A Case Study in Optimizing HTM-Enabled Dynamic Data Structures: Patricia Tries

Thomas J. Repetti Maurice P. Herlihy

Department of Computer Science
Brown University
{trepetti,mph}@cs.brown.edu

Abstract

The advent of multi-core microprocessors with restricted transactional memory (RTM) and accompanying compiler support allows us to revisit fundamental data structures with an eye to extracting more parallelism. The Patricia trie is one such common data structure used for storing both sets and dictionaries in a variety of contexts. This paper presents a concurrent implementation of a dynamically sized Patricia trie using a lock teleportation RTM fast path for `find`, `add` and `remove` operations, and a slow path based on atomic exchange spinlocks. We weigh the tradeoffs between alphabet size and tree depth inherent to tries and propose a novel means of determining the optimal number of retry attempts for specific operations on dynamically allocated data structures. The strategy proposed separates the retry policy governing operations that potentially interact with the operating system's memory management facilities from read-only operations, and we find that this transactional trie can support considerably higher multiprogramming levels than its lock-based equivalent. A notable result is that this scheme keeps throughput from collapsing at high thread counts, even when the number of threads interacting with the data structure exceeds the number of hardware contexts available on the system.

Keywords Patricia trie, symmetric multiprocessing, concurrent data structure, hardware transactional memory, restricted transactional memory

1. Introduction

1.1 Transactional Memory

Transactional memory is a synchronization paradigm, which effectively exends the atomicity of traditional atomic shared memory operations like compare-and-swap or fetch-and-add to generalized read-modify-write operations on arbitrary regions of memory [12]. Although it was originally conceived as an architectural feature to extend cache coherency protocols in hardware, until recently all implementations were strictly in software [29]. The composability of speculative regions of software execution alone is a benefit to the productivity of programmers writing concurrent software, but before the widespread commercial availability of hardware with transactional memory support, the full performance benefits of the technique could not be brought to bear. Currently available commercial hardware such as Intel's Haswell processors [14] and POWER8 architecture systems like IBM Blue Gene/Q [11] and System z [16] all support best-effort hardware transactional memory, meaning there are no guarantees of forward progress, and a transaction may abort for any reason, the cause of which may be opaque to the programmer. Avni and Kuszmaul preface [1] with a good summary of the variety of issues that may trigger an abort for unspecified reasons under Intel's Transactional Synchronization Extensions (TSX) RTM. The reasons for transactional aborts under the TSX scheme that do not have an cause visible to the programmer may include cache misses, TLB misses and interrupts. For these reasons, it is common to use pre-allocation strategies when investigating data structures under HTM in order to avoid interference from the operating system. The implementation presented here, however, uses dynamic memory allocation at runtime as one would expect from a normal data structure in the field.

1.2 Tries

Tries [9] are tree data structures used to store a set of arbitrary length keys. The root of such a tree is a node corresponding to the null string key. Each node, including the root, has a number of possible children determined by the number of characters in the alphabet from which strings are composed. A string present in the set will have a succession of non-null pointers to character nodes starting at the root which match each character in its sequence. This assumes the presence of a string termination signifier such as the "`\0`" character from the C string model or a flag within the node signfying the end of a string. Without a signifier of this kind, it could be inferred that prefixes of strings in the set were themselves keys in the set when in fact they were not [3]. In their simplest form, therefore, tries have exactly one node for every character in a unique suffix of a given string within the set. Shared string prefixes then share the prefix nodes descending from the root since their unique suffixes will only branch at the node corresponding to the character at which the strings themselves diverge.

1.3 Patricia Tries

A Patricia trie [24] (also known as a radix tree or prefix tree) is a compact trie, in which any only child node can be eliminated by incorporating it into its parent node. A string with a suffix unique to the set can consequently be stored with a single node regardless of the length of the suffix. By the same logic, common internal substrings need only be represented by a single node as well, which further reduces memory overhead. This modification requires that the data structure keep track of the omitted characters in the compressed portions of the string on a node-by-node basis. Due to their modest time complexity for key lookup, Patricia tries are often used in IP address lookup [26] [30], as well as as natural language processing applications such as approximate string matching [28]. They also often serve as an intermediary lookup data structure for more intricate objects such as the string B-tree [8].

1.4 Alphabet Size

The most natural choice of alphabet, in which there are 256 possible characters for each byte, may be efficient in the overall number of pointer references necessary to store a given string, but this incurs an additional memory cost because of the per-node stor-

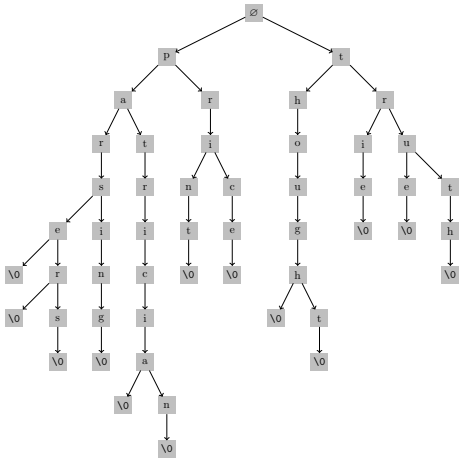

Figure 1. An uncompressed trie.

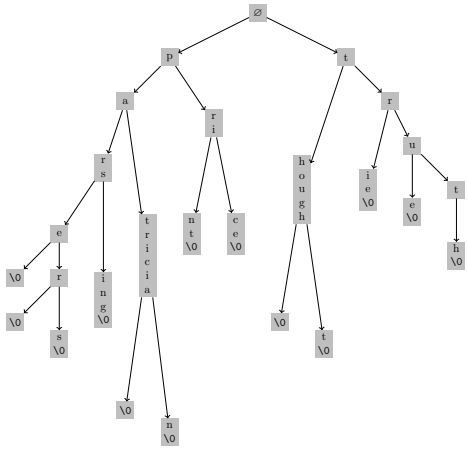

Figure 2. A Patricia trie.

age required to store the large alphabet. Each node's data structure must account for the 256 possible child nodes with 256 initially null pointers. On 64-bit architectures, this gives each node an inherent 2 KB memory overhead. Although small in absolute terms, this prohibits a node from fitting in a single cache line on all current consumer systems. While cache locality is important for any data structure, we will discuss why it is particularly important for hardware transactional memory in the section on data structure optimization.

Bitwise Patricia tries (also known as binary Patricia tries) all but eliminate this overhead in individual nodes by reducing the alphabet size to two, 0 and 1. It is partly due to this minimal memory footprint that bitwise Patricia tries have seen use in operating system infrastructure. One notable example is Doug Lea's Malloc (`dlmalloc`) [20], which uses a bitwise Patricia trie to efficiently index free chunks by their sizes [21]. A proposed intermediate alphabet reduction approach suggests using a hexadecmial alphabet to reduce memory overhead to 128 B [3]. Several proposed prefix trees and Patricia tries use an alphabet smaller than a byte alphabet but larger than a binary alphabet [2] [5]. While not having a strictly minimal memory footprint like tries based on binary alphabets, Patricia tries with alphabet size 2^k have, on average, half the maximum tree depth of those with alphabet size 2^{k-1}. Read operations are performed in $O(l)$ time where l is the length of the search string in alphabetic units, and modify operations are performed in $O(kl)$ time. As touched upon in discussing node size, n strings each with length l_i are stored with $O(nk + \sum_{i<n} l_i)$ space complexity [3]. We consider tries over a range of practical alphabets: binary, quarternary, hexadecimal and byte, each with 1-bit, 2-bit, 4-bit and 8-bit character sizes, respectively.

2. Related work

Shafiei describes a scalable, lock-free implementation of binary Patricia tries using compare-and-swap operations [27]. The initial implementation presented here was directly inspired by her thorough description of the issues in developing concurrent tries. As of the time of writing, however, we are unable to find any lock-free implementations for Patricia tries with arbitrary alphabet sizes.

Boehm et al. introduce the the uncompressed *Generalized Prefix Tree* (GPT), which takes advantage of an array of "jumper" pointers to find the nodes representing known prefixes without needing to lock nodes near the root in concurrent execution. They couple this with a node preallocation strategy to improve scalability. [2]

Leis et al. describe the *Adaptive Radix Tree* (ART), a novel solution to the trade-off alphabet size presents between tree-depth and node size, in which the number of child pointers in a given node is free to vary with the number of suffixes that happen to diverge from a particular node's prefix [22]. They also demonstrate the improved scalability of their data structure using HTM, specifically Intel's RTM, and propose the use of memory allocators with thread-local buffers to mitigate aborts caused by the allocator's trapping to the operating system [23]. The work thoroughly explores the implications of an HTM-enable ART within a main memory database, but it does not provide experimental data on the implications of concurrent workloads in which the multiprogramming level significantly exceeds the number of hardware contexts available on a given system, which is a primary aim of our work presented here.

3. Implementation

3.1 Data structure

The top-level trie data structure only needs to be aware of the size of the alphabet its nodes have and to maintain a reference to the root node of the trie. We also include a size parameter to allow for convenient memory usage housekeeping.

```
typedef struct pt {
    // Number of characters in the alphabet.
    uint8_t alphabet_size;
    // Data structure size in bytes.
    size_t size;
    // Root (empty string) node.
    pt_node_t *root;
} pt_t;
```

Listing 1. `pt_t`.

We compose the contents of our trie out of two defined types, a generic string and a Patricia trie node. The use of a generic string as opposed to native C strings is warranted since we are interested in sequences of values at arbitrary granularities. Therefore all strings mentioned in the following algorithmic description include a length field defined in terms of alphabetic units.

```
typedef struct string {
```

```
    // An array of bytes.
    char *string;
    // String length in alphabetic units.
    uint32_t length;
} string_t;
```

Listing 2. `string_t`.

Fundamentally, a trie node must have a generic label string that specifies the prefix it represents, a flag for whether the node's label itself is a key in the set and an array of child nodes whose size is specified by the alphabet we are using. Our trie implements a set for testing purposes, but all that would be needed to create a dictionary instead is a pointer at leaf nodes to some arbitrary payload, and this would require minimal changes to the layout of `pt_node_t`.

```
typedef struct pt_node {
    // Mutual exclusion lock.
    pthread_mutex_t lock;
    // Prefix and prefix length.
    string_t label;
    // Whether the node is a leaf.
    uint8_t leaf;
    // Number of immediate child nodes.
    uint16_t n_children;
    // Child nodes.
    struct pt_node *next[ALPHABET_SIZE];
} pt_node_t;
```

Listing 3. `pt_node_t` for a mutex-based trie.

Figure 3 shows the logical layout of the several internal nodes within the trie data structure as described. Note that internal nodes may be leaves if they are so marked.

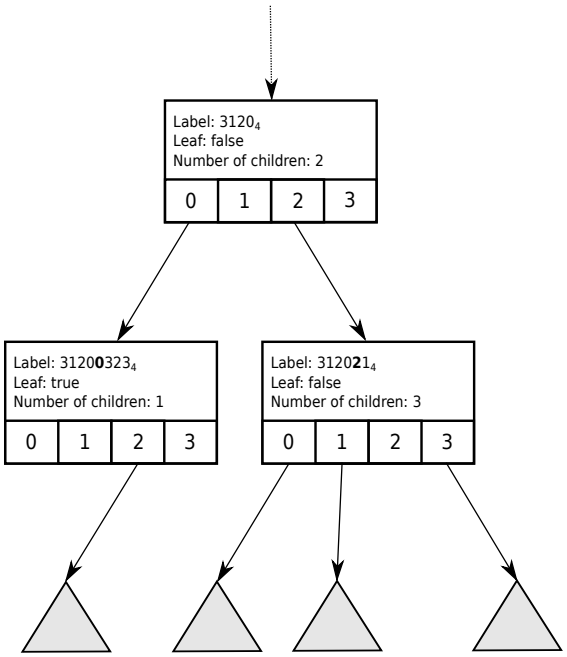

Figure 3. An example of the logical layout of a portion of a quarternary alphabet Patricia trie. The characters at which child node labels diverge from their parent node label are shown in bold.

3.2 Hand-over-hand Locking

Hand-over-hand locking is a well understood synchronization paradigm for list- and tree-like concurrent data structures [4]. We based our implementation of a lock-based Patricia trie off of a simplified (not lock-free) implementation of Shafiei's lock-free binary Patricia trie [27] and generalized it to arbitrary alphabet sizes. Hand-over-hand locking for our trie only needs to satisfy the invariant that a thread may only modify or dereference a pointer if it has a lock on the node containing said pointer. Since the mechanism behind such a locking scheme is commonplace, what follows is just a summary of the assumptions and effects of each primitive leaf operation and how they are composed into operations on keys in the set.

`search()`:

- Arguments: a pointer to the trie and a pointer to the query string.
- Returns: a custom data structure with the grandparent, parent, current node in the search routine, and a Boolean integer representing the current node represents an exact match or merely the parent of where a prospective match might go. All the nodes in the structure are locked upon return.
- Assumptions: the trie exists, and the query string conforms to the specifications of the `string_t` type.

`insert_leaf()`:

- Arguments: the presumptive grandparent of the node to insert (i.e., parent in the result from `search()` and the presumptive parent of the of the node insert (i.e., current in the result from `search()`).
- Returns: number of bytes added to the trie data structure, with all argument nodes and inserted nodes having been unlocked.
- Assumptions: the grandparent and parent nodes exist and are locked, and the insertion string conforms to the specifications of the `string_t` type.

`delete_leaf()`:

- Arguments: the grandparent of the node, the parent of the node, and the actual node to delete.
- Returns: number of bytes removed from the trie data structure, with all argument nodes having been either unlocked or destoyed.
- Assumptions: the grandparent node, the parent node and the deletion node exist and are locked.

From these basic operations, we can easily construct an external `find_string()` function to determine if a key is in the set the trie represents by calling `search()`, unlocking the grandparent, parent and result nodes and returning whether there was a match. Similarly, it is straightforward to construct an `add_string()` function by calling `search()` and passing its output into `insert_leaf()`. By the same logic, a `remove_string()` function is the equivalent combination of `search()` and `delete_leaf()`.

4. Baseline Transactional Impementation

4.1 Experimental Setup

All of the experimental results presented in the following sections have been generated on an Intel Core i7-4770 3.40 GHz Haswell CPU, with four discrete cores. Each core has 64 B cache lines, a 32 KB eight-way associative L1 cache, a 256 KB eight-way associative L2 cache and two hardware contexts, giving the chip a total

of eight simultaneous multithreading hardware threads or Hyper-Threads. The four cores share an 8 MB L3 cache and beneath that, 8 GB of main memory. At the time of writing, the experimental computer was running Debian 7 "Wheezy" and version 3.10.11 of the Linux kernel, and the C code was compiled under GCC 4.8.10 with `-O2` optimization enabled. All node allocations have been aligned to cache line boundaries using `posix_memalign()`, and unless otherwise specified all keys are randomly generated 128-bit integers so as to simulate a probable use case, the storage of IPv6 addresses.

4.2 Algorithmic Description

"Lock teleportation" [13] forms the basis of our speculative search implementation. We define individual speculative critical sections for `search()`, `insert_leaf()` and `delete_leaf()` as those regions of execution in which hand-over-hand locking proceeding from the root would—in a mutual exclusion implementation—obey the invariant that a pointer may only be dereferenced or modified by a thread holding the containing node's lock. A lock teleportation traversal of the trie in `speculative_search()` simply traverses without locking, knowing that its speculative reads would be invalidated should it observe an inconsistent state. Such states include, among other things, a pointer to a region of memory which has since been set to null and deallocated in another thread's cache, a fact which eliminates the need for locking on the basis of concurrent memory safety. Before commiting its speculative transaction, a thread in `speculative_search()` just needs to modify the lock variable of the result node and its parent to mimic the scenario in which it had actually traversed the entirety of the trie, locking each node along the way. Thus a thread, which successfully commits the transaction, will have traversed the trie having seen a state consistent with a linearizable execution while not excluding other threads from operating on those same nodes along its path. This property allows for considerably more parallelism than we would observe in a method in which other threads are barred form accessing nodes through mutual exclusion during concurrent traversals.

Should `speculative_search()` fail to commit, we may retry the search operation for a set number of times before falling back to a serialized spinlock-based implementation `locking_search()`. The retry policy, as it applies to each type of abort, reflects the likelihood of another attempt committing in a style based on Avni and Kuszmaul's implementation in [1]. For instance, given the abort code explanations in [14], if encouraged to retry by an `_XABORT_RETRY` abort code, it would be prudent to make another attempt, but an `_XABORT_CAPACITY` abort code indicates the read set may have exceeded the hardware limits, and another attempt would be wasteful. The initial default configuration allows for a maximum of ten retries per speculative critical section as a worst case. Listing 4 describes our retry procedure in detail by showing the source code for `search()`; `insert_leaf()` and `delete_leaf()` follow a similar prototype.

The `search()`, `insert_leaf()` and `delete_leaf()` functions composed into `find_string()`, `add_string()` and `remove_string()` operations commit their own transactions and have their own retry policies. Despite requiring two speculative critical sections for modify operations like `add_string()` and `remove_string()` and creating a kind of globally consistent waypoint between the two phases, this extra step allows us to decouple our retry policy between the read and modify sections of trie operations, and allows for performance tuning covered in Section 5.2.

4.3 Choice of Memory Allocator

The lack of standard library transaction-friendly memory allocation has until very recently been an issue preventing the wide application of hardware transactional memory to dynamically sized data structures [17]. Most memory allocators, including the default allocator in the version of `glibc` available on our test system at the time of writing (see Appendix), make frequent system calls to `brk()` and `sbrk()`, potentially trapping to the operating system through a fault handler in order to acquire more space on the heap. Such behavior, however, causes any speculative execution at the time of the system call to abort. Thread caching allocators are a proposed [17] [23] solution to this issue in that a thread may request more heap space from a private cache of free chunks without having to explicity invoke operating system facilities for small allocations. Below we detail the performance of the transactional binary alphabet implementation of the trie when paired with the standard `glibc malloc` implementation, and two thread-caching allocators, FreeBSD's `jemalloc` [7] and Google's `tcmalloc` [10]. Our test has eight threads operating on a set of initially 1,000,000 128-bit random keys with 100% `add` operations. All transactional aborts, except conflict and capacity aborts, are retried ten times.

memory allocator	mean ops / sec	mean abort rate
`glibc` 2.13	273,541	100.0%
`jemalloc` 3.0.0	732,995	100.0%
`tcmalloc` 2.0	839,010	58.2%

Across all three allocators, the majority of aborts were due to either capacity aborts (either traversing the trie itself or the underlying chunk layout data structure in the system's memory allocator) or unspecified aborts (presumably related to trapping to the operating system), with approximately four times as many unspecified aborts as capacity aborts. `glibc malloc` and `jemalloc`, which both had a 100.0% abort rate, differed in that `jemalloc` posted approximately four times the number of aborts as `glibc malloc`, which suggests that aborts occurred earlier in execution although ultimately failing. `tcmalloc` is the only allocator which performed successful allocations in speculative execution, although its abort rate was still quite high.

A method of analysis suggested by Doeppner in [6] to track the frequency of system calls used by concurrent memory allocators with `strace` proves very useful here. The primary trapping system calls that may be at fault for unspecified aborts in the memory allocator include `futex()`, `mmap()` and `brk()`. A call to `futex()` may only trap to the operating system if the mutex in question is under contention and has a wait queue already established, but it will otherwise remain in user mode (hence *Fast Userspace muTEX*). `mmap()`, which is primarily used for large changes to the process heap, and `brk()`, which is used to for smaller increases in heap size, are both prone to triggering page fault interrupts. If we monitor the number of system calls invoked using a non-transactional spinlock execution with eight threads (`strace` does not seem to operate correctly with TSX programs) and compare the system call activity of a binary linked against `glibc`'s `ptmalloc` to that of one linked against Google's `tcmalloc`, we can see the following trend.

memory allocator	futex() calls	mmap() calls	brk() calls
`glibc` 2.13	22	50	6196
`tcmalloc` 2.0	3912	301	409

Although in this test there were many more calls to `futex()` in the `tcmalloc` version, it is very likely that a thread will only be accessing its own cached heap's mutex and thus there will be no wait queue and no need to trap to the kernel to involve the scheduler. The more telling statistic is the fact that `glibc`'s allocator made over fifteen times as many trapping calls to `brk()`. Instead of small incremental increases to the heap, `tcmalloc` allocated large amounts

```
pt_result_t search(pt_t *trie, string_t *query_string)
{
    // Transactional memory variables.
    int retries, status;

    // Algorithm variables.
    pt_result_t result;

    // Entry point for speculative execution.
    retries = 0;
    search_retry:

    // Begin speculative execution.
    status = _xbegin();

    // --- RTM Fast Path ---

    if (status == _XBEGIN_STARTED) {
        result = speculative_search(trie, query_string);
        _xend();

    // --- Spinlock Slow Path ---

    } else {
        // Retry the transaction under certain conditions if the retry count has not been exceeded.
        if (status == _XABORT_RETRY) {
            retries++;
            if (retries <= NUM_RETRIES_SEARCH)
                goto search_retry; // Always retry if encouraged to do so.
        } else if (status == _XABORT_EXPLICIT && _XABORT_CODE(status) == 0xff) {
            retries++;
            if (retries <= NUM_RETRIES_SEARCH)
                goto search_retry; // One of the teleported locks was busy, so always retry.
        } else {
            if (status != _XABORT_CONFLICT && status != _XABORT_CAPACITY) {
                retries++;
                if (retries <= NUM_RETRIES_SEARCH)
                    goto search_retry; // Unspecified abort, so always retry.
            }
        }

        // If we get here, we should just perform hand-over-hand locking.
        result = locking_search(trie, query_string);
    }

    return result;
}
```

Listing 4. `search()`.

of memory in the form of calls to a `mmap()` to divide among the threads as needed, and crucially it needed to do this far less often than `glibc`'s `ptmalloc` called `brk()`. In short, `tcmalloc` seems far less likely to trap to the operating system and abort a transaction. In light of these results, we will use `tcmalloc` as the baseline allocator against which to link the test binaries.

4.4 Baseline Performance

As seen in Figure 4, the transactional implementation matches the scaling of its slow-path spinlock implementation at thread counts below the number of available hardware contexts, but unlike the the spinlock version, its throughput does not collapse as the number of concurrent threads increases. In fact for all concurrent thread counts less than thirteen, its performance surpasses that of all other synchronization methods outright. Since there does not appear to be a notable increase in abort rate at the highest thread counts, some of the decline in throughput may be due to context switching overhead and thrashing that is difficult to quantify in the absence of a TSX-enabled profiler (see Appendix).

All lock-based schemes use standard POSIX thread locking facilities, but the adaptive mutex lock is a special, non-portable form of mutex available in `glibc` which can switch dynamically between exponential backoff spinlock behavior and traditional wait queue mutex behavior [18]. We will show later that, as it turns out, the hexadecimal alphabet shown here is the best among those considered, but there are further benefits that can be gained from modifying the retry strategy.

5. Areas for Optimization

5.1 Alphabet Size

Using a baseline number of retry attempts with ten attempts for read operations and ten for write operations, we may assess the throughput of each alphabet size superficially in Figure 5 before looking at the primary sources of performance variation. There is a known correlation between alphabet size and tree depth already discussed in Section 1.4, and following that line of logic without considering cache friendliness would suggest the optimality of large alphabets. The superior throughput of moderate alphabet sizes observed in Figure 5, however, shows that a higher rate of L1 and L3 cache misses incurred by large alphabets is not negligible. We have simulated the cache behavior of various tries with `cachegrind` profiling tool [25] using our default spinlock imlementation with eight concurrent threads.

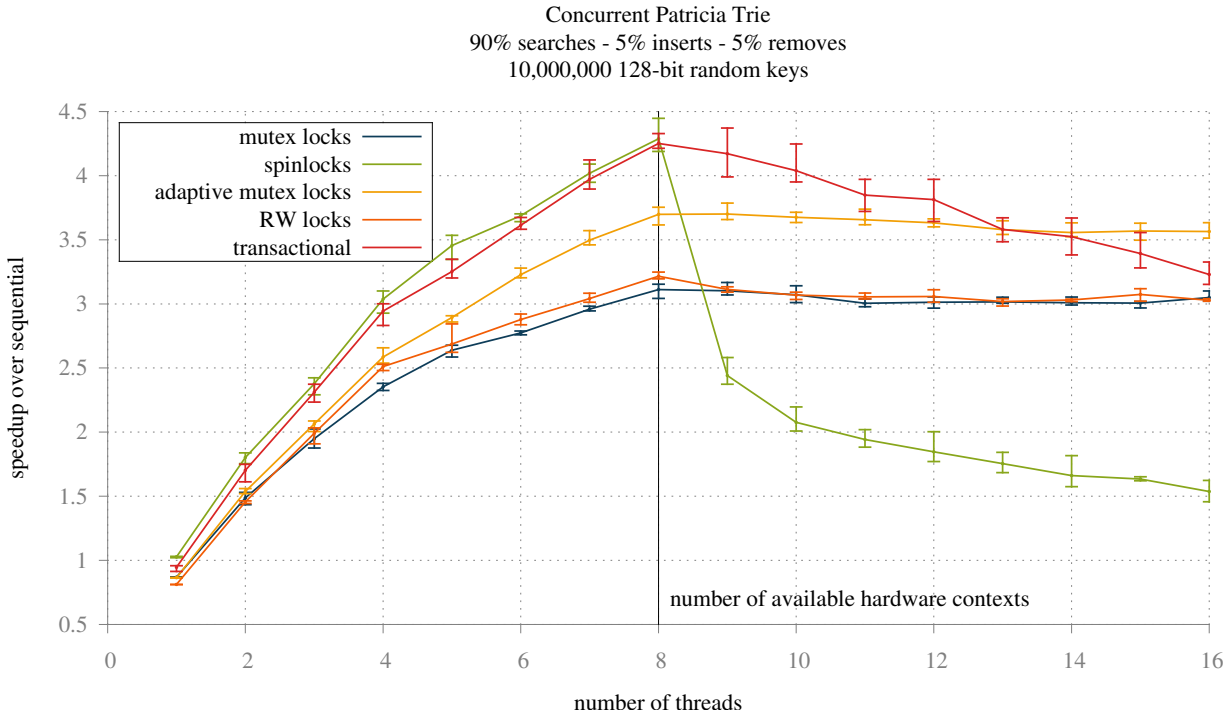

Figure 4. Baseline configuration of the transactional algorithm compared with other locking synchronization primitives. Data points are the mean throughput over ten runs in each configuration, and error bars reflect the minimum and maximum throughput recorded. Note the inflection in throughput when the number of OS threads exceeds the available number of hardware threads on the system.

alpha. size	L1 D-cache miss rate	L3 D-cache miss rate
2	2.4%	0.7%
4	3.8%	1.3%
16	4.1%	1.8%
256	7.4%	5.6%

Since a last-level cache miss on modern architectures may take several hundred times longer than an L1 hit, it follows that the high L3 data cache miss rate we observe with the byte alphabet may have an unduly large impact on throughput figures. It is also stands out that the poor cache behavior of the byte version is not consistent: looking at the error bars, there are executions in which it attains 14 million operations per second, which is well above the performance of other alphabet tries. That said, the significant drop in mean performance between the hexadecimal and byte alphabets at eight threads is still mainly due to the much larger node and data structure size we see with larger alphabets in the table below.

alpha. size	node size	trie size with 1,000,000 128-bit keys
2	48 B	107,963,659 B
4	64 B	114,748,505 B
16	160 B	225,471,634 B
256	2,080 B	2,284,996,369 B

Since the hexadecimal alphabet version of the trie offers the best performance at full hardware thread residency (eight threads) and shows only modest throughput deterioration at higher multi-programming levels, we will use it as the basis of futher retry policy optimizations in the following section. A strange observation, however, is that the hexadecimal alphabet despite having the highest absolute throughput, does not have the consistent performance of smaller alphabets as the number of threads increases. This seems to be more pronounced on less populated tries (1,000,000 initial keys in Figure 5 as opposed to 10,000,000 used in Figure 4). The rationale behind the smaller initial key population in the alphabet comparison is entirely due to memory constraints: the byte alphabet trie is simply too memory intensive to populate more fully.

5.2 Retry Policy

Having separate retry policies for `search()`, `insert_leaf()` and `delete_leaf()` lets us vary the number of retry attempts allowed for a given task as seen fit. In order to define a retry policy design space, we have divided the basic operations broadly into two categories: a *read* operation, `search()`, and two *modify* operations, `insert_leaf()` and `delete_leaf()`. Read operations are not able to invoke the memory allocator and only interact with existing objects allocated on the heap, while modify operations may invoke the memory allocator, although they are not guaranteed to do so. This distinction allows us to tune our retry policy to minimize potential interference from the memory allocator, and represents a new approach to dealing with dynamically sized data structures which support hardware transactions. We explore the correlation between retry rate policies in a realistic (90% search, 5% insert and 5% remove) database workload on the set.

5.2.1 Database Workload (90-5-5), 8 Threads

With eight threads there is little correlation between retry policy, throughput and abort rate.

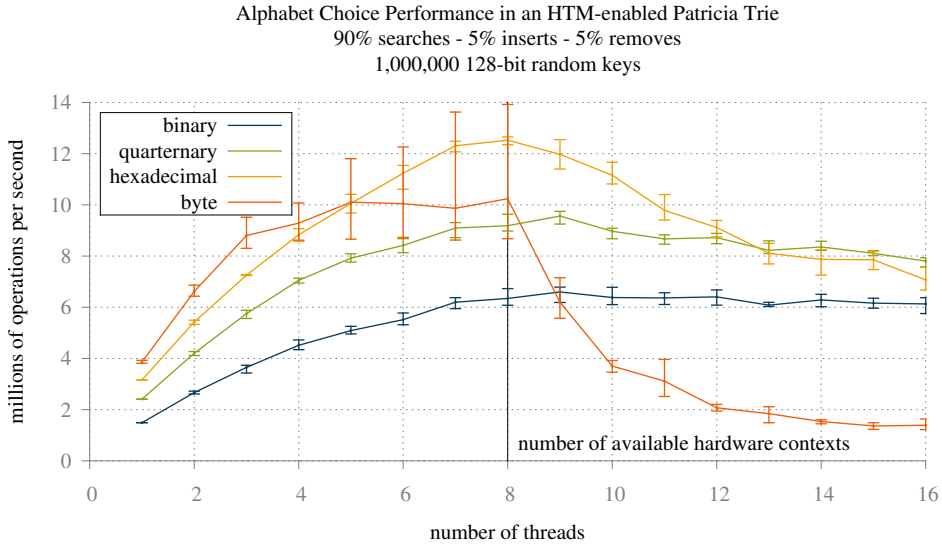

Figure 5. Alphabet size scaling comparison with ten retries each for both read and modify operations. Data points are the mean throughput over ten runs in each configuration, and error bars reflect the minimum and maximum throughput recorded. Note the inflection in throughput when the number of OS threads exceeds the available number of hardware threads on the system.

Constant: no retries per modify operation.

read retry rate	mean ops / sec	mean abort rate
0	9,607,605	25.0%
2	9,718,439	23.9%
4	9,883,110	23.1%
6	9,990,135	22.9%
8	9,865,361	23.0%
10	9,836,080	23.5%

Constant: no retries per read operation.

modify retry rate	mean ops / sec	mean abort rate
0	9,508,619	24.8%
2	9,445,998	26.4%
4	9,492,777	25.1%
6	9,346,614	26.7%
8	9,385,468	26.2%
10	9,199,493	28.2%

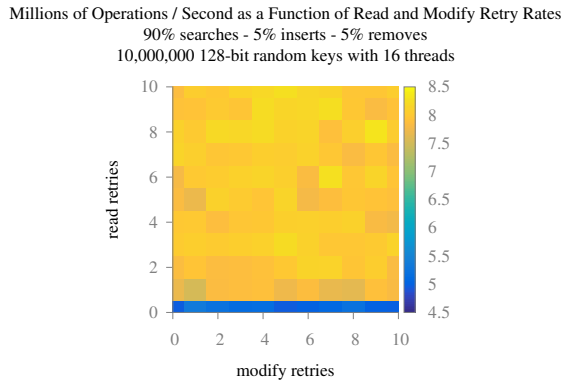

Figure 6. Heatmap visualization of throughput and retry rate dependence with less than ten retry attempts.

5.2.2 Database Workload (90-5-5), 16 Threads

At sixteen threads, however, there is a somewhat stronger correlation between the number of read operation retries and the throughput as seen in Figure 6. For instance, along the modify retry rate $=$ 5 line, average throughput goes from 4,305,519 operations per second with an abort rate of 34.5% at $(5, 0)$ to 7,908,826 operations per second with an abort rate of 22.1% at $(5, 10)$. Maximal improvement from the baseline retry policy's $(10, 10)$ throughput of 7,806,431 occurs at $(8, 9)$ with 8,339,432, which represents a 6.8% increase over the baseline. Although the data has considerable noise and the improvement over the baseline not overwhelming, such improvements at all suggest there is the potential to tailor retry to the multiprogramming level to augment performance.

6. Conclusions

We have introduced a concurrent, dynamically sized implementation of the Patricia trie that uses Intel's TSX instruction extensions as a means for improving shared memory multiprocessing support. The TSX-enabled version yields a significant improvement in performance at high multiprogramming levels compared with other synchronization primitives under realistic database workloads, and we attempted to mitigate the negative affects of OS-level memory management by isolating retried modify operations with their own local policy, which also brought some benefit over the baseline.

An extended version of this project might focus more on scaling behavior under high-contention workloads with short keys and see if the orthogonal retry policy knobs give more leverage in that scenario. It might also be interesting to see if the more stable throughput of smaller alphabet tries seen in Figure 5 holds up in these circumstances.

We have only performed a basic exploration of the space of possible retry strategies, but it is not hard to imagine situations where there may be interactions between read and modify retry strategies that are not so clear. To address such situations, it might be useful to use heatmaps like the one exemplified in the retry/throughput

chart in Figure 6. More rigorously, there are a variety of tools from multivariate optimization that may be applicable which are beyond the scope of the analysis presented here. The ability to modify retry policy at runtime also presents the opportunity for dynamic workload characterization that may point to optimal retry policies under certain conditions in a form of self-tuning data structure.

As it seems that most hardware transactional memory systems will be best-effort for the foreseeable future, developers are left to speculate about the causes of unspecified transactional aborts, but analysis of this kind can provide insight into factors affecting performance on a case-by-case basis. More generalized insight into the interaction between runtime memory allocation and hardware transactions is an open area of future research that will become more feasible as future transactional hardware becomes more stable and standard library support improves. A potential path for future inquiry might be the characterization of the transaction-friendliness of a variety of allocators including more recent versions of `glibc` (see Appendix) under a less data-structure-centric workload.

Appendix

The most efficient way to profile a TSX-enabled application is through the `perf` profiling tool, which allows access to hardware counters with minimal performance impact on the program under test [19]. The version of GNU `binutils` containing a TSX-aware version of the `perf` utility, however, requires a more recent version of the Linux kernel than the 3.10.11 kernel on the test system. We were reluctant to update the kernel on the test system in light of recent widely published bug in the RTM implementation on consumer Haswell chips [15]. Intel has been disabling RTM functionality on many active Haswell processors through microcode updates due to the hardware errors, and updating the test system past this kernel version did not seem prudent given the the risk of disabling the feature altogether. To circumvent this issue, we relied on runtime abort and commit rate accounting that can potentially impact throughput values, so figures and results comparing throughput and abort/commit rates are decoupled, and the collection of the two data points was performed in different executions on modified binaries which enabled thread-local performance metric tracking.

Simliarly, all cache hit and miss rate figures were provided by simulating the spinlock implementation in the `cachegrind` environment when it would have been simpler and more accurate to use `perf` alongside the transactional TSX-enabled binary.

A secondary side effect of using an older kernel version was the inability to use a more recent version of `glibc` ($\geq$2.15) in which experimental support for RTM allocations has been enabled. We suspect that the results using this more recent version with the improved `ptmalloc` implementation may be similar to those observed with `tcmalloc` in this study.

Acknowledgments

I thank my advisor Prof. Maurice P. Herlihy for his time and commitment to my growth as a student. He would make sure I understood the nuances of the problems at hand and gave me the opportunity to learn by doing, and I deeply appreciate his patience and guidance. I would also like to thank Prof. Thomas W. Doeppner for generously sharing his knowledge of and depth of experience with the POSIX threads API and for fruitful discussions. I greatly appreciated the assistance of Irina Calciu and Elias Wald while setting up the test environment as well. Outside of the Department, Prof. Sherief M. Reda and Prof. R. Iris Bahar in the School of Engineering went out of their way to introduce me to the study of computer architecture, and for that I am very grateful.

References

[1] H. Avni and B.C. Kuszmaul. "Improving HTM scaling with consistency-oblivious programming." *TRANSACT* 6 (2014): 5.

[2] M. Boehm et al. "Efficient in-memory indexing with generalized prefix trees." *BTW.* Vol. 180, 2011.

[3] P. Brass. *Advanced Data Structures*. Cambridge: Cambridge University Press, 2008.

[4] N.G. Bronson et al. "A practical concurrent binary search tree." *ACM SIGPLAN Notices.* Vol. 45. No. 5. ACM, 2010.

[5] J. Corbet. "Trees I: Radix trees," (Linux kernel data structures). Linux Weekly News. `http://lwn.net/Articles/175432/`.

[6] T.W. Doeppner. *Computer Science 0330: Introduction to Computer Systems*. Brown University, Fall Semester 2014. `http://cs.brown.edu/courses/csci0330/docs/lectures_f14/34Threads5.pdf`.

[7] J. Evans. "A scalable concurrent malloc (3) implementation for FreeBSD." *Proc. of the BSDCan Conference, Ottawa, Canada.* 2006.

[8] P. Ferragina and R. Grossi. "The string B-tree: a new data structure for string search in external memory and its applications." *Journal of the ACM (JACM)* 46.2 (1999): 236-280.

[9] E. Fredkin. "Trie memory." *Communications of the ACM* 3.9 (1960): 490-499.

[10] S. Ghemawat and P. Menage. "tcmalloc: Thread-caching malloc." `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[11] R.A. Haring et al. "The IBM Blue Gene/Q compute chip." *Micro, IEEE* 32.2 (2012): 48-60.

[12] M. Herlihy and J.E.B. Moss. "Transactional memory: Architectural support for lock-free data structures." *ACM SIGARCH Computer Architecture News* Vol. 21. No. 2. ACM, 1993.

[13] M. Herlihy. *Computer Science 1760: Introduction to Multiprocessor Synchronization*. Brown University, Fall Semester 2014. `http://cs.brown.edu/courses/cs176/lectures/chapter_18.pptx`.

[14] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2013.

[15] Intel Corporation. *Desktop 4th Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family: Specification Update (Revision 014)*, 2014.

[16] C. Jacobi et al. "Transactional memory architecture and implementation for IBM System z." *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on. IEEE,* 2012.

[17] A. Kleen. "Adding lock elision to Linux." *Linux Plumbers Conference,* 2012.

[18] A. Kleen. "Lock elision in the GNU C library." Linux Weekly News. `https://lwn.net/Articles/534758/`.

[19] A. Kleen. "TSX anti-patterns in lock elision code." Intel Software. `https://software.intel.com/en-us/articles/tsx-anti-patterns-in-lock-elision-code`.

[20] D. Lea and W. Gloger. "A memory allocator," 1996.

[21] D. Lea and W. Gloger. `malloc-2.7.2.c`, comments in source code, 2001.

[22] V. Leis et al. "The adaptive radix tree: ARTful indexing for main-memory databases." *Data Engineering (ICDE), 2013 IEEE 29th International Conference on. IEEE*, 2013.

[23] V. Leis et al. "Exploiting hardware transactional memory in main-memory databases." *Data Engineering (ICDE), 2014 IEEE 30th International Conference on. IEEE*, 2014.

[24] D.R. Morrison. "PATRICIA: practical algorithm to retrieve information coded in alphanumeric." *Journal of the ACM (JACM)* 15.4 (1968): 514-534.

[25] N. Nethercote and J. Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." *ACM SIGPLAN Notices. Vol. 42. No. 6. ACM*, 2007.

[26] R. Sangireddy, et al. “Scalable, memory efficient, high-speed IP lookup algorithms.” *Networking, IEEE/ACM Transactions on* 13.4 (2005): 802-812.

[27] N. Shafiei. “Non-blocking Patricia tries with replace operations.” *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on. IEEE*, 2013.

[28] H. Shang and T.H. Merrettal. “Tries for approximate string matching.” *Knowledge and Data Engineering, IEEE Transactions on* 8.4 (1996): 540-547.

[29] N. Shavit and D. Touitou. “Software transactional memory.” *Distributed Computing* 10.2 (1997): 99-116.

[30] M. Waldvogel et al. “Scalable high speed IP routing lookups.” *ACM SIGCOMM Computer Communication Review* Vol. 27. No. 4. ACM, 1997.