A Small Cache of Large Ranges: Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags

Mohit Tiwari Banit Agrawal Shashidhar Mysore Jonathan Valamehr* Timothy Sherwood
Department of Computer Science, University of California, Santa Barbara
{*tiwari, banit, shashimc, sherwood*}*@cs.ucsb.edu*
* *Department of Electrical and Computer Engineering, University of California, Santa Barbara*
jkv@ece.ucsb.edu

Abstract

Dynamically tracking the flow of data within a microprocessor creates many new opportunities to detect and track malicious or erroneous behavior, but these schemes all rely on the ability to associate tags with all of virtual or physical memory. If one wishes to store large 32-bit tags, multiple tags per data element, or tags at the granularity of bytes rather than words, then directly storing one tag on chip to cover one byte or word (in a cache or otherwise) can be an expensive proposition. We show that dataflow tags in fact naturally exhibit a very high degree of spatial-value locality, an observation we can exploit by storing metadata on ranges of addresses (which cover a non-aligned contiguous span of memory) rather than on individual elements. In fact, a small 128 entry on-chip range cache (with area equivalent to 4KB of SRAM) hits more than 98% of the time on average. The key to this approach is our proposed method by which ranges of tags are kept in cache in an optimally RLE-compressed form, queried at high speed, swapped in and out with secondary memory storage, and (most important for dataflow tracking) rapidly stitched together into the largest possible ranges as new tags are written on every store, all the while correctly handling the cases of unaligned and overlapping ranges. We examine the effectiveness of this approach by simulating its use in definedness tracking (covering both the stack and the heap), in tracking network-derived dataflow through a multi-language web application, and through a synthesizable prototype implementation.

1. Introduction

The ability to tag and track data as it pumps through high performance microprocessors opens the door to a host of new dynamic analyses. Recent research has shown that dataflow tracking enables run-time checks for malicious code-injection [5,17], that it can help uncover cross-site scripting attacks [7,12], that it makes privacy easier to manage [18], that memory errors can be more effectively uncovered and tracked [20,16], and that full systems can be sliced and examined in novel ways [11]. All of these techniques rely on ISA-level extensions – shadowing all architecturally visible state with tags; creating dataflow rules that ensure tags are effectively tracked during execution; and forming policies around these tags that solve problems that face modern software developers.

Most dataflow tracking schemes require that all architecturally visible storage locations (including the registers, cache, and main memory) are extended with *tag bits*. These bits store a form of metadata, for example they can be used to indicate whether a particular piece of data is "trusted" or "untrusted". Bits are initialized and propagated according to some specific policy created and enforced to solve a specific problem. For example a simple code-injection prevention policy might be (1) mark all data from the network as untrusted, (2) ensure that all operations using one or more untrusted operands have their produced data marked as untrusted, and (3) prevent jumps to code in untrusted memory. Of course this is an over simplification, real policies have to deal with many exceptions to these rules, and there are many past works that have explored how tag tracking can be efficiently integrated into various stages of the pipeline to meet various security or analysis goals [2,5,11,12,17,18,19,20,21]. In this paper we concentrate specifically on the problem of *how large multi-bit tags can be stored and queried efficiently* and we present an approach that could serve as a drop-in replacement for more straightforward, but less space efficient, tag storage schemes.

Specifically, the goals of the system we present here are to support: **fine-grain tracking** to allow tags to be tracked not just with each word, but even each byte; **flexible tracking granularities** to allow dataflow tags to be tracked per-byte or per-word as required by the systems; and **efficient on-chip storage of large multi-bit tags** to enable the next generation of dynamic analysis techniques that require 32-bit tags or larger. The key idea behind our approach is to store tags not with individual addresses, but for *ranges* of address. We show that tag bits naturally demonstrate a very high degree of spatial-value locality, and that by storing and operating on the tags directly as ranges we can very efficiently store attributes over arbitrary regions of memory (in fact we show in Section 5 that keeping even 4 ranges provides better coverage than a cache that can store 1000 individual elements). Once we have a way of querying these ranges to see which addresses map to them, we can then associate arbitrarily "big" tags on each of these ranges, effectively compressing the tag storage to a very large degree. The challenge is that those ranges need to be rapidly stitched together on the fly because addresses that share a common tag are *created piecemeal* through a series of writes (as stores write those tags to arbitrary locations) rather than as a bulk allocation (such as malloc). This means that the underlying ranges could potentially grow, shrink, split, or merge with any store operation.

In this paper we show that many dataflow tracking applications exhibit very significant **range locality**, where long blocks of memory addresses all store the same tag value, a fact we discuss and explore in Section 2. This range locality is helpful even in schemes that mark a few individual memory addresses because even then the *gaps* between these addresses can be stored as a single range. Furthermore, we show that a **range cache** (a cache where arbitrary ranges of tags are dynamically swapped in and out as requested), can effectively exploit this locality.

Implementing the range cache required a novel method for storing and querying non-power-of-two aligned memory ranges so that it works will work with unconstrained reads and updates. Using the techniques described in Section 4, it is possible to keep the resulting set of stored ranges optimally minimized at all times regardless of how range updates overlap (overlaps occur when a range to be inserted is not disjoint with the set of existing ranges). While complex overlaps between ranges are possible, we show that they occur quite infrequently and need not slow down the operation of the most frequently occurring range queries. The resulting system is effective and of relatively low complexity as evidenced by our experiments over a variety of workloads (from user level SPEC and JDK executions, to a full OS-level server running a Ruby-on-Rails application) and though our analysis of a fully synthesizable hardware implementation.

2. Background and Requirements

2.1. Currently Proposed DIFT Architectures

DIFT [17], Minos [5], Taintcheck [12], Raksha [7], LIFT [13] and a host of other proposals use dataflow tracking to track the flow of untrusted network, file and user inputs through memory. These tools assign a 1-bit tag with every word of physical memory to indicate whether the word stores "untrusted" data; whenever an untrusted memory word is used for a sensitive operation like a jump address condition or a system call, the tool generates a warning for the user. Analogous to these tools, Memtracker [20] uses a tag bit with every word in virtual memory to detect memory errors, where an uninitialized address is used dangerously. The wide applicability of this technique has given rise to generic taint tracking frameworks that allow configurable tag initiation and tracking ability. Dytan [4], GIFT [9], Taint-Enhanced Policy Enforcement [23], Raksha [7] and Flexitaint [19] are some examples of such schemes. While some rely on source code or compiler support to insert dataflow tracking logic into programs [23], others work at the program binary level [12,13]. Some binary level schemes even propose hardware support to lower the performance impact of dataflow tracking [5,7,18,19,20]. Operating at the binary level allows these schemes to provide an end-to-end system that can be useful in production environments, is compatible with existing code and can work with dynamically loaded third-party libraries and robust dataflow tracking tools have been demonstrated to detect both zero-day exploits and memory corruption errors. In this paper, we address an emerging concern for such hardware-assisted binary-level dataflow tracking tools.

Logically, these tools store the memory tags in a tag-table and propagate them through the table according to tool specified rules. In actual implementations, this tag storage functionality can be broken down into three components, each corresponding to a level in the memory hierarchy. The first component interacts with the processor pipeline and includes tag-bit extensions for all registers and extensions to the processor pipeline with logic (either integrated or as an in-order pre-commit stage) to check and propagate these tags [7,19]. To enable flexible propagation rules, Memtracker [20], FlexiTaint [19] and Raksha [7] introduced the notion of a programmable table to store the tag propagation rules, and a low overhead software handler to be called in case a tag-usage policy is violated. The second component is a software data structure that stores the tags for the entire address space. In case of 1 or 2 bit tags, this is usually a bitmap (i.e a packed array), while for tools that require 32-bit tags, this structure can be a forward-mapped page table (Mondriaan Memory Protection[22], Origin Tracking[15]) to reduce space requirement. This component usually resides in a protected part of the program's virtual memory itself. The third, and the performance critical component is an *on-chip tag cache*, where tags are kept either with the words in the cache or in a separate but analogous hardware component. This is then used to cache the tags for recently used addresses, and has been shown to work with very low miss rates and performance overheads given a 1, 2 or even 4 bit tag per word in certain scenarios like file input tracking [19,20]. The software data structure (the second component) then acts as secondary storage for tags. In [19] and [20], this tag cache stores cache lines containing chunks of packed arrays and derives great advantage from being able to pack tags for a larger set of addresses than an equal sized data cache line. Consequently, a tag cache can have very low missrates, and combined with a bitmapped secondary store, a 4KB tag cache has been shown to have a performance hit of only 1.8% [19] for a file input tracking tool running SPEC benchmarks.

While one and two bit tags are no doubt useful, many analysis methods are enabled by the use of both larger 32bit tags and finer granularity tracking (at the byte level).

2.2. Analysis Techniques using Large Tags

Depending on the granularity of tracking, a tag could be required for every word or even every byte of memory in the system (at all levels of the memory hierarchy). Most prior schemes are hardwired to support a single level of granularity because tags are kept uncompressed in a structure very similar to a standard cache. If only one bit of tag is required per word in the system a simple one-to-one storage method (where each word has its own instance of a tag) may be an acceptable expense. However, while past hardware techniques have considered primarily binary tags (e.g private and not-private, or trusted and untrusted), many security policies make a finer distinction between levels (e.g. unclassified, classified, secret, and top-secret or adversary, neutral, ally, and fully-privileged), and many dataflow analysis techniques make use of larger tags to store relevant semantic information (PCs, address, stack pointers, or bitmaps). Often, the 1 or 2 bit tag scheme is good enough to "detect" an anomaly, but it may not be sufficient to reveal information about the source of the anomaly. For instance, recent dynamic taint tracking tools [6,3,21,24] work by assigning all sensitive or untrusted information in the system

a *unique* tag, where each tag is a 32-bit pointer to a structure that stores the execution context when the untrusted information had entered the system. These tags are assigned to all memory locations that contain the untrusted information and are propagated dynamically. Such schemes are then able to precisely identify the sensitive information that is being leaked or expose the untrusted input that has led to a security policy violation.

A similar scheme has also been proposed to track null pointer exceptions at a memory location back to the program counter that stored the null value [2]. Whenever an instruction stores a null value to a memory location, the tool records the program counter as a 32-bit tag. Such a tool is considerably more powerful than Valgrind-Memcheck [16] or Purify [14] because it indicates the source of a null pointer exception accurately even when the exception manifests at a much later point in program execution. It is interesting to note that origin tracking tool suffers from considerable loss in accuracy if the tag value is less than 32 bits. The original tool was implemented with the tag size being equal to the data it was tracking, and suffered upto 78% loss in accuracy by encoding a PC in 8 or 16 bits, but with a 32 bit tag it has been shown to work with very high precision [15].

In addition to security and debugging, 32 bit tags have been used for understanding complex software systems involving multiple runtime environments distributed across a network. This technique, termed full system tomography [11], attributes a *unique tag to each byte* of an incoming network packet and tracks these tags across language runtimes and operating system boundaries and over the network through the entire distributed system. This has been used to extract and visualize dynamic interactions between various components of a full featured web application. For instance, tomography can be used to find out all the pieces of software that "touch" some information, to extract application level semantics at the hardware level and even to identify data transfer points across functions, processes and machines [11].

2.3. The Gap Between Current Architectures and Applications

The Need for Fine-Grain Tag Tracking: In addition to using 32bit tags, all the tools mentioned in the previous subsection use tags at *byte* granularity. The dynamic taint tracking tools track where every byte of sensitive information flows, the origin tracking tool tracks memory errors at byte granularity and the tomography tool assigns a unique tag to every incoming network byte. Performing this at word level would result in a trade-off between conservative propagation of tags (with the resultant false positives) and a loss in tracking accuracy. In their "Minos" work for instance, Crandall et al [5] found that Sun Java SDK uses 8 and 16 bit immediates to generate control data. Consequently, a word granularity tagging scheme gave a large number of false positives for even a simple Hello World program. They proposed either changing the JIT to use 32-bit immediates or adding a compatibility mode in the dataflow tool that marks all sub-word immediates as "safe". In our tests, we observed a considerable difference in the frequency of sub-word level accesses – though on an average a sub-word access is performed once in every 1904 x86 instructions for SPEC benchmarks, `bzip` and `parser`

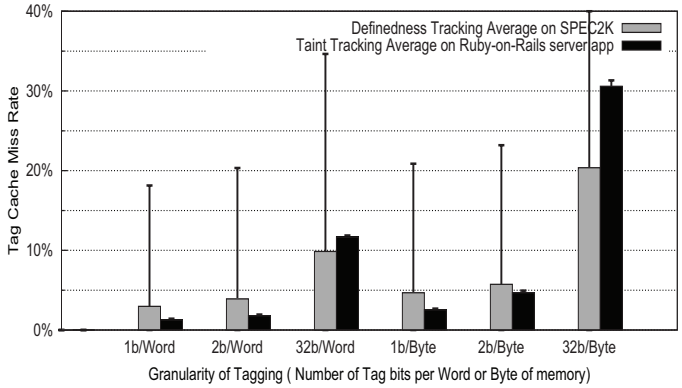


Figure 1: As dataflow tracking tools move from using "1 bit tag per word" to "32 bit tags per byte", the tag cache will incur an increasing number of misses. Each tag cache miss is then serviced by a handler that retrieves tags from memory and adds runtime overhead to the tracked program. We run two dataflow tracking tools, a memory-definedness tracker and a network taint tracker, and show the average (in boxes) and the maximum recorded miss rate (vertical bar) for a benchmark suite. The average miss rate increases by $7X$ for the definedness tracking tool and by $24X$ for the network taint tracking tool.

have a considerably high frequency of sub-word accesses (once in every 8 and 18 x86 instructions, respectively). Some other popular programs (especially those that work on strings) like JDK and Firefox show similar trends (sub-word accesses once in every 10 and 16 x86 instructions, respectively). Our approach, because it no longer maintains a direct one-to-one mapping between individual addresses and tags, can easily assign and track tags at whatever granularity the program exhibits.

Efficient Storage of Large Tags: To enable these increasingly rich dataflow tracking techniques, we need an efficient way of keeping large tags with the data. One way of performing this is to simply scale the size of the tag cache. A 4KB cache at 1-bit per word would obviously have the same miss rate as a 512KB cache operating on 32-bit tags per byte. Another way to view this same problem is to keep the cache size constant (in this case 4KB), and adjust the granularity accordingly. In Section 5, we present two classes of dataflow tracking tools to drive our tests, one tracks network dataflow while the other tracks the set of memory addresses that are "defined" (written before read). Both these tools assign a 1,2 or 32 bit tag with each tagged location (words or bytes). Figure 1 shows, for a tool using 1 bit tag per word, the tag cache has an average miss rate of 1.26% for the tomography tool and 2.97% for the memory error detector and matches up with previous findings [20,19]. As the tag sizes grow from 1 to 32 bits (and hence the number of slots in the cache decreases), the average miss rate increases to 12% and 10% for both the tools. At 32b tags per *byte*, the average miss rate for the tomography tool is 30%, while the memory error tool has a miss rate of 20%. Considering that every tag cache miss has to be serviced by a secondary tag store access (that walks a trie to locate the appropriate tags) this miss rate corresponds directly to a very large performance hit. By storing the tags as a cache of tagged address *ranges*, our approach takes up no more space than a 4KB cache yet can store 32-bit tags per byte with miss rates on the order of 1 or 2%.

2.4. The Opportunities and Challenges of Range-Based Tag Caching

While conventional (bit-mapped) tag caches have the advantage that they are intuitively understandable and reasonably straightforward to implement, they possess neither the degree of flexibility nor the compressed representation that we would prefer to have. However, as pointed out in prior work (such as Mondriaan Memory Protection (MMP) [22] and iWatcher [25]), another option for associating metadata (permission or watchpoints respectively) with the address space is to divide the addresses into a set of distinct ranges, and to associate metadata with the ranges instead.

The main idea from MMP is that address spaces can be partitioned into a series of nested aligned power-of-two sized ranges (MMP uses a multi-bit trie with 10, 10, 6, and 6 bits at levels 1 through 4 respectively). Any arbitrary (aligned or unaligned) range can then be decomposed into an equivalent set of aligned power-of-two ranges and memory attributes can be stored with those ranges instead of on each and every byte in the machine. MMP was originally proposed for defining and enforcing many fine-grain protection domains, going so far as to provide *byte* level granularity. Since the protection domains MMP seeks to enforce tended to be *defined directly* in terms of ranges (for example through "malloc" and "free" or at page boundaries), range-based storage offers dramatic compression over a full bit-mapped representation. Moreover, it achieves good performance by caching power-of-2 aligned sub-ranges and handling reads fast using network processing techniques. Since the trie is updated infrequently (e.g. when new memory ranges are allocated) and searched often, (e.g. by every load and store) the runtime overheads are very low (12% for tracking mallocs/frees).

However, the problem is that dataflow tracking does not conform to this kind of operation because ranges are not *declared* as long ranges, rather they are built up one by one from a series of smaller accesses. Initially, during a function call for example, two word-sized ranges representing the return address and the stack frame pointer are tagged. Later, as the function initializes some of its variables one-by-one (not particularly in order by address), more word-sized ranges become tagged (again one-by-one). An aligned trie-based scheme will have difficulty discovering that tagged stores to the address 9, 11, 8, and 10 could in fact be represented by a contiguous range [8,11]. In addition to wasting space, range fragmentation also leads to many unnecessary operations. If a range changes in size, say from size 16 to size 15, an MMP-style cache will break down an original range $[0, 15]$ into power-of-2 aligned sub-ranges $[0, 7], [8, 11], [12, 13], [14, 14]$. Likewise, if a range of size 15 is extended to size 16, then the prior set of 4 ranges need to be merged into a single range $[0, 15]$. Thus for every update, a trie based scheme has the time cost of determining and inserting the power-of-2 aligned sub-ranges for the new range, and the space cost of storing multiple entries for one unaligned range.

Table 1 attempts to quantify the problem of highly frequent updates in dataflow tracking as compared to the stress test for MMP (updating permissions at mallocs and frees). Dataflow tracking has one update every 4-6 x86 instructions on average, which means trapping to a software handler would be very expensive. In comparison, if we were to update tags at only mallocs and frees, there is one update per 4.8 Million x86 instructions. Our solution to this problem is to create a range cache that does not require a power-of-two alignment so that ranges can be split and merged quickly as needed without worrying about alignment (in fact in the common case we can update a range as required with zero or two cycle stalls).

3. Our Software Test Environment

In an attempt to quantify the conditions driving our design, we need to discuss the applications that we target and the methods used throughout this paper. The first tool we examine monitors the flow of tainted data as it arrives over the network and records a 1-bit "taint" tag for all addresses that store a value that is directly tainted or has been indirectly generated from tainted values [5,7,12] . This taint tracking infrastructure is built within QEMU [1], an x86 virtual machine, running an online bookstore application developed on the Ruby-on-Rails framework, using a Mongrel web server, with features for customers to view the store page, add to shopping cart and checkout books, along with administration features. For this first tool, we use a simple 1-bit tag that indicates whether the address content is tainted or not. The second tool is a memory error detector similar to Valgrind-Memcheck that attributes a 2-bit tag to all memory that is "allocated" and "initialized" by SPEC benchmarks. We also include a Java program that parses XML documents to consider the effect of a workload that is both important and quite different than traditional SPEC programs.

For our third and final tool, used to stress the effectiveness of our range-based design, we use the dataflow tomography tool considered in [11] that attributes a 32-bit tag to each byte of an incoming network packet. This tool is used to correlate usage of individual network data bytes, and so each time an address is attributed a tag, the tag propagation logic creates a new tag that encodes the overall range of tags that the address has stored for the entire execution. This tool is tested with network applications like ssh, scp, lynx and traceroute, in addition to the Ruby-on-Rails based online bookstore application discussed above.

4. Range Management for Tag Tracking

As described in the previous section, tags naturally group into contiguous address ranges, but they do so incrementally through a barrage of stores (as opposed to a single allocation). Table 2 shows that, in fact, the total number of ranges that are created range from a few tens to a few thousands. While the least number of ranges across benchmarks is surprisingly small, the largest number precludes storing *all* the ranges on-chip. This means we still have to use Range Cache as a *cache* for tags and deal with 'misses' on tag reads and evictions when new tag ranges are created. In this section we describe a novel Range Cache architecture that captures the vast majority of dataflow tags accesses while gracefully handling this incremental range aggregation problem.

Storing ranges of tags in a range cache improves the efficiency of dataflow tracking tools to a great extent (as will be shown in Section 5), but not before we address the following challenges -

Program	Instrns/update	Refs/update
bzip	7.43	1.75
crafty	6.97	1.94
gcc	5.03	1.34
gzip	6.38	1.14
mcf	13.84	3.56
parser	5.28	1.49

Program	Instrns/update	Refs/update
vpr	5.38	1.74
ammp	9.69	1.7
applu	5.82	2.09
swim	10.79	3.05
wupwise	6.33	1.43
average	6.54	2.07

Program	Instrns/update	Refs/update
admin login	3.38	1.34
book add	4.43	1.42
checkout	3.41	1.31
store view	6.08	1.29
average	4.33	1.34

Table 1: Frequency of tag updates in dataflow tracking: Updates to the tag store are very frequent, and are problematic for existing trie-based tag stores. A range based tag store will require handling updates with very high throughput.

Storing point values in a cache is simple, but how do we store ranges in a cache (Section 4.1)? With the cache storing arbitrary ranges, what are the ramifications of fetching or evicting a range (Section 4.2)? How do we handle cases where ranges overlap (Sections 4.3, 4.4)? This section discusses how the Range Cache architecture tackles each of the above questions.

4.1. Storing Arbitrary Ranges

The Range Cache stores a set of ranges (on the order of 128 ranges, each having a $start$ and end address), each of which has an associated metadata (which in our prototype is 32-bits). Requests, either reads or updates, are handled by the Range Cache in a very similar manner to a normal cache, except that each and every request is a range and has both a $start$ (called $newstart$) and end (called $newend$) of its own. All of these ranges (both stored ranges and requests) can be of arbitrary size and of arbitrary alignment. A *read request* should return the tags of the ranges it overlaps. The most common case (quantified later in Section 5.0.1) is that reads only overlap a single stored range, but more complicated overlaps are certainly allowed and their handling is discussed more below. An *update request* sets the new tag value to be now associated with new range of addresses. The tag values previously associated with those address are overwritten (i.e. each individual address has one and only one tag associated with it). This new range can overlap with existing ranges in complex ways that may require the existing ranges to be split up (again discussed more fully below).

The set of ranges in the Range Cache could be stored in many different ways, but fundamental to the performance of the system is the ability to search them very quickly. The lowest complexity way to do this is to store ranges directly as a set of memory cells for $start$ and end with a separate comparator on each memory address. Deciding whether a given address overlaps with a range in the set then translates into a simple parallel comparison of the query address over the set of stored $start$ and end values. Figure 2 shows the basic 2-stage pipeline at the core of our approach.

The first stage of the pipeline interfaces to an input controller (not shown) which decodes the incoming range instructions. An address is queried across the set of comparators and only the containing range is returned. The first stage compares the point input (either $newstart$ or $newend$) with all the stored $start$ and end registers in parallel and stores the matching start and end indices for use by the second stage. It also records if the match was an exact match (which hits flush on the edge of an existing range), so that the second stage can decide the number of new ranges to be inserted and if the new range can be aggregated with any existing range. Note that the comparison need not be a full subtraction operation, it only has to be able to identify the greater of two numbers, a task for which a variety of small and fast circuits apply [8]. The state machine in the second stage acts as the controller and handles all of the overlapping ranges that can occur (explained in detail in Section 4.3). Two pipelined accesses are only dependent on one another if they access the same ranges, but by speculating that all access are independent, and squashing those that are not, we can reduce the stall cycles by 32% on average. While we had to omit a more detailed analysis of this effect, the range cache policies had a far larger effect on performance.

4.2. Managing a Cache of Ranges

Cache Updates: As we have already seen, updates account for roughly 35% of the tag requests in the dataflow tracking tools we have tested and to provide any reasonable amount performance most updates need to be handled in the range cache only. This will rely on the range cache's ability to detect all overlaps to dynamically stitch the new range into the existing ranges if possible. This updating, similar in mission to a write-back policy, can be used even if the address range is *not* found or only partially found in the cache; the range cache simply updates the cache to include the given range entry. Thus updates should not ever directly result in a range cache miss. However, there is a complication: changing the tag of a stored range or entering a new range may increase the number of stored ranges, and if this number overflows the available range entries in the cache, an entry has to be evicted to the secondary store (all ranges are assumed to be dirty). Here we use a simple LRU based policy to decide the range to be evicted and written back to the memory hierarchy.

Cache Reads: Once update misses are handled, the next natural question then is what happens on a cache read miss? A read access is considered a miss if *any* of the addresses between $newstart$ or $newend$ are not present in the range cache, in which case the memory hierarchy needs to be accessed to find the matching tag. However, because this is a cache, it also means that we need to bring a range in from memory and replace some entry in the cache. The complication arises when we go to fetch that range from memory and pull it into the range cache. Unlike a normal cache where a fetched cache line will not overlap an existing line in the cache, the fetched range entry being brought in from the memory hierarchy might possibly overlap multiple cached ranges which have been written to, and inserting this range entry into the cache directly will overwrite the most up-to-date tag information. However, because we know the size of the gap we are attempting to fill, and because this case occurs infrequently, this case is handled by inserting only the truncated range $[\max(newstart, gapstart), \min(newend, gapend)]$. In order to

Program	Tagged Memory	Ranges
bzip	15.10MB	21
crafty	2.60MB	8236
gcc	3.46MB	3513
gzip	7.43MB	16
mcf	97.22MB	7053
parser	32.25MB	183

Program	Tagged Memory	Ranges
vpr	0.74MB	3597
ammp	7.47MB	39596
applu	199.31MB	54
swim	200.17MB	42
wupwise	184.84MB	38
defined-avg	57.94MB	4861

Program	Tagged Memory	Ranges
admin login	70771B	7558
book add	151582B	16958
checkout	29908B	2740
store view	44039B	4053
taint avg	74075B	7827

Table 2: Compression potential of storing ranges: The maximum number of tagged ranges over a program's run is much lower than the tagged memory footprint for both dataflow tracking tools tracking SPEC Int, FP and Ruby-on-Rails server programs.

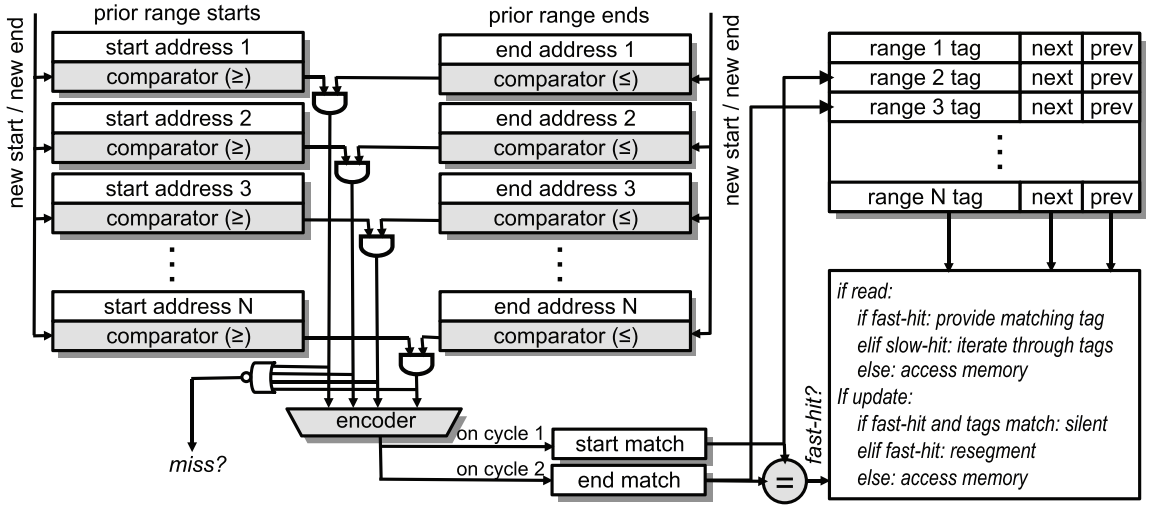


Figure 2: Range Cache Architecture: The first pipeline stage matches the start and end points of the new range, while the second stage is a state machine that performs tag lookup and decides on the aggregated range entry to be written back.

minimize the cost of fetching back the missed range, we limit the size of the fetched range to a maximum of 64B. Further, similar to a cache update, fetching in a new range into the cache might also require evicting a stored range in case the cache overflows. Both these cases require a write back to the secondary tag store.

Behind the Cache: Central to the notion of any cache is that it stores a subset of the complete set of information, and the range cache is no different. Prior tag cache approaches read from a full set of tags stored in protected memory when there is a miss, and for the purposes of a fair comparison we assume that both a tag cache and our range cache are served by the same non-range based tag storage. Specifically, we assume a two level bitmap with the first level containing tags for 64B aligned address ranges, and the leaf nodes containing byte level information if the first level is partially tagged. When a range read misses in the cache, the miss handler walks the trie and fetches a range from a 64B size chunk back into the cache. In case of a write, we use similar trie semantics to store the tags in base and leaf nodes.

4.3. Handling Range Overlaps

As is (hopefully) clear, the key to building a range cache based dataflow tracking scheme is handling all of the different range-overlap cases correctly. When a new read or update range request comes in (with start address $newstart$ and end address $newend$), the first step is to find all of the stored ranges that it overlaps with. Because stored ranges do not overlap with one another, finding the overlaps between the new range and the stored ranges can be done quite quickly.

The second stage of the pipeline in Figure 2 sketches how a state machine start and end match indices, looks at the incoming tag and the tag of the overlapped ranges, and inserts new range(s) with their corresponding tags into the stored array. For both read and update operations, the $newstart$ and $newend$ of the new range are injected into the two stage pipeline over two consecutive cycles to find the containing ranges. There are thus four possible scenarios: both $newstart$ and $newend$ are found in the range cache and both are contained by the same stored range (in which case we know we need only care about the interaction between these two ranges), both $newstart$ and $newend$ are found in the range cache but are contained by different stored ranges (in which case we overlap 2 or more stored ranges), only one of them is found in a stored range (while the other one misses in the cache) and finally, neither of them is found to lie in any of the ranges in the cache. Once the overlap is classified into one of these scenarios, the request type (read or update) and the tag of the new range must be considered.

Reads Overlaps: If the read request is fully contained within a single range that is present in the range cache, the operation is quite simple: simply return the tag of the containing range. If one of the end points of the read range is not present in the cache, then the appropriate range needs to be fetched from the memory hierarchy and inserted into the range cache (identical to the update case described below). If the read range overlaps multiple stored ranges then there are several ways of handling this depending on

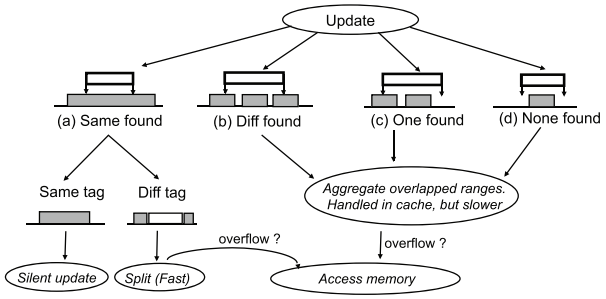


Figure 3: Different types of range overlaps possible for a Range Update Request. Silent updates are ones that write back the same tag to an existing range (Common and handled very fast). Writing a different tag to an existing range could split it into (possibly) three new ranges (Common, and handled fast). Other update types may overlap multiple existing ranges or may not even be found in the cache.

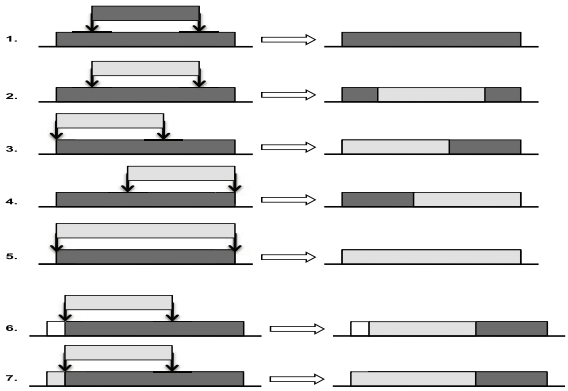


Figure 4: Along with the base cases described in Figure3, there are many specific subcases that all require handling. This figure shows a subset of those. 1) Silent hits; 2) Updates directly into the middle of an existing range require 3 ranges to be touched; 3-4) Updates to the edge only require one new range to be created; 5) A fully covering range requires only a change in the tag; 6-7) are special cases of 3 and 4 above where either the neighboring range is extended or not. All these cases can be handled in one or two cycles by our controller

the policy being enforced or analysis being performed. The tags can be prioritized (for example, returning the most "conservative" tag value), they could be joined through a reduce operation (similar to the method considered in Rifle [18]), a programmable rule table could be used (similar to Flexitaint or Rakha), or all of the results could be returned to a software handler through an user-level interrupt. As demonstrated by these past works, the "correct" behavior here is really tied to the intended semantics of the dataflow policy, and because this case is so infrequent (in Section 5.0.1 we show that it accounts for less than 0.001% of accesses across several different tools and tracked programs) we believe this matter to be orthogonal to the method by which tags are associated with addresses. However, for the purposes of evaluation we assume that all the tags need to be read and joined into a single conservative estimate.

Updates Overlaps: An update request starts, like read, by determining the overlap scenario. However, unlike read, the updates can require modification to many different ranges in the cache all at once. To make this more clear, Figure 3 enumerates the different overlaps that can occur during a range operation.

Case (a): The first case is when the new range to be written lies completely inside an existing range. This is both the simplest and most common case, and has two further sub-cases: In the first subcase, the tag of the new range is identical to that of the stored range it is contained within. In this case the update has no effect on the stored ranges, and we refer to it as a *silent update*. This case is surprisingly common[1], as tagged variables are often updated, reassigning the same tag back to the same address, an observation also noted in [19] to reduce tag-updates induced by inter-processor communication. The other sub-case is also very common, and involves an incoming range completely contained within a stored range, but with a different tag. This will cause the stored range to fragment into a set of new ranges as shown. This could result in up to two new ranges, the original range broken into the new smaller range and two "leftovers" on the either side of the inserted range. However, if the $newstart = start$ or $newend = end$ less than two new ranges will result from the update.

Case (b) happens when the start and end addresses of the given range are found to be present in two separate address ranges, with possibly multiple ranges existing between them. (When the start and end hit in adjacent ranges then there exists none). In this case, our architecture will need to remove the entries for these intermediate ranges, and create a new 'aggregated' range.

Case (c) represent situations when the input range extends one of the existing ranges in either direction. Here, our architecture performs a search for start and end of range, and the existing range is found. It will then extend this existing range entry to new boundary and remove other ranges that are overlapped by the input range.

Case (d) show situations where none of the end-points of the given range exist within any range in the Range Cache. In theory, this case is not that bad because it is an update, and as an update, we know the values that need to be written into the Range Cache. The problem is that we don't know *where* to put the range relative to the other ranges. In practice this means searching through the range cache to find the elements between which it should be inserted. The order of the elements in important for Case (c) above so that ranges that lie between two ranges can be quickly discarded when required. Once we find the proper place for insertion, a new range is inserted as appropriate.

Handling simple queries like inserting into single stored ranges requires only minimal intelligence on the part of the Range Cache, but managing the remaining overlap types without ever fragmenting or inserting redundant ranges requires some modifications to the design.

1. Silent Stores [10] are stores that write the same value back into memory that was already there, and thus can be safely ignored. In the case of dataflow tracking, silent operations (inserts and deletes) are even more common because a store instruction is very likely to have the same tag over time, and very likely to write back to the same memory addresses. Consider definedness tracking where we store the set of all valid address ranges. Here, initialization of a data-structure would result in a new range being created, while every subsequent update to the initialized data-structure will produce an insert request where the range is already marked defined. For taint tracking, addresses are very commonly marked "untainted" when the addresses were never marked tainted to begin with.

4.4. Complex Overlaps

Range Updates Covering Multiple Existing Ranges: Consider Case (b) in Figure 3. Here the input range overlaps multiple stored ranges, and the desired result is a single entry that represents the entire tagged region. We would like to delete entries for all ranges that are overlapped by the inserted range, but the problem here is that we are processing only the end-points of the input range. As a result, we will know which two existing ranges the $start$ and the end of the input range intersected with, but we will not know which other existing ranges were overlapped by the input range. To find all the overlapped ranges, we need a sorted index table (analogous to a linked list in software terms). For example, in order to insert $[7, 30]$ in the range-set $[5, 11], [29, 31], [21, 27]$, the Range Cache would have to know that the order of existing ranges was $[5, 11], [21, 27], [29, 31]$. 7 would hit $[5, 11]$ and 30 would hit $[29, 31]$, and by traversing the list of ranges the Range Cache can invalidate all existing ranges and insert a combined range $[5, 31]$. The sorted index table allows us to quickly find the indices that lie between the two ranges containing the end points. Such a situation can also arise when only the $start$ of an input range intersects an existing range while the end overlaps multiple existing ranges, or vice versa. Cases (c) and (d) in Figure 3 shows such overlaps, and eliminating redundant entries requires traversing either forward or backward through the index table with a state machine. It thus takes as many cycles as there are ranges to be invalidated.

Handling Missing Endpoint Information: Cases (c) and (d) mentioned above present a further challenge. One or both of $newstart$ and $newend$ points of the new range many not fall in a range in our range cache. For instance, consider inserting range $[4, 28]$ in the same set ($[5, 11], [29, 31], [21, 27]$). Since neither 4 nor 28 would intersect $[5, 11]$, $[21, 27]$ or $[29, 31]$, it would be impossible to detect that the $[5, 11]$ and $[21, 27]$ entries have to be invalidated when $[4, 28]$ is inserted. In these cases the state machine must traverse through the index table to find the location at which those ranges should be inserted into the cache (i.e. find the ranges r_1 and r_2 such that $r_1.end <= newstart$, $r_2.start >= newend$ and that no other range lies between r_1 and r_2. Once these are found, then $r_1.next$ and $r_2.prev$ are both set to $newentry$. Clearly this operation is not going to be very fast, but it also occurs very infrequently. In our prototype RTL design this is handled by a simple but slow state machine.

We have found that when such overlaps and misses occur they usually span only a few ranges, and our tests (Figure 5) show that while such updates are very uncommon in definedness tracking, these comprise almost 10% of accesses in network taint tracking in the Ruby-on-Rails application. So handling these cases using a state machine, while being slower than the simple case where both ends are found in the same range, is still much faster than invoking a software interrupt to invalidate overlapped entries.

5. Evaluation

In the architectural descriptions of our range cache design, we built around the fundamental assumptions that the most complex types of ranges overlaps are also the most infrequently occurring. Here we provide data across all of the benchmarks used to show that this is indeed the case, and we evaluate the miss-rate and estimate the performance impact across several dataflow tracking tools (1-bit Taint Tracking, 2-bit Definedness Tracking, and 32-bit Dataflow Tomography).

5.0.1. A Breakdown of Range Access Types. Figure 5 shows the relative breakdown of the different types of range accesses for 2 different dataflow tracking tools. The update requests are broken down into silent updates, fast updates (updates that only operate on a single existing range), other updates (that span multiple ranges), and update misses (where the endpoints were not found in any stored range and requires a variable length number of cycles to perform the insert). The read requests are broken down into read hits (where access is confined to a single cached range), others (where the read spans successive ranges) and read misses (when some part of the requested range is not found in the cache). Both definedness tracking and taint tools show a very distinct request mix, but the common cases are read hits and silent or fast updates for both tools and are handled fast in the cache. Tag read requests are predominant across all benchmarks and all tools. 62% of the definedness tracking tool's requests are reads, while 56% requests by the taint tracking tool are reads. Of the SPEC integer programs we evaluated, 0.007% reads miss, while floating point programs (owing primarily to `ammp`) miss 1.43% on average. The taint tracking tool run on the ruby bookstore applications performs well too and average miss rate is only 0.15%.

While read behavior is fairly similar between the definedness and taint tracking tools and benchmarks, the updates are quite different. Definedness tracking has fairly even mix of silent and non-silent updates, while taint tracking has a more significant proportion of silent updates. In the former, silencing the superfluous inserts removes almost 20% of the requests on average, and leaves about 15% of the requests actually modify state. Most updates overwrite the tag for part of a single stored range, which as mentioned before, are handled fast in the range cache. This leaves only 0.85% of SPEC integer programs and about 2% of floating point tag requests that require a small 2 or 3 cycle pipeline stall. The taint tracking tool is very different. Since it tracks network data, more memory addresses remain untagged as compared to the definedness tracking tool, where hundreds of MBs of the address space gets tagged. Since a lot of memory is untagged, silent updates to untagged memory comprise a significant proportion of its accesses, and eliminating these removes about 32% of the tag requests. Non-silent taint updates are about 10.75%, of which 10.6% are accounted for by updates where neither of the new endpoints are found in the cache and the remaining 0.15% comprise mainly of effective simple updates. This behavior is explained by the bursty nature of tagged data usage in network programs. These programs create on the order of $10,000$ ranges in the secondary store, and many of these are very small in size. These ranges sometimes fill up the range cache and cause incoming inserts to not overlap any range; thus taint tracking benchmarks are a good complement to the more range-intensive definedness tools. However, even though many accesses are to small ranges (the tiny islands of data that are marked as tainted), the range cache still performs well since a lot of the untainted address space then

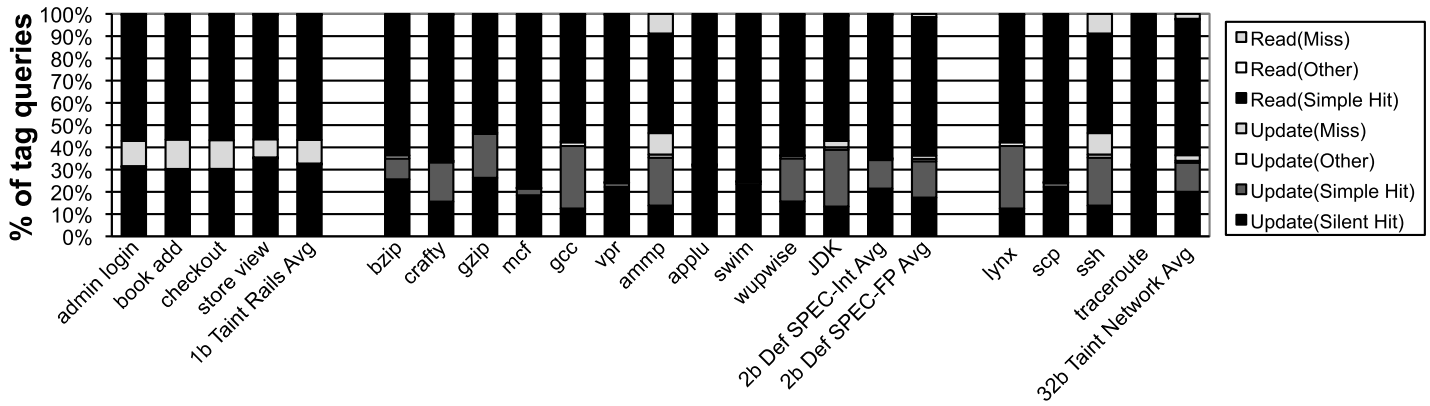


Figure 5: The relative proportion of various range queries for Dataflow Tracking. The SPEC applications (`bzip` *through* `wupwise`*) are run with the definedness tracking tool, while the Ruby applications (*`admin login` *through* `store view`*) and network applications (*`lynx` *through* `traceroute`*) are run with network input taint tracking. The vast majority of accesses fall into "fast" categories where the updates are simple and the reads are hits to a single range.*

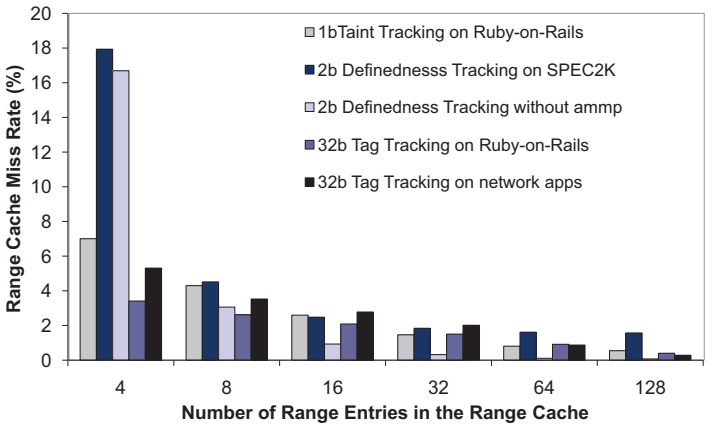


Figure 6: The graph shows miss rate of Range Cache for varying number of range entries

clusters into ranges and forms the target of a majority of accesses. Later we will examine the performance of this approach on a burst of network traffic and show that, even though the number of misses is high, the fact is that the tag data is still compressed enough to allow you to to get far lower miss rates than a more conventional tag caching method. This effect is only further magnified as we move to larger and larger tags.

5.1. Evaluation of the Architecture

One concern in creating an architecture, such as our proposed range cache, that address a very specific set of problems is the tradeoff between the complexity of the design and the benefit that it provides. Given all of the complex types of overlaps that need to be handled on read misses and updates that overlap multiple ranges, it might seem a dauntingly complex task to implement in hardware. Fortunately, as we have seen in section 5.0.1, the most commonly occurring cases in practice are actually quite straight forward (silent updates, direct range splits, simple reads, etc.). The real complexity occurs in the uncommon cases, but these can be handled slowly through either a state machine or even a more general tiny hardwired controller. To ensure that we have not missed anything in our design, we have implemented a synthesizable RTL model (in verilog) and tested it across a variety of input and output data. We also use this RTL model to count the exact number of cycles required in those cases where complex operations are required (such as deleting multiple range entries for example) so that we can accurately reflect those counts in our performance estimation. This hardware design also allowed us to estimate the area of our approach. The controller itself is just under 3000 logic gates (which does not include the memory for the actual tag storage itself), and when combined with the required memory, a 128-entry range cache storing 32-bit tags is approximately the same size as 4KB of memory.

Typical range sizes: We have shown in the motivation section that a small number of ranges are sufficient to store the tag information for a large set of addresses, which makes storing the tags as simpler page-aligned ranges a potential option. However, a snapshot of a 128 entry range cache for definedness tracking on `gcc` reveals that over 100 out of 122 stored ranges (82%) are below 64B in size, whereas a page size is 4096B. For the same snapshot, the largest range is almost 2MB in size. Thus, even though the maximum *number* of tagged ranges is small, there is great variation in the *sizes* of the ranges and this makes using a fixed page-sized range cache entry an expensive proposition.

Performance evaluation model: There are two primary sources of performance overhead for a hardware assisted dataflow tracking scheme. There is no performance penalty as long as the tag read or updates hit a single range in the cache. However, each time a tag read misses in the tag cache, a software or firmware handler must be invoked to access the memory hierarchy to fetch the tag cache line. This need to fetch this data from the memory hierarchy is common to all of these approaches, and can be handled in the same way. Range caching does not affect this at all. In the work on Raksha [7], a light-weight exception handler that runs in protected mode and costs only as much as a function call in userspace is proposed, but there is still a cost in terms of instructions executed and pollution of the memory hierarchy (fetching those tags from memory may displace regular program

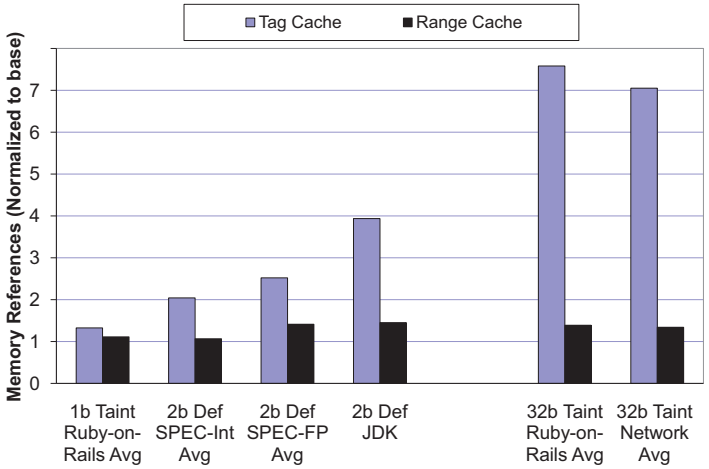


Figure 7: Extra memory references in tools with 1/2-bit tag per byte to 32-bit tags per byte: A tag cache miss triggers a secondary store access and adds extra memory references to the native program execution. This graph compares the increase in memory references while using 1-bit taint, 2-bit definedness and 32-bit tag tracking tools. The taint tool runs Ruby-on-Rails server apps while the definedness tool runs SPEC-Int,FP and a JDK program. The 32-bit tools runs the Ruby-on-Rails server apps and some other network apps. A range cache has better cache performance and will result in fewer calls to the backing store handler. So it has a smaller impact on the number of loads and stores.

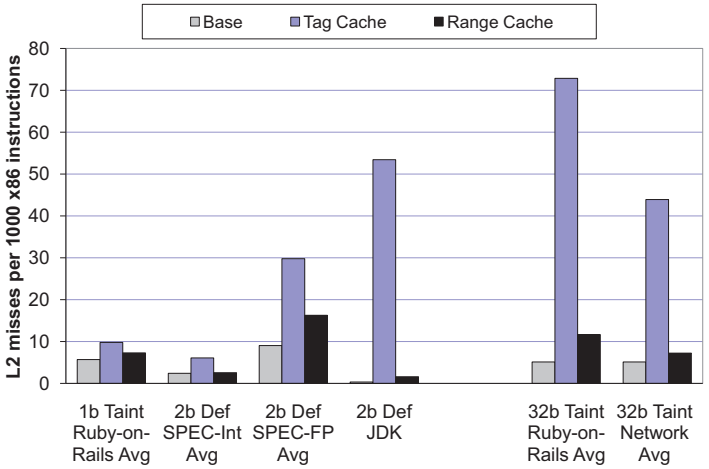


Figure 8: L2 misses per 1000 instructions in tools with 1/2-bit tag per byte to 32-bit tags per byte: The extra memory references shown before not only add extra instructions to be executed, but affect the memory hierarchy performance of the native program. This graph shows L2 misses per 1000 native x86 instructions for both the tag cache and range cache based dataflow tracking tools that use a 1-bit taint, 2-bit definedness or a 32-bit tomography tag.

data from the cache). We quantify both of these effects through memory hierarchy simulation.

To accurately characterize the effect of range caching – which relies directly on the amount of range locality in complex full systems across all of memory – we use trace driven simulation (to capture the full OS stack) and estimate performance with a relatively simple and conservative model (as done in other full-system evaluations such as MMP [22], Minos [5]). We assume a processor executing one x86 instruction per cycle, with a 16KB L1 Data cache and 4MB L2 cache, and we report the number of extra instructions executed, the increase in the number of additional L2 stalls (caused both by extra traffic from the tag and range caches, but also from the interference due to those accesses), and a rough estimate of the resulting performance. Stalls are tabulated by extracting those additional memory accesses required to service the tag storage in main memory and inserting them through the memory hierarchy simulator to measure the increase in the number of L2 misses (each causing a stall of 200 cycles), the extra instructions required, and the impact of extra L2 accesses. For these simulations, we have assumed a 4KB tag cache and a 128 entry range cache.

Cache Hit Rates Versus Number of Ranges: We test the range cache by using it with the three dataflow tracking tools under test, and measuring the miss rate while varying the number of range entries in the cache. Figure 6 shows the results of this experiment. With 128 range entries, the 1-bit taint tracking tool has a miss rate of 0.54% (even with only 4 ranges, it's miss rate is just 7%). The average miss rate for 2-bit definedness tracking on spec benchmarks is 1.57%, starting from 18% at 4 ranges. Among the Spec programs, `ammp` has the worst miss rate (18%) and contributes greatly to the average; sans `ammp` the overall average drops from 1.57 to 0.07%. While a range cache averages 0.54% for taint tracking, a 4KB tag cache averages 2.5%. For the definedness tracking tool, the averages for range and tag cache are 1.5% and 5.7% respectively. We observed that the miss rates begin to plateau out at 32 range entries, and the improvement thereafter is very small. While these tag cache miss-rates are arguably small enough, the real difference comes in when we have large 32-bit tags to store. For such tags, range cache miss rates of 0.5% contrast with those for a simple tag cache in Figure 1, which misses 10% when tracking per word and 30% when tracking per byte. It is interesting to note that the miss rate of the range cache is lower for the 32-bit tool than for the 1-bit tool for smaller cache sizes, while they settle down to similar values for 32 ranges and beyond. This is because, ideally, being independent of the tagging granularity, a range cache should give similar miss rates for both 1 and 32-bit tools that run the bookstore application. However, since the bookstore application is interactive, slight changes in the execution trace for different runs are expected and these show up as different miss rates, especially for small range cache sizes.

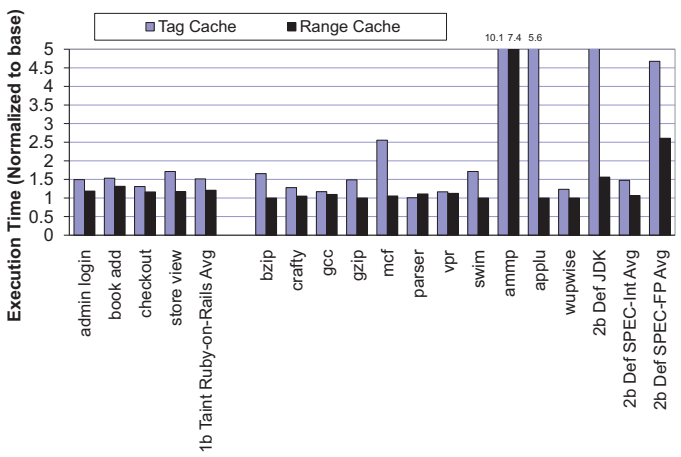
Extra Memory Accesses: Figure 7 shows the extra memory accesses that have to be performed by the program in order to access the secondary tag store for both tag-cache and range-cache. We measured this by implementing a software tag loader and then making calls to it when the tag cache requires service (on misses or evicts). This in turn will inject new accesses into the memory hierarchy which increase the load on the memory system. These experiments quantify the extra number of memory accesses generated by this loader for the different applications. A conventional tag-cache has a low miss rate for most programs as long as the tag sizes are just 1-bit (e.g. 1b Taint tracking tool in Figure 7) because there is so much data packed into the cache. However, as the tag size increases, so do the miss rates and thus

so do the extra memory references that arise from handling these misses (e.g. 2-bit tags in SPEC-Int, SPEC-FP and JDK in Figure 7). Figure 7 also shows the results for the 32-bit tools. Here we compare our range cache approach to a simple tag cache approach where both of the caches occupy approximately 4KB worth of space (we give the benefit of the doubt to the tag cache approach and assume that all of the area required goes directly to storing tags). The key difference here is that we now are storing full 32-bit tags which of course severely handcuffs the direct tag storage methods because of all the space required to store redundant tags. However, even in this small amount of space, the range cache can store enough entries to keep the inflation on the number of memory accesses to about 1.5X for a 32-bit tool as compared to 1.2X for the 1 and 2 bit tools. This is obviously a non-negligible amount of extra memory accesses, but remember it is also allowing full 32-bit tags to be kept on every physical address (at the byte level) in the entire system.

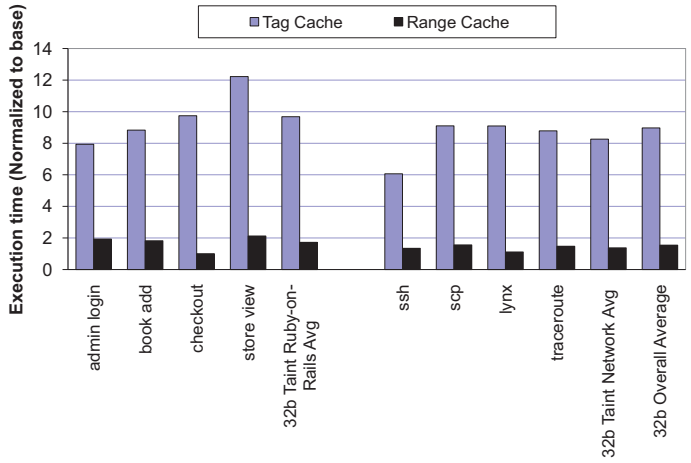
Extra L2 Misses: While data on the extra memory accesses helps us understand the extra load placed on the memory hierarchy, an equally important characteristic is the number of additional L2 cache misses encountered. There are two ways the number of L2 misses could be impacted, either a) directly through access to tags that are not stored in the L2 or b) indirectly by accessing tags and pulling them into the L2 that displaces other data that was needed later. Figure 8 shows the additional L2 misses incurred by the program due to accessing the secondary tag storage (mapped into the memory hierarchy) for both tag-cache and range-cache. The figure, as the others before it, shows the 1-bit taint tracking on Ruby applications and 2-bit definedness tracking on SPEC benchmarks. Figure 8 also shows L2 misses for the 32-bit case as above. All results are reported in number of L2-misses per 1,000 x86 instructions. For all of the schemes the range cache results in less misses than the standard tag cache, but for most of the 1-bit and 2-bit tools the results are comparable (at least within a factor of 2). The exception to this is JDK, which has more than a factor of 10 less L2 misses even for the 2-bit tool. Again, the biggest difference is with the 32-bit tools (Figure 8), where our range cache results in lesser cache misses on average.

Performance Slowdown: Putting the number of memory accesses and misses together with the stall cycles incurred by complex range operations we can draw an approximate picture of the performance of the system. Again we show the 1, 2, and 32 bit tracking tools in Figure 9. While some past schemes tracked only the heap [20], some others stored the secondary tag store as a flat bitmap so that tag cache misses can be handled in hardware [20,19,7]. In our evaluation, we are tracking all tags through all of physical or virtual memory (both stack and heap) and (like MMP) handle tag cache misses using a software handler (this allows the secondary store with 32b tags per byte to be compressed). Hence some performance estimates appear slower than prior estimates.

For 1 and 2 bit tags, we see the trend that both tag cache and range cache have low performance overhead. The tag cache, however, performs poorly on `mcf`, `ammp`, `applu`, and JDK. Some of these values are truncated off of the top of the graph. A



(a) Performance for 1-bit (Taint tracking on Ruby-on-Rails Application) and 2-bit (Definedness tracking for SPEC apps and JDK).



(b) Performance for 32-bit Dataflow tracking tools on Ruby-on-Rails Application and other network applications

Figure 9: Comparison of execution time of dataflow tracking tools using a Range Cache compared to Tag Cache.

range cache of equal size will miss a great deal less and will have few extra stalls (due to infrequent complex accesses as we discussed previously). While the range cache runs quite a bit slower than native on `ammp` (7.4X), it still improves over a tag cache (10.1X). `mcf`, `applu`, and JDK, on the other hand, all see large improvements. Further, while there are moderate gains when tracking 1 and 2 bit tags, the gains are far more visible with 32-bit tags, with the average slowdown coming down from around 9X to 1.52X. In handling large dataflow tags (or any memory tags that are frequently written) a range cache can provide a significant benefit.

6. Conclusions

As dynamic dataflow tracking evolves as a technique, we are likely to continue discovering surprising and powerful software analysis uses – each requiring specialized, and sometimes large, data tags. Caching these large tags on-chip in the conventional manner leads to a substantial performance hit (average 9X for a

4KB cache). Rather than store these tags as a large flat array, we show how by associating attributes with arbitrary ranges we can keep a very compressed representation of memory tags, and that by caching these ranges we can make very effective use of a very small amount of on-chip memory. Dataflow tags naturally exhibit a high degree of spatial-value locality, and storing memory tags as ranges (which cover a non-aligned contiguous span of memory), we can perform both reads and updates efficiently without ever taking the tag data out of its highly compressed form. Even in cases where tags are not likely to naturally fall into contiguous regions, for example when we tag every byte of network input with a unique identifier, there is still a large degree of range behavior *between* those identifiers. If you cache the tags directly, even elements that have uninteresting tags will occupy space in the tag store (you still need to check if those addresses have tags), where as in our range based scheme all of those elements can be efficiently summarized with a single entry.

Through our novel dynamic range caching techniques, a small range cache of only 128 entries (requiring 2KB of storage and about an equal amount of additional hardware) results in an average miss rate of around 1.5%. This miss rate is comparable with a 4K entry tag cache, which if storing 32-bits tags per byte would require at least 128KB of memory (just for the metadata tags). Of course storing this metadata as ranges comes at the cost of some added complexity, but in practice we have found the most complex overlapping cases are exceedingly rare and as such can be handled through a simple but slow controller. Across the applications we examined such cases occurred only 0.5% of the time. As such, achieving good performance comes down to handling the simple overlap cases quickly where a range is inserted into the middle of a single pre-existing range. As the devil is always in the details, we have demonstrated that it is indeed feasible to implement such a scheme by designing one for ourselves. Our synthesized device, while not highly optimized, is fully functional and the controller requires less than 3000 gates. In the end, we expect these techniques will be useful in associating any large and highly dynamic metadata with the bytes spanning the entire memory system, an ability which may turn out to be useful in many other contexts as well.

Acknowledgment

The authors would like to thank Fred Chong, Susmit Biswas, Bita Mazloom and the anonymous reviewers for providing useful feedback on this paper. This work was funded in part by NSF Career Grant CCF-0448654, CNS-0524771, and CCF-0702798.

References

[1] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, April 2005.

[2] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. *SIGPLAN Not.*, 42(10):405–422, 2007.

[3] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[4] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[5] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society.

[6] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 235–248, New York, NY, USA, 2005. ACM.

[7] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *34th Intl. Symposium on Computer Architecture (ISCA)*, 2007.

[8] J.-Y. Kim and H.-J. Yo. Bitwise competition logic for compact digital comparator. In *IEEE Asian Solid Stated Circuits Conference*, 2007.

[9] L. C. Lam and T. Chiueh. A general dynamic information flow tracking framework for security applications. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 463–472, Washington, DC, USA, 2006. IEEE Computer Society.

[10] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *International Symposium on Microarchitecture*, pages 22–31, 2000.

[11] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and Visualizing Full Systems with Data Flow Tomography. In *ASPLOS-XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[12] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS '05)*, 2005.

[13] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting General Security Attacks. In *Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[14] Rational Software. IBM Rational Purify for Unix. ROI Analysis, 2000.

[15] J. Seward. Origin tracking tool, valgrind release-3.4.0. Pre-release at svn co svn://svn.valgrind.org/valgrind/trunk.

[16] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX'05 Annual Technical Conference, Anaheim, California*, 2005.

[17] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[18] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254. IEEE Computer Society, 2004.

[19] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Fourteenth International Symposium on High Performance Computer Architecture (HPCA)*, pages 196–206, New York, NY, USA, 2008. ACM.

[20] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *13th International Symposium on High-Performance Computer Architecture (HPCA-13)*, February 2007.

[21] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. February 2007.

[22] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, New York, NY, USA, 2002. ACM Press.

[23] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.

[24] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[25] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, 2004.