Efficient Tagged Memory

Alexandre Joannou*, Jonathan Woodruff*, Robert Kovacsics*, Simon W. Moore*, Alex Bradbury*, Hongyan Xia*,
Robert N. M. Watson*, David Chisnall*, Michael Roe*, Brooks Davis†, Edward Napierala*,
John Baldwin†, Khilan Gudka*, Peter G. Neumann†, Alfredo Mazzinghi*,
Alex Richardson*, Stacey Son†, A. Theodore Markettos*

*Computer Laboratory, University of Cambridge, Cambridge, UK †SRI International, Menlo Park, CA, USA
Website: www.cl.cam.ac.uk/research/comparch Website: www.sri.com

***Abstract*—We characterize the cache behavior of an in-memory tag table and demonstrate that an optimized implementation can typically achieve a near-zero memory traffic overhead. Both industry and academia have repeatedly demonstrated tagged memory as a key mechanism to enable enforcement of powerful security invariants, including capabilities, pointer integrity, watchpoints, and information-flow tracking. A single-bit tag shadowspace is the most commonly proposed requirement, as one bit is the minimum metadata needed to distinguish between an untyped data word and any number of new hardware-enforced types. We survey various tag shadowspace approaches and identify their common requirements and positive features of their implementations. To avoid non-standard memory widths, we identify the most practical implementation for tag storage to be an in-memory table managed next to the DRAM controller. We characterize the caching performance of such a tag table and demonstrate a DRAM traffic overhead below 5% for the vast majority of applications. We identify spatial locality on a page scale as the primary factor that enables surprisingly high table cache-ability. We then demonstrate tag-table compression for a set of common applications. A hierarchical structure with elegantly simple optimizations reduces DRAM traffic overhead to below 1% for most applications. These insights and optimizations pave the way for commercial applications making use of single-bit tags stored in commodity memory.**

I. Introduction

Hardware support for tagged memory has been implemented from early days of computer architecture [1], [2], and tagged memory is used by many research systems to enforce security invariants in nearly unmodified programs, including tracking pointer integrity [3], [4], enabling unforgeable capability tokens [2], [5]–[8], tracking programmable information-flow [9]–[11], and even general-purpose watchpoint systems to support both debugging and software-defined security invariants [12], [13]. However, the costs associated with tagged memory have been unclear. Tag-storage access patterns are unique, with each bit potentially representing many bits of data memory. Although some research projects have recommended tag storage in standard memory, and a few have developed implementations, none have characterized single-bit tag access patterns sufficiently to inform implementations or further optimizations.

For simplicity, we identify three points in the tagging design space: no tag, a *single-bit tag* (SBT), or a *multi-bit tag* (MBT) per word. This paper demonstrates that SBT systems can be nearly as efficient as untagged memory. We do not attempt to optimize MBTs, although some of the principles here will also apply to small MBT systems.

The contributions of this paper include:

- A survey of proposed implementations of SBT systems identifying a practical approach: an in-DRAM tag table with a tag cache next to the DRAM controller, including tags with metadata in data caches.
- A characterization of the dynamic workload of tag-table caches whose hit rates can be surprisingly high, considering that we are below the last-level cache so most temporal and spatial locality has already been exploited. We sweep parameter spaces and evaluate against a range of benchmarks with diverse characteristics.
- A characterization of an elegantly simple and highly effective compression scheme for three tag use cases, finding that it reduces overhead for tag-memory traffic to nearly zero for most applications.

Benchmarks run on our FPGA implementation confirm the simulation results, and demonstrate that an SBT memory can be implemented using commodity memory at near-zero performance cost.

II. Single-bit Tagged Memory

Tags are often stored in a *shadowspace* that holds M-bits of metadata in a hidden memory for every N-bytes of conventional visible memory. Tags in a shadowspace can provide integrity (popular for security) because the shadowspace cannot be named by instructions from the host architecture and is therefore naturally protected from tampering without impeding program execution. Tagged memory enables a number of ambitious and useful functions that solve difficult problems in computer systems with high performance. We might divide tag use cases into information flow [3], [9], [11], [14]–[16], memory safety [4], [17]–[19], capability protection [6]–[8], instrumentation [12], [13], and general-purpose [20]–[25].

Various tagged architectures share the requirement of a single-bit tag (SBT) shadowspace [6]–[8], [11], [14], [16]. SBT shadowspace designs either require exactly two hardware types (and therefore interpret the tag bit directly [11], [12]), or use the tag bit to indicate a complex typed word (i.e., a word

This work is part of the CTSRD project sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. We also acknowledge the EPSRC REMS Programme Grant [EP/K008528/1], the EPSRC Impact Acceleration Account [EP/K503757/1], an ARM iCASE award and Google, Inc.

with embedded metadata [5]–[8]). While some effort has been spent optimizing large shadowspaces for MBT designs [9], [10], [14], [15], [18], [26], we find that an SBT shadowspace can achieve far lower overhead in a simple implementation, and note that many applications of large tags can be implemented more efficiently by embedding tag metadata within data words with some special handling in software.

Embedded vs. External Tag Architectures

The shadowspace mechanism both *isolates* tags from interfering with data and *protects* the tags from data manipulations. While isolated metadata must be stored entirely in the shadowspace, protection of an arbitrary amount of metadata can be achieved with a single bit. A system that needs an M-bit MBT on certain words (where M is less than word size) can simply embed this M-bit field in each word and mark these tagged words with an SBT in the shadowspace. Hardware will then naturally propagate the M-bit tag with the word, and can prevent these tag bits from being overwritten by legacy operations if the SBT is set. Clearly, these M bits are not *isolated* from data, as less data is available in this word to the application; however, these MBTs can be *protected* by an SBT in the shadowspace. This approach is often used in capability architectures to protect capabilities in memory, such as in the IBM System/38 and AS/400 [2], the M-Machine [5], Aries [7], Low-fat Pointers [8], and CHERI [6]. We may call these *embedded* tag architectures, as they embed metadata within the word itself, as opposed to *external* tag architectures that store metadata entirely in the shadowspace.

Embedded-tag architectures require tagged words to be explicitly handled in software. If an application can accommodate tags that are visible in data, embedded tags may be used in place of a large shadowspace. The primary benefit of embedded tag architectures is that the major cost of metadata storage scales with use. That is, if metadata is embedded directly in words of memory, and only when needed, metadata consumes memory space only to the extent that it is explicitly used. This technique is exploited by the CHERI capability system to allow 192-bits of metadata per pointer with acceptable overheads [6]. Intel MPX uses an external shadowspace for similar structures but incurs a fixed 400% memory overhead for all memory that could contain pointers [27], with predictably high overheads. However, it should be noted that CHERI requires extensive compiler support for wide pointers and perturbs application binary interfaces, while Intel MPX remains mostly undetectable in the layout of program memory.

Hardbound: Devietti *et al.* present an interesting hybrid between the embedded and external approaches [19]. Hardbound has both an MBT shadowspace for bounds that is twice the size of data memory, and also a SBT shadowspace to indicate whether the bounds are actually in the shadowspace or whether a short bound is embedded in the pointer itself. Most bounds can be embedded in just 11 bits of the pointer, avoiding the external lookup, allowing a very low overhead in the common case. The Hardbound study also found that if the SBT shadowspace grew to a 4-bit MBT shadowspace, runtime overhead due to additional memory accesses grew by about 250% on average for their benchmarks. The Hardbound example suggests that the embedded-tag approach is desirable, supporting SBT for integrity, but embedding the remainder of the metadata in the word for performance.

Embedded tags are most commonly used in pointers. Virtual addresses often leave some bits available for possible future use. Furthermore, pointers are crucially important for secure execution; all data is loaded and stored through pointers, so nearly all data protection properties can be expressed as pointer checks rather than checks on the data itself. CHERI, which we use in our hardware implementation, supports a few pointer-specific use cases (e.g., bounded code or data pointers), while also enabling software-defined applications (e.g., information flow, file descriptors) that may selectively use the hardware-supported fields and semantics.

III. Single-bit Tag Storage

Brief consideration of the tag-storage problem and a survey of literature yields a few potential solutions:

- *width* – Store tag bits in a wider physical memory, e.g., 65-bit words.
- *in-page* – Store tag bits in borrowed bytes of DRAM pages, slightly shrinking each DRAM page and skewing the DRAM address mapping.
- *dedicated* – Store tag bits in a dedicated memory.
- *table* – Store tag bits in a table in DRAM.

The *width* option is ideal; historical systems such as the Burroughs B6000 simply widened system memory [1] using bespoke components. While we could imagine re-purposing Error-Correcting Code (ECC) DRAM to implement a wider memory today, ECC RAM is not economical for many applications, or else is needed for its original purpose. Furthermore, ECC is not even available in many formats such as the common low-power double data-rate (LPDDR) standard. Today's environment of commodity memory with power-of-two widths pushes us to explore more creative tag storage solutions.

The *in-page* option ensures that any tag access will hit in the page opened for the data access. This limits the worst case, but also limits tag cacheability by distributing the tag table, limiting tag cache line width and preventing the tag cache from exploiting broader spatial locality.

The *dedicated* option ensures that tag bits can always be fetched in parallel with data fetches, logically similar to a wider memory, but can also support caching nearby values to eliminate tag fetches when spatial locality allows. However it is hard to justify a dedicated memory interface and discrete chips for a memory that is a small fraction of the size of primary system memory.

The *table* option fits conveniently into existing memory systems, but its viability depends on the cacheability and compressibility of the tag table to reduce DRAM access overhead. We optimize the *table* design in this paper, as it is a very practical approach – although our results also inform *in-page* and *dedicated* designs that can also exploit caching and compression.

IV. Caching Single-bit Tags

Once we establish SBT storage in memory, we must settle on a caching strategy on chip. There is some confusion and disagreement in the tag literature concerning how tag metadata should be cached. While exploring the many applications of tagged memory, two major strategies have emerged to handle the tag extension to each word of memory: the *split*-cache hierarchy and the *merged*-cache hierarchy (Figure 1).

A. Split-Cache Hierarchy

The split hierarchy approach inserts a level-one tag cache beside the existing instruction and data caches to respond to tag requests [9]–[12], [20], [28]. Many of these systems allow tag lines and data lines to mingle in the L2 cache, naturally competing for space. Furthermore, these split-cache designs generally maintain tags for virtual addresses, possibly influenced by implementations in user-mode instruction-set simulators, maintaining tables for each address space in the address space itself [9]–[11], [19], [20]. The split-cache design is both *problematic* and *unnecessary* for SBT architectures.

The split-cache design is problematic for both the pipeline and for memory. It is problematic for the pipeline because the new tag-memory access and its dependencies must be handled separately from the data memory access. For example, FlexiTaint and MemTracker insert two extra pipeline stages to access the tag cache along with a new register file [11]. FlexiTaint argues that this avoids extending memory access width and physical registers, but extending physical registers by the one or two bits they had proposed is easier than fitting a second memory access into the pipeline and implementing rename and forwarding logic for another register file.

The split-cache design is equally problematic for memory. Consistency between tag and data becomes a problem when the two exist in separate cache lines in the memory hierarchy. This metadata consistency challenge has spawned its own strain of research with no obvious, efficient solution [29], [30]. FlexiTaint also discovered a new false sharing effect due to level-one tag cache lines that each cover a page of memory [11]. A multi-kilobyte granularity for coherence is a high price to pay for tagged memory.

The split-cache design is also unnecessary. As tagged memory logically extends each memory word, separate tag and data cache lookup operations are almost entirely redundant. Unlike DRAM, caches are not commodity logic and can simply be widened. It is reasonable and far more efficient for SBT systems to simply implement slightly wider memories in the entire cache hierarchy, storing tags alongside the metadata already held for coherence.

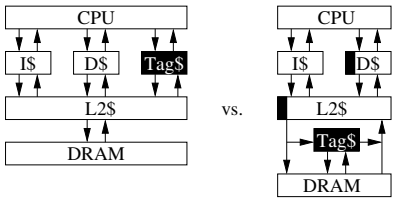

Fig. 1. Split-cache hierarchy (left) and merged-cache hierarchy(right)

Virtual vs. Physical: Split-cache designs have tended to tag virtual addresses rather than physical. Tagged virtual memory requires a tag table for every virtual address space with a complex multi-level structure to avoid storing tags for unmapped space. In contrast, tagging physical memory limits tag metadata to a fixed proportion of the available memory. Physical tags also enable memory clients that do not support translation to correctly handle tagged memory [31]. Physical tags are safer, enable a static tag table, and also enable tags to merge with the physically-indexed cache hierarchy.

While we are critical here of the split-cache design, we note that the primary contribution of the projects referenced in this section were compelling applications for tagged memory rather than their tag-storage approach – with the exception of Range Caches, which focused on tag caching exclusively and had a more considered analysis [10]. These authors argue that widening caches for *large tags* wastes storage and power when tags can be redundant across large regions of memory for many applications, yet for *small tags* they recommend a merged hierarchy and a simple bit-vector table for storage. We agree that a merged-cache hierarchy is clearly the best option for SBT systems, and recommend embedding MBTs if possible, though even some implementations of large external MBTs have chosen a merged-cache design [24].

B. Merged-Cache Hierarchy

Merging tags with the existing CPU cache hierarchy yields a more practical design. A number of research projects have taken the merged-cache approach [1], [3], [4], [6], [13], [16], [17], [21], [25]. Unlike the split-cache collection, five of these research projects include hardware implementations that run operating systems [4], [6], [16], [21], [25], indicating that the merged-cache approach is indeed the more practical approach for actual implementations.

While the merged-cache approach is simple, it is not obvious how to make memory performance fast when every data access requires an additional tag access. The Raksha extension of the Leon processor simply generates extra DRAM accesses for tags on every data access [21]. For example, a data write would generate a single external access for data and both a read and write of the tag table to fold a few bits into a byte of DRAM. CHERI introduces a dedicated tag cache next to the DRAM controller to more efficiently emulate a wider physical memory [6], with the hope that some locality is left to be gleaned at the bottom of the cache hierarchy. The lowRISC open-source system-on-chip project has adopted the CHERI tag cache approach [25], and has inspired the RISC-V HDFI extension that implements both a tag cache and rudimentary tag-table compression [16]. Further development of the HDFI design would particularly benefit from our characterization of tag-table traffic. The Rocket processor used in the HDFI work has only a 16KiB L1 data cache, and does not have an L2 cache. The tag cache is only 1KiB and contains only 16 lines in total. While their caching performance study

shows overwhelming DRAM access overheads due to basic tagging (37.9%-373%), the limitations of their design demand a more thorough analysis. We provide this analysis and include an improved implementation of compression to demonstrate that overheads can be much lower than the published HDFI approach and to give the root cause for these overheads. With work progressing toward practical implementations, it is important that we understand the caching properties of tag tables in general, and SBT tables in particular, to inform future tagged architectures that may be adopted commercially.

V. Tag-Table Cache Characterization

We maintain tag bits in a simple table in DRAM with one bit per word in memory. If these table accesses are cached, we can take advantage of both temporal and spatial locality of table accesses. The idea of a table cache might make us wary, as the astute reader will realize that the table cache behaves exactly like a last-level cache, seeing only memory accesses that have missed in the upper levels of cache hierarchy, and that last-level caches generally have poor hit rates. A 2MB L3 cache behind a 256KB L2 cache was found to have an average miss rate of 48% in SPEC2006, for example [32]. Such a miss rate would be disastrous for our tag-table cache as this would imply at least a 48% increase in DRAM transactions and potentially incur a similar penalty in average latency.

A. Dynamic Tag-Table Cacheability

To understand dynamic SBT table cache behavior, we have designed an experiment to model a tagged memory using DRAM traces from unmodified applications. Any trace of DRAM data accesses implies a hypothetical trace of shadowspace table accesses to provide the tags for each data line. For a given word width, this caching behavior is identical for any SBT scheme as all schemes simply emulate a memory that is wider by one bit.

We instrumented gem5 to trace all accesses to DRAM and replayed these traces against a set of simulated cache structures with varying parameters. Here we tag 64-bit words, giving us a tag cache *amplification factor* of 64. For example, a 1KiB tag cache will cover 64KiB of data memory and should behave like a data cache of that size. We have selected two ARMv8 systems for this analysis. The *small* system has a single core with a 256KiB L2 last-level data cache, and the *big* system has four cores with an 8MiB L3 last-level data cache with prefetching enabled. For this analysis we select two benchmark loads. Earley-Boyer of the Octane suite under Google V8 was chosen because we have found that Javascript has very unfavorable pointer distributions (which is significant when compression is applied) and because the Earley-Boyer test is particularly unfavorable among the tests of the Octane suite, as demonstrated in Figure 5. Each instance of Earley-Boyer under Google V8 uses approximately 60MiB of memory which is almost entirely a large heap with complex access patterns yielding poor cacheability. In contrast, FFMPEG is a media-centric C program with fewer pointers and more regular memory access patterns and also has a data set of approximately

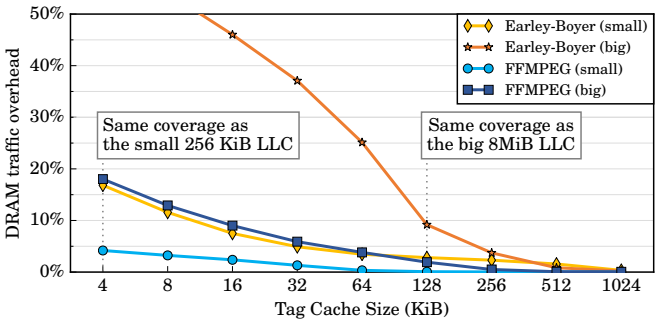

Fig. 2. DRAM traffic overhead vs. tag cache size for *big* and *little* memory configurations

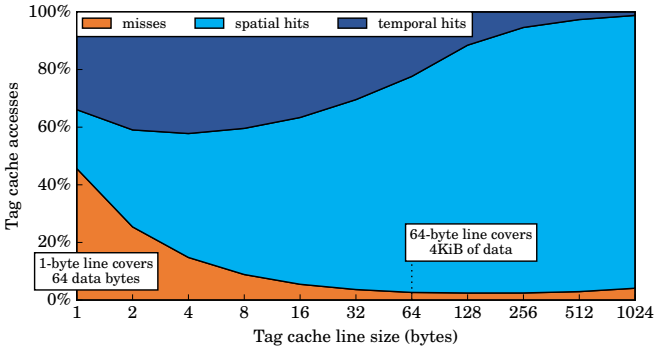

Fig. 3. Temporal and spatial hits vs. line size for Earley-Boyer *big* 256KiB cache, 8-way associative

60MiB. We run three instances of Earley-Boyer in the four-core case to ensure all cores are generating traffic, expecting the V8 engine to do book-keeping on the fourth core, but maintain a single instance of FFMPEG as it is multithreaded. The three independent instances of Earley-Boyer particularly allow us to stress caches, disrupting locality that might be gleaned by the tag cache. Nevertheless, we discover that a tag cache can achieve much higher hit rates than a typical last-level cache.

Figure 2 shows DRAM transaction overhead versus cache size, using a standard 64-byte line size in the tag cache. All of our simulations used 8-way set-associative, physically-indexed tag caches. We note that for even modestly sized caches we achieve a surprisingly low transaction overhead. For example, a tag cache size of 128KiB covers the same amount of data as the 8MiB L3 cache that it serves and yet manages to hit over 90% of the time, inflating DRAM accesses by less than 10% for both fills and writebacks.

If we choose 5% as a threshold for acceptable DRAM traffic overhead, we need a 32KiB tag cache for the *small* system to serve the 256KiB last-level data cache, and a 256KiB tag cache for the *big* system to serve the 8MiB last-level data cache. Note that the tag cache for the *big* system only covers twice as much data as its last-level data cache. At these sizes, with 64-byte lines, Earley-Boyer and FFMPEG have 4.91% and 1.31% overheads respectively for the small system, and 3.72% and 0.50% for the big system.

To uncover the reason for these unusually high hit rates,

Figure 3 graphs the temporal and spatial hits in the tag cache as the line size grows for the Earley-Boyer *big* case (256KiB tag cache, 8-way associative). Spatial hits are on tags that have not previously been accessed in the cache, i.e. that have been brought in due to a miss on a nearby tag. Temporal hits are on tags that have previously been accessed and are re-accessed due to lack of capacity in the upper layer of cache. The graph begins with a tag cache line that covers one data line. As the line size increases, spatial hits continue to increase consistently until we reach lines of 512 tags (64 bytes) which each cover a 4KiB page of data. Bigger lines benefit spatial hits more then they harm temporal hits until lines of approximately 4096 tags (512 bytes) which each cover 8 pages of data memory. After that point, no more spatial locality seems to be harvested from larger lines, but the number of temporal hits still decreases, harming overall hit-rate. Thus the tag cache can exploit spatial locality at page granularities to reduce overhead from an expected 50% of DRAM traffic to less than 5%, even for an unusually small capacity.

Silent-Write Elimination: Writes that rewrite the existing value, or *silent writes*, are more common for tags than for data and are more problematic. Silent tag writes are common since tag metadata is often unchanged through data writes, e.g., when updating untagged data. Tag lines are also much more likely to be dirty than data lines, as the coarse line granularity increases the probability that some bit will be written. Our simulated tag cache eliminates these silent writes. This optimization reduces dirty lines from 80% to 4% in the pointer-sparse FFMPEG case, and from 60% to around 30% for the pointer-heavy Earley-Boyer case. This feature makes writeback traffic dependent on the value of the tags. Figure 8 includes several use cases, one of which sees a 30% reduction of traffic overhead without compression due to tags changing less frequently.

B. Hardware Implementation

We rebuilt the tag controller engine in the open-source CHERI processor (http://www.bericpu.org/), and added performance counters to the CHERI cache. CHERI is instantiated with 32KiB L1 caches and a 256KiB L2 cache, all 4-way set associative with 128-byte lines. CHERI requires a tag bit for each 256-bit word, resulting in a natural caching amplification factor of 256. Our new tag controller includes a lookup engine backed by a 32KiB 4-way set-associative cache with 128-byte lines, matching the burst size in the CHERI system. Since each cached tag bit covers 256 bits of data memory, each 128-byte line in the tag-table cache provides tags for 32 kilobytes of data memory. We restricted ourselves to a standard cache instantiation for the tag controller which did not allow silent-write elimination so this feature was not evaluated in hardware.

Benchmark results for this basic FPGA implementation are shown as the *Uncompressed* case in Figure 9. All of our benchmarks were compiled to use 256-bit CHERI capabilities for all pointers, though the tag values do not affect hit rates for the uncompressed case. Our benchmarks include a selection of Octane benchmarks running under the Duktape interpreter and of MiBench benchmarks running natively. DRAM overhead was below 3% for programs with data sets contained in the multi-megabyte reach of the tag cache. The Splay benchmark with a working set of over 100MB still maintained an overhead of less than 8%.

VI. Tag-Table Cache Compression

Tag-table compression reduces cache footprint by taking advantage of likely patterns in adjacent tag-bit values. Our focus is on compression for caching rather than reducing the size of the table in memory, as the table itself occupies a very small proportion of DRAM, and the full capacity is required in the worst case. As compressibility depends heavily on probable distributions, we must select a tag use case to gain concrete insights into compressibility.

Three prominent approaches have been taken for tag compression. The *Range Cache* approach compressed arbitrary ranges of tags with the same value, and was particularly useful for large MTB systems [10]. The *Multi-granularity* tagging approach indicates the presence or absence of tags using the TLB, eliminating tag lookup for the majority of cases. Most of these systems keep tags on virtual memory such that tag storage is entirely under software control [9], [15], [33]. Our approach is a fully hardware-managed[1] *hierarchical tag table* in physical memory that performs compression while emulating a flat tag space.

A. Hierarchical Tag Table

To optimize for regions that contain no tags, we may implement a two-level table where a bit in the root level indicates whether any bits are set in a group of leaf level bits. In the example in Figure 4, one bit in the root level can be cleared to indicate that 512 bits in the leaf level are all zero and need not be accessed on a read or on a write of zero. We refer to the group granularity as the *grouping factor* "GF", as this is the factor by which the tag footprint can be compressed for groups with no pointers. All tag-table lookups must access the root level, but only addresses that lie in a group including a tagged word must access the leaf level. It is simple to maintain such a hierarchy. Each time we clear a tag bit in the leaf level, we must check whether the rest of the tags in the group are zero, clearing the bit in the root level if this is the case. On boot up, we must clear only the root level of the table to clear the tag bits on all of memory.

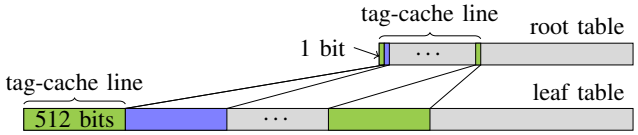

Fig. 4. Hierarchical table structure for grouping factor of 512

Crucially, this scheme eliminates table-cache pressure for applications that do not use tagged pointers. In addition, this

[1]WHISK demonstrates that the root level of a two-level tag table can be managed in software at the cost of flushing tag caches on root updates [31].

scheme can greatly reduce the cache footprint of sparse tags. While we found that a two-level table performs well, our simulator and FPGA implementations are parameterized for tables with arbitrary levels and grouping factors at each level to enable a thorough investigation.

Empty Line Fabrication: Our simulated tag controller can also fabricate lines in the tag cache when a leaf group becomes non-zero for the first time, and destroy it without writing back when the group becomes fully zero, avoiding accesses to the backing memory. This optimization requires that eliminated groups correspond precisely to tag cache lines, further enforcing the *line-group* principle we will present in Section VI-C. While this optimization reduces cache fills by 0.5-75% for our benchmarks, depending on pointer density, our FPGA tag controller implementation instantiates a standard cache which was not able to implement this behavior.

B. Static Memory Analysis

Tiwari *et al.* observed strong patterns for MBTs and developed the range cache in response [10]; we find that SBTs also allow efficient compression of contiguous regions with a hierarchical structure. To study typical pointer distributions, we core-dump programs (running on x86-64 under FreeBSD) near peak memory usage to provide a static memory map. Our core-dump analysis tool identifies likely pointers by matching 64-bit words against a list of all allocated virtual addresses for the process, thereby deriving an upper limit on the number and distribution of pointers to be tagged. False positives should be rare, as it is unlikely that non-pointer values will match a valid 64-bit address. To assess compressibility of a program's tag table, we analyzed what percentage of tag groups have at least one pointer at various group granularities. As shown on Figure 5, we discovered that, for many C-language applications, fewer than 10% of the groups contained pointers, even for large groups (large GF). For these applications, more than 90% of the tag data could be eliminated from the working set, and the reach of a tag cache would be amplified by a factor of more than 10 above the natural amplification factor of 64 (or 256 for CHERI with a tag for every 256-bits) . For applications in higher-level languages, such as Javascript and C++, we saw a much higher concentration of pointers, with some approaching as much as 25%, with over 75% of the groups containing pointers for very large GFs. Nevertheless, even in pointer-rich applications, we find that pointers tend to cluster together, i.e., the grouped pointer densities are lower than for a random distribution. Figure 6 allows us to compare with a 1% random pointer distribution which surpasses all the application samples at large GFs.

C. Dynamic Hierarchical Cache Study

We return to our gem5 trace engine to study dynamic caching behavior of our compressed tables. We compared each DRAM data write against a list of valid virtual addresses compiled from a trace of TLB misses to identify pointers in DRAM traffic. This pattern of virtual addresses

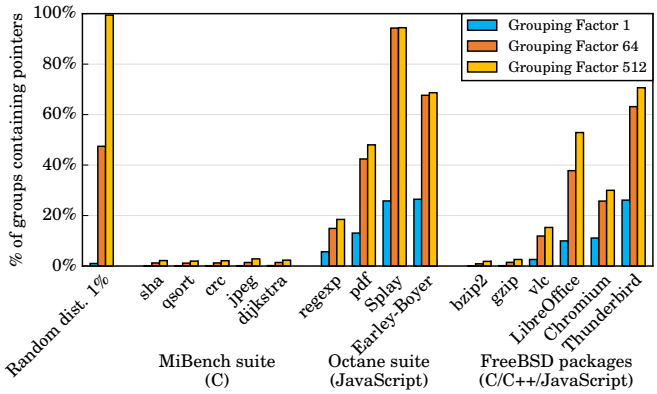

Fig. 5. Pointer concentration for GFs 1, 64 and 512

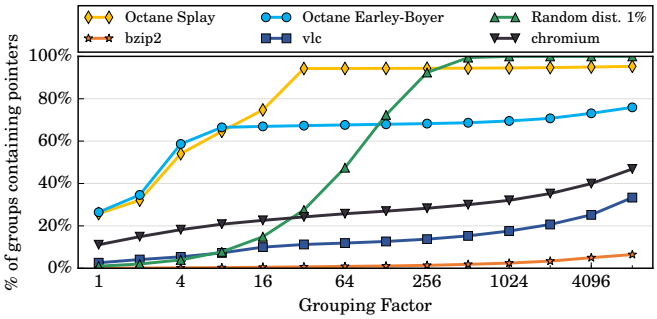

Fig. 6. Measured pointer densities vs. a 1% random distribution

written to memory constitutes a trace of hypothetical tag-table traffic that slightly overestimates pointer density due to false pointer matches. By replaying this trace of tag-table traffic against a hierarchical table engine backed by a standard cache, we are able to simulate dynamic cache behavior for a hierarchical table structure. Figure 7 reports DRAM traffic overhead without compression and for a hierarchical table with a GF of 512, according to the *line-group* principle from Section VI-C. For this study we include our primary example of tagging all pointers, and introduce two additional tag use cases: *code pointers*, and *zeroes*. The *Code pointers* case models control-flow integrity systems that tag only code pointers [3], [4], and the *zeroes* case tags zeroed cache lines to avoid accessing DRAM when a cache line is entirely zero, improving performance and power.

The results in Figure 7 show that data-centric programs such as FFMPEG see pointer-tag misses collapse to very near zero when using compression, while pointer-heavy programs such as Earley-Boyer running under Google's V8 engine see more measured improvements proportional to those predicted by the static study. When tagging only code pointers, the hierarchical compression successfully eliminates additional leaf-level groups compared to the all-pointer case, with overhead collapsing in all but the Earley-Boyer *big* case.

The *zeroes* case has the interesting implication that such a system could cause a net reduction in DRAM traffic. Identifying and eliminating all loads and stores of zeroed lines to DRAM costs less than 0.2% DRAM traffic overhead for these cases, and zeroed lines comprised between 1.5% and

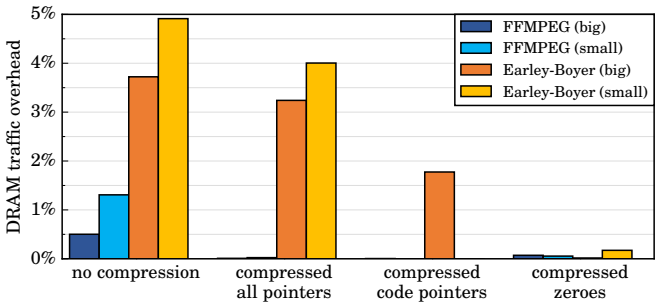

Fig. 7. Traffic Overhead with Table Compression Across Tag Use-cases

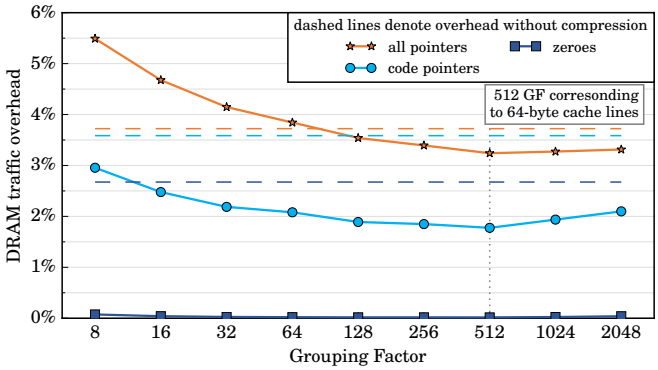

Fig. 8. Earley-Boyer *big* DRAM Traffic Overhead vs. Grouping Factor

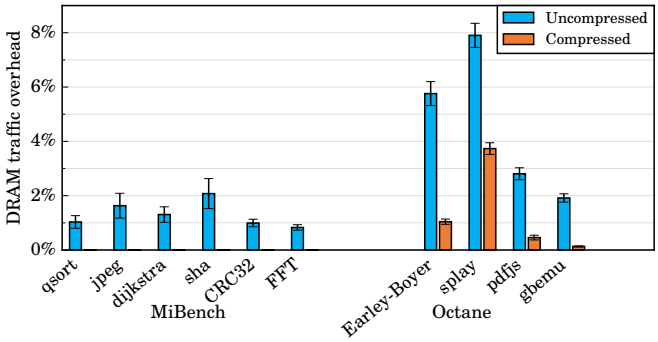

Fig. 9. DRAM Traffic Overhead in FPGA Implementation
Note: MiBench overheads with compression are approximately zero

2.5% of DRAM traffic.

Line-group Principle: The optimal grouping factor (GF) will almost always be equal to the cache line size. We may call this the *line-group* principle. Figure 8 shows that traffic overhead for our most demanding benchmark, Earley-Boyer *big*, decreases as the grouping factor approaches the line size, 64 bytes (512 bits). After this point, the overhead sometimes increases. This is due to a relationship between the compression granularity and the caching granularity. The hierarchical mechanism reduces the size of the tag working set by eliminating groups of tags. However, the cache is filled on a line granularity and no group smaller than a line can be usefully omitted from the cache. Therefore, we cannot take practical advantage of grouping factors that are smaller than the line size of the cache. For example, an analysis of pointer density may indicate that a GF of 8 will yield a smaller tag working set than a GF of 512, but the cache will still fetch on a line granularity and the "eliminated" groups will still consume cache space if they share a line with an active group.

Nevertheless we may desire a finer granularity for the grouping factor to simplify table maintenance. For example, in our FPGA implementation we have chosen a GF equal to the word size of our cache, 256 bits, rather than the line size (1024 bits). This allows us to check if a leaf group is zero in a single cycle when a tag is cleared, so that we can clear the bit for that group in the higher level.

D. Performance Results from Hardware Implementation

We extended the tag-table manager for the CHERI open-source prototype with a parameterizable hierarchical lookup engine. We used a two-level table with a GF of 256. We compiled our benchmarks to use tagged capabilities for all pointers, providing a real tag working set.

For the MiBench suite in C, the cacheability of the table is amplified to the extent that the working set fits easily into the table cache, and overheads entirely collapse. The pointer densities of the MiBench benchmarks are well below 5% even with large GFs according to Figure 5, giving us a cache amplification factor of over 20. Our tag cache of 32KiB could therefore have an amplified capacity of 640KiB for tags. Such a cache would cover 160MiB of data memory due to the natural amplification factor of 256 for CHERI. For the Javascript applications that do not collapse due to both pointer density and working set size, the miss rate is at least halved. We emphasize that we selected Javascript as a difficult case for pointer compression, and Splay and Earley-Boyer, because they extensively exercise the pointer-rich Javascript heap. Gbemu and pdfjs are typical of the rest of the Octane suite and have better tag caching properties.

The hierarchical instantiation of the tag controller on FPGA consumed 15.4% more logic elements than the flat instantiation, averaged across 10 builds. Memory resources were identical since the tag cache size remained constant. The overhead consists of comparison logic and buffering registers to maintain the hierarchical structure. FPGAs have peculiar logic consumption patterns compared to ASIC so this overhead should not be considered representative of hierarchical tag caches in general. A 32KiB cache in a 14nm ASIC would be on the order of $0.1mm^2$ [34]; a small fraction of a modern memory controller.

VII. Conclusion

Tagged memories are commercially viable for mass-market implementations. Tagged architectures have long held promise for enforcement of strong security invariants, but their adoption has been hindered by a lack of understanding of their affect on memory subsystems. We have characterized the straightforward design of a single-bit tag table in physical memory cached at the memory controller, measuring a less than 8% DRAM traffic overhead on an FPGA implementation despite pessimistic intuitions due to poor performance of last-level caches. Using simulations, we demonstrated that this

surprising performance is due to spatial locality on a page scale that is enabled by cache lines that hold tags for kilobytes of data.

To push overheads down further, we exploited patterns in tables of single-bit tags that make them amenable to compression. This compression allowed us to avoid table accesses for regions that do not use tags, or that use tags only briefly, to bring the common case well below 1% overhead. Crucially, applications that do not use tags see almost zero DRAM traffic overhead. With such low overheads demonstrated for single-bit shadowspace implementations, researchers should be encouraged to further develop applications relying on small tags and industry should be encouraged to attack security challenges directly by enforcing non-bypassable security properties enabled by tagged memory.

REFERENCES

[1] A. J. Mayer, "The architecture of the Burroughs B5000: 20 years later and still ahead of the times?" *ACM SIGARCH Computer Architecture News*, vol. 10, no. 4, pp. 3–10, 1982.

[2] B. E. Clark and M. J. Corrigan, "Application System/400 performance characteristics," *IBM Systems Journal*, vol. 28, no. 3, pp. 407–423, 1989.

[3] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *37th International Symposium on Microarchitecture (MICRO-37)*. IEEE, 2004, pp. 221–232.

[4] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer, "Defeating memory corruption attacks via pointer taintedness detection," in *Proceedings. International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2005, pp. 378–387.

[5] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," in *ACM SIGPLAN Notices*, vol. 29, no. 11, 1994, pp. 319–327.

[6] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 457–468.

[7] J. Brown, J. Grossman, A. Huang, and T. F. Knight Jr, "A capability representation with embedded address and nearly-exact object bounds," Project Aries Technical Memo 5, Tech. Rep., 2000.

[8] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr, and A. DeHon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proceedings of the ACM SIGSAC conference on Computer & communications security*, 2013, pp. 721–732.

[9] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *ACM Sigplan Notices*, vol. 39, no. 11, 2004, pp. 85–96.

[10] M. Tiwari, B. Agrawal, S. Mysore, J. Valamehr, and T. Sherwood, "A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 94–105.

[11] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, 2008, pp. 173–184.

[12] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin, "A case for unlimited watchpoints," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, 2012, pp. 159–172.

[13] P. Zhou, F. Uin, W. Liu, Y. Zhou, and J. Torrellas, "iwatcher: simple, general architectural support for software debugging," *IEEE Micro*, vol. 24, no. 6, pp. 50–56, Nov 2004.

[14] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory." in *OSDI*, vol. 8, 2008, pp. 225–240.

[15] R. Shioya, K. Daewung, K. Horio, M. Goshima, and S. Sakai, "Low-overhead architecture for security tag," *IEICE TRANSACTIONS on Information and Systems*, vol. 94, no. 1, pp. 69–78, 2011.

[16] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-Assisted Data-flow Isolation," in *2016 IEEE Symposium on Security and Privacy (SOSP)*, 2016, pp. 1–17.

[17] K. Piromsopa and R. J. Enbody, "Secure bit: Transparent, hardware buffer-overflow protection," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 4, pp. 365–376, Oct 2006.

[18] E. Witchel, J. Cates, and K. Asanović, *Mondrian memory protection*. ACM, 2002, vol. 30, no. 5.

[19] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language," in *ACM SIGARCH Computer Architecture News*, vol. 36, no. 1, 2008, pp. 103–114.

[20] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "Memtracker: Efficient and programmable support for memory access monitoring and debugging," in *IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007, pp. 273–284.

[21] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, 2007, pp. 482–493.

[22] H. Shrobe, A. DeHon, and T. Knight, "Trust-management, intrusion-tolerance, accountability, and reconstitution architecture (TIARA)," DTIC Document, Tech. Rep., 2009.

[23] S. Chiricescu, A. DeHon, D. Demange, S. Iyer, A. Kliger, G. Morrisett, B. C. Pierce, H. Reubenstein, J. M. Smith, G. T. Sullivan *et al.*, "Safe: A clean-slate architecture for secure systems," in *IEEE International Conference on Technologies for Homeland Security (HST)*, 2013, pp. 570–576.

[24] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight Jr, B. C. Pierce, and A. DeHon, "Pump: a programmable unit for metadata processing," in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2014, p. 8.

[25] W. Song, A. Bradbury, and R. Mullins, "Towards general purpose tagged memory," June 2015, 2nd RISC-V Workshop.

[26] D. Y. Deng, *Flexible and efficient accelerator architecture for runtime monitoring*. Cornell University, 2016.

[27] C. W. Otterstad, "A brief evaluation of Intel® MPX," in *9th Annual IEEE International Systems Conference (SysCon)*, 2015, pp. 1–7.

[28] H. Shrobe, T. Knight, and A. d. Hon, "TIARA: Trust Management, Intrusion-tolerance, Accountability, and Reconstitution Architecture," 2007.

[29] H. Kannan, "Ordering decoupled metadata accesses in multiprocessors," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 381–390.

[30] B. Rajaram, V. Nagarajan, A. J. McPherson, and M. Cintra, "Supercop: a general, correct, and performance-efficient supervised memory system," in *Proceedings of the 9th conference on Computing Frontiers*. ACM, 2012, pp. 85–94.

[31] J. Porquet and S. Sethumadhavan, "WHISK: An uncore architecture for dynamic information flow tracking in heterogeneous embedded SoCs," in *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2013, p. 4.

[32] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," *Web Copy: http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf*, 2010.

[33] V. Karakostas, S. Tomic, O. Unsal, M. Nemirovsky, and A. Cristal, "Improving the energy efficiency of hardware-assisted watchpoint systems," in *50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, May 2013, pp. 1–6.

[34] T. Song, W. Rim, J. Jung, G. Yang, J. Park, S. Park, Y. Kim, K.-H. Baek, S. Baek, S.-K. Oh *et al.*, "A 14 nm FinFET 128 Mb SRAM With Vmin Enhancement Techniques for Low-Power Applications," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 1, pp. 158–169, 2015.