A First Exploration of Fine-Grain Coherence for Integrity Metadata

Per Ekemark*, Alberto Ros†, Konstantinos Sagonas*‡, and Stefanos Kaxiras*

*Dept. of Information Technology
Uppsala University
Uppsala, Sweden

†Computer Engineering Dept.
University of Murcia
Murcia, Spain

‡School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece

*{per.ekemark,kostis,skaxiras}@it.uu.se †aros@ditec.um.es ‡kostis@cs.ntua.gr

***Abstract*—Memory integrity protection is intended for secure execution, and it is typically associated with programs running on a single core. However, with the emergence of multi-processor systems-on-chip and chiplets, extending memory integrity protection to cache-coherent multiprocessors becomes essential.**

In this work, we explore for the first time the design space for maintaining coherence in fine-grain integrity metadata at the block level. We discuss various policies for updating the integrity tree using the underlying coherence protocol, and examine how these policies affect coherence traffic. We introduce the concepts of proactive and reactive update initiation, and discuss their implications for data and integrity-tree blocks. We also investigate the trade-offs between eager and lazy update propagation policies, focusing on coherence transactions such as invalidations and downgrades to analyse the pros and cons of different approaches. What we observe is that for some benchmarks the choice between the eager and the lazy update initiation policy does not make much difference, while for many other benchmarks one policy is better over the other, depending on how the benchmark shares its data.

I. Introduction

Memory integrity protection is intended for secure processing which is typically thought of as a secure program running in a single core in an enclave. Typically, one tends *not* to associate memory integrity protection with shared-memory parallel systems (running parallel workloads) where cache coherence would be needed. However, compelling cases, where we would care to extend memory integrity protection to a multiprocessor cache-coherent system, are now emerging:

- *Multi-processor systems-on-chip* (MPSoC) applications where multiple embedded processors may need to share memory. According to Hassan et al. [1, 2], one of the most challenging burdens faced by the designers of mixed-criticality systems and MPSoCs is the problem of maintaining correctness of shared data stored in the memory hierarchies of multiple-PE platforms. Such platforms find extensive use in areas such as transportation (smart vehicles), infrastructures (smart power grids), healthcare (implantable devices), and industrial environments (robots) [1, 2].
- Obtaining and integrating into an SoC *chiplets* from untrusted sources [3]. The trend in computing systems is shifting towards 2.5D designs, which involve acquiring diverse hardware IPs, referred to as chiplets, from multiple suppliers and integrating them using an interposer. Industry has successfully showcased that these 2.5D designs reduce manufacturing expenses, allowing for continued scalability beyond the constraints of Moore's Law. Chiplet-based designs rely heavily on cache coherence for coherent data communication, as the cache hierarchy is distributed across chiplets which may straddle security boundaries. In upcoming 2.5D designs, there is a strong potential for the adoption of standard coherence, e.g., Arm AXI Coherency Extensions or Compute Express Link (CXL) [4] to enable collaborative operation through shared memory. Acquiring chiplets from various sources introduces the need for increased memory integrity protection for critical computations that span more than one chiplet.

It is exactly in cases such as these where our research study becomes relevant. More specifically, in this paper, we delve into the concept of coherence for integrity metadata, particularly focusing on the common practice of adhering to "standard" coherence protocols, like the familiar invalidation-based MESI or MOESI protocols[1]. Often, coherence is standardised, as exemplified by widely used frameworks like AXI and CXL. In practice, these standard coherence protocols become considerably more intricate and convoluted when additional states are introduced, making them challenging to implement and manage. Still, there are scenarios where custom coherence protocols can be devised, offering opportunities to explore alternative approaches, e.g., using update-based protocols instead of invalidation-based ones, but this is left for future work.

In contrast to prior research, which partitioned datasets into non-shared segments making the problem of applying coherence on an integrity scheme a relatively straightforward task [5], the unique aspect of our work lies in its emphasis on fine-grain sharing at cacheline granularity. In this context, and without changing the coherence protocol, the main question is how the updates to integrity metadata are initiated and scheduled in relation to the coherence events that affect the data.

Memory integrity relies on constructing message authentication codes (MACs) (i.e., keyed hashes) for data blocks. To verify integrity, a MAC is computed for a data block and checked against a stored MAC in memory. However, ensuring MAC integrity becomes challenging as adversaries can tamper with both data blocks and their corresponding stored MACs. To address this issue, *integrity trees* are introduced that incorporate MACs in each node: Merkle Trees (MTs), Bonsai Merkle Trees (BMTs), with BMTs using version counters to reduce MT size, or Intel SGX-like integrity trees, which combine MACs and counters in the same blocks.

The challenge is to keep all blocks, both the data blocks and the integrity tree blocks, under a single coherence domain. The problem is that this is expensive: to verify a data block, a complete path to the root of the integrity tree must be available; to expose a modified data block to other sharers, a complete path to the root of the tree must be updated. These two operations, even for independent data blocks, interact via *false-sharing* in the integrity tree nodes: reading and writing completely independent data blocks by different cores, leads to coherence downgrades and invalidations higher up in the integrity

[1]The MESI and MOESI protocols are named after the included, stable coherence states: *Modified*, (*Owned*), *Exclusive*, *Shared*, and *Invalid*.

tree. This in turn leads to substantial interference and ping-pong effects for the nodes of the higher levels of the integrity tree.

The choice of when to propagate updates in the integrity tree, therefore, becomes critical. On the one hand, eagerly updating the integrity tree on a data block update, invalidates parts of the integrity tree in other caches which are not actively sharing the data block in question. On the other hand, lazily updating the integrity tree only when it is absolutely necessary transfers the penalty (appearing as a serialised invalidation latency) to the reading side, where latency is critical for performance.

In this work, we examine this trade-off for the different integrity tree update strategies from the point of view of the amount of coherence transactions generated, as a first-order approximation of the incurred cost of coherence. We introduce proactive and reactive integrity tree update initiation policies and analyse their impact on data and integrity-tree blocks. We also explore the trade-offs between eager and lazy update propagation policies, with a focus on coherence transactions like invalidations and downgrades. Our findings suggest that the choice between these policies has varying significance, depending on the benchmark's data-sharing characteristics.

II. Background

In this section, we give a quick recap of the most important elements for memory integrity to set the stage for the coherence discussions that follow.

At its most basic, memory integrity for a data block `b` is guaranteed by constructing a keyed hash of the value of `b`. These keyed hashes are usually called message authentication codes (MACs), a terminology that comes from secure networks.

The integrity check for `b` consists of: reading `b` and its MAC from memory, computing anew the keyed hash of `b` and checking to see if it matches the MAC read from memory.

Only the holder of the secret key can generate the correct MAC for a particular value of `b`. It is therefore important that the key is only stored within a secure domain (e.g., in a register within a processor core). Key transfers must ensure confidentiality and can thus only be done within the secure domain (e.g., within a core or across a trusted bus) or while encrypted (i.e., when using an untrusted bus).

A MAC is smaller than the data block it protects. Typically a 64-byte block can be reasonably well protected by a 16-byte MAC. To save memory and manage MACs more easily, multiple MACs are packed in the space of a data block. In our example, four MACs, protecting four 64-byte data blocks, are packed in a block of 64 bytes.

It would seem that complementing every memory block with a corresponding MAC would guarantee the integrity of all memory, but the problem now is who guarantees the integrity of the MACs? If adversaries are able to write both a data block and its MACs, they can simply corrupt memory by replacing the latest data with a stale version of the block and its corresponding (valid) MAC. To protect against such *replay* attacks, the concept of *integrity trees* is applied.

A. Merkle Trees

Assuming that the MACs that protect data blocks form a "first level" of MAC blocks (multiple MACs per block), a second level of MACs is needed to protect the first level of blocks, and a third level to protect the second, and so on, all the way to a "root" MAC. The root cannot be protected yet with another MAC that is public (otherwise we would go on ad infinitum), so the root MAC needs to be kept secret within the secure domain (and be handled similarly to the key for the MACs). This tree of MACs is known as a Merkle Tree (MT) [6] and is shown in the top half of Fig. 1.

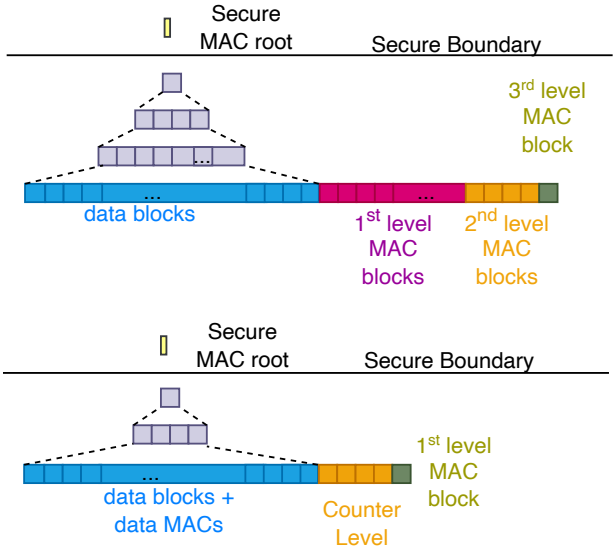

Fig. 1. Top: Merkle Tree (MT); Bottom: Bonsai MT (BMT); Logical layout shown in grey, memory layout in colour.

A Merkle Tree is characterised by its *arity*, which corresponds to how many MACs can fit in the space of a data block. For example, in a cache hierarchy with 64-byte blocks (cachelines) and a 16-byte MAC per block, one can fit four MACs in a block. Thus, the *arity* of the MT in this case is four.

B. Bonsai Merkle Trees

Bonsai Merkle Trees (BMT) [7, 8] for memory integrity, aim to reduce the size of the Merkle Tree while preserving its properties. In a BMT, the first level of MACs that protect the data blocks are not directly protected by a tree. Instead, each data block is associated with a *version* counter that changes every time the data is updated. Corresponding data and counter are concatenated before computing their combined MAC. To prevent replay attacks with stale data, it is sufficient to protect the counters with a Merkle Tree. The version counters can be much smaller than MACs; 16 or 64 counters can fit in a 64-byte block. Thus, the arity at the counter level can be higher (e.g., 64) than for the rest of the tree levels (e.g., 4), allowing the height of the tree to be reduced. An example of a BMT is shown in the bottom half of Fig. 1.

C. Intel SGX Integrity Tree

The Intel SGX integrity tree [9, pp. 166–201] combines the MAC and the counters it protects in the same block. This increases arity throughout the whole tree at the expense of having slightly smaller counters, since some of the space of the block is taken by the corresponding MAC. This type of integrity trees have a constant arity throughout the tree.

D. This paper

In this paper, we discuss an abstract MT with its arity as a parameter. At first approximation, the coherence transactions for an MT and a BMT (or other tree with counters) are identical. While differences can arise depending on when the counters and the MACs are updated, here we adopt a uniform view and assume that the counters and the corresponding MACs are updated at the same time.

TABLE I
TRUST MODELS

Name	Trust	Comment
High	shared cache	coherence inside secure boundary
Medium	all private caches	(private) cache-to-cache is secure
Low	own private cache	all coherence insecure

III. THREAT MODELS AND COHERENCE

A. Threat Models

We consider three threat models (or trust models) that can apply, which affect coherence in different ways. They are shown in Table I and described below.

High-Trust: If cores trust each other and the shared memory (read Last Level Cache (LLC)), the coherence is not affected as it operates entirely within the trusted domain.

Low-Trust: At the other extreme, where cores do not trust each other, nor any other part of the system, all data entering a private cache must be verified (ultimately against the root MAC). Similarly, any modifications must also update the integrity tree before release. Updates in the integrity trees can be done lazily, only one level closer to the root at a time. This is useful when voluntarily evicting data, but on invalidation another core is likely going to force writes all the way to the root MAC, as that core wants to verify the data it reads by invalidating.

Medium-Trust: We can also consider an intermediate trust level. If cores trust each other but not shared memory (i.e., coherence data transfers are trusted), the integrity tree only has to be read and updated when data is read from and written back to, respectively, the shared memory, not on core-to-core transfers. This model opens the possibility of sharing already verified (and potentially dirty) data between cores without a lookup in the integrity tree. The same policy applies for integrity tree nodes as well.

In this paper, we chose to examine the low-trust model because it represents the greatest challenge in terms of coherence, and fits the motivating cases where coherence for integrity would be needed (cf. §I). As we show, in a low-trust scenario, a transfer of dirty data between two cores requires the sender to make writes throughout the full height of the integrity tree and the receiver to read the same tree path in its entirety.

B. Coherence Threat Model

The critical function of coherence when it comes to memory integrity protection can be understood by contrasting the non-coherent and the coherent scenarios:

- In a **non-coherent** setup, *exposing* (evicting and writing to memory) requires updating all integrity tree nodes and writing out all updated nodes to memory. This is essential to enable a valid integrity check. It is important to note that the order in which nodes are exposed does not matter; an adversary can disrupt correct integrity checks regardless of the order in which integrity tree nodes and data are written to memory.
- In a **coherent** system, *exposing* (evicting, or causing a coherent transfer on a read or write) only necessitates changing the state of the integrity tree nodes in a manner that allows coherence to handle the correct exposure of the required integrity tree nodes during an integrity check. In other words, if we trust coherence transactions and the directory, *there is no need to write out the integrity tree nodes to a shared level of the coherent hierarchy*. Our primary concern is ensuring that the only valid copy of any node is deterministically located, which is precisely the function of coherence.

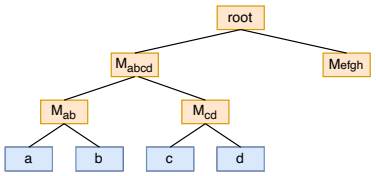

Fig. 2. Abstract integrity tree example. Here. the arity is two, M_{ab} holds the MACs of cachelines a and b, and M_{abcd} holds the MACs of M_{ab} and M_{cd}.

Thus, we assume that coherent transactions and the information stored in the directory are safe from tampering by an adversary. In other words, while we may not trust any data values delivered by a different core/cache, shared cache, or memory, we trust the validity and integrity of the coherence transaction (e.g., Read, Write, Invalidate, Downgrade, Ack, etc.) both in terms of the address it refers to and the coherence command. Safety and security for coherence transactions is a valid assumption as it has been demonstrated by Olson et al. [10, 11, 12]; their solutions can equally well apply in our case.

Thus, given a low-trust model and a trusted coherence layer, the overall threat model can be summarised as:

Threat Model

Coherence transactions are trusted, but the values of the transferred cachelines are not.

Safeguarding against adversaries in our threat model, necessitates an integrity mechanism that ensures the integrity of the data and the integrity of its metadata (i.e., the nodes of the integrity tree), *all of which are subject to the same coherence protocol*. As far as the coherence protocol is concerned, it is agnostic to the type of data in the cachelines.

In this paper, the only differentiation we permit between ordinary data cachelines and integrity metadata cachelines strictly concerns the freedom to schedule integrity tree updates in the local private caches so as to implement policies that affect the resulting coherence traffic.

IV. INTEGRITY PROPAGATION POLICIES

After updating data protected by an integrity tree, it is necessary to (eventually) document the new data in the integrity tree to make the data verifiable in the future. There are numerous possible policies imaginable for when to update the integrity tree and at what pace.

We illustrate our discussion regarding the various policies concerning the update of the integrity tree, and subsequently how these policies affect coherence, with an *abstract integrity tree*. An abstract integrity tree consists of data nodes (in leaves), and of integrity nodes which can be MACs, counters, or a combination thereof. An example is depicted in Fig. 2.

We explain the various policy choices, illustrated in Fig. 3, on a case-by-case basis. In Fig. 3Ⓐ, we show two cores caching two different data cachelines (a and b) but sharing the same integrity tree path to the root. The cached data blocks and integrity tree nodes are shown in bold and colour, while uncached nodes (in each core respectively) are shown as faded grey nodes to illustrate the whole integrity tree.

Proactive or Reactive? To reason about policies that control when an update takes place, we start from the question: "If we update the data cacheline a, do we also cause an *immediate* update to the integrity tree?" Or, do we wait until the data cacheline a needs to be

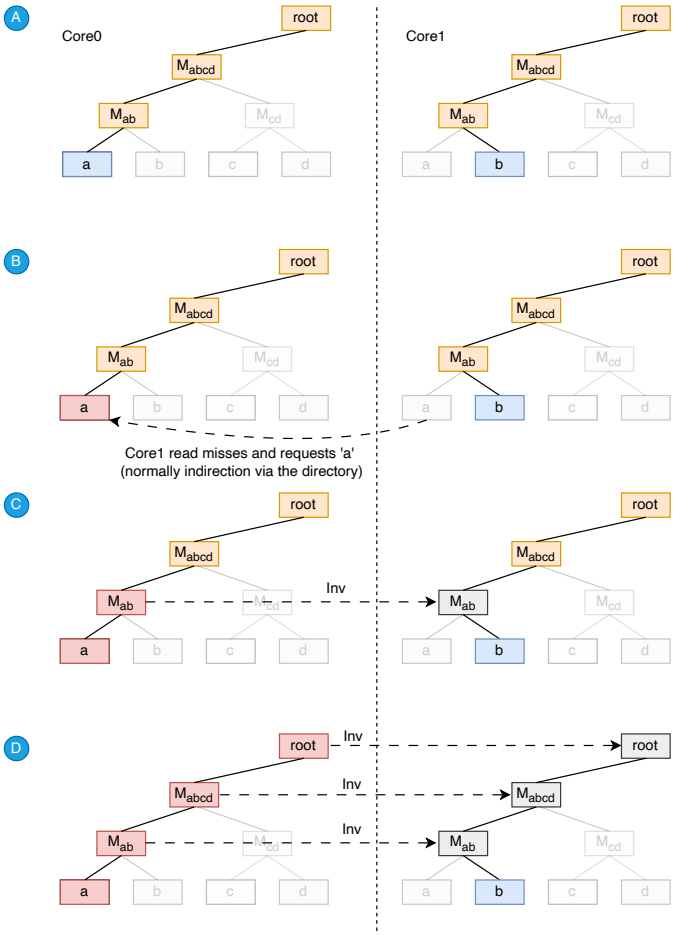

Fig. 3. Two and two choices: Reactive/Proactive — Eager/Lazy.

exposed either by a request from some other node or because of an eviction? (Note that we will treat these two cases separately later on.) For example, in Fig. 3Ⓑ, the data cacheline `a` is written to by Core0 but the integrity tree is not updated until Core1 requests this cacheline (shown as a dashed arrow). This is a **reactive** update initiation policy. In contrast, when cacheline `a` is written in Fig. 3Ⓒ, an immediate update of the integrity tree is proactively initiated upwards. This is a **proactive** update initiation policy.

The question can be generalised for any node in the integrity tree: When a node is updated, is the update **proactively initiated** towards the root, or deferred until the node is exposed, that is, **reactively initiated**?

The proactive and reactive initiation policies can be mixed *independently* per level, or even node, of the integrity tree. However, a more reasonable (and useful) division would be to apply different policies to data blocks and tree node blocks. Since having spatial (and temporal) locality for data is a common pattern, it makes sense to wait for as long as possible to coalesce multiple data modifications into a single integrity update. This implies a reactive policy for data cachelines, but it is unclear what policy should be followed for the integrity tree.

Once an integrity update is needed, either an **eager** or **lazy** policy can be applied to *propagate* the update in the tree. While we refer to the update initiation policies as **proactive** and **reactive**, to differentiate the policies of propagating an integrity update within a tree, we use the respective terms **eager** and **lazy**:

- An **eager** update policy for the integrity tree implies that a proactive initiation policy is used in *all* tree levels.
- A **lazy** update policy for the integrity tree implies that a reactive initiation policy is used in *all* tree levels.

Between the two extremes (eager and lazy), other policies can be constructed that switch between proactive and reactive propagation depending on the level of the tree. While such policies may have interesting properties, we defer their examination for future work.

Figure 3Ⓒ and Ⓓ demonstrate the lazy and eager update policies for the integrity tree (in other words, the updates either just go to the next level above or all the way to the root). In Fig. 3Ⓒ, the **lazy** policy, the update is stopped at the first level node (M_{ab}) and does not propagate upwards unless this node is exposed to the outside world. In Fig. 3Ⓓ, the **eager** policy, the update to the integrity tree propagates from the node that was affected all the way to the root uninterrupted.

A. Eager vs. Lazy Trade-off

Suppose that a lazy integrity propagation policy were in place, as in Fig. 4. When Core1 (right) requests dirty data `a` from Core0 (left) ①, Core0 must first ensure that an initial update has been made into the integrity tree, specifically tree node M_{ab} ②, before releasing `a` ⑤. The update of M_{ab} requires Core0 to obtain write permission for it by invalidating other cores or requesting an exclusive and writable copy of the integrity node ③④. Once Core1 receives `a` ⑥, it must verify the data since it comes from outside Core1's zone of trust ⑦. As the MAC in M_{ab} was just updated by Core0, it cannot already be cached by Core1. As Core1 requests M_{ab} ⑧, a new cycle, similar to the request of the data, begins: Core0 must get write permission for the node one level up in the tree, M_{abcd}, and update it before Core0 can send out M_{ab} ⑨–⑬. This process with two round trips per cycle repeats until the root of the integrity tree is reached.

Observation I

A lazy propagation policy *doubles* the request/response round trip latency of a reader from the bottom level to the top level of the tree. This is because, for each level of the tree, the request/response latency (of reading a modified node), is serialised by the interjection of the invalidate/ack latency of the node in the next level up. On the other hand, a lazy policy does not create interference in nodes higher up in the tree, potentially affecting other cores who are trying to validate using the same nodes at the same time.

As an alternative to this observation, consider instead an eager propagation policy, illustrated in Fig. 5. In the bottom diagram, Core1 requests, as before, the dirty data `a`. At this point, Core0 starts an internal process (top diagram) and requests write permission for M_{ab} ①–③. However, armed with the knowledge that in a low-trust scenario, a transfer of dirty data between two cores requires the sender to make writes throughout the full height of the integrity tree and the receiver to read the entirety of the same tree branch, an *eager* policy of updating the integrity tree on the sender's side can overlap the latency of the transfer.

Writes to the integrity branch must be made in order from leaf to root (①, ⑤, ⑨) as write permissions are acquired for each node (②–③, ⑥–⑦, ⑩–⑪). The leaf-to-root order, guarantees absence of deadlocks even in the case when write permissions would be held for

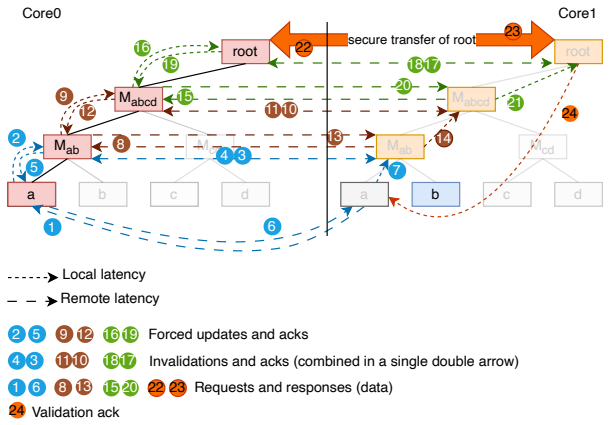

Fig. 4. Lazy update of the integrity tree. Updates in the integrity tree on the left (e.g., cached in Core0) are propagated upwards in the local tree only when the corresponding nodes are requested from another core (e.g., from Core1 as part of integrity check on the same path).

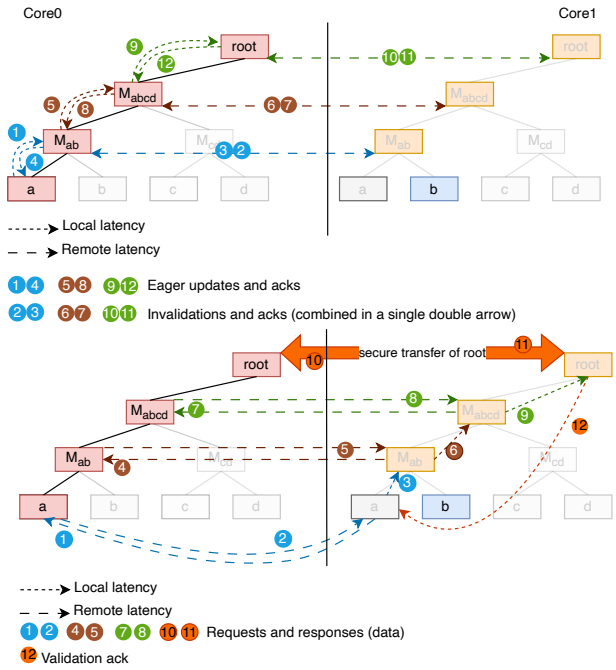

Fig. 5. Eager update of the integrity tree. Assuming sufficient separation, the update of the tree happens (top diagram) independently of the read (bottom diagram). In contrast to the lazy update, eager update does not weave the latency of the invalidation phase into the latency of the read phase.

the lower-level nodes until the top of the tree is reached.[2] However, holding on to permissions until the root of the tree is reached is not necessary, as the single-writer, multiple-reader (SWMR) invariant guarantees correct update of the tree regardless of the number of cores trying to update the same path simultaneously. Thus, similar to the lazy update, a cacheline can be exposed as soon as there is confirmation that the write of the next level has been *performed*. In coherence, a write is performed when the invalidation has been acknowledged (all invalidation acks received).

As Core0 completes updates to integrity nodes, the node in the level below is released. As a is sent to Core1 (Fig. 5, bottom, ②), Core1 can climb the integrity tree and request necessary integrity nodes from Core0. Since we know that Core1 will eventually request all integrity nodes on the relevant path, requests for these nodes (bottom diagram ④, ⑦, ⑩) can be considered implied and the nodes be sent from Core0 to Core1 when available.

In the lazy policy, there have to be four requests/responses (two round trips) between cores *for every level* of the integrity tree, strictly serialised. With the eager policy, one of the four requests *per level* can be implied and the remaining requests/responses only have to be serialised *within* each level but different levels can overlap.

Observation II

An eager propagation policy *decouples* the invalidation/ack latency from the request/response latency as experienced by a reader. Given sufficient separation between the invalidation phase (by the writer) and the read phase (by the reader), the read latency is minimised.

B. Secure Root Transfer

In Figs. 4 and 5, the transfer of the root is marked differently from the transfers of the other tree nodes. This is because there are no further nodes in the tree to check the integrity of the root against. Since the root is always kept securely in the processor and never written to memory, root transfers always take place between processors. We do not propose any new solutions to facilitate secure transfers between processors over the insecure interconnect; instead we note that there already are solutions to this. For example, Rogers et al. [13] achieve secure processor-to-processor transfer by attaching a MAC to the message generated using a nonce to protect against replay attacks. They also use the address to protect against splicing attacks, which we can omit since we only need to transfer the root.

C. Evictions

In a typical coherence protocol, a modified (dirty) cacheline, kept locally in the private cache in state M (Modified), which signifies that it is the only copy of the data block in the system (SWMR invariant), can only be exposed to the outside world (other cores or memory) in one of the following ways:

1) Invalidation: Another core wants to write the cacheline.
2) Downgrade: Another core wants to read the cacheline.
3) Eviction: The cacheline is evicted to memory and it is unknown who (if any) is going to read or write it next.

There is a difference between eviction and remote-demand-request (Invalidation–write or Downgrade–read) because in the latter case, it is expected that the rest of the path to the root will soon be needed, while in the former there is no such expectation since it is unknown who is reading the data next.

While the eager policy clearly has benefits when a core receives a demand request for dirty data, this is not the case when a core itself generates an eviction. It is entirely possible that the evicting core itself will read the same cacheline back on its next use. Assuming the core can keep (at least part of) the integrity branch for the data node, it would be unnecessary to update the entire branch at eviction time. For this reason, it would make sense to use a lazy integrity propagation policy for evictions. If another core reads the evicted cacheline, it will request the corresponding integrity node, at which point the eager policy can be used to transfer the rest of the integrity branch.

[2]The reason is that for any two tree paths that share a number of nodes to the top, there is a unique strict order (a "lexicographical" order of, e.g., ascending physical addresses of the nodes to the root), in which write permission "locks" can be taken.

TABLE II
RELATION OF EVENTS FOR INTEGRITY TREE COHERENCE.

Event	Relation to Invalidations/Downgrades
Read	The number of reads to the nodes of the integrity tree (per level). Reads can hit; or in a miss, in which case the relevant cacheline must be fetched.
Read Miss	Read misses correspond to data transfers. A read miss may result in a downgrade of a dirty node in a different cache (the previous writer) but write misses are a better proxy for the downgrades.
Write	The number of writes to the nodes of the integrity tree (per level). Writes can result in a hit (write permissions already present) or a write miss. The write misses are permission misses, a portion of which can also be missing cachelines.
Write Miss	Write misses are a good proxy for both invalidations (every write miss sends an invalidation) and downgrades (since, eventually, the soon-to-be writer is downgraded by a future read). Invalidations can result in the invalidation of one or more cached copies. In general, the eager policy has an average degree of sharing upon invalidation between 1 and 2, while the lazy policy more than 2.

V. EVALUATION

A. Methodology

The design space for how to keep integrity metadata coherent is large. In this work, we limit our exploration to fine-grain coherence (at the block level) without changing the underlying coherence protocol (i.e., its states, transactions, and directory information). Besides the practical aspect of performing a first exploration in such a large design space, focusing on keeping the coherence protocol unmodified is justified by the case of standardised protocols as we discussed in the introduction.

Thus, in our evaluation, we focus on simple metrics that can determine the advantage or disadvantage of one approach over another. These metrics focus on coherence transactions such as invalidation and downgrade messages that incur latency and are the cause of coherence misses (i.e., read misses on invalidated cachelines, and write-misses on cachelines that do not have write permissions).

We use the Ruby memory model [14] to generate a detailed trace of L2 cache events in a 3-level invalidation-based cache coherence protocol modelled in the SLICC language. The L1D (32 KB, 8-way associative) and the L2 (256 KB, 8-way associative) are private to the core. The LLC (8 MB, 16-way associative) is shared across the cores. We model a system with 16 out-of-order cores. We run our simulator without memory integrity protection for a wide range of benchmarks from the PARSEC [15] and SPLASH-3 [16] benchmark suites for 16 threads and with the simsmall input.

The traces include the following L2 events: reads (i.e., cache fills), dirty evictions, invalidations, and downgrades. Dirty cachelines that are evicted, invalidated or downgraded from L2 (last level private cache) require an integrity tree update. Cachelines that are filled into L2 (for either reading or writing) require a verification against the integrity tree.

The collected traces are then processed using a tool we developed that models an abstract integrity tree. Because the volume of results is considerable, for clarity, we chose to present only reads/writes and read-misses/write-misses, from which the rest of the events (invalidations, downgrades) can be inferred. Table II shows the relationship among the various coherence events.

Since the memory access traces concern a fixed cache hierarchy, we cannot retroactively model the cache interference that would be generated by keeping integrity metadata cachelines in the same cache hierarchy with the data cachelines. Thus, a simplifying assumption of our model is that the integrity metadata do not suffer from evictions and always fit in the cache (without evicting data cachelines). This allows us to model the pure coherence communication (e.g., invalidations and downgrades) that is incurred by the integrity protection mechanism assuming ideal caching for its metadata.

Our analysis tool simulates a Merkle tree and keeps track of what cores cache each node of the tree and in what state. The tool reads the aforementioned cache trace, where each trace entry indicate that some cache has read a cacheline and must verify it or has exposed dirty data (eviction, invalidation, or downgrade) and must update the integrity tree. The tool can operate with either the *eager* or *lazy* propagation policy. In the eager mode, an update in the integrity tree implies that a core fetches and writes all tree levels immediately (before processing the next trace entry). In the lazy mode, an update only implies fetching (and validating, if necessary) and writing the leaf level of the tree, writes higher in the tree are delayed until they are forced by another trace entry, causing a downgrade of the dirty tree node. Regardless of mode, a validation operation implies reading the appropriate leaf node and, if necessary, continue validation higher in the tree until hitting a node that is already cached. While processing trace entries, our tool keeps statistics of coherence events for every node in the tree and ultimately presents a summation of the event per level of the integrity tree.

We explore different arities with our tool: 4, 16, and 64. An arity of 4 is practical as it allows storing four 128-bit MAC in a single 64-byte cacheline. Higher arities would require increasing the node size, weakening the MAC, or using (small) counters. Our tool does not simulate the size of the tree nodes nor the integrity mechanism (i.e., calculating and comparing MACs), thus the arity is only relevant to see the coherence effect of different tree topologies.

B. Microbenchmarks

To characterise specific scenarios, we have constructed artificial traces of integrity events that we feed into our integrity simulator.

Continuous Independent Writes: Several cores (16) each write an amount of memory (1 MB) that exceeds the cores' private cache capacity (256 KB). Each core writes its own *continuous* part of the memory. The cores begin to read from memory, causing verification operations, reading branches in the integrity tree to verify. As the cache fills up, cores evict old data, causing integrity updates mixed with verification for new data. When the writing is done, the caches flush (i.e., evict) dirty data, continuing the integrity updates without the interspersed verifications.

Interleaved Independent Writes: Several cores (16) each write an amount of memory (1 MB) that exceeds the cores' private cache capacity (256 KB). Each core writes its own cachelines, but the cores' cachelines are interleaved in memory. The operation is the same as in the previous scenario, only with different addresses.

Blocked Interleaved Independent Writes: It is similar to the previous microbenchmark, but instead of interleaving *every* cacheline, cachelines are interleaved in blocks of four. This eliminates contention for leaf integrity node (when arity is 4).

Lock Variable Contention: Several cores (16) contend over a single Test and Test-and-Set (TATAS) lock. Each core reads the lock (each causing an integrity verification). As they see an available lock, they all write (TAS) to attempt to acquire the lock (causing an integrity update and verification each as they invalidate each other). All but two cores (the one that acquired the lock and the last that failed) have to read (and verify) the lock again. (The last writer is downgraded.) While the core with the lock keeps it, there are no more coherence transactions (due to the lock). When

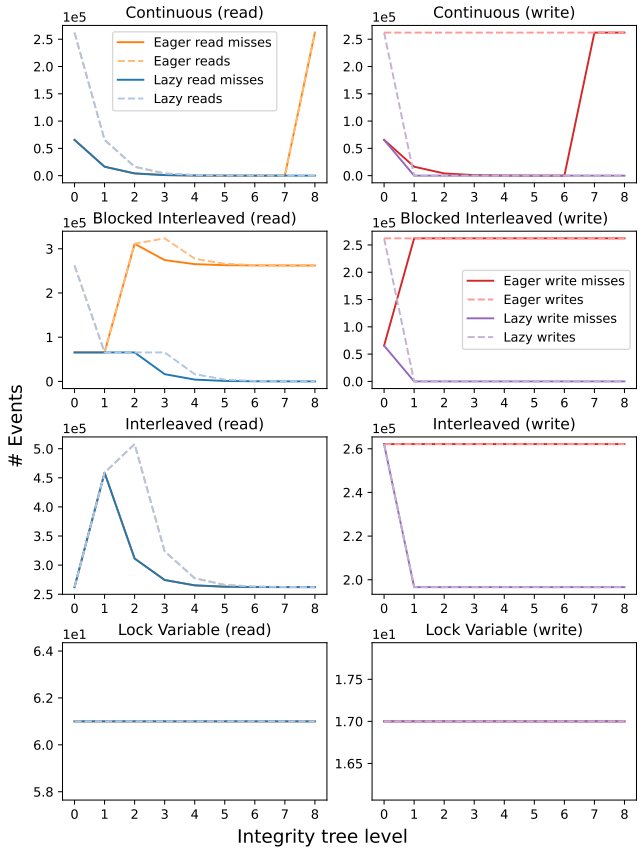

Fig. 6. Results of microbenchmarks.

the lock is released, the lock is read (and verified) and written. After being invalidated, the other cores request (and verify) the lock after it has been downgraded (and integrity updated) by the former lock holder.

Figure 6 shows the analysis result of our microbenchmark traces, integrity tree read characteristics to the left, write characteristics to the right.

To understand the *Continuous* write benchmark, it is important to realise that each core has a portion of the integrity tree private (i.e., no other core is neither reading nor writing that part). We use 64-byte cachelines and arity 4 in the tree, and since each core is using 1 MB of data, all tree nodes at level 6 and below are private to each core. Each core starts reading their private memory, causing verifications (i.e., reads) in the integrity tree. In the private subtree area, both the eager and lazy policy follow the same curve, with read misses being exactly $1/4$ of the reads (an effect of using arity 4 in the tree). At the root end of the plot, we can see that the reads in the lazy policy stay very low, while the reads for the eager policy, with 100% miss ratio, go high. To understand this sharp incline we must consider the updates (i.e., writes in the integrity tree). The lazy policy only has to write in the leaf nodes as we do not model evictions and no other core are requesting theses nodes as per the design of the benchmark. The eager policy, on the other hand, is forced to write in all levels every time a data cacheline is evicted. Within the private subtrees, the eager write misses can still stay low, but at level 7 and up, cores start contending over the same integrity nodes and invalidating each other. This also explains the sharp rise in eager reads at level 8: every time a cache fetches a node at level 7 it must verify it by reading level 8. This will always be a miss since level 8 is also experiencing severe write contention.

The *Blocked Interleaved* microbenchmark is similar to the *Continuous* in that all cores have private integrity subtrees, but instead of having one spanning many levels, the cores now have many subtrees only involving the leaf and data nodes. The plots demonstrate the similarity to the *Continuous* microbenchmark: the same rise in eager write misses we saw at level 7, here shows up at level 1, the first non-private tree level. This in turn, causes a similar eager read spike at level 2 (instead of level 8).

The (fully) *Interleaved* microbenchmark does not allow for any private subtree for cores. This makes the eager and lazy policy have the almost same characteristics: As soon as a core has made a write to a leaf integrity node under the lazy policy, the next core will request the same node, causing a cascading update to the root, as we showed in Fig. 4. The exception to this is that only 4 of the 16 cores compete over each leaf node (due to using arity 4). Since the microbenchmark is only doing one pass over the memory, the lazy policy can avoid an update propagation $1/4$ of the time, resulting in fewer writes for the lazy policy by $1/4$. The read characteristics are, however, the same for both eager and lazy.

In the final microbenchmark, the cores are contending over a single lock variable. The important thing to notice is that all cores are trying to read and write the exact same memory location. Every time a core writes, it invalidates the integrity nodes on all levels for every core, causing a read at every level when another core tries to read the value again. As a result, the eager and lazy policy become effectively the same, and the plot becomes flat.

To summarise, if cores work on separate memory areas, they can avoid interfering with each other in the integrity tree. The bigger the private memory areas are, the bigger the interference-free integrity subtrees can be. In eager mode, all integrity updates will propagate throughout the height of the tree, hitting in cache within the private subtree, but interfering with other cores higher up in the tree. In lazy mode, it is possible to remain interference free in the entire tree, as long as the private subtree can be held in cache.

C. Benchmarks

Figures 7 and 8 show the results of our analysis of the PARSEC and SPLASH-3 benchmark suites, for reads and writes, respectively. Note that we put emphasis on misses in solid lines, as these represent events that would cause coherence traffic, and an account for total number of events (including both hits and misses) in weaker, dashed lines.

In both Figs. 7 and 8, the benchmarks are sorted according to which policy is "better" in a column-major order (top-down, left-to-right) with "eager better" to the (top) left (e.g., *Canneal*) and "lazy better" to the (bottom) right (e.g., *Swaptions*), with a spectrum in-between. To be precise, the sorting criteria is the relative difference between the read (misses) of the two policies at the root level (level 12) of the integrity tree. Additionally, we can also highlight an orthogonal "read heavy" group, marked with "*".

We would particularly like to draw attention to the following insights:

1) The leaf write miss ratio influences policy performance.
2) Page size and placement influence integrity tree sharing.
3) Large read-only data benefit from good integrity tree caching.

We discuss each of these insights in more depth below:

1) Policy performance differentiation: While Figs. 7 and 8 only show integrity tree statistics, the fetches and exposures of data (which cause integrity tree activity) can be seen as an echo in the reads and writes, respectively, to the leaf level (level 0) of the integrity tree.

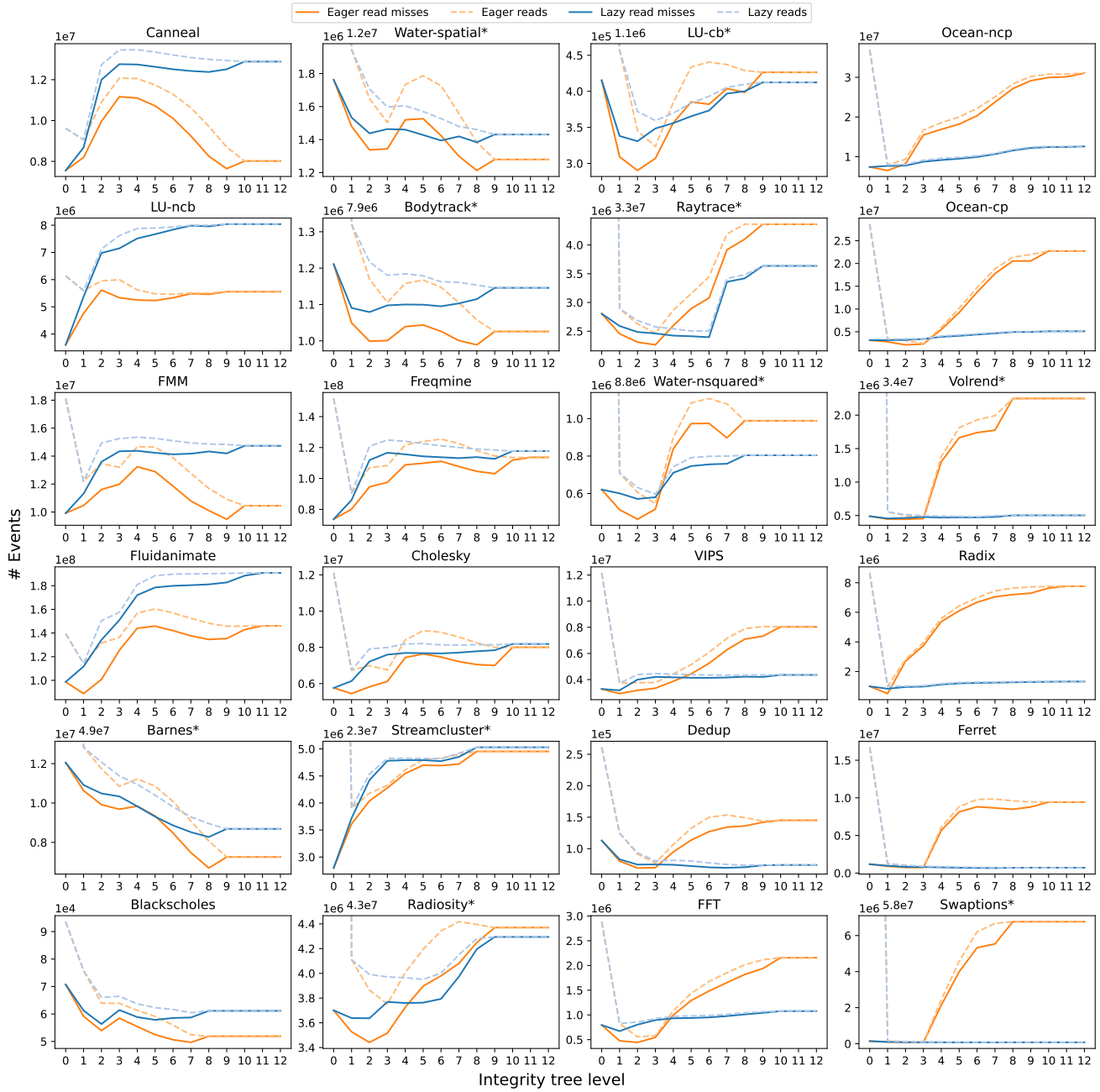

Fig. 7. Results from selected PARSEC and SPLASH-3 benchmarks: reads and read-misses for both eager and lazy propagation. Ordered according to characteristics in column-major order (top-down, left-to-right). Eager better to the left, lazy better to the right. Benchmarks marked with "*" are read heavy in the leaf node and the outlier on level 0 has been left outside the plot with a note of its value (equal for both eager and lazy).

This is the reason there is no difference between the propagation policies at the leaf. Similarly, it is the misses that cause fetches at a particular level, which propagate as reads to the next level of the integrity tree. Write propagation depends on the policy: either every write are *eagerly* propagated as a write in the next level or only exposures (caused by misses on dirty integrity tree nodes) are *lazily* propagated as writes in the next level.

Since our model does not limit cache size for the integrity tree, all write misses at the leaf level are compulsory or coherence misses (the later due to communication between cores or false sharing). This is beneficial for the lazy propagation policy as writes only propagate on exposures, which are caused by coherence misses. It is even more useful when the write miss ratio at the leaf level is low as it limits the traffic (both reads and writes) that propagates to higher tree level. As a result, we find the benchmarks with a low leaf write miss ratio to the right (i.e., the "lazy better" side) in Fig. 8.

Conversely, when the leaf write miss ratio is high, the lazy policy approaches the write volume of the eager policy. However, it is already at about the 50% leaf write miss-ratio mark that read characteristics become similar between the policies (middle of Fig. 7). At even higher ratios, read characteristics favour the eager policy. A possible explanation is that the eager policy keeps the integrity tree fresh at all

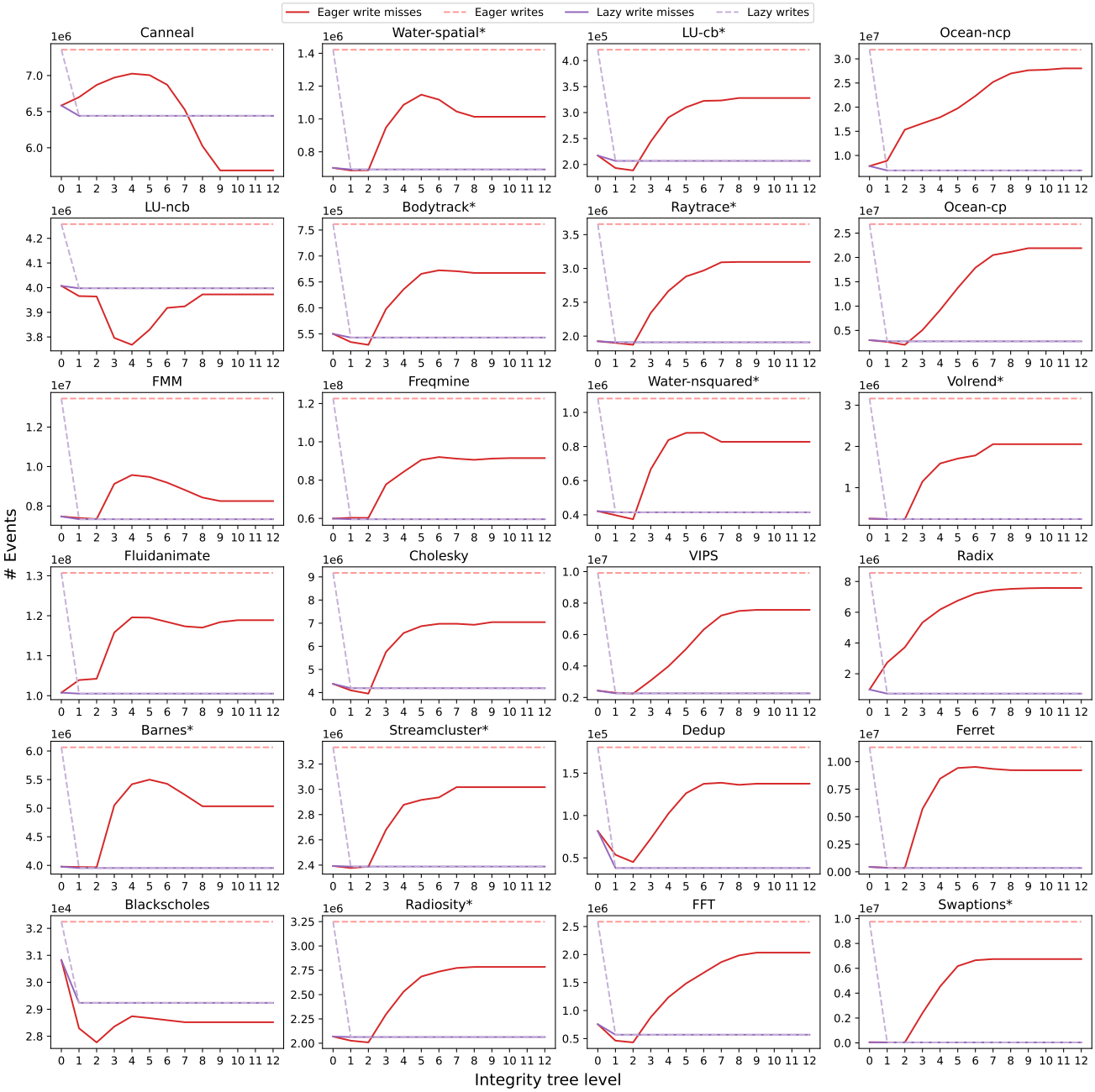

Fig. 8. Results from selected PARSEC and SPLASH-3 benchmarks: writes and write-misses for both eager and lazy propagation. Ordered according to characteristics in column-major order (top-down, left-to-right) in the same way as in Fig. 7. Benchmarks marked with "*" to match Fig. 7 but irrelevant here.

times, which would allow cores to sometimes fetch and keep a fresh integrity tree node rather than loose the node due to an invalidation by a lazily delayed update.

We can see the same relation between policy performance and leaf write miss ratio already in the microbenchmarks (Fig. 6). In the *Blocked Interleaved* and *Continuous* benchmarks, the miss ratio is low and as a result, there are fewer lazy read (misses) than eager (at the root). In the *Interleaved* and *Lock Variable* benchmarks, the miss ratio is 100% and read (misses) for the benchmarks are the same for both policies. The artificial perfect interleaving does not allow any coincidental prefetches for the eager policy.

2) Page influence on integrity tree sharing: In Fig. 8 we can see that most benchmarks have more write misses in the eager policy compared to the lazy policy, particularly near the root level (level 12). This is natural as every exposure of data causes a write in every level of the integrity tree under the eager policy. As there are fewer nodes per level near the root, collisions causing coherence misses become increasingly likely when climbing toward the root.

Interestingly, with only a few exceptions (e.g., *Radix* and *Canneal*), most benchmarks start the eager miss climb (departing from the lazy miss level) *after* level 2. This is the same behaviour we see in the *Continuous* and *Blocked Interleaved* microbenchmarks (cf. Fig. 6). The *Continuous* microbenchmark starts the climb *after* level 6 as it

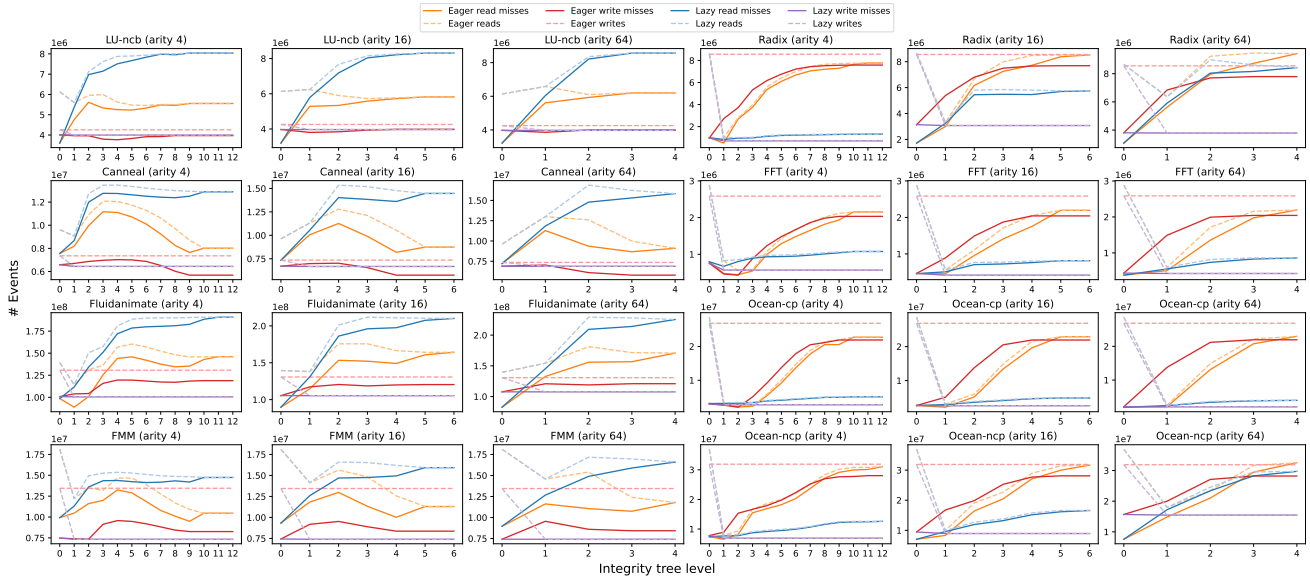

Fig. 9. Results from a random selection of eight PARSEC and SPLASH-3 benchmarks: Reads and writes (and misses) for both eager and lazy propagation for different integrity tree arity (4, 16, 64). Trends remain the same across different arity values but for much shorter trees with higher arity, which indicates higher contention for hot-spot nodes.

has continuous private blocks of 1 MB, the exact amount of memory covered by a level 6 integrity tree node. In the *Blocked Interleaved* microbenchmark the climb starts *after* level 0 as the continuous private blocks are 256 B (four cachelines).

Since level 2 integrity tree nodes cover 4 KB, we can conclude that most benchmarks exhibit 4 KB private blocks. However, this is likely tied to the fact that we have 4 KB pages in our simulation. It is possible that using larger pages, or placing pages used by a particular core consecutively in memory, we can delay the climb of eager write misses to higher integrity tree levels, similarly to the *Continuous* microbenchmark. That would decrease coherence overhead incurred by both write and read misses for the eager policy.

3) Integrity tree caching importance: In Fig. 7, the benchmarks marked with "*" exhibit a count of leaf reads that exceeds both read misses and writes many times over (the value is printed between the scale and benchmark name). As the leaf reads represent fetches of data to the private cache, the volume indicate that the benchmark's data set does not fit in the private cache. Since our simulation caches the integrity tree separately, it does not compete for capacity.

However, it is important to realise that if data and the integrity tree did compete for space and the data were allowed to replace integrity tree nodes, particularly at the leaf, many more fetches would cause misses and reads on many more levels. The lesson we can learn is that it is important to balance the cache storage for data and integrity nodes.

D. Sensitivity to the Arity of the Integrity Tree

Higher arity for integrity nodes results in a shallower integrity tree. With arity 16 and 64 every level represents two or three level of the arity-4 tree, respectively. In Fig. 9 we show the trends for some (randomly selected) benchmarks for higher arity. The statistics are similar for levels where nodes cover a specific amount of memory (e.g., level 0 with arity 64 corresponds to level 2 with arity 4), but skew toward the statistics of the levels that were skipped. However, coherence traffic is generally lower overall with higher associativity.

VI. Directions for Future Work

A. A Universal Policy

It is evident from the evaluation that, without taking into account evictions for the integrity tree, some benchmarks benefit from the eager update policy and some from the lazy one (while for a few it makes no significant difference). But what about if we account for evictions?

We can generalise the lazy/eager update propagation policies beyond the data cachelines to the whole integrity tree. For any cacheline, regardless of whether it contains data or is an integrity tree node at any level:

- On a dirty eviction, a *lazy* update of the next level is preferable (just to save the information that the evicted block has changed but not interfere with any other sharer).
- On a demand request, an *eager* update *upwards* in the integrity tree should be selected (as the requester will verify shortly after).

We can therefore propose the following:

Universal Policy

Lazy on evictions, eager on demand requests, for any cacheline, whether data or integrity tree cacheline, independently of tree level.

B. Data-Race Freedom

Another compelling reason to prefer the lazy integrity propagation policy on eviction is the effect of data-race–free programs [17]. Data used by one core will remain private, even when evicted, until it is released using synchronisation. After synchronisation, it is possible —and sometimes even likely— that another core will read previously private data. Assuming the integrity mechanism is aware of the synchronisation, it can trigger a flush of integrity metadata. Starting from the leaves, one level at the time, any dirty data or integrity metadata updates the integrity metadata in the level above. Updating in this order maximises the coalescing in the integrity tree as different branches converge. The entire integrity flush only requires one update

of the root. Without the synchronisation-triggered integrity flush, every request for a dirty cacheline or integrity node would cause an update of the relevant integrity branch, including the root.

C. Implications of Trees with Counters

An improvement to purely MAC-based integrity trees (e.g., Merkle trees) is to introduce one or more levels with counter nodes (e.g., Bonsai Merkle Trees). This modification also affects when it is feasible to start an integrity update.

When the integrity of a cacheline is protected by a MAC only, it is possible to be proactive by updating the MAC on every write. This is, however, very wastefully (especially when paired with eager integrity propagation). A better alternative is to be reactive and only make the integrity update when the cacheline is about to leave the cache. The drawback is instead that the entire integrity branch has to be updated *after* a request for a cacheline arrives, a time-consuming operation at a time-critical point.

With counters, data is associated with both a MAC and a counter, with the MAC computed from both the data and the counter. For the same reason as before, it would be wasteful to update the MAC proactively and it still makes more sense to follow a reactive strategy for it. The counter, however, only has to be incremented once per (exposed) MAC update, and *before* the MAC update. Since we know that the MAC has to be updated eventually once the cacheline becomes dirty, this is an excellent time to increment the counter proactively. The updated counter can be saved and used later when updating the associated MAC.

When a counter is involved, the MAC is not really part of the tree, that is, the MAC is not used to construct the MAC a level up in the tree. Instead, the counter serves as the link to the rest of the tree. This means that updates in the integrity branch can be propagated as soon as the counter is incremented. The MAC can be updated later, independently.

VII. Conclusion

Motivated by the need to extend memory integrity protection to cache-coherent multiprocessors, we performed a design space exploration of the policies for updating integrity trees. We presented both a proactive and a reactive update initiation in the integrity tree, as well as an eager and a lazy update propagation policy. We analysed how those policies affect coherence traffic, discussed their pros and cons, and concluded that both data cachelines and integrity tree cachelines (independently of tree level) should propagate updates lazily on evictions and eagerly on demand requests from another core. This work is a first step towards a systematic evaluation of fine-grain coherence for integrity trees, potentially enabling a slew of future work (§VI) to truly enable secure cache-coherent systems.

References

[1] M. Hassan, "Heterogeneous MPSoCs for mixed criticality systems: Challenges and opportunities," *arXiv preprint arXiv:1706.07429*, 2017.

[2] M. Hassan, A. M. Kaushik, and H. Patel, "Predictable cache coherence for multi-core real-time systems," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2017, pp. 235–246.

[3] S. H. Gade, M. Sinha, M. Kumar, and S. Deb, "Scalable hybrid cache coherence using emerging links for chiplet architectures," in *2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID)*. IEEE, 2022, pp. 92–97.

[4] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. H. Loh, M. Subramony, and S. White, "Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families: Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 57–70.

[5] E. Feng, D. Du, Y. Xia, and H. Chen, "Efficient distributed secure memory with migratable Merkle tree," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 347–360.

[6] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* IEEE, 2003, pp. 295–306.

[7] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, "Making secure processors os-and performance-friendly," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 5, no. 4, pp. 1–35, 2009.

[8] 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 *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007, pp. 183–196.

[9] Intel Corporation, "Intel® software guard extensions (intel® sgx)," ISCA '15 tutorial, https://community.intel.com/legacyfs/online/drupal_files/332680-002.pdf, Jun. 2015, reference no. 332680-002, Accessed: 2024-04-13.

[10] L. E. Olson, S. Sethumadhavan, and M. D. Hill, "Security implications of third-party accelerators," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 50–53, 2015.

[11] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *Proceedings of the 48th International Symposium on Microarchitecture*. IEEE, 2015, pp. 470–481.

[12] L. E. Olson, M. D. Hill, and D. A. Wood, "Crossing guard: Mediating host-accelerator coherence interactions," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 163–176, 2017.

[13] B. Rogers, M. Prvulovic, and Y. Solihin, "Efficient data protection for distributed shared memory multiprocessors," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '06. New York, NY, USA: ACM, 2006, pp. 84–94.

[14] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Nov. 2005.

[15] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, Jan. 2011.

[16] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2016, pp. 101–111.

[17] S. V. Adve and M. D. Hill, "Weak ordering – A new definition," in *17th International Symposium on Computer Architecture (ISCA)*, Jun. 1990, pp. 2–14.