamic BMFs.

LLC Capacity Sensitivity Figures 18 and 19 show the impact of LLC capacity on our *dbmf* scheme with *sp* and *o3* models respectively normalized to *secure_wb*. As expected, performance improved as the LLC capacity was increased, with a 19.1% overhead reduction observed with *sp-dbmf* and a 8.8% overhead reduction for *o3-dbmf*.

Metadata Cache Capacity Sensitivity We vary all three metadata cache (MDC) capacities to study the impact on our *sp-dbmf* (top) and *o3-dbmf* (bottom) schemes, with our results presented in Figure 20 and Figure 21. For *sp-dbmf*, the performance overhead was reduced by 8.1% when comparing a 128KB to 32KB MDC, while

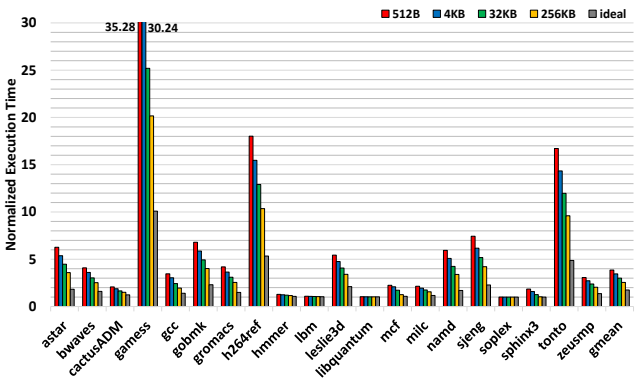

Figure 16: Execution time of Static BMF assuming strict persistency with different nvMC sizes.

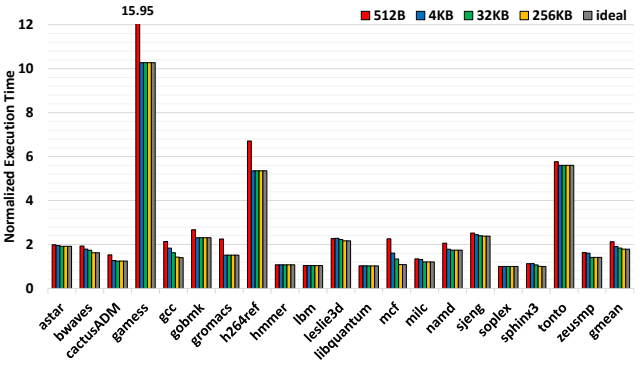

Figure 17: Execution time of Dynamic BMF assuming strict persistency with different nvMC sizes.

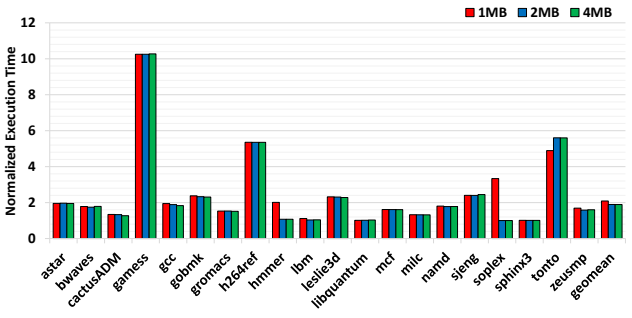

Figure 18: Normalized execution time of *sp-dbmf* with different LLC capacities

an improvement of <1% was observed for *o3-dbmf*. This shows that the MDC capacity does not have a significant impact on our *dbmf* schemes.

8 CONCLUSION

Integrity verification and memory encryption are critical components of securing persistent memory. While integrity tree root update ordering was previously exposed as being primarily responsible for performance overhead in securing NVMM, prior work still demonstrated significant performance slowdowns when reducing

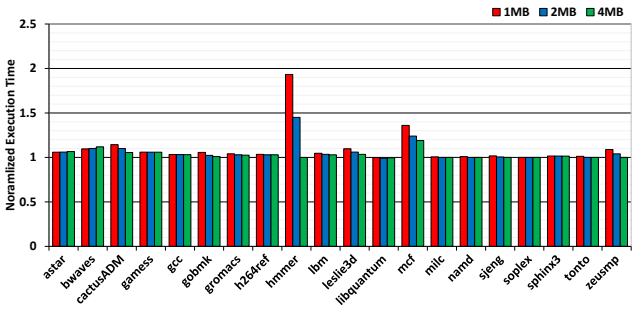

Figure 19: Normalized execution time of *o3-dbmf* with different LLC capacities

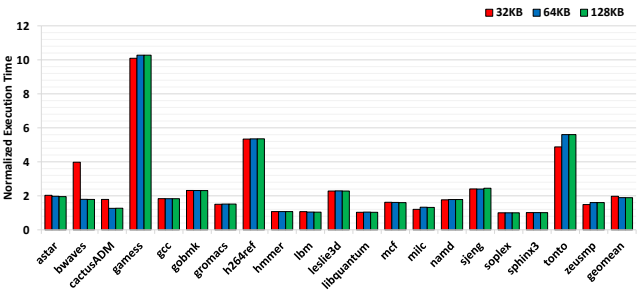

Figure 20: BMT cache miss rate of our *dbmf* scheme with *sp* with different MDC capacities

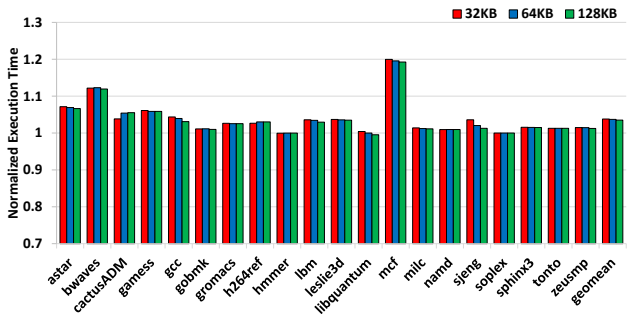

Figure 21: BMT cache miss rate of our *dbmf* scheme with *o3* with different MDC capacities

this overhead. In this work, we presented *Bonsai Merkle Forests* (BMF) and proposed utilizing on-chip non-volatile metadata caches to support a forest topology in place of conventional trees for integrity verification. With our Bonsai Merkle Forests, overheads for strict persistency models were reduced from 426% to 89%, while epoch persistency models observed a reduction from 8.5% to just 3.5% while using a 4KB non-volatile cache. These mechanisms significantly reduce root update ordering overheads, and provide a practical approach to high performance secure NVM.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their insightful feedback and comments. The NCSU team is funded in part by NSF grants 1717550 and 1908406, while the UCF author is funded in part by ONR grant N00014-20-1-2750.

REFERENCES

[1] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. 2021. BBB: Simplifying Persistent Programming using Battery-Backed Buffers. In *The 27th IEEE International Symposium on High-Performance Computer Architecture (HPCA-27)*.

[2] Mohamad Alshboul, James Tuck, and Yan Solihin. 2018. Lazy Persistency: A High-Performing and Write-Efficient Software Persistency Technique. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*.

[3] M. Alwadi, K. Zubair, D. Mohaisen, and A. Awad. 2020. Phoenix: Towards Ultra-Low Overhead, Recoverable, and Persistently Secure NVM. *IEEE Transactions on Dependable and Secure Computing* (2020).

[4] AMD. 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. (2020).

[5] Amro Awad, Laurent Njilla, and Mao Ye. 2019. Triad-NVM: Persistent-Security for Integrity-Protected and Encrypted Non-Volatile Memories (NVMs). In *Proceedings of the 46th International Symposium on Computer Architecture*.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* (2011).

[7] Guy E. Blelloch, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2018. The Parallel Persistent Memory Model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*.

[8] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*.

[9] Siddhartha Chhabra, Brian Rogers, and Yan Solihin. 2009. SHIELDSTRAP: Making Secure Processors Truly Secure. In *Proceedings of the 2009 IEEE International Conference on Computer Design*.

[10] Siddartha Chhabra and Yan Solihin. 2011. i-NVMM: A secure non-volatile main memory system with incremental encryption. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*.

[11] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*.

[12] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086.

[13] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*.

[14] Alexander Freij, Shougang Yuan, Huiyang Zhou, and Yan Solihin. 2020. Persist Level Parallelism: Streamlining Integrity Tree Updates for Secure Persistent Non-Volatile Memory. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[15] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Martin van Dijk, and Srinivas Devadas. 2003. Caches and hash trees for efficient memory integrity verification. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*.

[16] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. 2008. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX Security Symposium*.

[17] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* (2006).

[18] Intel. 2015. Intel and Micron Produce Breakthrough Memory Technology. (2015).

[19] Intel. 2021. Intel® Architecture Memory Encryption Technologies. (2021).

[20] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD Memory Encryption. (2016).

[21] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Zhenghong Wang. 2005. Architecture for protecting critical secrets in microprocessors. In *32nd International Symposium on Computer Architecture (ISCA'05)*.

[22] Tamara Silbergleit Lehman, Andrew D. Hilton, and Benjamin C. Lee. 2016. PoisonIvy: Safe Speculation for Secure Memory. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*.

[23] Chen Liu and Chengmo Yang. 2015. Secure and Durable (SEDURA): An Integrated Encryption and Wear-leveling Framework for PCM-based Main Memory. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM*.

[24] Sihang Liu, Aasheesh Kolli, Jinglei Ren, and Samira Manabi Khan. 2018. Crash Consistency in Encrypted Non-volatile Main Memory Systems. *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2018).

[25] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. 2019. Janus: Optimizing Memory and Storage Support for Non-volatile Memory Systems. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*.

[26] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.

[27] Xiang Pan, Anys Bacha, Spencer Rudolph, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2018. NVCool: When Non-Volatile Caches Meet Cold Boot Attacks. *2018 IEEE 36th International Conference on Computer Design (ICCD)* (2018).

[28] Steven Pelley, Peter Chen, and Thomas Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*.

[29] Joydeep Rakshit and Kartik Mohanram. 2017. ASSURE: Authentication Scheme for SecURE Energy Efficient Non-Volatile Memories. In *Proceedings of the 54th Annual Design Automation Conference 2017*.

[30] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2007. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[31] Brian Rogers, Milos Prvulovic, and Yan Solihin. 2006. Efficient Data Protection for Distributed Shared Memory Multiprocessors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*.

[32] Brian Rogers, Chenyu Yan, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2008. Single-Level Integrity and Confidentiality Protection for Distributed Shared Memory Multiprocessors. In *in Proceedings of the 14th International Symposium on High Performance Computer Architecture (HPCA-14*.

[33] Andy Rudoff. 2016. Deprecating the PCOMMIT Instruction. (2016).

[34] Andy Rudoof. [n. d.]. Persistent Memory Programming Without All That Cache Flushing. In *SDC 2020*.

[35] Steve Scargall. 2020. Programming Persistent Memory: A Comprehensive Guide for Developers.

[36] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*.

[37] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

[38] G. Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*

[39] G. E. Suh, C. W. O'Donnell, and S. Devadas. 2007. AEGIS: A Single-Chip Secure Processor. *IEEE Design Test of Computers* (2007).

[40] S. Swami and K. Mohanram. 2018. ACME: Advanced Counter Mode Encryption for Secure Non-Volatile Memories. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*.

[41] Shivam Swami and Kartik Mohanram. 2018. ARSENAL: Architecture for Secure Non-Volatile Memories. *Computer Architecture Letters* (2018).

[42] Jakub Szefer and Sebastian Biedermann. 2014. Towards Fast Hardware Memory Integrity Checking with Skewed Merkle Trees. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*.

[43] Meysam Taassori, Rajeev Balasubramonian, Siddhartha Chhabra, Alaa R. Alameldeen, Manjula Peddireddy, Rajat Agarwal, and Ryan Stutsman. 2020. Compact Leakage-Free Support for Integrity and Reliability. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*.

[44] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*.

[45] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[46] S. Vig, R. Juneja, and S. Lam. 2020. DISSECT: Dynamic Skew-and-Split Tree for Memory Authentication. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*.

[47] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. 2006. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*.

[48] Jun Yang, Youtao Zhang, and Lan Gao. 2003. Fast Secure Processor for Inhibiting Software Piracy and Tampering. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*.

[49] Mao Ye, Clayton Huges, and Amro Awad. 2018. Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories. In *51st Annual IEEE/ACM International Symposium on Microarchitecture*.

[50] Kazi Abu Zubair and Amro Awad. 2019. Anubis: Ultra-low Overhead and Recovery Time for Secure Non-volatile Memories. In *Proceedings of the 46th International Symposium on Computer Architecture*.

[51] Pengfei Zuo, Yu Hua, and Yuan Xie. 2019. SuperMem: Enabling Application-transparent Secure Persistent Memory with Low Overheads. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*.