A Parallel Tag Cache for Hardware Managed Tagged Memory in Multicore Processors

Wei Song, *Senior Member, IEEE*, Da Xie, Zihan Xue, and Peng Liu, *Member, IEEE*

Abstract—Hardware-managed tagged memory is the dominant way of supporting tags in current processor designs. Most of these processors reserve a hidden tag partition in the memory dedicated for tags and use a small tag cache (TC) to reduce the extra memory accesses introduced by the tag partition. Recent research shows that storing tags in a hierarchical tag table (HTT) inside the tag partition allows efficient compression in a TC, but the use of the HTT causes special data inconsistency issues when multiple related tag accesses are served simultaneously. How to design a parallel TC for multicore processors remains an open problem. We proposed the first TC capable of serving multiple tag accesses in parallel. It adopts a two-phase locking procedure to maintain data consistency and integrates seven techniques, where three are firstly proposed, and two are theoretical concepts materialized into usable solutions for the first time. Single-core and multicore performance results show that the proposed TC is effective in reducing both the extra amount of memory accesses to the tag partition and the overhead in execution time. It is important to provide enough number of trackers in multicore processors while providing extra trackers is beneficial for running HTT/TC ineffective applications.

Index Terms—Tagged memory, hierarchical tag table, tag cache, parallel accesses, search order, two-phase locking, data consistency.

1 Introduction

A tag[1] is a metadata attached to each data unit. Tagged memory is a method of supporting metadata processing in the processor hardware. It is actually an idea dating back to the early days of computers [1], [2] where tags were usually used to label the types of the data that they attached to. One well-known case of tagged memory was the Symbolics Lisp machines [2] made in the 80's, where tags were used by the garbage collector to distinguish addresses from numbers. In recent years, tagged memory has been adopted as a viable mechanism to support security related techniques, including tracking pointer integrity [3], dynamic information flow tracking [4], pointer capabilities [5], spatial and temporal memory safety [6], thwarting cache side-channel attacks [7], hardware-assisted address sanitizer [8], and even general-purpose watch-points [9]. The latest commercial processors begin to utilize tagged memory as well, such as the SPARC ADI [10], the memory tagging extension (MTE) implemented in ARMv8.5-A [11] and the Arm Morello SoC [12] designed for the next-generation Arm devices.

Hardware-managed tagged memory is the dominant way of supporting tags in current processor designs [5], [7], [10], [13]–[16]. It expands all data storage elements, from the processor registers to the physical memory, with a tag field. Tags stored in this field are managed by the processor hardware and hidden from user-level software (even the kernel). Fig. 1 shows an exemplary implementation of hardware-managed tagged memory in a dual-core processor. Cache

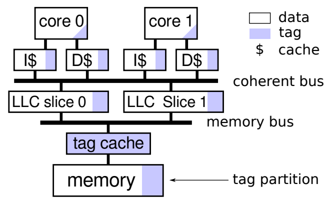


Fig. 1. Hardware-managed tagged memory.

blocks in all cache levels above the memory bus, including the private level-one (L1) caches and the shared last-level cache (LLC), are widened to store data along with their tags.

Since the memory is an array of DRAM chips scheduled by an on-chip memory controller typically shipped as a hard IP, it cannot be easily widened, and storing tags in the memory becomes a challenge. It might be possible to widen the memory if both DRAM chips and the memory controller are customizable [2]. Otherwise, re-purposing ECC bits for tags is also viable if just the memory controller is customizable. However, most processors cannot afford such non-standard memory. A more prudent alternative is to keep the memory untouched but reserve a hidden tag partition dedicated for tags, as depicted in Fig. 1. This has become the state-of-the-art solution [5], [7], [10]–[16], but brings a crucial problem: Each LLC memory access is divided into a data access for the data stored in the normal memory region and a tag access for the tag stored in the tag partition. The memory throughput demand is doubled. To avoid unacceptable overall performance of hardware-managed tagged memory, both the research community and the industry have been adopting the idea of "a small tag cache (TC) sitting between the LLC and the memory."

Substantial progress has been made recently in designing such a TC for single-core processors. Assuming a tag is much smaller than a machine word, a small TC, as shown in Fig. 1, can serve most of the tag accesses. For the SPEC CPU 2006 benchmark suite [17], it was reported that 94% of the

- *W. Song, D. Xie, and Z. Xue are with Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS and School of Cyber Security, University of Chinese Academy of Sciences. W. Song was also with Computer Laboratory, University of Cambridge between 2014 and 2017. E-mail: {songwei, xieda, xuezihan}@iie.ac.cn*
- *P. Liu is with Pennsylvania State University. E-mail: pxl20@psu.edu*

1. Please note this tag is not the higher address bits used for block matching in caches. The latter is called an address tag in this paper.

TABLE 1
Summary of tag cache related techniques.

Name	Description	Section	Initial Idea	Usable Solution	Support in Hardware HDFI [15]	CHERI [19]	this work
Hierarchical tag table	Introduce tag map bits.	3.1	HDFI	HDFI	✓	✓	✓
Uniformed TC	Store blocks of all HTT levels in a shared cache.	3.2	CHERI	CHERI	×	✓	✓
Parallel tag accesses	Support parallel read/write tag accesses.	5	this work	this work	×	×	✓
Bottom-up search order	Search an HTT from TTs.	4.3	this work	this work	×	×	✓
Dynamic search order	Dynamically choose the search order.	4.4	this work	this work	×	×	✓
Avoid redundant store	Avoid tag writes when tags are untouched.	4.1	CHERI	this work	×	×	✓
Avoid empty accesses	Avoid reading/writing empty TC blocks.	4.2	CHERI	this work	×	×	✓

extra tag accesses can be avoided using a 128KB TC [18]. Several recent studies have also explicitly acknowledged this performance benefit [7], [16], [19], [20] and the recently released Arm CoreLink CI-700 coherent interconnect [21] contains an optional TC to support Arm MTE [11]. Furthermore, since most tags are unused (zero by default) or tagged zero [13], a large proportion of tags cached in a TC are empty, which waste the precious TC space and can be avoided by cache compression [22]. According to the latest search [15], [19], storing tags in a hierarchical tag table (HTT) [19] in the tag partition allows efficient compression of the empty cache blocks inside a TC.

However, how to design a parallel TC for multicore processors remains an open problem. The goal of this work is to propose the first parallel TC for multicore processors. Our design is motivated by the following new challenges: **Challenge 1.** Severing tag accesses sequentially is unacceptable for multicore processors. All existing TCs [15], [18], [19] are sequential ones. On single-core processors, the tag access latency is likely hidden because it is shorter than the data access latency when the access hits in the TC. However, the amount of parallel tag accesses drastically increases on multicore processors. The data access latency is almost unchanged as modern memory controllers are made to serve multiple in-flight accesses simultaneously [23], but the time slot available for each tag access shrinks significantly if they are served sequentially. Performance degrades substantially when the tag access latency cannot be hidden. **Challenge 2.** For the first time, we demonstrate in Section 5.1 that naively supporting parallel tag accesses in an HTT-adopted TC introduces data inconsistency issues ultimately resulting in data corruption. The use of HTT transfers a tag access from a single access to a tag table into a search of an HTT by a sequence of TC accesses. There is no straightforward way to serve multiple tag accesses simultaneously without considering the interaction between related TC accesses. **Challenge 3.** On other hand, we would like to keep using HTTs in parallel TCs as, otherwise, they would suffer from significant waste of space.

We have discovered a couple of insights in the process of resolving these challenges: **Insight A, room for further efficiency improvement.** Existing TCs fail to exploit the full benefit of HTTs. An integration of HTT and six other techniques could enable us to produce an adequately efficient transactional design of a parallel TC. **Insight B, maintaining consistency by parallelizing transactions.** The sequence of TC accesses fulfilling one tag access is effectively a transaction. Severing parallel conflicting transactions is a classic problem in database systems and we can resolve the consistency issues by reusing the techniques proven effective. Based on these insights, we propose the first parallel TC capable of serving multiple tag accesses simultaneously. It adopts a two-tier tracker structure. Each LLC memory access is served by one of multiple tag transaction trackers (1st tier), which relays the data access directly to the memory controller and schedules the tag access (transaction) locally in the TC, which locates the tag in an HTT by a sequence of TC accesses. Each TC access is offloaded to one of multiple TC access trackers (2nd tier). Data consistency is maintained by enforcing a two-phase locking procedure typically used in database systems [24].

As summarized in Table 1, the proposed TC integrates seven techniques, where three are firstly proposed, and two are theoretical concepts materialized into usable solutions for the first time. Rather than searching an HTT always in a top-down order, we propose a new bottom-up search order and show that it can outperform the top-down order in certain scenarios. A hardware monitor is used to dynamically choose the optimal search order. It should be noted that, no viable solution for avoiding redundant store and empty accesses has ever been proposed. Considerable engineering work is needed to make them work in a sequential TC, and supporting them in a parallel TC is painstakingly complicated, as shown in Section 5.1 and 6.

Paper organization: Section 2 reviews the existing TC designs in the literature. Section 3 provides a thorough explanation of HTTs, especially its layout both in memory and in the TC. Based on **insight A**, Section 4 proposes four optimization techniques to further improve the efficiency of the TC. Following **insight B**, Section 5 demonstrates the potential consistency issues and presents our solution for maintaining data consistency. As illustrated in Section 6, the proposed TC has been implemented into the lowRISC SoC [25]. The performance of the proposed TC in both single-core and multicore processors has been evaluated in Section 7. Finally, the paper is concluded by Section 8. This TC is open-sourced at:

https://github.com/comparch-security/lowrisc-tag-v0.4

2 Related Work

Historically speaking, there are two ways of storing tags in the processor's cache hierarchy: *split* or *merged* caching. In a cache adopting the split caching, tags of all cache blocks are stored in a separate tag array side-by-side with the data array [14], while the merged caching expands each cache block to store data and tags in the same block [5], [15]. The split caching scheme allows the separate tag array to be compressed for reduced cache space, which would be a compromising benefit against the merged cache scheme considering that the majority of data are not tagged or tagged zero [13]. However, it is difficult to maintain data consistency between the data and the tag arrays because they are accessed separately. This problem is further exaggerated by the need of maintaining cache coherence in modern multicore processors. As a result, most of nowadays hardware-managed tagged memory designs adopt the merged cache scheme [5], [7], [10]–[12], [15], [16]

A problem raised by the merged caching scheme is the doubled memory traffic. As described in the introduction, an extra tag access is issued to fetch tags from the tag partition reserved in memory. To reduce this overhead on memory bandwidth, a TC is added between LLC and the memory controller. Early TCs are traditional set-associated caches [5], [15]. Although such TCs are effective in reducing the amount of the extra memory accesses, their sizes are quite large. It was reported that a 128KB TC was able to avoid 94% of the tag-incurred memory traffic in a single-core processor with an LLC of 256KB [15].

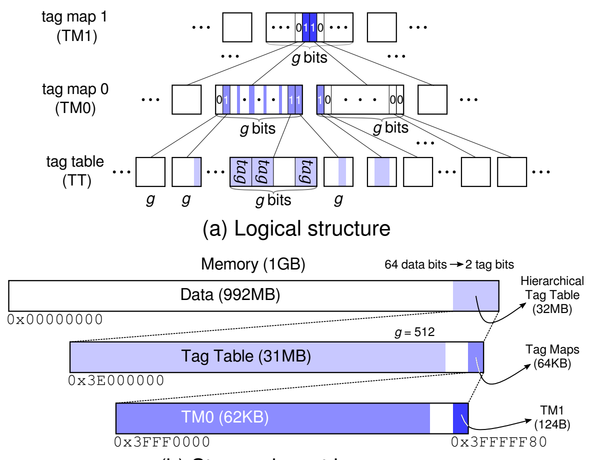
HDFI (hardware-assisted data-flow isolation) [15] is the first tagged memory design adopting an HTT in the tag partition. For its 512MB memory, 8MB was reserved for a tag table and an extra 128KB was reserved for a meta tag table. Each 64B entry in the tag table was linked to 1 bit in the meta tag table, which was set to 1 if the 64B tag table entry was non-empty. In the TC, if the corresponding meta tag table bit was found 0, the tag table entry was not stored to save cache space. Since the majority of data are tagged zero, the size of the tag table was significantly reduced.

In HDFI's TC, entries of the tag table and the meta tag table were stored in two separate 1KB caches. This static space partition may lead to inefficient cache utilization, e.g., the cache for the tag table may be fully occupied while that for the meta tag table is almost unused, because applications may present high spatial and temporal locality in their tag access pattern. To alleviate such inefficiency, a uniformed TC was used in an updated CHERI (capability hardware enhanced RISC instructions) processor [19]. Entries of both the leaf table (equivalent to HDFI's tag table) and the root table (equivalent to HDFI's meta tag table) were stored in the same cache. Therefore, space allocation between tables can adapt to the changes in the tag access pattern.

However, both the TCs implemented in HDFI and CHERI are sequential ones that serve only one tag access at a time. As explained in the introduction, this limits their effectiveness in multicore processors.

3 Storing and Caching Tags Using HTTs

Before introducing our solution for further improving the efficiency of TCs and maintaining data consistency when serving tag accesses in parallel, it is necessary to obtain a thorough understanding of the HTT, especially its layout both in memory and in the TC. Assuming a small and fixed sized (t bits) tag is attached to each 64-bit word in a 64-bit machine, each tag-extended cache block (used by L1 and L2/LLC) contains 64 bytes of data and $8t$ bits of tags. These tags are backed by the memory using a hidden tag partition typically reversing $t/64$ of the memory. To reduce the number of accesses to the tag partition, a TC is added between the LLC and the memory controller. This TC is found compressible as most tags are not in use or set zero [13]. HTT is the state-of-the-art memory layout [15], [19] for supporting such TC compression scheme. This section describes the details of the memory and the cache layouts for storing/caching tags using HTTs.



(a) Logical structure

(b) Storage layout in memory

Fig. 2. Structure of a hierarchical tag table (HTT).

3.1 Layout in Memory: Hierarchical Tag Table

Let us consider a hardware-managed tagged memory processor where the memory size is 1GB, each 64-bit word is attached with a 2-bit tag and a 3-level HTT is used. The logical structure of this HTT is shown in Fig. 2a while the storage layout in memory is depicted in Fig. 2b.

In the 1GB memory, the top 32MB ($1024 \times 2/64$) is reserved as the hidden tag partition, which is slightly more than enough to store the tags of the remaining 992MB data ($31 = 992 \times 2/64$). These tags are stored in a tag table (TT) which is located at the beginning of the tag partition (`0x3E000000`). Each TT entry is 2-byte wide storing the tags attached to a 64B data (cache) block. The mapping between a data block and its tags is linear, e.g., the tags attached to the data block at `0x00000100` is stored in the 4th TT entry ($256/64$) located at `0x3E000008`.

Besides the tag table, an HTT introduces extra tag map levels formed in a tree structure, such as the tag map 0 (TM0) and tag map 1 (TM1) as shown in Fig. 2a. Consecutive TT entries are grouped into fixed-sized TT nodes, each of which is g-bit wide. A single bit in TM0, which is also linearly mapped to TT, is used to record the status of the mapped TT node (1 for non-empty). Therefore, the 31MB TT needs a TM0 of ($31 \times 1024/g$) KB. Recalling that TT takes only 31MB of the 32MB tag partition, the higher 1MB space is left unused as shown in Fig. 2b. TM0 is located at the higher $32 \times 1024/g$ KB range of the tag partition. As long as $g \geq 64/t$, the unused space is large enough for storing TM0. As for the example processor ($t = 2, g = 512$), TM0 is 62KB located at `0x3fff0000`. Similar to TM0, which is used to record the status of TT, TM1 can be introduced to record the status of TM0. Using the same node size and the linear mapping, the 62KB TM0 is grouped into 992 TM0 nodes requiring 992 TM1 bits. The size of TM1 is only 124B. It is stored in the higher 2KB unused memory located at `0x3fffff80`.

The benefit of using an HTT is its natural support for TC compression. When a TT node is empty, caching the corresponding TM0 bit rather than the TT node consumes

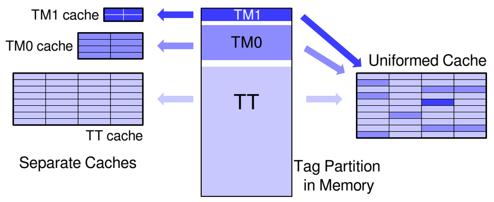


Fig. 3. Separate and uniformed TC layout.

significantly less cache space. Of course, TM0 bits are not cached individually but in cache blocks. In our design, the size of a node is deliberately set equal to a TC cache block (64B). Replacing an empty TT node with a TM0 node introduces no space overhead but space reduction as long as the number of zero bits in this TM0 node is > 1. Similarly, caching TM1 nodes when some TM0 nodes are empty can further boost the compression ratio. In addition, HTT introduces no storage overhead in the memory. As long as $g \geq 64/t$, TM0 can be stored in the unused memory space left by TT. The same holds true for TM1 as well.

3.2 Layout in Cache: Uniformed Tag Cache

There are two possible methods of storing tags and TM bits in a TC: *separated caches* or a single *uniformed cache*. Adopting the *separate caches* method, a separate cache is used for each HTT level, as depicted on the left-hand side of Fig. 3. This allows HTT levels to choose different cache sizes, block sizes and g. For a specialized processor dedicated for running a small set of targeted applications, the best storage trade-off can be achieved by choosing the optimal cache size, block size and g for each level at design time. However, designers cannot predict the applications running on a generic processor in advance. A static partition of the cache space leads to sub-optimal storage trade-off and unsatisfactory performance for most applications. It would be better to make the partition dynamically adapt to the tag access pattern.

A *uniformed cache*, as shown on the right-hand side of Fig. 3, provides such flexibility. Blocks from all HTT levels can be stored in a shared cache as long as they use the same block size. Consequently, the cache space occupied by each level is dynamically adjusted by the replace policy (pseudo-LRU in our TC), effectively adapting to the tag access pattern as blocks with high locality from all HTT levels are prioritized. The use of a uniformed cache usually leads to the same g as well [19]. A cache block is the smallest data unit managed by the cache. When g is larger than the block size, multiple blocks are represented by a single TM bit, requiring extra data consistency maintenance. When g is smaller than the block size, a cache block is represented by multiple TM bits, reducing the space efficiency of TM blocks. As a result, the optimal g is always the block size for all HTT levels.

3.3 Overall Structure of the Proposed TC

Adopting a uniformed cache, the structure of the proposed TC is shown in Fig. 4. Blocks of all HTT levels are cached in the same *data array* with its cache `state` (invalid, clean or dirty) and address tag (`ctag`) stored in the *metadata array*. Painted in different colors, all access types described in this paper are summarized in Table 2.

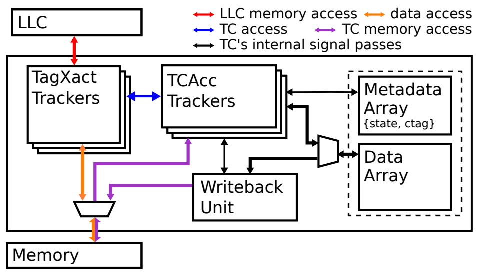


Fig. 4. Block diagram of the proposed TC.

TABLE 2
Summary of access types.

Access Type	Description
LLC memory acc.	LLC to TC: fetch/writeback a data block and its tags.
Data access	TagXact tracker to memory: fetch/writeback the data of an LLC memory access.
Tag access	A transaction scheduled by a TagXact tracker: fetch/writeback the tags of an LLC memory access.
TC access	TagXact to TCAcc trackers: access an HTT block.
TC memory acc.	TC to memory: fetch/writeback an HTT block.

An LLC memory access is served by one of the multiple tag transaction (*TagXact*) *trackers*.[2] It is divided into a data access, immediately relayed to the memory controller, and a tag access to locate the tag in the HTT (cached in the TC) by a sequence of TC accesses. Let us consider searching a non-empty tag in a top-down order (from TM1 to TT), the TT block containing this tag is accessed after its corresponding TM1 and TM0 blocks are sequentially examined. A tag write access is more complicated as it may update both TT and TM blocks after reading them. Two tag accesses become related when they need to access the same TM blocks, or even the same TT block. Consequently, a tag access becomes a transaction, and a parallel TC must guarantee that transactions are executed atomically [30]: A tag access must access shared HTT blocks without interfere with other related tag accesses. A tag access is either fully executed by updating all HTT blocks requiring changes; otherwise, the tag access is temporarily blocked, and no block is touched. Such atomicity is achieved later in Section 5.2 by enforcing a two-phase locking procedure when multiple HTT blocks are updated by one tag access. Letting a TagXact tracker directly schedule TC accesses presents inherent challenges. They are offloaded to multiple second-tier TC access (*TCAcc*) *trackers*. Since writing back a cache block happens less frequently than accessing a cache block, a shared *writeback unit* is used to handle all writeback requests.

4 Further Improvement of Efficiency

As described by **insight A** in the introduction, it is challenging to make a transactional design of a parallel TC adequately efficient. We present four techniques to improve the efficiency of a TC in Sections 4.1 to 4.4, respectively.

4.1 Avoid Redundant Store

When a dirty cache block is written back to memory by the LLC, the attached tags might be clean. It was reported that 50∼90% of all LLC write accesses are with clean tags [19],

2. The term "tracker" is used by the Rocket chip generator [26], [27] and dates back to the Intel 870 chipset [28]. In practice, a tracker fulfills the function of a MSHR [29] in generic non-blocking caches.

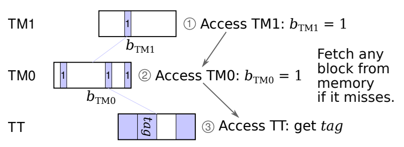

TM1
TM0
TT
① Access TM1: b_{TM1} = 1
② Access TM0: b_{TM0} = 1
③ Access TT: get *tag*
Fetch any block from memory if it misses.
b_{TM1}
b_{TM0}
tag

Fig. 5. Access a non-empty tag using the top-down order.

[31]. A simple and efficient optimization is to enforce read-before-write for all tag write accesses; therefore, a tag write access without modifying any tag can be detected as redundant and reduced to a read access. Since most data inconsistency issues are due to concurrent TC write accesses, reducing the amount of tag write brings down both the occasions needed to block conflicting TC accesses and the average tag access latency.

4.2 Avoid Empty Access

In an HTT, an empty non-top-level block is always labeled by its map bit with a zero. This is the reason why an HTT can be used to compress the TC as storing empty non-top-level blocks can be avoided. In an ideal scenario, only top-level blocks and non-empty blocks are stored in the TC.

However, the write-back policy complicates the scenario with two corner cases: a non-empty block might become empty due to a tag write, and an empty block is fetched from memory when it becomes non-empty. The first case wastes a block space until it is written back and both cases lead to memory fetch/write of empty blocks. To remove such unnecessary TC memory accesses, a TC can invalidate a cache block when it becomes empty and create an empty cache block without fetching it from memory.

4.3 Search Order

All existing TCs adopting HTTs locate tags using a *top-down* search order. As shown in Fig. 5, to access a non-empty *tag* in a 3-level HTT, TM1 is accessed first to check the top-level map bit b_{TM1}. If $b_{\text{TM1}} = 1$, TM0 is accessed to check whether $b_{\text{TM0}} = 1$. Finally, *tag* is accessed in TT. A total of three TC accesses are needed for reading a non-empty tag. The top-down search order is optimal when most tags are empty. In this case, a significant amount of tag accesses would finish with just one TC access as $b_{\text{TM1}} = 0$.

However, top-down might not be the optimal order for all applications. As shown later in Fig. 14, the memory accesses of many SPEC CPU 2006 benchmarks are heavily tagged in our tag use cases, including some of the memory heavy ones, such as 410.bwaves, 433.milc, 462.libquantum, and 470.lbm. For these applications, a bottom-up search order might outperform the top-down order as most memory accesses are attached with non-empty tags.

When the TT block hits, as shown in Fig. 6a, using the bottom-up order saves 2 TC accesses, i.e. TC accesses ① and ② to the TM blocks needed by the top-down order. Since these TM blocks are not accessed, they can actually miss in the TC. In other words, using the bottom-up order potentially save two TC memory accesses. When the TT block misses in the TC, as shown in Fig. 6b, the numbers of TC accesses are equal for both orders, but using the bottom-up order may save one TC memory access if the TM1 block misses in the TC. Only when both the TT and the TM0 blocks misses in the TC, as shown in Fig. 6c, using the bottom-up search order incurs two extra TC accesses, i.e., the TC accesses ⓐ and ⓑ to TT and TM0 respectively.

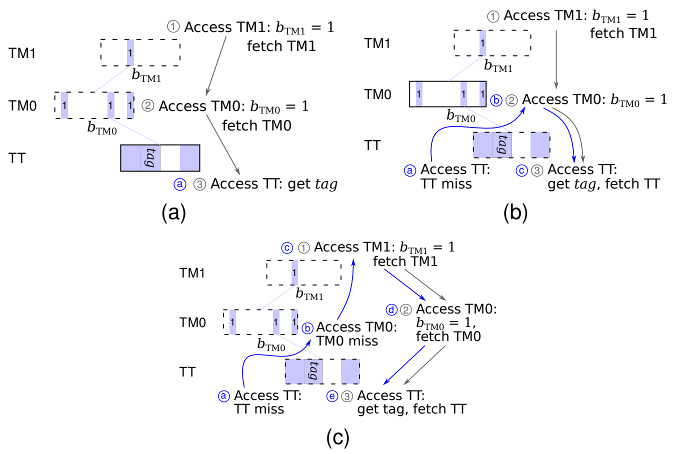

TM1
TM0
TT
① Access TM1: b_{TM1} = 1 fetch TM1
② Access TM0: b_{TM0} = 1 fetch TM0
ⓐ ③ Access TT: get *tag*
(a)
① Access TM1: b_{TM1} = 1 fetch TM1
ⓑ ② Access TM0: b_{TM0} = 1
ⓐ Access TT: TT miss
ⓒ ③ Access TT: get *tag*, fetch TT
(b)
ⓒ ① Access TM1: b_{TM1} = 1 fetch TM1
ⓑ Access TM0: TM0 miss
ⓓ ② Access TM0: b_{TM0} = 1, fetch TM0
ⓐ Access TT: TT miss
ⓔ ③ Access TT: get tag, fetch TT
(c)

Fig. 6. Access a non-empty *tag* using the bottom-up/top-down order when (a) the TT block hits, (b) the TT block misses but the TM0 block hits, and (c) both TT and TM0 blocks hit. TC accesses are sequenced according to the search order: The top-down order is sequenced by numbers while the bottom-up order is sequenced by alphabets. Missing blocks are circled in dashed lines.

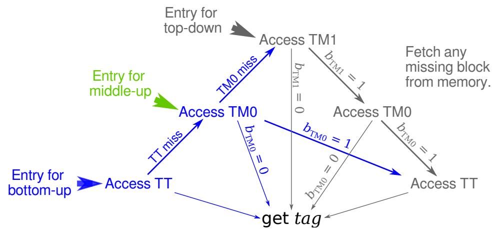


Fig. 7. Considering the bottom-up order as adding speculative TC read accesses (colored in blue) before falling back to the top-down search order (colored in gray).

It is important to note that missing blocks are not fetched from memory when the search is going upwards using the bottom-up order, i.e., TC accesses ⓐ in Fig. 6b, and ⓐ and ⓑ in Fig. 6c. If these missing blocks are fetched from memory, the HTT is reduced to a plain tag table, eliminating all the benefits. As shown in Fig. 7, we can consider the extra TC read accesses (colored in blue) introduced by the bottom-up order as speculative reads intended for finishing a tag access with a short-cut. If all short-cuts fail, the search falls back to the top-down order. A failed speculative TC read typically costs only 2$\sim$3 cycles for checking the cache status in the metadata array. Considering the long memory access latency, using the bottom-up order is cost-effective when most tag accesses can finish with a short-cut.

Finally, as indicated by Fig. 7, there exists another search order for 3-level HTTs, labeled as the *middle-up* order. Introducing a single speculative read to TM0, it is a trade-off between the bottom-up and the top-down orders.

4.4 Dynamically Select a Search Order at Runtime

As described in Section 4.3, the top-down order is optimal when most LLC memory accesses are untagged while the bottom-up search order may outperform it when most LLC memory accesses are tagged. However, designers cannot foreknow the applications running on a generic processor. It would be better to choose a proper search order suitable to the running application. To achieve this, we have derived

TABLE 3
Cost of using different search orders.

Category	Symbol	Bottom-Up	Middle-Up	Top-Down
TT served	S_{TT}	0	1	2
TM0 served	S_{TM0}	1	0	1
TM1 served	S_{TM1}	2	1	0
Cost of This Order		$S_{\mathrm{TM0}} + 2S_{\mathrm{TM1}}$	$S_{\mathrm{TT}} + S_{\mathrm{TM1}}$	$2S_{\mathrm{TT}} + S_{\mathrm{TM0}}$

a set of functions to estimate the cost of different search orders. It is then possible for a TC to dynamically choose the optimal order by implementing these functions in a hardware monitor.

As shown in Fig. 7, the value of a *tag* is confirmed only when one of its map bits (b_{TM0} or b_{TM1}) is found 0 or the TT block is accessed. Thanks to the avoidance of empty accesses as described in Section 4.2, all non-top-level blocks stored in a TC are non-empty and the block finally accessed to infer the value of a *tag* is unique, no matter which search order is used. For example, when $tag = 0$, $b_{TM0} = 0$, and $b_{TM1} = 1$, the *tag* is confirmed 0 only after accessing TM0, no matter it is a bottom-up/middle-up speculative read or a top-down normal read. The TT block must miss in the TC as it is empty. Utilizing this uniqueness, we can categorize a tag access by the block finally accessed: i.e., a tag access is *TT served* if the final block is a TT block. Consequently, we can define the cost of accessing a *tag* using a specific order as the number of potentially wasted TC accesses as listed in Table 3. For simplicity, we further assume the speculative reads for non-empty and top-level blocks always hit as the miss rate is low. Take a TM0 served access for example, the middle-up order introduces no extra TC access as the first block accessed is the final block. Using the bottom-up order leads to one wasted TC access to a missing TT block while using the top-down order wastes a TC access to TM1. In hardware, the numbers of tag accesses served in each category (S_{TT}, S_{TM0} and S_{TM1}) are recorded by three performance counters (PFCs). Assuming a total of $S = S_{\mathrm{TT}} + S_{\mathrm{TM0}} + S_{\mathrm{TM1}}$ tag accesses are served in a monitor period, the overall cost C (wasted TC accesses) of a certain search order can be estimated as the bottom row of Table 3. The optimal search order for the monitor period must incur the lowest cost, which is used to derive the order selection criteria:

$$\text{optimal order} = \begin{cases} \text{bottom-up} & \text{if } S_{\mathrm{TT}}/S > 50\%; \\ \text{top-down} & \text{if } S_{\mathrm{TM1}}/S > 50\%; \\ \text{middle-up} & \text{otherwise.} \end{cases} \quad (1)$$

Assuming the tag access pattern does not change drastically between consecutive periods, the optimal order of the previous period is likely a good choice for the current period.

5 Maintaining Data Consistency

It is crucial to serve multiple tag accesses in parallel for efficiently supporting tagged memory in multicore processors, however, it is not easy to achieve. Following **insight B** in the introduction, we demonstrate the potential consistency issues when parallel tag accesses are naively served in parallel and propose our solution for maintaining consistency.

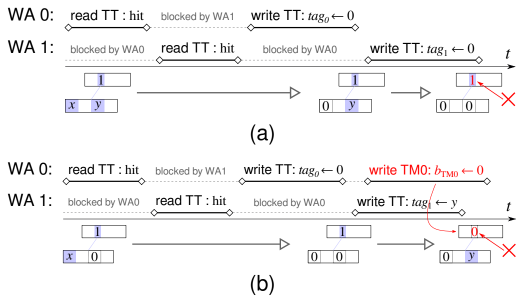


Fig. 8. Two inconsistent cases caused by two parallel tag write accesses: (a) persistent or (b) missing b_{TM0}.

5.1 Data Consistency Issues

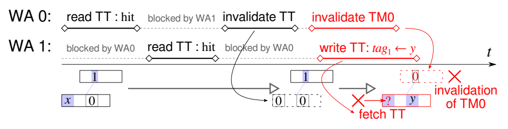
Unique to TCs adopting HTTs, serving a tag write access simultaneously with another tag access potentially leads to inconsistency issues. Any tag update that also updates its map bits must guarantee that all other related tag accesses observe both updates at the same time. For simplicity, all cases demonstrated in this Section are presented in a 2-level HTT. They become even more complicated in 3-level HTTs.

Serving two related tag write accesses may leave the HTT with a corrupted tag map bit. In the case depicted in Fig. 8a, both tag write accesses (WA0 and WA1) want to clear their tags (tag_0: $x \to 0$ and tag_1: $y \to 0$), which are the last two tags stored in the same TT block mapped to b_{TM0}. Recall that all tag writes must read the block before write to avoid duplicated write operations (Section 4.1). Assuming WA0 and WA1 finishing reading TT around the same time, both of them must assume that the TT block would remain non-empty after their own write operations. As a result, none of them would proceed further to clear b_{TM0}, leaving it in a stale and wrong state of 1.

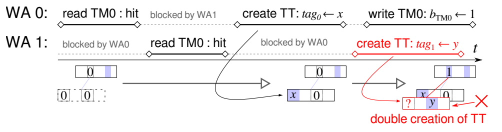
This b_{TM0} could be mistakenly cleared as well, as shown in Fig. 8b. In this case, WA0 still wants to clear its tag (tag_0: $x \to 0$) which is left as the last tag in a TT block while a parallel WA1 want to initiate a tag (tag_1: $0 \to y$) in the same block. Let us assume that WA0 and WA1 have finished reading TT around the same time, and WA0 would clear its tag ($tag_0 \leftarrow 0$) before WA1's tag initiation ($tag_1 \leftarrow y$). Since WA0 would believe it has cleared the TT block, it would go further to clear b_{TM0} unaware of WA1. Finally, the non-empty tag_1 is mistakenly mapped to a zero b_{TM0}. For both cases, the map bit b_{TM0} is mistakenly left untouched or cleared because the parallel and related tag write accesses are unaware of each other, and finish their accesses based on potentially outdated observation of the related HTT blocks.

Unfortunately, supporting the avoidance of empty accesses (Section 4.2) makes the situation more complicated. As demonstrated in Fig. 9, parallel tag writes may lead to fetching stale blocks, invalidating non-empty blocks and double creation of existing blocks. Please note that we demonstrate only the straight-forward cases while the complicated ones are omitted due to page limit.

When the WA0 in Fig. 8b is allowed to invalidate empty blocks, the situation is worsened to the one shown in Fig. 9a. The two TC write accesses of WA0 are replaced with invalidation operations, which mistakenly invalidate the TM0 block containing b_{TM0}. What is worse, WA1's TT write access would find the block missing in the TC. Believing the block were evicted due to cache replacement, WA1 would fetch

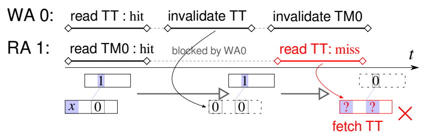


(a)

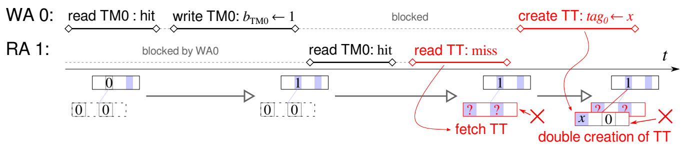


(b)

Fig. 9. Further inconsistent cases caused by avoiding accessing empty blocks: (a) inconsistent fetch and invalidation, and (b) double creation.



(a)



(b)

Fig. 10. Two inconsistent cases caused by parallel read and write accesses: (a) inconsistent fetch and (b) double creation.

it from memory and initiate the tag ($tag_1 \leftarrow y$). However, the fetched block is stale as the latest copy has just been invalidated by WA0 without a writeback. The final TT block might be seriously corrupted.

Another case relating to block creation is shown in Fig. 9b where both WA0 and WA1 want to initiate two tags (tag_0: $0 \rightarrow y$ and tag_1: $0 \rightarrow y$) in the same TT block and mapped by b_{TM0}. After simultaneously reading TM0, both WA0 and WA1 create a TT block as $b_{\text{TM0}} = 0$, resulting in two inconsistent copies of the same TT block.

Serving a tag read access simultaneously with a related tag write access can cause consistency issues as well. They can be summarized into two scenarios: (a) a stale block is fetched from memory due to a late removal of a tag map bit. and (b) double creation of a block due to an early initiation of a tag map bit. An example of the first scenario is depicted in Fig. 10a, where WA0 tries to clear the last remaining tag (tag_0: $x \rightarrow 0$) in a TT block while a parallel tag read access (RA1) tries to read the same TT block. Unfortunately, RA1 reads b_{TM0} before it is cleared by WA0. Since $b_{\text{TM0}} = 1$, RA1 proceeds further with a read to the TT block just after it is invalidated by WA0. As the block misses in the TC, a stale copy is mistakenly fetched from memory. An example of the second scenario is illustrated in Fig. 10b. WA0 tries to initiate a new tag (tag_0: $0 \rightarrow x$) in an empty TT block while a parallel tag read access (RA1) tries to read the same TT block. If WA0 updates TM0 before TT, RA1 may observe the prematurely updated $b_{\text{TM0}} = 1$ and consequently issues an access to the TT block before WA0 gets a chance to create it. As a result, RA1 would mistakenly fetch a stale copy from memory and WA0's creation would be a duplicated one. For both scenarios, the culprit is the stale/premature observation of b_{TM0} by the parallel RA1.

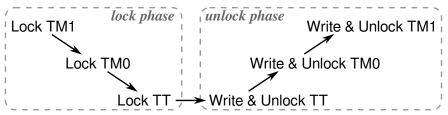


Fig. 11. Update a tag using the two-phase locking procedure.

5.2 Two-Phase Locking

Unlike in database systems where partial transactions can be safely aborted, updating a HTT block in the TC cannot be rolled back. A transactional TC must be able to avoid inconsistency: A tag access is either finished atomically with all HTT blocks being updated, or it is blocked before making any update. Our solution is to enforce a conservative *two-phase locking* procedure for updating a tag along with its tag map bits. As shown in Fig. 11, the two-phase locking procedure contains a lock phase, where all the blocks potentially updated by a tag write are locked in a top-down order, and an unlock phase, where these blocks are written and unlocked in a bottom-up order. A locked block cannot be written or read by any tag access except for the one locking it. This guarantees that the content of a block is untouched when a related block is updated. It stops a related tag access from reading a stale/premature block as well.

The concept of *two-phase locking* was initially proposed for resolving conflicting transactions in large-scale relational database management systems [24]. Assuming a transaction may modify multiple data entries and related transactions can occur simultaneously, two-phase locking requires all data entries to be modified by a transaction are locked by the transaction before any entry is modified or unlocked. Data consistency is maintained because locking a new entry after unlocking any previously locked entry is prohibited. This strict division between lock and unlock phases ensures no cyclic dependency in the finished transactions. The system is therefore consistent. However, two-phase locking does not guarantee deadlock-free for the unfinished transactions, because two transactions may mutually wait for the entries locked by the other in the lock phase.

Since it is difficult in hardware to abort a transaction once it is started, the proposed transactional TC guarantees deadlock-free by enforcing a top-down lock order and a reversed unlock order. Whenever two tag write accesses are related, they must share the same b_{TM1} in a 3-level HTT. The top-down lock order requires the TM1 block containing this b_{TM1} to be locked first, which blocks all other related tag write accesses, effectively serializing the related write accesses, and ensures that all tag accesses can finish in a finite duration. Only unrelated tag write accesses are allowed to proceed in parallel. A bottom-up lock order would work as well. However, TT and TM0 blocks might be deliberately invalidated without being written back. When a block misses in the lock phase, it cannot be fetched from memory without first check the tag map bit. The order effectively falls back to top-down and the control logic is unnecessarily complicated. Another problem is the unnecessary blockage of related tag read accesses. A bottom-up lock order implies the top-down unlock order which makes TT blocks the last to unlock. All tag read accesses to this block are blocked during the whole tag write process. On the contrary, this TT block is actually never locked using the top-down lock order. Tag read accesses can proceed in parallel using the bottom-up search order.

5.3 Reasoning the Two-Tier Tracker Structure

The proposed TC adopts a two-tier tracker structure where each LLC memory access is served by one TagXact tracker and TC accesses are offloaded to TCAcc trackers. Although letting TagXact trackers handle TC accesses directly may result in a smaller design, a two-tier structure is more modular, which normally leads to less design errors. It is also difficult to achieve the required control flexibility without the two-tier structure.

TCAcc trackers also help resolve a consistency corner case. Two related tag write accesses may request to lock the same TM1 block in the same clock cycle, which leads to double lock and potential deadlock. If TC accesses are directly handled by TagXact trackers, the TC needs to detect whether two (or more) TagXact trackers are requesting the same block and grant only one of them, which is difficult to handle with a large number of TagXact trackers. With the help of TCAcc trackers, it is possible to divide cache sets into banks and map each bank with a dedicated TCAcc tracker. This tracker then becomes a serialization point for all TC accesses to the bank, resolving the corner case, at the cost of a small conflict rate as unrelated TC accesses may conflict on the same bank. This also ensures that a TC access always successfully finishes in a finite duration once it is allocated to a TCAcc tracker. Note that this banking scheme cannot resolve the inconsistent cases described in Section 5.1 (e.g. Fig. 9b) as it cannot enforce a proper TC access order for related tag accesses.

6 Hardware Design

The proposed TC has been implemented into the Rocket chip generator [27] with some necessary modification in the core pipeline, cache hierarchy and the on-chip interconnects to support tags and extra PFCs. The overall structure of the proposed TC has been discussed in Section 3.3. This section details the designs of the core components.

A *TagXact tracker* divides an LLC memory access into a data access to the memory and a tag access to the local TC. For an LLC memory write access, the LLC is immediately acknowledged once the data write access is relayed to the memory. The associated tag write access proceeds in the background and the tracker becomes available again once the access is finished. For an LLC memory read access, the LLC is acknowledged when both the data and the tag accesses are finished. If the tag access takes longer time than the data access, extra memory access latency is introduced.

The tag access is scheduled by a state machine depicted in Fig. 12, which can be configured to support the dynamic selection of search orders and different levels of HTT (up to 3) at synthesis time. For all states, except for **IDLE**, a TC access fulfilling a certain command is issued to a TCAcc tracker. As listed in Table 4, there are eight commands in total and the possible commands issued by each state is labeled as a superscript colored in blue.

In a bottom-up search, a tag access starts with a state transition from **IDLE** to **TTR** (**IDLE**→**TTR**). In the worst case, both speculative reads miss ($\textbf{TTR}^{\text{R}}\xrightarrow{miss}\textbf{TM0R}^{\text{R}}\xrightarrow{miss}\textbf{TM1FR}$),

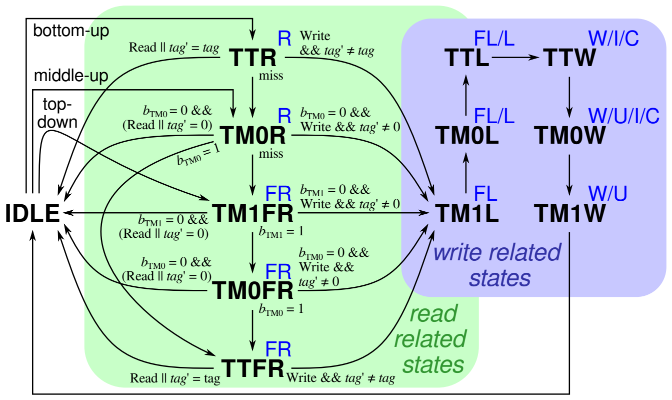


Fig. 12. State machine of the TagXact tracker for a 3-level HTT.

TABLE 4
Available commands for TC accesses.

	Description		Description
R	Speculative read.	FR	Forced read. Fetch when miss.
C	Create a block.	L	Speculative read and lock.
I	Invalidate a block.	FL	Forced read and lock. Fetch when miss.
U	Unlock a block.	W	Write a block. Fetch when miss.

and the search falls back to the top-down order with all missing blocks fetched from the memory: $\textbf{TM1FR}^{\text{FR}}\rightarrow\textbf{TM0FR}^{\text{FR}}\rightarrow\textbf{TTFR}^{\text{FR}}$. The state machine skips states whenever possible. For example, $\textbf{TM0R}^{\text{R}}\xrightarrow{b_{\text{TM0}}=1}\textbf{TTFR}$ when TT misses but $b_{\text{TM0}}=1$. Tag accesses immediately finish whenever a tag is inferred empty for a read, such as $\textbf{TM0R}^{\text{R}}\xrightarrow{b_{\text{TM0}}=0}\textbf{IDLE}$ and $\textbf{TM1FR}^{\text{R}}\xrightarrow{b_{\text{TM1}}=0}\textbf{IDLE}$, or whenever a write is found redundant, such as $\textbf{TTR}^{\text{R}}\xrightarrow{tag=tag'}\textbf{IDLE}$ and $\textbf{TM1FR}^{\text{R}}\xrightarrow{b_{\text{TM1}}=0\ \&\&\ tag'=0}\textbf{IDLE}$, where tag' denotes the new tag. The state transits to **TM1L** immediately when a write is confirmed necessary, such as $\textbf{TTR}^{\text{R}}\xrightarrow{tag\neq tag'}\textbf{TM1L}$ and $\textbf{TM1FR}^{\text{R}}\xrightarrow{b_{\text{TM1}}=0\ \&\&\ tag'\neq 0}\textbf{TM1L}$.

As described in Section 5.2, a tag, along with its tag map bits, is written using a two-phase locking procedure. All related blocks are locked in a top-down order ($\textbf{TM1L}^{\text{FL}}\rightarrow\textbf{TM0L}^{\text{FL}}\rightarrow\textbf{TTL}^{\text{FL}}$) and written (unlocked) in a bottom-up order ($\textbf{TTW}^{\text{W}}\rightarrow\textbf{TM0W}^{\text{W}}\rightarrow\textbf{TM1W}^{\text{W}}$). For each TM block, the tracker checks whether the TM need an update during the lock phase. If not, the command is reduced to U, which is significantly faster than W. When a block is to be cleared, the W command is replaced with a faster I command to invalidate the block. Similarly, a block is created using C instead of W when the it is confirmed empty by its TM bit. Note that this optimization is not applied to the top-level TM blocks as there is no map bit to back them up. If a TM bit is found zero in the lock phase, all blocks on lower levels are empty and FL is reduced to L to avoid fetching them. Finally, the TT block is not locked in state **TTL**, because no other tag write accesses would update the same tag between **TTL** and **TTW**, and leaving the block unlocked allows for parallel tag read accesses.

When the search order is not bottom-up, tag accesses starts with different entry states: $\textbf{TM1FR}^{\text{FR}}$ for top-down or $\textbf{TM0R}^{\text{R}}$ for middle-up. The TagXact tracker can be statically configured to support a 2-level HTT or a plain tag table by removing unnecessary states and commands. For a 2-level HTT, states **TM0R**, **TM1FR**, **TM1L** and **TM1W**, along with

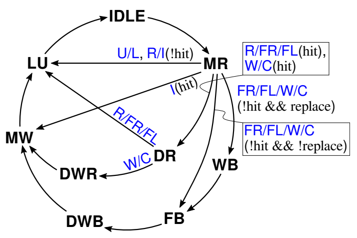


Fig. 13. State machine for of a TCAcc tracker.

commands L for **TM0L** and I/C for **TM0W**, are removed. The top-down order follows the path of the middle-up order. Reducing the 2-level HTT into a plain tag table results in the further removal of **TTR**, **TM0FR**, **TM0L** and **TM0W**, along with L for **TTL** and I/C for **TTW**.

TCAcc trackers are responsible for fulfilling TC accesses according to the commands issued by TagXact trackers. The tracker is controlled by a state machine as depicted in Fig. 13. All TC accesses start by reading the metadata in state **MR** to check whether the requested block hits in the TC. Depending on the command and the cache status, an access is dispatched on different paths. For a read access (R/FR/FL), the data array is accessed in **DR**, if the block hits in the TC. Otherwise, the block is fetched from memory in **FB**, refilled in **DWB** and recorded to meatadata in **MW**, if the read is a forced one (FR/FL). Of course, replacing a dirty block leads to its writeback in **WB**. When a speculative read (R) misses in the TC, the access finishes without fetching the missing block. For a lock-related access (L/U), a block is either locked (L) or unlocked (U) in **LU** without accessing the data array. **LU** also serves as a common exiting point for all TC access, where a response containing the requested tag (map bit) is assembled. For write accesses (W/C/I), W and C share the same path. If the block hits in the TC, the tag (map bit) is updated in a read-before-write process (**DR**→**DWR**). Otherwise, the block is fetched from memory in **FB** (only for W), refilled in **DWB** and recorded to meatadata in **MW**. The new tag (map bit) is first written to the fetch buffer in **FB** and then refilled to the data array in **DWB**. Recall that C does not fetch the missing block. The fetch buffer is simply cleared in **FB**. For block invalidation (I), the metadata is cleared in **WB** if the block hits in the TC. For all TC writes, the block is unlocked afterwards in **LU**. To further reduce latency, all TC write accesses are early acknowledged in **MR**.

A vector of lock registers are added to support the L command. Each register records an address to be locked, a valid flag and its owner (a TagXact tracker). When a block is locked by a TCAcc tracker under the request of a TagXact tracker, a free (invalid) lock register is validated, and filled with the block address and the requesting TagXact tracker (owner). Whenever a TagXact tracker issues a TC access, the address of the requested block is compared with all valid lock registers in parallel. The access is blocked if a match is found and the TagXact tracker is not the owner. A valid lock register is invalidated when the locked block is unlocked (U) or written (W/C/I) by the lock owner. All TC accesses pending on this lock are unblocked afterwards. There should be sufficient number of lock registers to avoid deadlock. The optimal number is $2m$ where m is the number of TagXact trackers. In the worst case when all TagXact trackers are serving tag write accesses, each tracker would lock two blocks. A total of $2m$ lock registers ensures that a free lock is always available. An optimization is made to boost the number of parallel tag accesses by shrinking the lock granularity from a block to a single bit. For the default configuration listed in Table 5, a TM1 block covers 0.5GB memory while it is only 1MB for a TM1 bit. Locking by blocks effectively serialize all tag write accesses (and some read accesses) in a 0.5GB range of memory while reducing the granularity to a bit shrinks the range to 1MB, which significantly reduces the probability for a tag access being blocked by others. For a 2-level HTT, this range shrinks further to just 2KB.

A rich set of PFCs have been added to the cache system (including the TC). The original Rocket chip supports only a limited set of PFCs for monitoring events in the processing core. Depending on the number of available control registers, e.g., four in a SiFive U74 core [32], only a small number of events can be monitored in parallel. We need much more PFCs for exploring the design space of the TC. 38 PFCs are therefore added to all cache levels from L1 to TC. They occupies a separate address space accessible by only three control registers [33]. Multiple PFCs can be read in a burst and the round trip latency is around 2 to 4 cycles. Regarding the TC, PFCs monitor events including read, read miss, write, write miss and writeback in each HTT level, and record the numbers of tag accesses served by each HTT level and total TC accesses.

7 Performance Evaluation

The tag-enabled Rocket chip is ported to a Diligent Gensys-2 FPGA board. However, only one core is implemented due to the limited resources. To overcome this limitation, we managed to collect memory traces of the SPEC 2006 benchmark [17] running on a 4-core processor and feed this trace to a register-transfer level (RTL) simulation of the TC to evaluate the multicore performance. Table 5 lists the parameters configurable at synthesis time along with their default values. The maximal tag size of 8-bit is due to the use of 64-bit wide memory blocks for the data array. Larger tags can be supported by using wider memory blocks. By default, the search order is dynamically chosen at runtime ($o = 3$). The area overhead is analyzed in Table 6. Supporting tagged memory introduces 22.6% extra logic and 7.2% extra memory where 19.6% and 2.2% of the logic and the memory are introduced by the TC while the rest is incurred by supporting tags in the original Rocket chip. Table 6 reveals also the area of a single TagXact tracker, a single TCAcc tracker and PFCs. Together they take almost all the logic of a TC.

7.1 Description of the Tag Use Cases

To evaluate the performance of the proposed TC, three demonstrative tag use cases exploiting a 2-bit tag per 64-bit machine word are added to the GNU GCC compiler used to compile the SPEC 2006 benchmark. Since the use cases utilize tags in different memory regions (code, stack and heap), they share the same 2-bit tag. Fig. 14 reveals the breakdown of (tagged and non-tagged) memory regions and the total size of the occupied memory by each program. The tagged

TABLE 5
Parameters available at synthesis time.

Name	Default	Constraint	Description
B	64	≥ 8	TC block size in bytes.
S	32	≥ 4	Number of sets.
W	4	≥ 2	Number of ways.
t	2	[1 : 8]	Tag bits per every 64-bit word.
l	3	[1 : 3]	Levels of HTT (plain tag table if $l = 1$).
o	3	[0 : 3]	Search order: 0, top-down; 1, bottom-up; 2, middle-up; 3, dynamic.
m	2	≥ 1	Number of TagXact trackers.
c	2	2's power	Number of TCAcc trackers.

TABLE 6
Breakdown of area overhead.

	Slice	Overhead	RAM Block	Overhead
Rocket-Chip	16177	—	160	—
Rocket-Chip (tagged)	19841	22.6%	171.5	7.2%
Tag Cache (8KB)	3170	19.6%	3.5	2.2%
A TagXact Tracker	654	4.0%	0	0%
A TCAcc Tracker	786	4.9%	0	0%
PFC	607	3.8%	0	0%

workload is relatively heavy as over 50% memory accesses are tagged for most benchmarks.

Tag call/return instructions: All indirect call/return instructions are tagged and untagged call/return raises exception at runtime. Although not a widely used defense, it demonstrates the possibility of tagging code in read-only pages, which is a rare feature in most tagged memory designs. It can be used to thwart direct code injection, derive data tags and raise precise exceptions for control-flow checking. The call and return instructions are identified by the linker and stored in binary using an extra section. The loader is modified to initialize the instruction tags before setting the code pages read-only.

Tag return addresses (RAs) on stack: This is a commonly utilized defense. When calling a function, RA is automatically tagged by the processor. This tag follows RA where ever it goes, either register or stack. An exception is raised if an untagged RA is used by a return. Tags on stack are explicitly cleared when the stack frame is released to avoid replay attacks. Hardware is responsible for tag generation, forwarding and checking while the compiler inserts extra tag removal operations in the function epilogue.

Tag the live data space in heap: The memory allocator is revised to tag the allocated data space and clear tags when it is released. This can be used to detect some forms of use-after-free attacks and potentially accelerate garbage collection algorithms, although such a tagging scheme is extremely heavy. For programs acquiring large data space at runtime, such as 410.bwaves, 401.bzip2 and 433.milc, almost all memory accesses are tagged as revealed in Fig. 14. As a result, the tag workload evaluated in this paper is much heavier than all of the previous evaluations [15], [19].

7.2 Single-Core Evaluation (FPGA)

For the evaluation of the single-core Rocket chip running on FPGA, the processor runs at 60MHz and is configured with two 32KB L1 caches, one 512KB LLC (L2) and a 8KB TC. The 1GB off-chip memory runs at 800MHz. We have successfully run 21 out of the 29 SPEC CPU 2006 benchmarks. Each benchmark case has been run for 10G instructions.

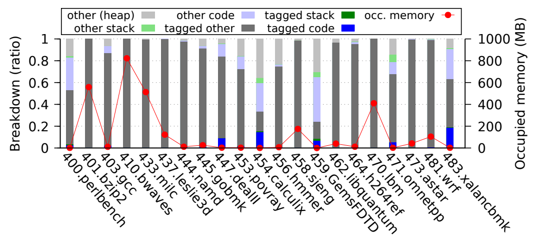


Fig. 14. Breakdown of the occupied memory (curve) by each program.

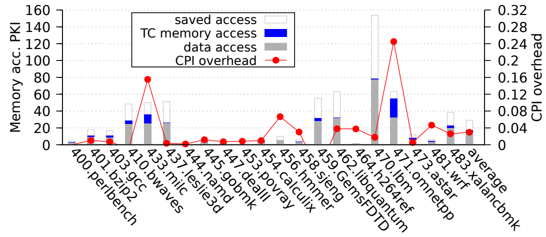
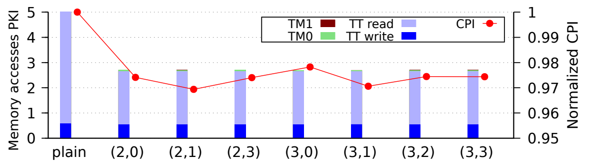


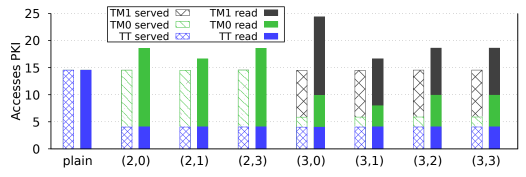
Fig. 15. Memory accesses PKI and CPI overhead (curve).

Fig. 15 shows the memory accesses per kilo instructions (PKI) and clock per instruction (CPI) overhead of running the tagged benchmark. The heavy use of tags in heap slightly increases the amount of L2 memory accesses, such as the heap-heavy cases 410.bwaves and 433.milc. Using a TC reduces the amount of TC memory accesses from 100% to 12.7% of the data accesses and the CPI overhead is around 2.97%. For most benchmarks, the CPI overhead is lower than 4%. The only two benchmarks suffering large CPI overhead are 433.milc and 471.omnetpp. Further investigation shows that they are the only benchmarks where >75% tag accesses are served by TT blocks while the average is only 47%. 471.omnetpp even suffers from the highest TT miss rate of 91.6%. Observable from Fig. 15, they generate the highest amount of TC memory accesses. They are the two cases where TC is ineffective due to the lack of locality in tag access pattern. For other benchmarks, the tiny 8KB TC is already efficient enough.

Fig. 16a demonstrates the reduced TC memory accesses by using an HTT, and the performance impact of using different HTT levels (l) and search orders (o). Introducing TMs besides a plain tag table has significantly reduced the amount of TC memory accesses from 5.0 to 2.7 accesses PKI. Most reduction comes from the reduced memory read for TT blocks while the amount of memory accesses to TM blocks (HTT's overhead) is marginal. The choices on HTT levels and search orders have nearly no effect on the amount of memory accesses but their impact on the CPI overhead is



(a) TC mem. access and normalized CPI (baseline: plain)



(b) Breakdown of tag access (left) and TC read (right)

Fig. 16. Reduced CPI and TC memory accesses by HTT. (x-axis: (l, o))

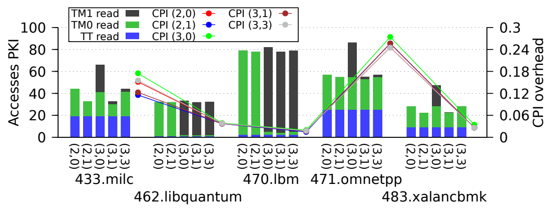


Fig. 17. TC read accesses (bar) and CPI (curve) of some memory heavy benchmarks (x-axis: (l, o)).

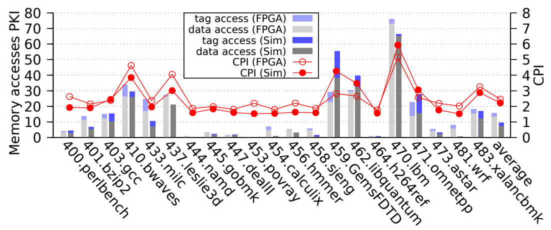


Fig. 18. Comparison of mem. accesses (bar) and CPI (curve) between FPGA (5G instr.) and simulation (5G warm-up + 5G sample cycles).

noticeable. Compared with using a plain tag table ($l = 1$), using extra TMs ($l > 1$) reduces CPI by ∼2.5%. The bottom-up search order ($o = 1$) outperforms other orders regardless of the number of HTT levels. The CPI overhead is reduced by 3% for both 2 and 3-level HTTs.

We have extracted the numbers of tag accesses served by (as defined in Section 4.4) and TC read accesses (hit or non-speculative) to each HTT levels, as presented in Fig. 16b. 73% of the tag accesses become TM0 served when using a 2-level HTT, while 82% of the those TM0 served become TM1 served when the level increases to three. Since the memory space covered by a cache block increases with the HTT level, the total amount of TC memory accesses is therefore reduced. The choice on search order has no impact on the proportion of tag accesses served by each level. What it actually affects is the amount of TC accesses to each HTT level. For both 2 and 3-level HTTs, the bottom-up order incurs the minimal numbers of TC read accesses, which is the reason for its lowest CPI.

Interestingly, dynamically choosing search order does not achieve the lowest CPI. For both 2 and 3-level HTTs, using the dynamic search order ($o = 3$) results in a sub-optimal CPI 0.5% higher than the optimal. Fig. 17 shows the numbers of TC read accesses to each HTT level and the CPI overhead of using different search orders for the memory heavy benchmarks also with high CPI overhead (>2%). The result shows that the choice of search order has a noticeable impact on CPI only when the TC read accesses to TT retain a large portion of total accesses, i.e. 433.milc and 471.omnetpp, indicating ineffective use of TM blocks. For 471.omnetpp, using dynamic search order with a 3-level HTT, $(l, o) = (3, 3)$, still achieves similar CPI with the optimal case of $(l, o) = (2, 1)$. 433.milc is the only case where dynamic search order leads to substantial CPI rise. Following investigation also shows that the memory access pattern varies frequently that the dynamic search order fails to follow the pattern changes. Nevertheless, the multicore evaluation presented in the next section shows that dynamic search order does adapt to various memory accessing patterns and provide concrete CPI reduction.

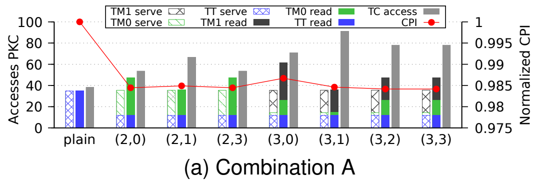
7.3 Multicore Evaluation (Simulation)

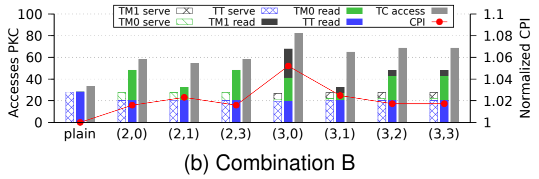
Supporting parallel tag accesses is one of the major benefits of the proposed TC. Since our FPGA board is only large enough for a single-core processor, we evaluate the multicore performance using a trace-based RTL simulation. A trace of the LLC memory accesses issued by four SPEC benchmark cases running on a 4-core processor (32KB L1s per core and a shared 2MB LLC) is collected using the Spike RISC-V simulator [34] with a latency-aware cache model [35]. Each trace entry records one LLC memory access, including the memory address, the attached tag, read/write and the estimated allowable latency. This trace is fed to a RTL simulation of the TC linked with a behavioral memory model to estimate the prolonged execution time. Instead of targeting an FPGA system where the core is significantly slower than the memory, the simulation is scaled to target an ASIC chip where the memory access latency is much longer than LLC access latency. Since the generated trace almost fills up our hard drive, we cannot run the same amount of instructions with the FPGA run. For each test, a trace is collected still from running 10G instructions per core, but the RTL simulation collects results from only 5G cycles after 5G warm-up cycles.

Fig. 18 compares the results from the FPGA and the simulation runs. For most benchmarks, the amount of memory accesses and CPI match closely between the two. The CPI differences are within 0.5 cycle. The simulation run produces slightly lower CPI than the FPGA run because the Spike simulator is inaccurate in simulating the conflicts happening on a pipelined core, leading to over-optimistic issuing of memory accesses. The trend is reversed for memory heavy benchmarks as the core frequency is scaled to target an ASIC chip, where the long memory access latency has a larger impact on CPI than in the FPGA run. Some benchmarks show larger than average differences, such as 433.milc, 437.leslie3d, 459.GemsFDTD and 462.libquantum. Since the simulation run executes only a portion of the instructions finished by the FPGA run, e.g., 3.3G for 437.leslie3d and 1.7G for 462.libquantum, they show different results as different ranges of instructions are executed. 433.milc and 459.GemsFDTD also shows frequent variation on their memory access pattern. Nevertheless, the overall result of the simulation run is still consistent and can be used to evaluate the performance of a TC.

Several combinations of benchmark cases have been evaluated using a 64KB TC configured with eight TagXact and eight TCAcc trackers, $(m, c) = (8, 8)$. Three representative combinations have been identified. ***Combination A****, memory heavy and HTT effective cases*: 410.bwaves, 437.leslie3d, 459.GemsFDTD and 470.lbm. ***Combination B****, memory heavy cases with one HTT ineffective case*: 459.GemsFDTD, 462.libquantum, 471.omnetpp and 483.xalancbmk. ***Combination C****, memory light cases with two HTT ineffective cases*: 403.gcc.input4, 403.gcc.input6, 433.milc.input0 and 471.omnetpp.

Most benchmark combinations behave similar to combination A as shown in Fig. 19a. Since a large proportion of tag accesses are served by TM blocks, HTT is effective in reducing the number of cached TT blocks, leading to significant drop in both TC memory accesses and CPI. All benchmark cases are memory heavy in combination A.


(a) Combination A


(b) Combination B

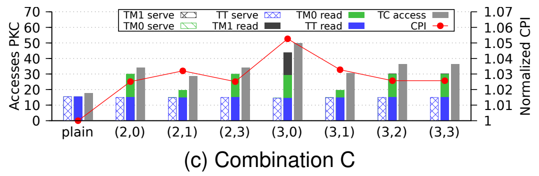

(c) Combination C

Fig. 19. 4-core performance with different levels and search orders (bar: tag access (left), TC read access (middle), total TC read access (right); curve: normalized CPI based on plain; x-axis: (l, o)).

Collectively they produce both the highest tag accesses of 38 per kilo cycles (PKC), and the highest TC accesses of 90 PKC. Using HTTs reduces CPI by ∼1.5% compared with a plain tag table. Top-down is the optimal search order for a 2-level HTT while it is middle-up for a 3-level HTT. Unlike the FPGA run, the dynamic search order successfully choose the optimal order for both 2 and 3-level HTTs.

Combination B and C are the abnormal cases identified in our evaluation. Instead of reducing CPI, using an HTT prolongs it when one or more HTT ineffective benchmark cases, i.e. 433.milc and 471.omnetpp, runs along with others. As shown in Fig. 19b, using an HTT instead of a plain tag table increases CPI by ∼1.9% when 471.omnetpp is included in combination B. It gets worse when combination C includes two HTT ineffective cases. CPI rises by ∼2.5% as shown in Fig. 19c. Although the amounts of tag/TC accesses of both combinations are lower than combination A, the proportion of TM served tag accesses shrinks significantly in combination B and becomes almost invisible in combination C, which directly causes the prolonged CPI. This result shows that HTT is not universally beneficial. When one or more HTT ineffective application runs on a multicore processor, dynamically disabling HTT might be a good choice.[3] Nevertheless, the dynamic search order successfully chooses the optimal search order for both combinations.

However, HTT ineffectiveness alone does not seem to explain everything. Although 471.omnetpp is HTT ineffective, using an HTT still reduces CPI by 0.12% in the single-core FPGA run. We believe another cause is the exhausted TagXact and TCAcc trackers. The default numbers of trackers, $(m, c) = (8, 8)$, are simply scaled from the single-core configuration of $(2, 2)$. However, concurrent applications may issue burst memory accesses around the same time, stressing the TC with a massive amount of tag accesses. Additional trackers may help alleviate the impact. Fig. 20 shows the result of rerunning combination B using suitable search orders with extra trackers, $(m, c) = (16, 16)$.

3. This can be potentially supported by our design as the TagXact state machine is already configurable at synthesis-time.

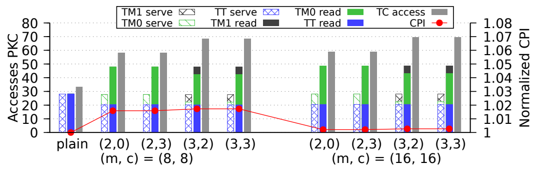


Fig. 20. Improving performance of combination B with extra trackers (figure notations are same with Fig. 19).

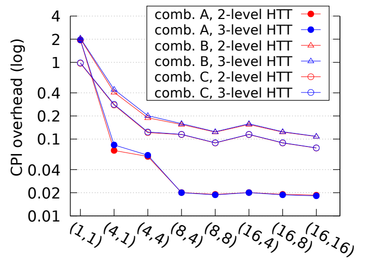


Fig. 21. CPI with extra TagXact and TCAcc trackers (x-axis: (m, c)).

All tag/TC accesses figures remain the same but the CPI overhead drops to just 0.27%. The number of trackers has a significant performance impact on multicore processors.

Fig. 21 reveals the CPI reduction with the increasing number of trackers. The number of HTT levels is almost irrelevant to CPI when the dynamic search order is used. By increasing the number of trackers, CPI drops substantially and it is most visible for combination A. A strictly sequential TC, $(m, c) = (1, 1)$, incurs a 200% CPI overhead. It drops to just 2% when the numbers of trackers are increased to $(8, 4)$. Introducing extra trackers earns marginal improvement, indicating that $(8, 4)$ is sufficient. The HTT ineffective combinations, i.e., combination B and C, gain less, but still significant, CPI drop compared with combination A. Interestingly, $(8, 4)$ is not enough to harvest the full benefit. By increasing trackers from $(1, 1)$ to $(16, 16)$, the CPI overheads of combination B and C are reduced from 206% and 97% to 10.8% and 7.6%, respectively. It is also observable that adding more TagXact trackers without companion TCAcc trackers becomes fruitless after $(8, 8)$. Overall, it is important to provide enough trackers and providing extra trackers is beneficial for running HTT/TC ineffective applications.[4]

7.4 Comparing with existing TCs

The TC of CHERI [19] is currently the most advanced in existing designs. It has several differences compared with our TC. It attaches a 1-bit tag to each 256-bit data and a TC block is 128-byte wide. LLC memory accesses are claimed bypassing the TC without being tracked, while the TC sequentially access tags in the meantime.[5] This is equivalent to $(m, c) = (\infty, 1)$ in our design. CHERI's TC is fixed to use the top-down order in a 2-level HTT, $(l, o) = (2, 0)$. The avoidance of both redundant store and empty accesses was not implemented. We ignore CHERI's preference on tag and cache block sizes as it is specialized for CHERI only. An equivalent TC using our design is configured to match all

4. Our TC is not limited to 4-cores. Running combinations B and C on 8-cores incurs only 4% CPI overhead comparing with running combination B on 4-cores. Also, as a TC can be considered as a cache attached a memory controller, its bandwidth should scale with the memory controller rather than the number of processing cores.

5. It was not described in [19] how tags and the data returned from memory are later combined, especially when a tag access finishes late.

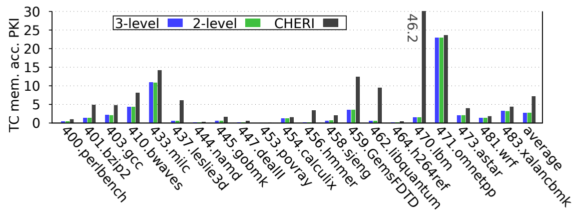


Fig. 22. Single-core comparison with CHERI (legend: HTT levels).

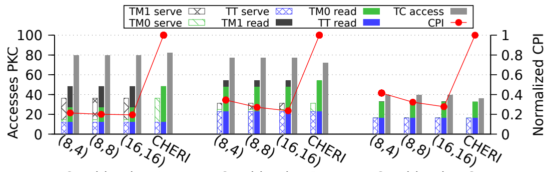


Fig. 23. 4-core comparison with CHERI (x-axis: (m, c), normalized CPI based on CHERI, other figure notations are same with Fig. 19)

other differences: $(l, o, m, c) = (2, 0, 2, 1)$ for the single-core FPGA run and $(2, 0, 16, 1)$ for the 4-core simulation run. The avoidance of redundant store and empty accesses is manually disabled.

Fig. 22 shows the amount of TC memory accesses of CHERI against our TCs using 2 or 3 levels of HTTs. Our TCs clearly outperform CHERI. The lack of avoiding redundant store and empty accesses leads to significant amount of TC memory accesses. CHERI incurs 164% extra TC memory accesses by average, or 49 times in the worst case, i.e., 456.hmmer. The marginal differences between 2-level and 3-level HTT configurations indicate that the choice on HTT levels is irrelevant when the search order is dynamic, against CHERI's suggestion [19]. CHERI's CPI is around 2.592 by average, while it is 2.551 and 2.550 for our 3-level and 2-level TCs, respectively, leading to a 1.6% overhead on CHERI.

Fig. 23 reveals CHERI's performance on a 4-core processor against our default TC with three tracker configurations, $(m, c) = (8, 4)$, $(8, 8)$ and $(16, 16)$. CPI results are normalized using CHERI as the baseline. Since top-down is the optimal search order for a 2-level HTT (as shown in Fig. 19), CHERI's TC performs similarly with our TCs in the number of TC read accesses and incurs slightly less TC accesses than ours TC for combination B and C. However, our TCs outperform CHERI in CPI performance by large margins. CHERI's CPI overhead is 9.4%, 46% and 28% for combination A, B and C, respectively. When our TC is configured to $(m, c) = (8, 4)$, the seemingly sufficient configuration for combination A, CPI overhead is reduced by 79%, 66% and 58% against CHERI for combination A, B and C, respectively. When the numbers of trackers increase to $(16, 16)$, CPI overhead is further reduced by another 2%, 10% and 14% for combination A, B and C, respectively.

HDFI [15] is the first design adopting a 3-level HTT in its TC. The cache space is statically allocated to each HTT level. Both TT and TM0 take half of the cache space while all of TM1 is stored in registers. It is a sequential TC fixed to the top-down order, and lack of the avoidance of redundant store for TT and empty accesses for both TT and TM0. Comparing with CHERI's TC, the statically allocated cache space is the major drawback, and we would like to evaluate its impact. We have reproduced HDFI's TC in our simulation model and compared it with our default TC. To make a fair comparison, the cache and memory sizes of HDFI have been

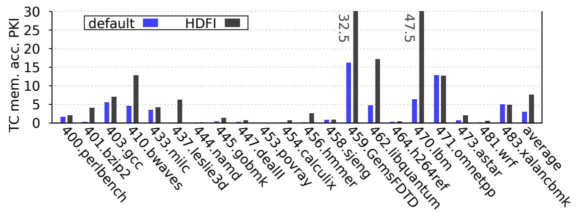


Fig. 24. Comparison with HDFI.

proportionally increased, while our TC is made sequential with the corresponding avoidance optimizations disabled. Fig. 24 demonstrates the amount of TC memory accesses of both TCs. Out of the 21 benchmark cases, 19 incur extra TC memory accesses using HDFI's TC. HDFI's average memory overhead is around 155% compared with our default TC while 456.hmmer achieves the worst overhead of 29.7 times.

8 Conclusion

In this paper, we proposed the first TC capable of serving multiple tag accesses in parallel, which is crucial for supporting tagged memory in multicore processors. The TC adopts a 2 or 3-level HTT which is searched by an order dynamically decided according to the tag access pattern. A two-phase locking procedure is used to ensure data consistency when serving related tag accesses in parallel. The proposed TC integrates seven techniques, where three are firstly proposed, and two are theoretical concepts materialized into usable solutions for the first time. Performance evaluation shows that the CPI overhead for single-core processors is just 2.97% while it is around 2% when four TC effective applications running on a 4-core processor. Even when some applications are TC ineffective, CPI overhead can be reduced to 7∼10% by adding trackers.

Acknowledgments

The initial hardware design of the tag cache was done by W. Song as a member of the lowRISC project [25] during his employment in the University of Cambridge between November 2014 and October 2017. The subsequent work was done collectively by all of the authors. The authors thank Prof. Robert Mullins and Alex Bradbury for their guidance on tagged memory, and Dr. Alexandre Joannou and Dr. Jonathan Woodruff for discussions on the design of tag caches. The subsequent work was partially supported by the National Natural Science Foundation of China under grant No. 61802402 and 62172406, and the CAS Pioneer Hundred Talents Program. Any opinions, findings, conclusions, and recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] E. A. Feustel, "The Rice Research Computer — a tagged architecture," in *Spring Joint Computer Conf.*, 1972, pp. 369–377.
[2] D. A. Moon, "Garbage collection in a large Lisp system," in *Proc. ACM Symp. LISP and Functional Programming*, 1984, pp. 235–246.
[3] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proc. IEEE/ACM Int'l Symp. Microarchitecture*, Dec. 2004, pp. 221–232.

[4] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proc. ACM Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, Oct. 2004, pp. 85–96.

[5] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *Proc. IEEE Symp. Security and Privacy*, May 2015, pp. 20–37.

[6] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *Proc. Int'l Symp. Computer Architecture*, Jun. 2012, pp. 189–200.

[7] A. Harris, S. Wei, P. Sahu, P. Kumar, T. M. Austin, and M. Tiwari, "Cyclone: Detecting contention-based cache information leaks through cyclic interference," in *Proc. IEEE/ACM Int'l Symp. Microarchitecture*, Oct. 2019, pp. 57–72.

[8] "Hardware-assisted addresssanitizer design documentation," The Clang Team, 2021, https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.

[9] J. L. Greathouse, H. Xin, Y. Luo, and T. M. Austin, "A case for unlimited watchpoints," in *Proc. ACM Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, Mar. 2012, pp. 159–172.

[10] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves C/C++ memory safety," *CoRR*, Feb. 2018, abs/1802.09517.

[11] "Armv8.5-A memory tagging extension," Arm, Aug. 2019.

[12] *Morello Prototype Architecture Overview*, 1st ed., Arm, Feb. 2022.

[13] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory," in *Proc. USENIX Conf. Operating Systems Design and Implementation*, Dec. 2008, pp. 225–240.

[14] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. K. Jr., B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," in *Proc. ACM Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, Mar. 2015, pp. 487–502.

[15] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *Proc. IEEE Symp. Security and Privacy*, May 2016, pp. 1–17.

[16] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco, S. Malik, M. Tiwari, and T. M. Austin, "Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn," in *Proc. ACM Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 469–484.

[17] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[18] W. Song, A. Bradbury, and R. Mullins, "Towards general purpose tagged memory," in *RISC-V Workshop*, Jun. 2015.

[19] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. D. Son, and A. T. Markettos, "Efficient tagged memory," in *Proc. IEEE Int'l Conf. Comp. Design*, 2017, pp. 641–648.

[20] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A. Sadeghi, "TIMBER-V: Tag-isolated memory bringing fine-grained enclaves to RISC-V," in *Proc. Network and Distributed System Security Symp.*, Feb. 2019.

[21] *Arm® CoreLink™ CI-700 Coherent Interconnect*, r3p0 ed., Apr. 2022.

[22] S. Sardashti, A. Seznec, and D. A. Wood, "Skewed compressed caches," in *Proc. Int'l Symp. Microarchitecture*, 2014, pp. 331–342.

[23] C. Natarajan, B. Christenson, and F. A. Briggs, "A study of performance impact of memory controller features in multi-processor server environment," in *Proc. Workshop Memory Performance Issues*, Jun. 2004, pp. 80–87.

[24] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, no. 11, pp. 624–633, 1976.

[25] J. Kimmitt, W. Song, and A. Bradbury, "Tutorial for the v0.4 lowRISC release," lowRISC, Jun. 2017.

[26] H. Cook, "Productive design of extensible on-chip memory hierarchies," Ph.D. dissertation, EECS Department, University of California, Berkeley, May 2016.

[27] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket chip generator," University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016.

[28] F. A. Briggs, M. Cekleov, K. Creta, M. Khare, S. Kulick, A. Kumar, L. P. Looi, C. Natarajan, S. Radhakrishnan, and L. Rankin, "Intel 870: A building block for cost-effective, scalable servers," *IEEE Micro*, vol. 22, no. 2, pp. 36–47, 2002.

[29] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proc. Ann. Symp. Comp. Architecture*, 1981, pp. 81—-87.

[30] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[31] C. Molina, A. González, and J. Tubella, "Reducing memory traffic via redundant store instructions," in *Proc. Int'l Conf. High-Performance Computing and Networking*, Apr. 1999, pp. 1246–1249.

[32] *SiFive U74-MC Core Complex Manual*, 21st ed., SiFive, Inc., 2021.

[33] Z. Xue, D. Xie, and W. Song, "Hardware performance counter based on RISC-V," *Computer Systems & Applications*, vol. 30, no. 11, pp. 3–10, 2021, (in Chinese).

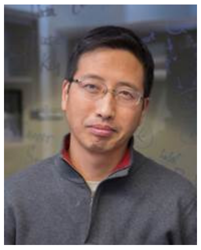
[34] A. Waterman, T. Newsome, C.-M. Chao, and others, "Spike RISC-V ISA simulator," 2021, https://github.com/riscv/riscv-isa-sim.

[35] W. Song and P. Liu, "Dynamically finding minimal eviction sets can be quicker than you think for side-channel attacks against the LLC," in *Proc. Int'l Symp. Research in Attacks, Intrusions and Defenses*, Sep. 2019, pp. 427–442.

Wei Song is an Associate Professor at the Institute of Information Engineering, CAS. He received his Ph.D. from the University of Manchester, and had worked as a postdoctoral researcher in the University of Manchester and the University of Cambridge. During his employment in Cambridge, he was the hardware lead for the first four releases of the lowRISC SoC platform. His current research focuses on secure computer architectures.

Da Xie is a Ph.D. student at the Institute of Information Engineering, CAS. He received his BE degree from Huazhong University of Science and Technology. His current research focuses on secure computer architectures.

Zihan Xue is a Ph.D. student at the Institute of Information Engineering, CAS. He received BE and ME degree from Southwest Jiaotong University and had worked as assistant engineer at CRSC Research & Design Institute Group Co., Ltd. His current research focuses on secure computer architectures.



Peng Liu received his BS and MS degrees from the University of Science and Technology of China, and his PhD from George Mason University in 1999. Dr. Liu is the Raymond G. Tronzo, MD Professor of Cybersecurity, and Director of the Cyber Security Lab at Penn State University. His research interests are in all areas of computer security. He has published over 350 technical papers. He has served on over 100 program committees and reviewed papers for numerous journals. He is currently the Co-Editor-in-Chief of Journal of Computer Security.