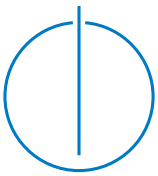
FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Scalable Database Concurrency Control using Transactional Memory

Martin Schrimpf



FAKULTÄT FÜR INFORMATIK

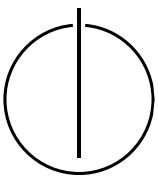
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Skalierbare Datenbanknebenläufigkeitskontrolle mittels transaktionalem Speicher

Scalable Database Concurrency Control using Transactional Memory

Author:	Martin Schrimpf
Supervisor:	Univ.-Prof. Alfons Kemper, Ph.D.
Advisor:	Prof. Uwe Röhm, Ph.D.
Submission Date:	15.07.2014



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Sydney, 25th of July 2014	MS
Place and Date	Martin Schrimpf

Acknowledgements

I want to express my great appreciation to Professor U. Roehm for providing me with continuous support throughout the whole project - the collegial atmosphere was a fantastic experience. My grateful thanks are extended to Professor A. Kemper for making it possible to work on this thesis in Sydney and for providing us with the necessary hardware. Also thank you Dr. V. Leis for the useful tips and the help related to the servers. Moreover, I also want to thank Professor A. Fekete and Dr. V. Gramoli for welcoming me in Sydney.

Finally, I wish to thank my dad for his support and encouragement throughout my study despite the distance of over 16 thousand kilometres.

Abstract

Intel recently made available the optimistic synchronization technique Hardware Transactional Memory (HTM) in their mainstream Haswell processor microarchitecture.
The first part of this work evaluates the core performance characteristics of the two programming interfaces within Intel's Transactional Synchronization Extensions (TSX), Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). Therein, a scope of application is defined regarding inter alia the transaction size which is limited to the L1 DCache or even less with wrongly aligned data due to cache associativity, the transaction duration restricted by Hardware interrupts and a limit to the nesting of transactions. By comparing common data structures and analyzing the behavior of HTM using hardware counters, the Hashmap is identified as a suitable structure with a 134% speedup compared to classical POSIX mutexes.
In the second part, several latching mechanisms of MySQL InnoDB's Concurrency Control are selected and modified with different implementations of HTM to achieve increased scalability. We find that it does not suffice to apply HTM naively to all mutex calls by using either HLE prefixes or an HTM-enabled glibc. Furthermore, many transactional cycles often come at the price of frequent aborted cycles which inhibits performance increases when measuring MySQL with the tx-bench and too many aborts can even decrease the throughput to 29% of the unmodified version.

Contents

1 Introduction **1**

2 Intel TSX **3**

2.1 Restricted Transactional Memory 5

2.2 Hardware Lock Elision 7

2.3 Linux perf Tool 10

2.3.1 Preliminaries 10

2.3.2 Basic profiling 10

2.3.3 Event profiling 12

2.3.4 Recording and reporting profiles 14

2.4 Micro-Usage of HLE 15

2.4.1 Lock variable type 16

2.4.2 HLE function call 18

2.4.3 Locking algorithm 19

2.4.4 TAS implementation 22

2.4.5 Resulting combined function 23

3 Evaluation of Core Performance Characteristics **25**

3.1 Necessity to keep the mutex in the read-set 25

3.2 Scope of application 26

3.2.1 Transaction size 26

3.2.2 Cache Associativity 29

3.2.3 Transaction duration 33

3.2.4 Transaction Nesting 35

3.2.5 Overhead 37

3.3 Isolated use cases 38

3.3.1 Closed banking system 39

3.3.2 Shared counter 41

3.3.3 Doubly linked List 44

3.3.4 Hashmap 47

3.4 Linking with a HTM-enabled glibc . . . 51
3.4.1 Installation . . . 51
3.4.2 Usage . . . 52

4 Database Concurrency Control using Intel TSX 55
4.1 InnoDB internals . . . 57
4.1.1 Multi-granularity locking . . . 58
4.1.2 Transaction locks . . . 58
4.1.3 Function hierarchy . . . 59
4.2 Modifications to apply HTM . . . 62
4.2.1 Targeted functions . . . 63
4.2.2 RTM implementation with fallback path . . . 63
4.2.3 Relinking with the HTM-enabled glibc . . . 66

5 Evaluation of HTM in MySQL/InnoDB 67
5.1 MySQL configuration . . . 67
5.2 Benchmarking with the txbench . . . 68
5.3 Comparison and analysis of the applied changes . . . 69

6 Conclusion 77
6.1 Conclusions on Intel TSX and its application in a Database Concurrency Control . . . 77
6.2 Lessons learned . . . 78

Appendix A Environment 81

Appendix B Technical pitfalls 85

Appendix C Omitted Benchmarks and figures 87

Appendix D Open Questions 101

Bibliography 107

CHAPTER 1

Introduction

"The future is parallel". This statement by Flynn and Rudd [1] in the year 1996 was a broad prediction for what would happen in the next few years in the computer industry. At the present day, Intel has decided to cancel its 4 GHz single-core developments and embrace multicore CPUs [2] which is a movement that holds true for the other major chip makers such as AMD and IBM as well [3], [4]. Earlier, computing performance has mostly been driven by decreasing the size of chips while increasing the number of transistors. "In accordance with Moore's law, this has caused chip speeds to rise and prices to drop" [3]. Due to physical limitations in the semiconductor technology, it is difficult however to continue shrinking the transistors [4], [5]. Marc Tremblay, former chief architect for Sun Microsystems once noted that "We could build a slightly faster chip, but it would cost twice the die area[1] while gaining only a 20 percent speed increase" [3]. Hence, it seems reasonable to "build chips with multiple cooler-running, more energy-efficient processing cores instead of one increasingly powerful core" although each single core is often not as powerful as a single-core model, their ability to handle work in parallel leads to an increased overall performance [3], [4], [6].

To parallelize a program, imperative languages usually allow the developer to start threads for concurrent execution. However, examples such as the banking problem illustrate issues that arise with parallel programs: consider a banking system where the two threads Alice and Bob share the same account with an initial balance of 1000$ and both want to deposit money. Firstly, both threads read the value of the account and store it in a local variable each. Then Alice makes a deposit of 100$, thus she adds this value to the stored value of the account and then writes the result back to the account that will then show a value of 1100$. Bob wants to deposit 50$ so he executes the same steps: add the deposit to the stored value of the account (1000$) and set the account's value to the result of 1050$. This leads to the loss of the 100$ that Alice payed in initially which has its cause in the lack of communication between the two parties (threads).

To solve this problem, one of the first papers regarding critical sections has been published in 1965 by Dijkstra [7]. He proposed a mutual exclusion algorithm that allowed only a single computer to be in

[1] semiconductor block on which an integrated circuit is fabricated

the critical section at a time which set the foundation for current mutexes that are e.g. implemented in the POSIX[2] standard. Other techniques use atomic instructions such as compare and swap or test and set [8] that guarantee for only one thread executing the atomic instruction. Using these atomic instructions, lock-free data structures can be implemented that allow to parallelize processes without the necessity to lock critical regions [9]. Java has its synchronized methods and monitors that allow for a high-level protection of sections that the Java Virtual Machine will then carry out in a specific implementation hidden from the programmer [10].

A different approach is transactional memory that allows the programmer to define critical regions without the requirement of implementing a complex locking system while maintaining or outperforming the performance of other locking techniques [11], [12].
So far, transactional memory has mostly been implemented in software due to the lack of mainstream hardware support. Yet, Software Transactional Memory (STM) has shown convincing performance results [13]–[15]. With the Intel Transactional Synchronization Extensions in the recently released Haswell processor, Hardware Transactional Memory (HTM) has become realistically available [16]–[18]. First analyses of this technique have shown how HTM can increase the performance of an application: Yoo, Hughes, Lai, *et al.* [17] have achieved a 1.4x speedup with high-performance computing workloads, Leis, Kemper, and Neumann [19] implemented it in main-memory databases and obtained nearly lock-free processing performance. Matveev and Shavit [20] implemented a hybrid algorithm with partly short hardware transactional paths where the HTM implementation of a Red-Black tree gained an approximate 10 times speedup compared to TL2 STM[3]. Log-based Transactional Memory has been proposed by Moore, Bobba, Moravan, *et al.* [22] which addresses the problem of many aborts inside transactions [23] and achieved a 4 times speedup compared to the default locks in the SPLASH-2 benchmark[4]. Similarly, Levy [24] addressed performance problems in the HLE implementation of HTM caused by transactional aborts by serializing only conflicting threads and improved the performance on STAMP[5] by 3.5 times compared to default Haswell HLE. Finally, Kleen *et al.* [25] implemented a glibc version using HTM so that existing applications can be re-linked to support HTM without any change in the source-code or the need to re-compile.

This work describes HTM in the Intel Transactional Synchronization Extensions and its usage in two different programming interfaces as well as the useful Linux profiling tool perf. We then evaluate several micro benchmarks to find the application area, followed by isolated use cases of data structures that show how data layout affects the performance. Ultimately, we choose the Concurrency Control in MySQL InnoDB for more complex benchmarks, reimplement the internal locking structures using HTM and evaluate the results with regard to profiling results.

[2]Portable Operating System Interface: defines the interface between operating systems and applications
[3]TL2: Software Transactional Memory algorithm based on a combination of commit-time locking and a global version-clock validation technique [21]
[4]SPLASH-2: a benchmark suite for parallel applications
[5]STAMP: Stanford Transactional Applications for Multi-Processing, a benchmark suite designed for Transactional Memory research

CHAPTER 2

Intel TSX

Since Transactional Memory allows for a dynamic determination whether critical sections of threads need to be serialized or not, it supposedly allows for the simplicity of coarse-grain locking at the performance of fine-grain locking [17], [26], [27]. In terms of implementation, the realization in hardware has some benefits over a software approach: the imposed overhead is much lower which in turn leads to a better performance, concurrency works at cache line granularity instead of object granularity [28], [29] and having a mainstream implementation in the Intel Haswell processor gives some sort of guarantee that HTM will stay around for a while [18].

With the new Haswell generation, the processor keeps track of the data that is accessed, i.e. read or modified, in critical regions [30], [31]. Such an execution of a critical region is also referred to as a transaction which implies the same as a database transaction where either everything is committed or everything fails [32, p. 321]. A transaction can thus be defined as "a sequence of operations that perform a single logical function" [33]. All updates inside a transaction are buffered by the hardware and will either be atomically committed at a later point or aborted [27] whereas during the execution, the hardware does not make any updates visible to other threads [34]. If conflicts occur within a transaction because other threads execute the same or a different transaction that accesses the same data, all but one transactions must be aborted and only a single transaction may commit. Ideally, this allows multiple threads to execute in parallel when no communication is required instead of executing serialized with a lock variable as shown in Figure 2.1.
To implement such a behavior, the existing cache and memory coherency protocol MESIF [36] can be utilized where the processor already keeps track of cache line conflicts between multiple cores [1] [28], [29, p. 3]. A data conflict occurs if either one thread modifies a cache line that another thread has accessed (read or write) or vice-versa if a thread accesses a cache line that another thread has modified [24], [29]. The thread that detects the conflict must then transactionally abort which leads to the rollback of the transaction [31]. A rollback can be expensive and issues with too many

[1] Although Intel did not make a definite statement over how HTM is implemented so far, these assumptions are quite straight-forward and very likely to be true [29, p. 3]

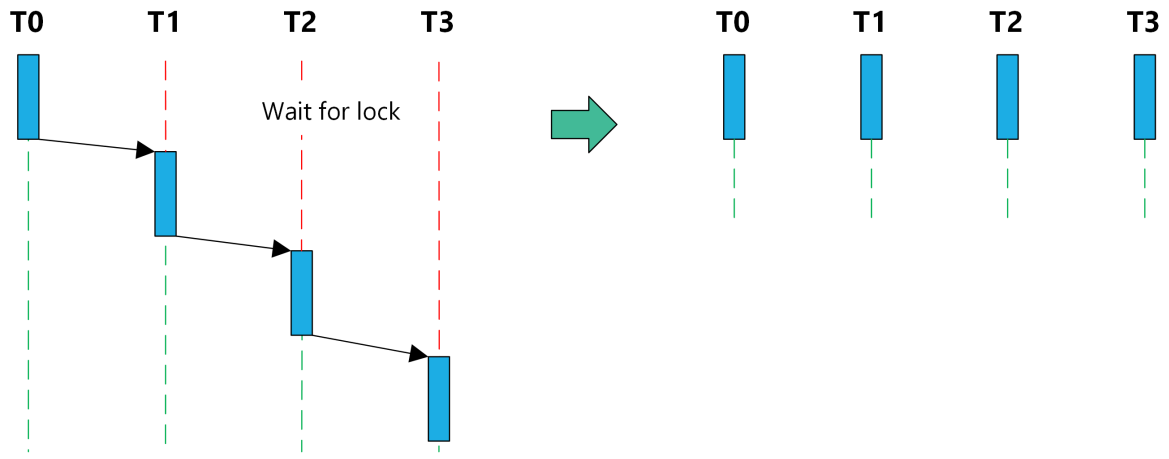


Figure 2.1: Serial (left) and transactional execution (right) [35]

rollbacks in HTM have been identified [22], [24]. Furthermore, cache lines that are read but not written in multiple transactions do not present a data conflict since nothing has been modified. Therefore, the processor keeps track of the 64 Byte cache lines in two separate sets, one for modified data and one for data that has only been read. This is done by monitoring the addresses of data inside a transactional region and is generally limited to the first level data cache [18], [29]–[31], [34], [37] which imposes a resource limit that STM does not have [28]. Since data might be more fine-granular than a cache line or overlap multiple cache lines, false conflicts can happen: if, for instance, two byte-values are stored in the same cache line and two transactions each modify a different byte-value, the processor recognizes this as an abort because it only knows about the cache line but not about the actual values inside it [37]. This problem can be solved partially by padding data - in our example we would put the byte-values inside a class (object-oriented) or a struct (C) that holds 63 more bytes and hence sums up to a total of 64 byte together with the actual value. Although instances of the modified structure are then 64 byte big, they also need to be aligned to begin at an address that is a multiple of the size of a cache line because otherwise the instance could just start in the middle of a cache line and thus overlap with other instances again. Moreover, with Hyper-Threading, the L1 cache is shared between two threads on the same core and an abort in one of the two threads can thus lead to an eviction in the other thread on the same physical processor [31, p. 12-2].
Besides data conflicts, there are several other possibilities of why an abort can occur: any system or I/O calls (e.g. `PAUSE` or `CPUID`), context switches, overflows and somewhat random interrupts will lead to a transactional abort [27], [29], [34], [38]. Aborts also occur when (de-) allocating memory with `new/delete` and `malloc/free` [2].

However, if no conflicts occur, critical sections can be executed concurrently where a mutex-based system might not allow such behavior due to a too coarse-grained lock.

[2] Also see Appendix Section C.1 for an evaluation of the (de-) allocation of memory

To use HTM, the Intel Transactional Synchronization Extensions provide two software interfaces to mark code regions for transaction execution: Hardware Lock Elision (HLE) adds instruction prefixes to existing lock calls whereas Restricted Transactional Memory (RTM) is a new instruction set interface [18], [29].

2.1 Restricted Transactional Memory

The RTM instruction set interface provides four basic operations [16], [18], [27], [29]:

_xbegin begins an elided transaction

_xend ends an elided transaction

_xtest tests whether the code is executed transactionally

_xabort aborts an elided transaction

To elide a critical region with RTM, we mark its beginning `_xbegin`. This function returns an integer code signaling the status of the elision - we have only entered an elision if the return value is equal to `_XBEGIN_STARTED`. This value can be set to one of the default abort reasons but it can also be passed via in the argument of the `_xabort` function [30], [39], [40]. Table 2.1 shows the meaning of return values as defined in the *Intel® C++ Compiler XE 13.1 User and Reference Guides* [30].

EAX register bit	Meaning
0	abort caused by `_xabort`
1	transaction may succeed on retry
2	data conflict
3	internal buffer overflow
4	debug breakpoint was hit
5	abort due to nesting

Table 2.1: Excerpt of RTM codes [30]

Adding to Table 2.1, bits 23:6 are reserved and bits 31:24 represent the `_xabort` argument. We further found that an interrupt does not set any bits[3], hence the return value for aborts caused by an interrupt is 0.
Finally, the end of a transaction is marked with `_xend` as shown in Listing 2.1.

Listing 2.1: RTM control flow

```
1 if(_xbegin() == _XBEGIN_STARTED) {
2   // perform elided transaction ...
3   _xend();
```

[3]The interrupt benchmark can be found in Section 3.2.3

```
4 } else { // aborted
5   /* fallback path... */
6 }
```

An important aspect to understand here is that the very first call of `_xbegin` will always return `_XBEGIN_STARTED` but if an abort such as data conflicts or an interrupt occurs before the `_xend`, the program jumps back to the start of the transaction, `_xbegin`, which will then return a value unequal to `_XBEGIN_STARTED`. The program does not know about this jump however, since all written variables inside the transaction are aborted [27], [41].

The fallback-path can naively be implemented with a simple retry so that the elision is started again after it aborted. But because RTM has no forward guarantee (the worst-case of a transaction that never elides can always happen), one should provide a maximum amount of retries and still define an alternative fallback path when this maximum has been reached or in the case of little tests at least output an error message.
With any fallback path, one needs to make sure that it correctly interoperates with any transactional execution. A simple way to do so when using locks is to put the lock variable in the read-set and have the fallback write to the lock variable which aborts all transactions [42].
Moreover, generalized lock/unlock methods can be implemented - Listing 2.2 shows the combination of a lock/unlock function with a retry routine. Therein, the elided path is attempted again if retries are available and the elision might succeed on a retry which is indicated by a set second bit of the `_xbegin` return value.

Listing 2.2: Generalized RTM lock/unlock

```
1  void rtm_lock() {
2    int failures = 0, max_retries = 123, code;
3    while ((code = _xbegin()) != _XBEGIN_STARTED) {
4      failures++;
5      if((code & 2) == 2 && failures <= max_retries) {
6        _mm_pause();
7      } else {
8        /* alternative fallback path... */
9        break;
10     }
11   }
12 }
13 void rtm_unlock() {
14   if(_xtest()) { // speculative run
15     _xend();
16   } else {
17     /* alternative unlock path... */
18   }
19 }
```

Although generalized methods might be easier to implement at first glance, one needs to keep the fallback-path in mind that can usually be implemented more efficiently if the specific use-case is known. Furthermore, the shown implementation is not compatible with mutexes which leads to the second HTM programming interface: HLE.

2.2 Hardware Lock Elision

Hardware Lock Elision (HLE) can be seen as a subset of RTM where the fallback path uses a lock [42]. It implicitly provides backwards compatibility with processors that do not support the Intel TSX and allows to use HTM in applications without changing the program logic [29], [30], [43].

Therefore, HLE introduces two new assembly instruction prefixes [30]:

XACQUIRE used in front of the instruction that acquires the lock protecting a critical section (marks the beginning of a transaction)

XRELEASE used in front of the instruction that releases the lock protecting the critical section (marks the end of a transaction)

Thus, the abstract structure of a program with HLE does not differ from the same program without HLE, the added prefixes present the only difference as shown in Listing 2.3.

Listing 2.3: Pseudo structure of a program using HLE prefixes

```
1 var mutex;
2 XACQUIRE lock(mutex);
3 execute_critical_section();
4 XRELEASE unlock(mutex);
```

HLE can also be used on a more abstract function-level rather than assembly, more on this topic can be found in Section 2.4.

When an instruction is prefixed with the processor hint `XACQUIRE`, the write associated with the lock operation is elided. Furthermore, instead of adding the mutex address to the write-set of the transaction, it is added to the read-set and the logical processor [4] enters transactional execution. Even though the write request has been elided, the eliding processor sees the mutex as occupied after the instruction whereas other processors will not notice a change in its value, i.e. if it was available before, it will remain available after the elided instruction. This allows other threads to enter the same critical region that another processor currently elides. As previously discussed, data conflicts inside transactions are detected by the processor and will lead to a transactional abort after which an instruction with an HLE prefix will be executed without elision. Thus, when data

[4] Logical processors: equal to the number of physical cores or twice their number with Hyper-Threading [44]

conflicts occur inside the transaction, an instruction with the `XACQUIRE` prefix will be executed twice: the first time elided without a write to the mutex and the second time after the abort and rollback as if the prefix would not be present [30]. If the instruction then writes to the lock variable, all threads speculating on this mutex will fail because the variable is located in their read-set [27]. An instruction prefixed with `XRELEASE` usually releases the mutex, thus writes e.g. zero to it. If the value of the mutex is equal to what it was in the beginning of the transaction, the write request will be elided again and the mutex address is not added to the write-set. Since an abort will be issued when the mutex variable is different from zero on release, it is desirable in terms of performance to immediately abort an operation that does not meet this condition [45]. Finally, the processors attempts to commit all buffered updates that happened during the transactional execution [27], [30], [34].

There are some important takeaways here: First of all, HLE is perfectly suited for an existing application that wants to make use of HTM since using it only involves prefixing lock instructions with the according prefixes - if locks are elided, the application becomes more scalable and non blocking [27]. For instance, Reindeirs [26] provides the example of a Hashmap that is protected by a global mutex. So far, this would have meant that every operation accessing an item of the Hashmap has to wait until it is the only operation on the Hashmap even if there are no collisions. With HLE, such a program could easily be extended to elide the mutex and thus increase the performance vastly. Figure 2.2 shows the example of HLE read operations on a Hashmap that contains the first names of some members of the Database Research Group at the University of Sydney. The names are hashed by the amount of characters. HLE ignores the global mutex and can perform all operations concurrently since there are no data conflicts.

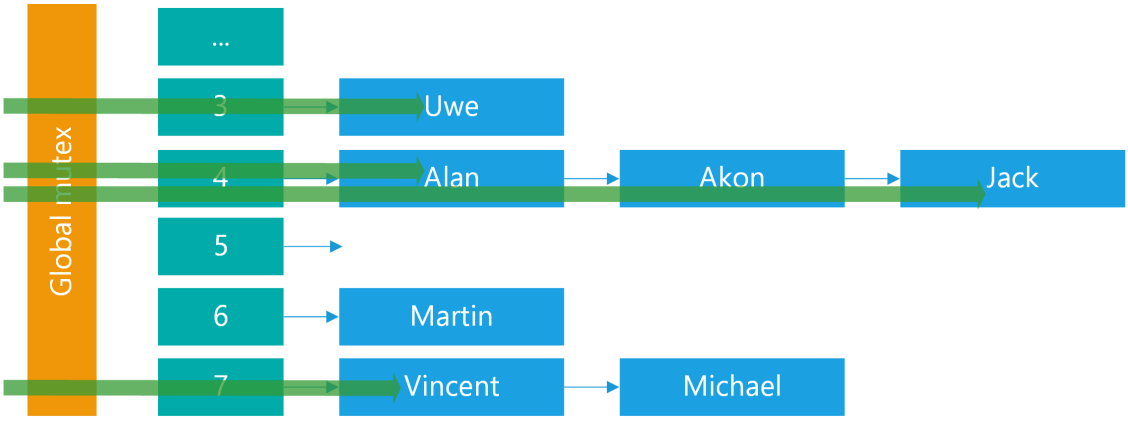


Figure 2.2: HLE operations eliding the global mutex on a Hashmap

Moreover, although a mix of HLE and non-HLE on the same lock variable is possible, it might not be beneficial since any write to the mutex will abort all transactional executions. Lock variables should also not be held in the same cache line of 64 Byte because the CPU tracks at exactly this granularity and thus multiple mutexes in the same cache line can not be tracked separately and

will abort [37]. When the instruction acquiring the mutex is executed in a loop, for instance with a spin-lock, and the mutex is already occupied by another non-transactional execution, elided and non-elided writes alternate [5].
In comparison with RTM, HLE is easier to implement and with its backwards-compatibility, it is a good fit for existing applications but it is not as flexible as RTM because the fallback path cannot be controlled [42].

As indicated at the beginning, HLE is a subset of RTM and by analyzing the control flow, we can write a naive implementation of a function that test-and-sets a mutex variable with HLE using RTM instructions. Thus, the HLE TAS function can be written using RTM for instance. TAS (test-and-set) is a method especially popular in low-level locking and lock-free data structures which sets a variable to the given value and returns the variable's previous content.

Algorithm 1 HLE TAS implementation using RTM

```
1: function TAS_LOCK(mutex)
2:     if _xbegin() = _XBEGIN_STARTED then
3:         return mutex
4:     else
5:         return non_elided_tas(mutex)
6: function UNLOCK(mutex)
7:     if _xtest then
8:         if mutex != 0 then
9:             _xabort()
10:        _xend()
11:    else
12:        set mutex to 0
```

The Algorithm 1 tries to execute an elided transaction using `_xbegin`. If this is successful, the value of the mutex is returned, otherwise the result value of the non-elided TAS function is executed. To unlock the mutex, we test whether the execution is transactional - if the mutex is not equal to zero therein, the transaction is aborted, otherwise we attempt to commit the transaction. In the non-transactional case where we used non-elided TAS in the lock function, the mutex has to be set back to zero.
Note that the value of the mutex will not appear to be set inside the eliding thread with this version, hence it is not perfectly equal to the actual HLE behavior.

The Lemming Effect We already discussed that when HLE falls back to explicitly acquiring the lock variable, other threads that elide the same mutex will be aborted. In the following, the other threads will also fall back to an explicit acquisition of the mutex. This can either fail as well if the mutex is still occupied and the thread would bounce between eliding the write and explicitly acquiring the mutex in the follow-up. If any of the explicit writes of all threads is successful, the

[5] A more detailed HLE spin-lock explanation can be found in Section 2.4

execution remains serialized and it can end up in a situation where no elision is successful for a longer time period [34]. The likelihood for a single thread to acquire the lock variable without eliding it is simply put 50% as it alternates between elision and no elision. Thus, n threads trying to acquire the mutex can be seen as picking n balls from an urn that are either white (elision) or black (explicit write) where the ball is put back after every pick. The likelihood that at least one mutex is written explicitly is equal to $p = 1 - P(\text{"all mutexes elided"}) = 1 - 0.5^n$. To put this into relation, with $n = 2$ threads, p is 75%, with 4 threads 93.75% and with 32 threads 99.999999976%, hence it is extremely likely that the execution will remain serial after an initial abort.
A solution to this issue is presented in Section 2.4.

2.3 Linux perf Tool

perf [6] is a profiling tool for Linux that allows to read and aggregate CPU hardware performance counters as well as tracepoints, software performance counters and dynamic probes [46].

Hardware performance counters are processor-registers that count certain hardware-events and thus allow for a low-level performance analysis. An important aspect of these registers is that they do not slow down the kernel or applications [46].

The measurable hardware events include e.g. the amount of cycles, cache-misses and most importantly Intel TSX events [18, p. 19-2] which will be shown in detail in Section 2.3.3. Another significant aspect is that the measuring can be performed with a low overhead [47]. The tool provides multiple commands to analyze an application or the full system such as a live event count, reporting, etc. [48]. We mainly focus on event counts with `perf stat` as well as recording and reporting events with `perf record` and `perf report`.

2.3.1 Preliminaries

Since perf is integrated in the Linux kernel, one might need to update to a new kernel. The 3.13 kernel includes all the features whereas the 3.11 and 3.12 kernels only contain a subset [49]. Our machine runs the `Linux version 3.11.0-custom+` which includes a patch for perf by one of the Intel engineers working on HTM, Andi Kleen.

2.3.2 Basic profiling

As mentioned, perf is capable of collecting and displaying events relevant to Hardware Transactional Memory. A general overview of elided transactions and transactional cycles can be produced with

[6] https://perf.wiki.kernel.org/index.php/Main_Page

`perf stat` and the `-T` flag. For instance, given a running process with the process id 17358, the call

```
perf stat -T -p 17358 sleep 120
```

collects general indicators such as the amount of executed instructions as well as HTM events over a timespan of 120 seconds [48]:

```
 Performance counter stats for process id '17358':

      2714,528891 task-clock            #    0,023 CPUs utilized
    3.658.016.172 instructions          #    1,05  insns per cycle
    3.486.716.405 cycles                #    1,284 GHz
      119.995.044 cpu/cycles-t/         #    3,44% transactional cycles
                0 cpu/tx-start/         #    0,000 K/sec
          866.442 cpu/el-start/         #      138 cycles / elision
       84.327.282 cpu/cycles-ct/        #    1,02% aborted cycles

    120,001087923 seconds time elapsed
```

The abbreviations of this output are explained in Table 2.2.

label	explanation
cycles	All recorded cycles
cycles-t	Started transactional cycles
tx-start	Started RTM transactions
el-start	Started HLE transactions
cycles-ct	Committed transactional cycles

Table 2.2: Explanation of abbreviations in the `perf stat` output [49]

One also needs to be aware that the comment part on the right is not to be read one to one to the event part on the left [50]. For instance, the last line on the left gives information about committed cycles whereas the comment part tells us about the percentage of aborted cycles relative to all cycles in the program.
Furthermore, the share of aborted cycles is relative to the total cycles and not to the transactional cycles as one might assume. Thus, the percentage is calculated as follows:

$$\begin{aligned} \text{proportion}_{\text{tot}}(\text{cycles-abort}) &= \text{proportion}_{\text{tx}}(\text{cycles-abort}) \times \text{proportion}_{\text{tot}}(\text{cycles-t}) \\ &= \frac{\text{cycles-t} - \text{cycles-ct}}{\text{cycles-t}} \times \text{proportion}_{\text{tot}}(\text{cycles-t}) \\ &= (1 - \frac{\text{cycles-ct}}{\text{cycles-t}}) \times \text{proportion}_{\text{tot}}(\text{cycles-t}) \end{aligned} \tag{2.1}$$

where $\text{proportion}_{\text{tx}}(\text{cycles-abort})$ is the proportion of aborted cycles relative to the transactional cycles and $\text{proportion}_{\text{tot}}(x)$ is the proportion of x relative to the total cycles. In our example output, that is $(1 - \frac{84{,}327{,}282}{119{,}995{,}044}) \times 3.44\% = 1.02\%$. To calculate the share of aborted cycles relative to the transactional cycles directly from the given percentages, we solve for $\text{proportion}_{\text{tx}}(\text{cycles-abort})$:

$$\text{proportion}_{\text{tx}}(\text{cycles-abort}) = \frac{\text{proportion}_{\text{tot}}(\text{cycles-t})}{\text{proportion}_{\text{tot}}(\text{cycles-abort})} \tag{2.2}$$

Calculating $\text{proportion}_{\text{tx}}(\text{cycles-abort})$ for the example, we receive $\frac{1.02\%}{3.44\%} = 29.65\%$ which differs only slightly from the exact value of $1 - \frac{\text{cycles-t}}{\text{cycles-ct}} = 1 - \frac{84327282}{119995044} = 29.72\%$ due to rounding errors in the percentages.

Unless explicitly mentioned otherwise, this work presents all transactional cycles relative to the total cycles and the aborted cycles relative to the *transactional* cycles (instead of total cycles) which is the more intuitive representation in our opinion.

2.3.3 Event profiling

We can also profile hardware statistics for particular events. This is achieved by using the syntax `perf stat -p PID -e '{EVENT_CODES}'`. A list of TSX performance events can be found at the Intel® 64 and IA-32 Architectures Software Developer's Manual under section 19-2, table 19-3. Table 2.3 lists some of the events in the `HLE_RETIRED` category on page 708 that are of interest to us.

Event Num.	**Unmask Value**	**Event Mask Mnemonic**
C8H	01H	HLE_RETIRED.START
C8H	02H	HLE_RETIRED.COMMIT
C8H	04H	HLE_RETIRED.ABORTED
C8H	08H	HLE_RETIRED.ABORTED_MISC1 (memory events, e.g. capacity/conflicts)
C8H	10H	HLE_RETIRED.ABORTED_MISC2 (uncommon conditions)
C8H	20H	HLE_RETIRED.ABORTED_MISC3 (unfriendly instructions or transaction nesting limit overflow [49])
C8H	40H	HLE_RETIRED.ABORTED_MISC4 (incompatible memory type, e.g. HLE executed inside RTM [49])
C8H	80H	HLE_RETIRED.ABORTED_MISC5 (other, e.g. interrupts)

Table 2.3: Excerpt of TSX Performance Events [18, Table 19-3]

The unmask values for RTM are the same whereas the Event Num. is `C9H` instead of `C8H` [18,

pp. 707–708].
To profile these events, an example `perf` call could be the following:

```
perf stat -p 12345 -e '{r01C8,r02C8,r04C8}'
```

The `r` has to be prefixed whenever events are described by using the unmask value and event num. One could also refer to the `HLE_RETIRED.START` event by using `el-start`, for example, where we would not need the prefixed `r`.

To verify that the flags actually represent their descriptions and to illustrate some outputs, we profile a few simple tests and check the abort reasons.

Data conflicts One of the most common reasons for aborts is obviously a data conflict where one thread writes data that other threads access (read or write) afterwards, which would lead to inconsistency. We create such a case where we have two threads that write to the same shared value, as shown in Listing 2.4.

Listing 2.4: Thread run function that writes to a shared variable

```
1 int shared;
2 void run(int repeats) {
3   if (_xbegin() == _XBEGIN_STARTED) {
4     shared++;
5     _xend();
6   }
7 }
```

The profiled events for this output in Listing 2.5 confirm that the abort reason `MISC1` indicates data conflicts.

We also observe that the sum of the abort reasons misc1 and misc5 is greater than the indicated aborted transactions and therefore conclude that the hardware events perf displays are not 100% accurate.

Listing 2.5: perf events for 1,000,000 repeats in two threads of RTM with data conflicts

```
1          3.156.174 r01C9 (started)
2          1.874.316 r02C9 (committed)
3          1.279.861 r04C9 (aborted)
4          1.279.920 r08C9 (misc1)
5                  0 r10C9 (misc2)
6                  0 r20C9 (misc3)
7                  0 r40C9 (misc4)
8                164 r80C9 (misc5)
```

Unfriendly instructions As previously mentioned, instructions such as system calls cannot be rolled back and therefore abort immediately. Listing 2.6 shows a setup where the transaction never succeeds.

Listing 2.6: RTM with unfriendly instructions

```
1 if (_xbegin() == _XBEGIN_STARTED) {
2   _mm_pause();
3   _xend(); // never succeeds
4 } else {
5   // failure
6 }
```

The according perf results are displayed in Listing 2.7 and confirm that `MISC3` is the event for aborts caused by unfriendly instructions.

Listing 2.7: perf events for 100 loops of RTM with `_mm_pause()` from Listing 2.6

```
1                 100 r01C9 (started)
2                   0 r02C9 (committed)
3                 100 r04C9 (aborted)
4                   0 r08C9 (misc1)
5                   0 r10C9 (misc2)
6                 100 r20C9 (misc3)
7                   0 r40C9 (misc4)
8                   0 r80C9 (misc5)
```

In terms of representativeness on the impact, cycles are more significant: a high abort rate per transaction is not important as long as the aborted cycles are low [50]. The reasons are based on transactions consisting of many cycles. So when a transaction aborts, any of its cycles has aborted. In HLE, that means that all modifications of the previous cycles have to be undone which can cost a lot if there have been many cycles before. But when one of the first cycles of the transaction aborts, it is not much effort (or maybe even no effort at all) to undo the changes made. Thus, cycles and their abort rate represent the performance more precisely and events are mostly useful to figure out abort causes in our experience.

2.3.4 Recording and reporting profiles

perf also allows to record a profile of a process that can then be used for in-depth analysis in a report. The `perf record` call writes samples of the measured activities to a file, typically `perf.data`. `perf report` then uses this file to display all sorts of information.

We use these two commands mostly for assembly-level analysis: perf is capable of displaying hot-spots of a certain event, such as a transactional abort. The report displays a percentage-weighted

list of functions where the event occurred and allows to see the call-chains of how this function has been called, again aggregated with their relative shares as well as annotating the assembly source that is again furnished with percentages representing how relevant an instruction is, i.e. how often the event occurred in this particular instruction [46]. Figure 2.3 shows such a list of

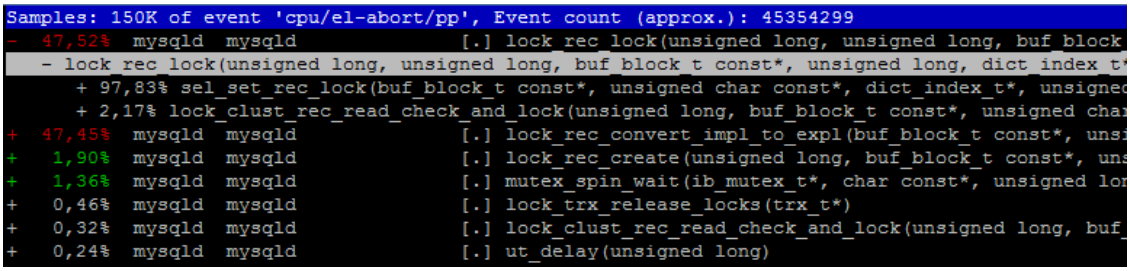


Figure 2.3: Exemplary perf report

functions where the samples have been recorded for the hardware event `el-abort`, i.e. aborted HLE transactions. In this example, 47.52% of elision aborts occurred in the function `lock_rec_lock` and in particular with 97.83% by calls to this method from the `sel_set_rec_lock` function. Every of these functions can then be annotated on assembly-level.

2.4 Micro-Usage of HLE

Section 3.3 compares HLE and RTM against POSIX and TAS implementations (which are identical to the HLE implementation but do not elide). Therefore, this section aims to determine the proper HLE and thereby also non-elided TAS calls that we will then use in more complex measurements.

Usually, there is more than one solution to a problem and our lock function implementation is not an exception to this rule. Table 2.4 shows the three categories for our case: HTM (n/y), modification and loop-definition.

$$\begin{matrix}\text{not-elided}\\ \text{HLE}\end{matrix} \times \begin{matrix}\text{TAS}\\ \text{EXCH}\end{matrix} \times \begin{matrix}\text{spin-lock}\\ \text{spin-read-lock}\end{matrix}$$

Table 2.4: Function combinations

To reduce this rather big amount of $2^3 = 8$ possible combinations, we run tests on a simple array of (padded) mutexes that will be locked/unlocked by different threads which leads to some contention but not every lock access will be in conflict. The accesses are herein completely random and the array is shared among all threads. Finally, we choose the best performing non-elided and HLE implementation that we will then use for following benchmarks.

In terms of implementation, the non-elided operations `test_and_set` and `exchange` are implicitly provided and HLE's equivalent functions can be accessed by either providing an additional

HLE flag in the atomic builtins which are supported in GCC since version 4.8 [27], [43] or by using the `hle-emulation` header. Three example calls are shown in Listing 2.8.

Listing 2.8: atomic_tas and hle_exch call

```
1 #include "hle-emulation.h" // required for all __hle functions
2 type *lock_var;
3 __atomic_test_and_set(lock_var, __ATOMIC_ACQUIRE);
4 __atomic_test_and_set(lock_var, __ATOMIC_ACQUIRE | __ATOMIC_HLE_ACQUIRE);
5 __hle_acquire_exchange_n4(lock_var, 1);
```

The proper `type` for the lock variable will be determined in the next section.

2.4.1 Lock variable type

The `hle-emulation` header offers four different definitions of a lock variable: `unsigned char`, `unsigned short`, `unsigned` and `unsigned long long`. The later is only available on 64-bit systems. To determine which type to use, we run two tests comparing all four types with the lock function shown in Listing 2.9. In following sections, this function will be further modified to determine the best implementation but the one shown suffices to decide the best type.

Listing 2.9: HLE lock macro

```
1 #define __HLE_LOCK(size, type)\
2 static void hle_lock##size(type *lock) {\
3   while(__hle_acquire_test_and_set##size(lock, 1)) {\
4     _mm_pause();\
5   }\
6 }
```

This macro is then expanded by calling it with the four (type, size) pairs: { `(unsigned char, 1)`, `(unsigned short, 2)`, `(unsigned, 4)`, `(unsigned long long, 8)` }. Listing 2.10 illustrates an example call for the first pair mentioned.

Listing 2.10: uchar macro expansion call

```
1 __HLE_LOCK(1, unsigned char)
```

In addition, we also aim for more control about the thread-scheduling and pin every thread to a core as shown in Listing 2.11.
Since we have 4 cores, this means that in the case of 4 threads each of them will each run on its own core exclusively and in the case of 16 threads, 4 threads will struggle for the resources on each core.

Listing 2.11: Thread pinning

```
1 static int num_cores = sysconf(_SC_NPROCESSORS_ONLN);
2 int stick_this_thread_to_core(int core_id) {
3   cpu_set_t cpuset;
4   CPU_ZERO(&cpuset);
5   CPU_SET(core_id, &cpuset);
6
7   pthread_t current_thread = pthread_self();
8   return pthread_setaffinity_np(current_thread, sizeof(cpu_set_t), &cpuset);
9 }
```

To test these methods, we run a fixed number of threads that each call the lock- and unlock-function repeatedly. The used mutexes vary from a single mutex up to 100 parallel mutexes, all of them shared among all threads. They are further padded to fill in a complete cache line of 64 Byte. Figure 2.4 shows the results of measuring all lock types where four threads continuously perform the operation of locking and immediately unlocking a random mutex.

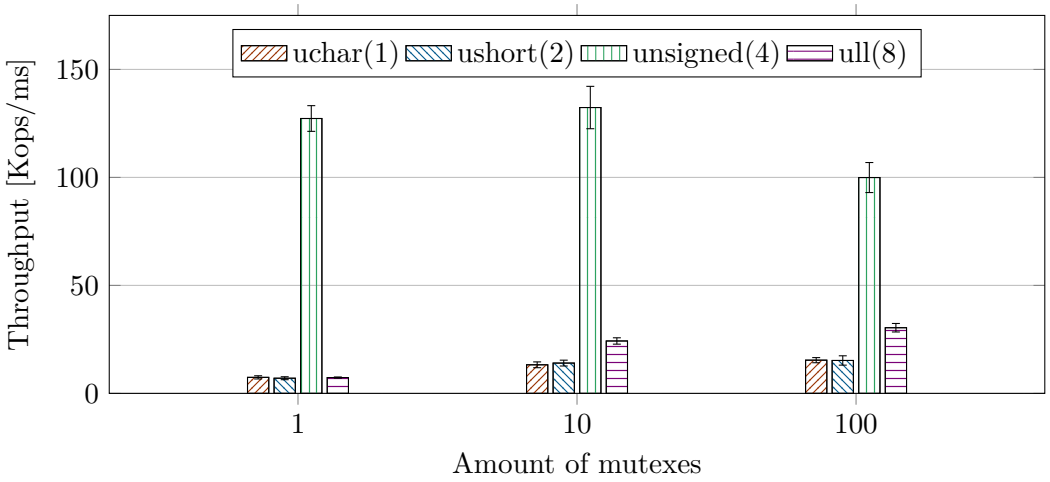


Figure 2.4: Comparison of lock variable types for a varying amount of mutexes

Apparently, `unsigned` with a size of 4 byte is the fastest lock variable type on our machine and we will use this type for our implementation. We assume that the reason for its performance advantage has to do with on how many bytes the CPU operates, i.e. in this case on sizes of 4 byte or 32 bit. Nonetheless, we can show how the share of transactional and aborted cycles is connected to the performance. In Table 2.5 which displays hardware indicators for a run of 10 mutexes, the type `unsigned` has more than twice as many transactional cycles as the other types and by far the smallest share of aborted cycles which is reflected in a throughput that is 4-5 times higher than the thoughput of the other types.

	uchar	ushort	unsigned	ull
Transactional cycles	24%	22%	54%	23%
Aborted cycles	12%	24%	6%	14%

Table 2.5: Transactional and aborted cycles for 10 mutexes

2.4.2 HLE function call

Kleen [51] defines two ways to use Hardware Lock Elision on a function-level: adding a HLE-flag to the `__atomic` operations or using the `hle-emulation` header. In terms of the `__atomic_exchange_n` function, the only difference in assembler-code with and without the `__ATOMIC_HLE_ACQUIRE` flag is a prefixed `xacquire`.

Listing 2.12: atomic xchg without HLE

```
__atomic_exchange_n(lock, 1,
    __ATOMIC_ACQUIRE);
```

```
xchg  eax, DWORD PTR [rdx]
```

Listing 2.13: atomic xchg with HLE

```
__atomic_exchange_n(lock, 1,
    __ATOMIC_ACQUIRE |
    __ATOMIC_HLE_ACQUIRE);
```

```
xacquire xchg eax, DWORD PTR [rdx]
```

Furthermore, we compare the assemblies of `__atomic_exchange_n` with a set HLE-flag against `__hle_acquire_exchange_n4`.

Listing 2.14: atomic HLE lock

```
__atomic_exchange_n(lock, 1,
    __ATOMIC_ACQUIRE |
    __ATOMIC_HLE_ACQUIRE);
```

```
xacquire xchg eax, DWORD PTR [rdx]
```

Listing 2.15: emulation HLE lock

```
__hle_acquire_exchange_n4(lock, 1);
```

```
.byte 0xf2 ;  ; lock ; xchg eax,
    DWORD PTR [rcx]
```

The only differences between the code in Listings 2.14 and 2.15 are firstly that `xacquire` has been replaced by `.byte 0xf2` which is the lower-level representation of `xacquire`. In addition, a `lock` has been added in front of the `xchg` call. Since the atomic assembly accesses the memory with `[rdx]` however, the lock prefix is implicit already [52]. The final notable inequality is that the `hle-emulation` assembly places semicolons between the instructions which are interpreted as newlines [53, p. 3], [54], [55].
We can therefore conclude that the two assemblies are equal and that it does not matter whether one uses the `__atomic` functions with a HLE-flag or the `__hle` functions from the emulation header.

2.4.3 Locking algorithm

Kleen [51] suggests a method similar to the spin-lock, but with an inner read loop (referenced to as TAS-T lock (as described later in this section). This section compares it with a naive spin-lock and explains why the TAS-T lock is suggested.

TAS lock the naive implementation is a trivial busy wait that continuously test-and-sets the value.

Listing 2.16: Pseudo HLE TAS spin-lock implementation

```
1 void spin_lock(var lock_var) {
2   while(hle_test_and_set(lock_var, 1))
3     pause();
4 }
```

It should be emphasized that HLE will always elide in the first place - even if the `lock_var` is set [45]. A set lock variable (i.e. TAS returns 1) indicates two things: first, at least one other thread is currently executing in a serialized manner, thus without elision, and more importantly second, if we were to elide, the `XRELEASE` operation would abort because it attempts to set the lock variable to zero but it is one (see Section 2.2). Because this transaction will not commit, we abort it immediately by calling `pause` and thus "loose" only a few cycles as compared to all cycles of the transaction.

TAS-T lock similar to the TAS spin-lock in the outer loop but with an inner loop where the mutex-value is only read instead of continuously writing it (the outer loop performs a *T*est-*A*nd-*S*et and the inner loop only *T*ests).

Listing 2.17: Pseudo HLE TAS-T-lock implementation

```
1 void tas_t_lock(var lock_var) {
2   while(hle_test_and_set(lock_var, 1)) {
3     var lock_value;
4     do {
5       pause();
6       lock_value = get(lock_var);
7     } while(lock_value is set);
8   }
9 }
```

An additional test in front of the loop to check if the `lock_var` is free in the first place does not increase the performance since the first TAS call of the loop tests the value anyways. And even if the lock variable is set and the transaction has to be aborted, no data has been modified within this

very short transaction, thus a rollback does not have to undo any changes and it makes more sense to perform the test-and-set in one operation instead of executing a test-test-and-set algorithm.
The `pause` is realized by calling `_mm_pause` (`xmmintrin.h`) which results in `rep nop` on an assembler level. In the case of multiple threads on one core, the `pause` instruction gives the other thread a chance to get the CPU and release us from waiting. It also reduces the CPU effort in busy waiting while taking only 0.4-0.5 clocks according to Intel [56]. HLE or atomic instructions replace the `test_and_set` method in specific implementations.

To determine whether a TAS- or TAS-T-spin-lock is the better lock-mechanism for our case, it does not make a difference for a trivial scenario where 4 threads lock a certain slot out of an array with 100 mutexes and unlock it again immediately after as shown in Figure 2.5. That is because

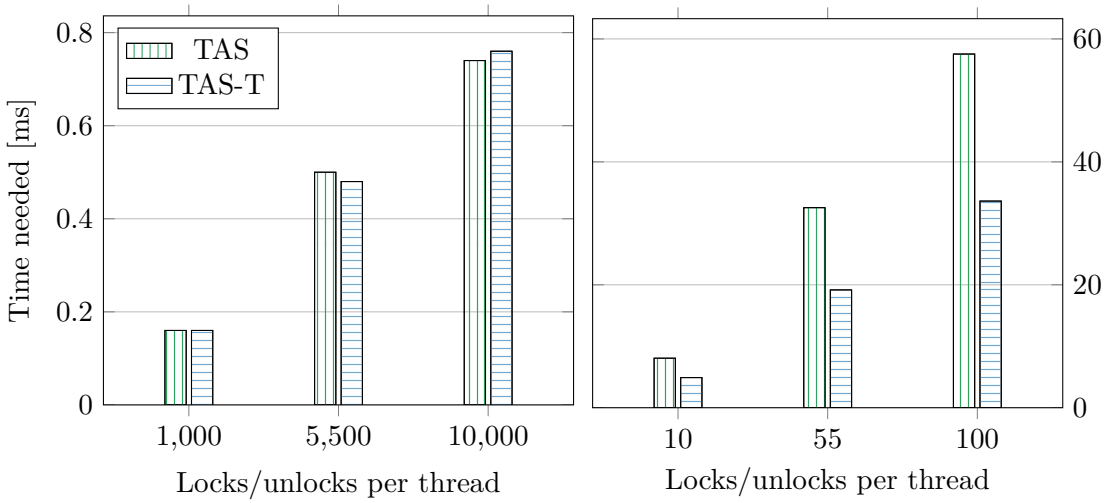


Figure 2.5: Immediate unlock on a size 100 mutex array (4 Threads, lower is better)

Figure 2.6: Delayed unlock on a single unsigned mutex (4 Threads, lower is better)

a blocked mutex is unlocked immediately again, so it is very likely that a subsequent write after the initial failed write is successful. However, the TAS-T-lock should have an advantage when it comes to mutexes that are not immediately unlocked. To test this assumption, we reduce the mutex array size to 1 and put a wait phase between the lock and unlock call, hence force a hotspot that is unlocked after a small period of time. The wait phase is implemented with the `nop_sleep` described in Section 3.2.3 and a parameter of 1,000, hence $1,000 \times 750$ `nops`. The observation could also be made with e.g. the parameter 10 but 1,000 makes the differences very clear. Furthermore, a value of e.g. 10,000 would already lead to lots of aborts because of too long transaction times. Thus, 1,000 seems to be a good balance between illustration and representative test results.
Figure 2.6 illustrates that the TAS-T-lock wins over the TAS-lock in such a setup. The reason is that there is a lower chance of successfully locking the mutex when the mutex is continuously written in the lock-function. On the other hand, adding a control mechanism to wait for the mutex to become free after it could not be written (TAS-T-lock), the mutex is only written when it is free (after the first failed write).

To analyze the improvement of the TAS-T-lock compared to the TAS-spin-lock seen in Figure 2.6, we use the `perf` tool once more. Table 2.6 compares hardware transaction indicators for the TAS- and TAS-T-spin-lock and shows that the abort rates for the TAS-T-lock are minimal in comparison to the TAS-lock (transactions as well as cycles).

	TAS (spin)	**TAS-T**
Started transactions	213.187.224	71.899
Aborted transactions	99.99%	12.15%
cycles	126×10^9	59×10^9
Transactional cycles	32.43%	85.48%
Aborted cycles	40.04%	5.68%
Instructions per cycle	1.64	3.59

Table 2.6: TAS- and TAS-T perf results for delayed unlocks with 4 threads on a single unsigned mutex (Figure 2.6)

For the following explanation, we have to keep in mind that HLE always elides [45] and every HLE instruction is executed twice in the case of aborts: once elided and once with an actual write-attempt to the mutex. Following, the TAS-lock does exactly the same again after a short `pause`: a new transaction is started with an elision but since the mutex is still occupied, the transaction is aborted and we perform the actual TAS call on the mutex. If the non-elided approach fails as well, we start over. This behavior provokes the start of a lot of transactions since we simply retry until it works at some point. But because the mutex is non-accessible for some time, a lot of these transactions (and with them their cycles) are aborted immediately by checking the mutex-value and aborting if it is set. Thus the high amount of started and aborted transactions and aborted cycles for the TAS-lock.
Furthermore, because the TAS-lock does not wait for the mutex to become free, there is an approximate 50:50 chance of executing an elided transaction or performing the actual set of the mutex (it is only an exact 50:50 chance if the time before the HLE TAS call, i.e. the `pause` instruction is equal to the time before the non-elided TAS call, i.e. the abort). In contrast, the TAS-T-lock waits for a free mutex which greatly increases the probability of an elided transaction since it is the first call after the mutex became free. This difference is observable in Table 2.6 as well: the TAS-lock has a lower share of transactional cycles than the TAS-T lock (the difference is close to 50%). Ultimately, the total cycles of the TAS-lock are approximately twice as much as compared to the TAS-T-lock which can be explained by the greater amount of elided transactions in the TAS-T-lock. Successful elisions mean that no writes to the mutex are involved which are the main component of this benchmark.
We also note that the TAS-T-spin-lock is able to execute more than twice as much cycles compared to the TAS-lock. This can be explained by instruction prefetching: every jump invalidates the next prefetched cache-line [57] and it has to be reloaded. For the TAS-spin-lock, this is expensive because the complete TAS function has to be loaded into cache again whereas the TAS-T-lock only has to reload a single pause instruction.

The measurements in Figure 2.5 and 2.6 have shown that the TAS-T-lock is vastly faster than the TAS-spin-lock in a scenario where the lock is not immediately unlocked and that it is not slower even in a TAS-spin-lock friendly scenario. Hence, we use the TAS-T spin-lock for future measurements.

2.4.4 TAS implementation

The TAS operation can also be implemented by using an exchange (EXCH) instruction which exchanges value `m` (typically the lock mutex) with another value `v` and returns the previous value of `m`.

Listing 2.18: Pseudocode TAS implementation

```
1 void test_and_set(var m, var v) {
2   var prev_m = m;
3   m = v;
4   return prev_m;
5 }
```

When comparing the example EXCH implementation in Listing 2.18, we observe that it is almost completely similar to the TAS implementation - in fact the `hle-emulation` header defines the `__hle_acquire__test_and_set`-functions as an `exchange` operation with an additional comparison of equality to one (see Listing 2.19).

Listing 2.19: ___hle__test__and__set implementation

```
1 #define __HLE_EXCHANGE(type, prefix, asm_prefix, size)      \
2 int __hle_##prefix##_test_and_set##size(type *ptr)          \
3 {                                                          \
4   return __hle_##prefix##_exchange_n##size(ptr, 1) == 1;   \
5 }
```

We suspect that the non-elided `__atomic_test_and_set` operation is implemented similarly and both `test_and_set` functions can therefore be simplified to an exchange operation instead. This is safe to do because an expression `if(a)` where `a` is a numeric value will only evaluate to false when `a` is zero which makes the additional comparison to one redundant.
To verify this assumption, we compare the performances of `TAS` and `EXCH` TAS-T-locks on an unsigned array size 100.

Figure 2.7 illustrates that there is no noticeable difference between `TAS` and `EXCH` for neither HLE nor non-elided calls. Thus, we stick with the `EXCH` lock and ignore `TAS` from now on. We also see a first tendency of HLE outperforming non-elided TAS as it is e.g. two times faster for 10^4 repeats.

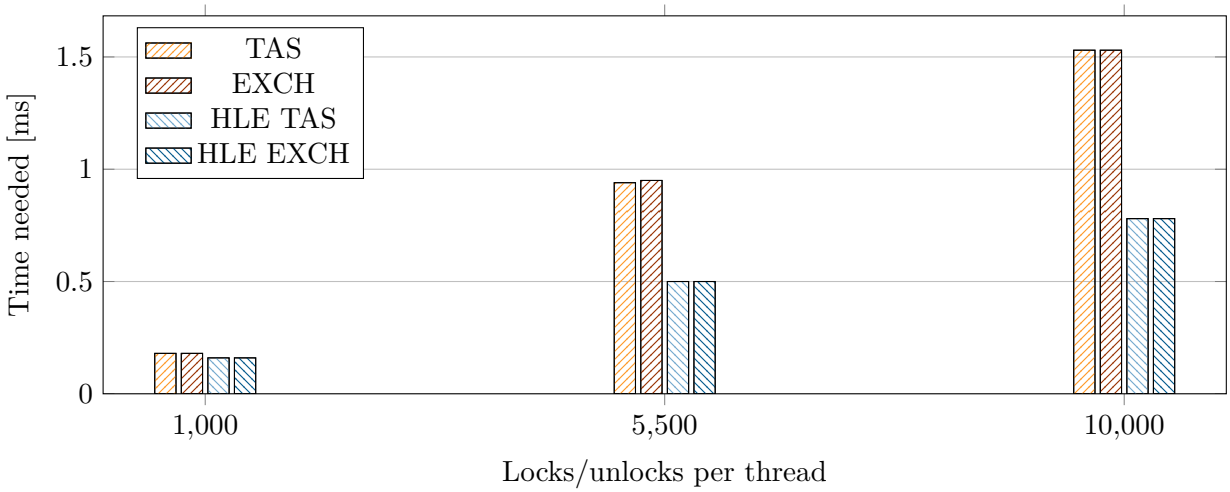


Figure 2.7: TAS-T-locks using TAS and EXCH on an unsigned array of size 100

2.4.5 Resulting combined function

Bringing all evaluations together, we use HLE and non-elided TAS implementations of a lock function that:

1. operates on a lock variable of type `unsigned`
2. is called via the `hle-emulation` header (in the case of HLE)
3. uses a TAS-T-lock with an outer TAS-loop and an inner read-only loop
4. writes to the lock variable using `exchange`

The implementation of this HLE function in C++ is shown in Listing 2.20.

Listing 2.20: Implementation of the final HLE function [51]

```
1  void hle_lock(unsigned *lock) {
2    while (__hle_acquire_exchange_n4(lock, 1)) {
3      unsigned val;
4      do { /* Wait for lock to become free again before retrying. */
5        _mm_pause(); /* Abort speculation */
6        __atomic_load(lock, &val, __ATOMIC_CONSUME);
7      } while (val == 1);
8    }
9  }
10 void hle_unlock(unsigned *lock) {
11   __hle_release_clear4(lock);
12 }
```

CHAPTER 3

Evaluation of Core Performance Characteristics

This chapter deals with the scope and the environment in that HTM is useful and therein also defines its boundaries. Furthermore, the performance of certain data structures with HTM is analyzed and we identify the best-suited cases for this technique.

3.1 Necessity to keep the mutex in the read-set

One thought that came to our mind in the development of this work is "why does the mutex even have to be in the read-set?" The idea behind this is that even if there are two different locking systems in place, e.g. a non-elided TAS and the new RTM approach, all conflicts are due to data - thus, why should HTM care about the mutex if only data matters? Hence, a function that forgets about the mutex can be written using RTM where we simply retry the locking until it succeeds.

Listing 3.1: RTM locking without loading the mutex in the read-set

```
1 void rtm_lock(type* mutex) {
2   int code;
3   while ((code = _xbegin()) != _XBEGIN_STARTED) {
4     _mm_pause();
5   }
6 }
```

However, running applications with mixed non-elided TAS locks and this modified version that ignores the mutex but operates on the same mutex as the TAS lock leads to inconsistency and we conclude that the mutex must be kept in the read-set. A definite reason for this can not be defined due to lack of open documentation but the assumption is that only one thread recognizes data conflicts and then aborts all other threads that keep the mutex in their read-set (which they always do with HLE). Hence, if the thread recognizing the data conflict runs a non-elided TAS lock

in the example, a different thread that runs the modified version would not be informed about the conflict and carry on with the elided transaction, thus not providing any concurrency guarantees anymore.

3.2 Scope of application

With HTM operating on the L1 cache only and being aborted by a lot of reasons, some of which occur randomly (e.g. interrupt) [18], [19], it is clear that there are limitations other than "real" data conflicts to this technology. This section goes into detail about the size of a transaction, its duration and the nesting of HTM instructions. It does not elaborate on aborts that either occur all of the time or not all, such as issuing system calls or other unfriendly instructions [58].

All benchmarks in this section are run in a single thread since we are interested in isolating single abort reasons.

3.2.1 Transaction size

The transaction size of a hardware transaction is limited by the size of the L1 cache which is 64 KB per core in our case. However, the cache is equally split into instruction and data cache, thus the effective cache size for data is 32 KB (see Chapter A). Furthermore, the associativity of the cache sometimes decreases the effective cache even further as shown in Section 3.2.2.

For our benchmark, we use a `unsigned char` array because it offers fine granularity in terms of its size in memory (`sizeof(unsigned char) = 1 Byte`). Every loop accesses all elements of the array from 0 to `array size-1` with either a write- or a read-access. We then increase the size of the array and measure the aborts.

Aside from distinguishing between read and write, we also use both initialized and uninitialized arrays because our tests showed differences therein. An initialized array is defined as an array whose values are set to any value. For instance, the array `a` in Listing 3.2 is initialized.

Listing 3.2: Initializing an array

```
1 unsigned char a[10];
2 for(int i=0; i<10; i++)
3   a[i] = 0;
```

In contrast, an uninitialized array has just been allocated but its values have not manually been set yet (however, the values are deemed to be 0 by definition).

Write-only

To begin with, we use a setup where we only write to the array. This is similar to the setup of Leis, Kemper, and Neumann [19, p. 5] except that we access the array sequentially instead of randomly.

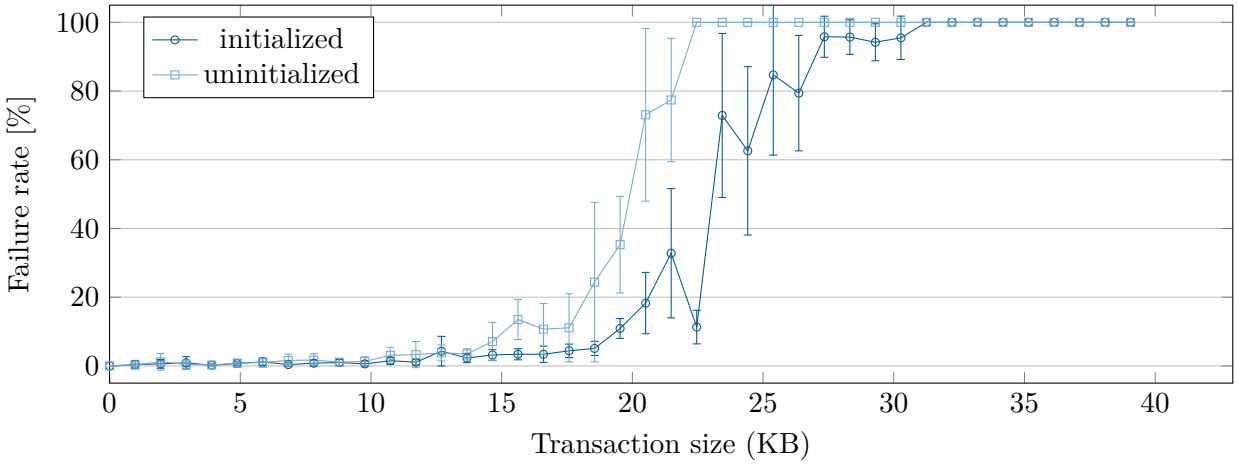


Figure 3.1: Sequential write access on a uchar-array

Figure 3.1 confirms that write-sets exceeding the size of the L1 DCache of 32 KB [31, p. 12.2] lead to an abort all of the times. We can conclude that the more data inside a transaction is modified, the more aborts occur. Writing data bigger than the L1 DCache size (32 KB) always aborts.

It is also shown that uninitialized data is more likely to abort. The reason for that is that uninitialized memory only exists virtually in the virtual memory page table but not in main memory. When it is accessed and the page table entry indicates that it is currently not in real memory, a page fault exception is raised which then leads to the initialization of the page [18, p. 4-44]. This hardware interrupt causes the transaction to abort, resulting in the increased abort rate of the uninitialized array in Figure 3.1.

By profiling this benchmark with perf (Listing 3.3), we see that a capacity conflict is represented in the same way as a data conflict, namely by the event `MISC1`.

Listing 3.3: perf events for write access on an array

```
1          123.532 r01C9 (started)
2            4.847 r02C9 (committed)
3          118.683 r04C9 (aborted)
4          117.959 r08C9 (misc1)
5                0 r10C9 (misc2)
6                2 r20C9 (misc3)
7                0 r40C9 (misc4)
8              721 r80C9 (misc5)
```

Read-only

For a read-only setup (contrary to write-only in the previous section), the resulting graph differs in multiple aspects.

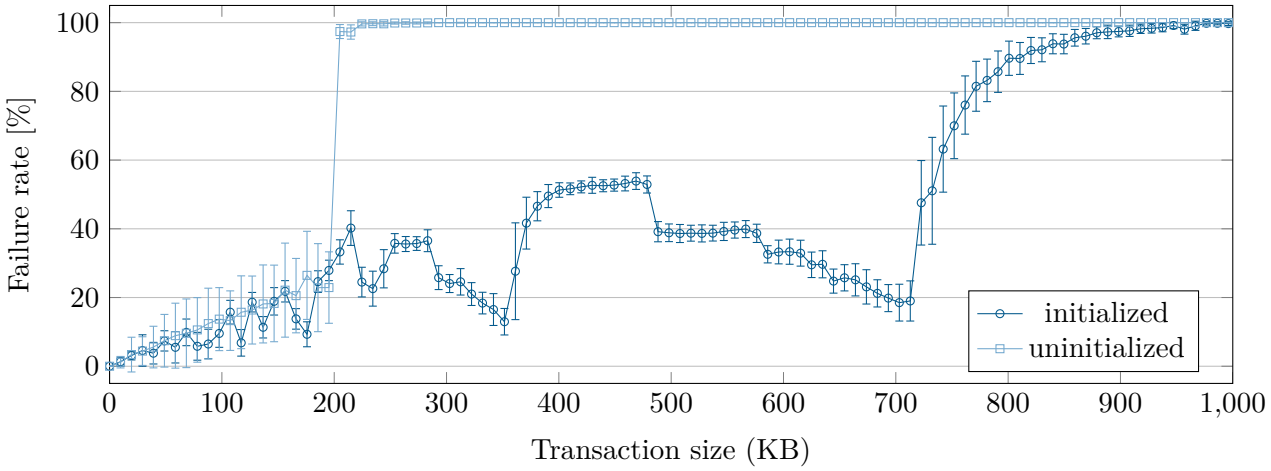


Figure 3.2: Sequential read access

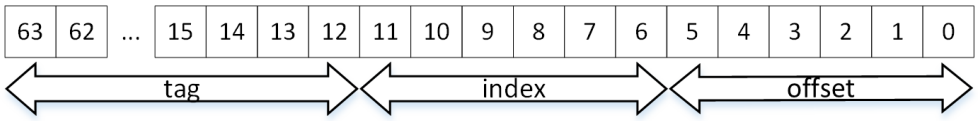
Data that is only read and not written can exceed the L1 data cache size under certain cirumstances: *Intel® 64 and IA-32 Architectures Optimization Reference Manual* [31, p. 12.1] states that "An eviction of a read set address may not always result in an immediate transactional abort since these lines may be tracked in an implementation-specific second level structure. The architecture however does not provide any guarantee for buffering and software must not assume any such guarantee". This statement is confirmed in Figure 3.2: Non-aborting transactions can be found with transaction sizes way beyond the L1 DCache of 32 KB. Although the shown graph is cut off at 1000 KB and it mostly stays at 100% afterwards, there were individual cases of successes beyond this point, e.g. for a transaction size of 1280 KB. Furthermore, multiple runs of the same test gave different (but still consistent in their category) results - with the two extremes of 1) the shown graph where a transaction with a size close to 1 MB still commits in some cases and 2) a result where every transaction size bigger than 32 KB failed. These results often depend on unexpected parameters such as the steps at which the array size is increased. As Intel states, software must not assume that the cache size can actually be exceeded.

The "rise and fall" pattern of the uninitialized data's failure rate could occur when the architecture decides to swap the tracking of some lines in the second level structure. According to this explanation, this only happens periodically (thus the spikes) and does not work forever since the failure rate reaches 100% at around 1 Megabyte of data.

3.2.2 Cache Associativity

Since the cache is set-associative, data that is aligned unfavorable can fail although its transaction size is smaller than the L1 DCache.

The set-associative scheme is often chosen because it compensates the disadvantage of fully associative caches where every slot has to be checked in a complex parallel manner and direct-mapped caches which may cause collisions of addresses to the same slot by being a hybrid of both [59], [60]. To find the set for a memory address, 6 bits of the memory address are used (32 KB cache / 64 Byte cache line size = 512 cache lines, 512 cache lines / 8 slots per set = 64 sets, ld 64 = 6 bits to represent each set). One byte can have 64 different offsets which again can be represented using 6 bits [61, pp. 133–136]. Hence, of a 64 bit address A, bits A_{63-12} are used for the tag and 6 bits each for the index (A_{11-6}) and offset (A_{5-0}).



However, this can also lead to the eviction of cache entries although theoretically all the data would fit into the cache. If, in our 8-way-set-associative cache, there are variables from more than 8 distinct cache lines mapping to the same cache set, one of the cache lines will no longer fit into the L1 cache, hence leading to an abort [37], [31, p. 12-2]. This is shown in Figure 3.3 where 8 cache lines can be stored in Set i but the 9th will lead to the eviction of one of the previous variables. In HTM this produces an abort because the data does not fit into L1 DCache all at once anymore.

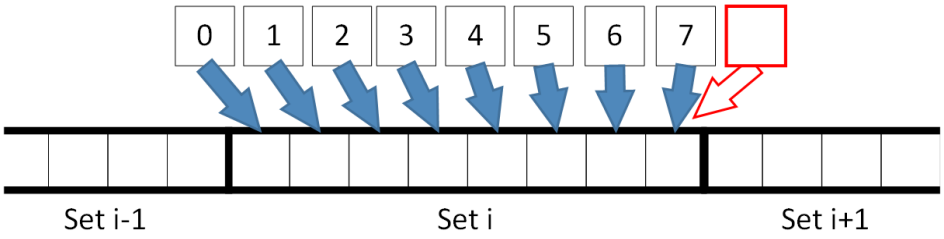


Figure 3.3: Too many variables mapping to the memory same set

We want to illustrate this phenomenon by comparing linear access on an array where only every `n`th item is accessed and thus, some elements are skipped. Each array item has the size of a cache line (64 B) and thereby the last 6 bits of every array element address are the same (but not necessarily zero). The 6 index-identifying bits are incremented by 1 per element. Hence, the item `i` maps to the memory set `i % 64`. When we access every single item, all memory sets are mapped (`0%64=0, 1%64=1, 2%64=2, ...`). By increasing the step size to 2, only half the set amount is actually reached because every second address is skipped (`0%64=0, 2%64=2, 4%64=4, ...`).

The following Figure 3.4 illustrates the "loss" of memory sets for a simplified cache with only 8 sets and an array that is accessed with different step sizes n, i.e. every, every second, every third and so on item is accessed. All slots that will be reached are marked with a bullet-point and blue colored slots are the first few items that are not mapped to the same set up to the red slot which marks the first repetition of a set.

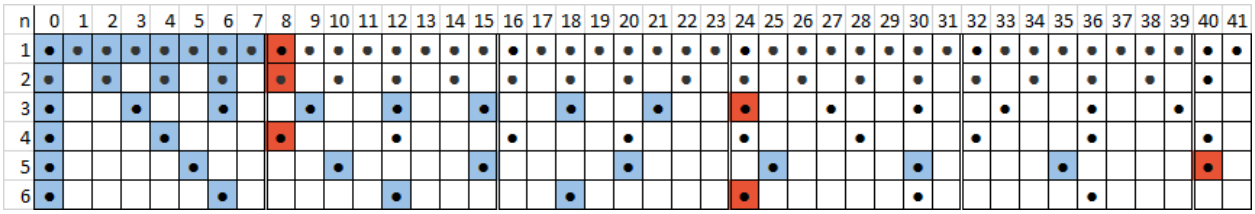


Figure 3.4: Cache-set mappings for array access of every nth item

It is shown that e.g. for $n = 2$ all odd memory-sets are omitted. All odd step sizes only repeat the set after 8 (amount of sets) items. Table 3.1 summarizes the figure by listing after how many elements an already occupied set is used again.

step size n	1	2	3	4	5	6	7	8	9
first set-repetition after r elements	8	4	8	2	8	4	8	1	8

Table 3.1: First set-repetition of skipped array access with 64 sets (Figure 3.4)

The pattern we can observe is that the first set-repetition r times the step size n is always a multiple of the amount of sets s. This gives us the following equation

$$(r \times n) \mod s = 0 \tag{3.1}$$

By finding the smallest $r > 0$ that matches the equation, we then get the number of sets that data is mapped to. In other words, we search for the least common multiple (lcm) of n and s and divide it by n to get the amount of sets that are being used.

$$r = \frac{\text{lcm}(n, s)}{n} \tag{3.2}$$

By finding an r for a given n and s, we are able to compare how skipped array access behaves in comparison to non-skipped access in terms of set-mapping. For instance, if for n', $r_{n'}$ is half the value of $r_{n=1}$, only half of the available sets are reached when we use only every n'th item. Since this reduces the effective available L1 DCache size, it presents an issue for HLE.

To apply these findings to a cache with 64 sets we apply Equation 3.2 with a changed s which gives us Table 3.2 (we could also multiply all values from Table 3.1 by $\frac{64}{8} = 8$).

After these theoretical calculations, we also want to observe the effects on the abort rate. Therefore,

step size n	1	2	3	4	5	6	7	8	9
first set-repetition	64	32	64	16	64	32	64	8	64

Table 3.2: First set-repetition of skipped array access with 64 sets

we use a class `CacheLine` with a size of 64 Byte.

Listing 3.4: CacheLine class

```
1 class CacheLine { // sizeof(CacheLine) = 64B
2   unsigned char value; // 1B
3   unsigned char padding[64-1]; // 63B
4 };
```

Similar to the setup in Section 3.2.1, an array of this class is allocated with a fixed size and its elements are written to. However, the write-loops only access every nth value where n is the step size as shown in Listing 3.5.

Listing 3.5: Write loop

```
1 // step = {1, 2, 3, 4, 5}
2 // size goes from 0 to 35 KB
3 CacheLine array[size];
4 int adjusted_size = size * step;
5 for (int i = 0; i < adjusted_size; i += step) {
6   array[i].value = 1;
7 }
```

By varying the step size, we are able to create certain environments where only limited cache sets are reached by the data.

First of all, Figure 3.5 confirms the maximum transaction size for writes of 32 KB that has been found in Section 3.2.1.
Furthermore, as predicted in Table 3.2, the odd numbers 1, 3, 5 behave exactly the same (therefore not shown as the values overlap completely) whereas a loop that skips every second element is only able to access half the transaction size of a loop accessing all array elements. When we skip every fourth item, only a fourth of the original transaction size can be achieved. Keep in mind that this does not continue (i.e. even skips can only access $\frac{1}{n}$ of the original transaction size) since e.g. $n = 6$ behaves the same as $n = 2$ and can only access $\frac{1}{2}$ of the transaction size for $n = 1$.
Another interesting fact to mention here is that the compilation with the flag `-O2` instead of `-O0` increased all transaction sizes by the factor two, $n = 1$ was capable of 32 KB instead of 16 for example.

In terms of stack- or heap-allocation with `new`/`malloc`, one can argue that since the data is allocated rather randomly, the differences are not that big. This is shown in Figure 3.6 where data

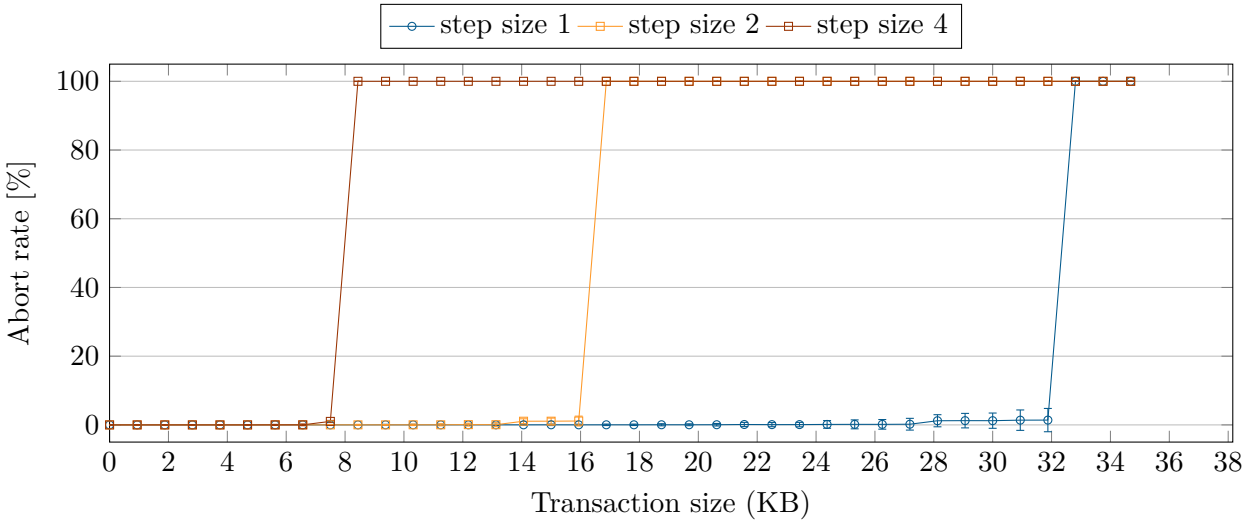


Figure 3.5: Sequential write access on an initialized static array

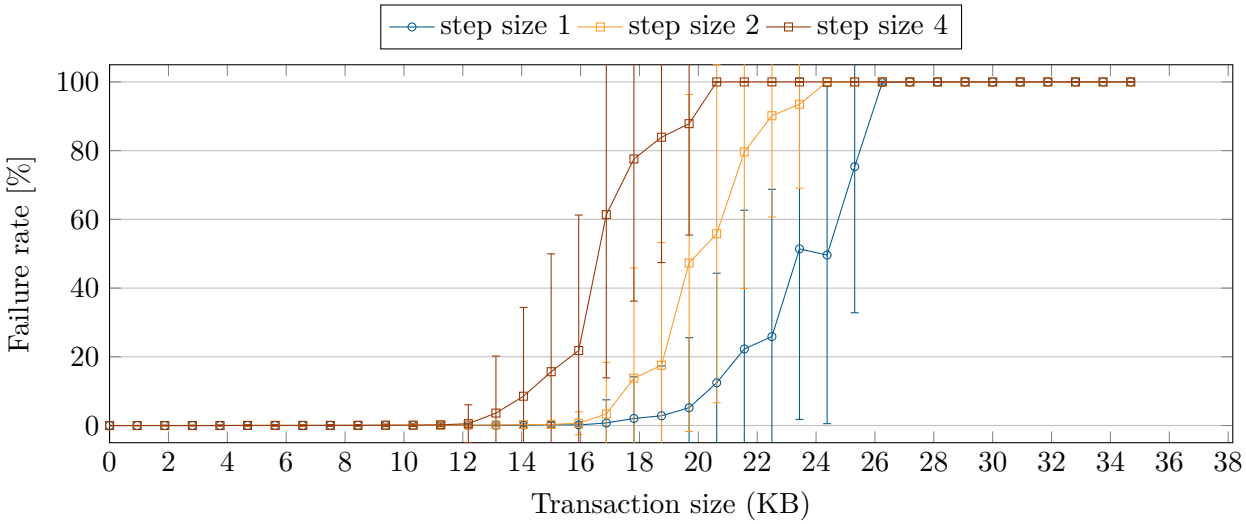


Figure 3.6: Sequential write access on an initialized dynamic array

is allocated consecutively using `new`: the worse cases where for instance only every fourth item is accessed are not as bad anymore but at the same time, the best-case of sequential access without skips looses out on its maximum transaction size.
We therefore conclude that a program can be optimized by properly aligning memory allocation.

To align data that is sequentially iterated, a potential naive implementation in C++ could be to use the `vector` class and reserve a fixed space upfront.

Listing 3.6: Pre-allocating data using vector

```
1  std::vector<MyClass> aligned_data;
2  aligned_data.reserve(pre_allocation_size);
3  MyClass dummy();
4  for(int i=0; i<pre_allocation_size; i++)
5    aligned_data.push_back(dummy);
6
7  int last_allocation;
8
9  MyClass * preallocated_new() {
10   int allocation_search_begin = last_allocation;
11   do {
12     last_allocation = (last_allocation + 1) % pre_allocation_size;
13     MyClass * data = &(aligned_data[last_allocation]);
14     if (! data->isUsed()) { // slot is not used
15       data->setUsed(1);
16       return data;
17     }
18   } while (last_allocation != allocation_search_begin);
19
20   /* no space in pre-allocation available */
21 }
```

However, this approach is neither thread-safe nor would a resize keep the references, pointers or iterators referring to the elements in the sequence[1]. A different solution could be to use the `aligned_alloc` function[2].

3.2.3 Transaction duration

One of the abort reasons being a system interrupt that is randomly issued, Leis, Kemper, and Neumann [19] found that the longer a transaction needs, the more likely it is to abort. Since system calls cannot be avoided in practice, even transactions that do not perform any operation other than waiting face this limitation. This section confirms that finding and also analyzes the standard deviation of interrupt aborts over time.

[1] see http://www.cplusplus.com/reference/vector/vector/reserve/
[2] see http://www.gnu.org/software/libc/manual/html_node/Aligned-Memory-Blocks.html

To make our transaction as simple as possible, we do not want to compute anything or perform memory operations. A sleep or wait operation is suited for that. However, we can neither use `usleep` nor `pause` as they perform a system call, which will always fail since a transactional execution is aborted by a system call [27], [62]. Another approach to keep the processor waiting is the assembly instruction `nop` (short for No Operation). This command effectively does nothing at all by telling the processor do nothing in this cycle [18, p. 898].

Listing 3.7 shows the test setup where the `nop` instruction is repeatedly called inside the HTM part. We test how long we can wait after the instruction will no longer succeed.

Listing 3.7: nop measurement

```
1 if (_xbegin() == _XBEGIN_STARTED) {
2   for (int i = 0; i < clocks; i++) {
3     asm volatile("nop");
4   }
5   _xend();
6 } else {
7   failures++;
8 }
```

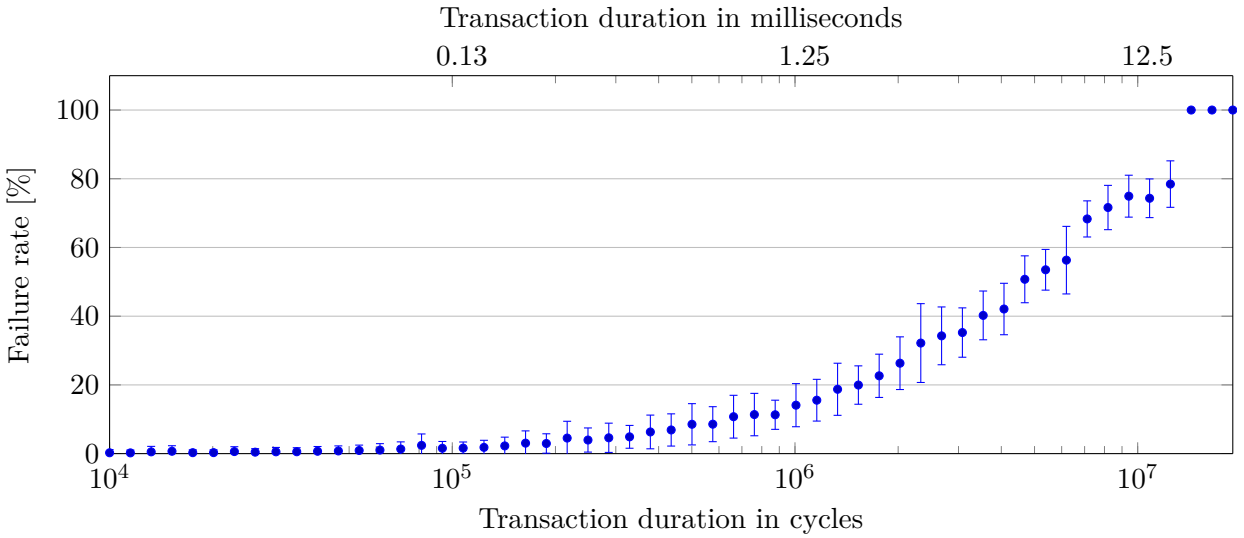


Figure 3.7: Frequency of aborts in relation to transaction durations and their standard deviations

As we can easily observe, failure rates rise with the duration of a transaction and thus conclude that the more time a transaction needs, the more likely it is to abort.

To convert clock cycles to microseconds, we have to take into account the machine's CPU clock speed. On our machine, that is 800 MHz = 800 $\times 10^6$ cycles per second. Thus, the duration of a cycle is $\frac{1\text{s}}{800 \times 10^6} = 1.25\text{ns}$. To calculate the duration of n cycles in nanoseconds, we simply calculate the result of the equation $n \times 1.25\text{ns}$.

Note that after 100,000 cycles (0.125 milliseconds), the failure rate is already above 10% and for over 13 million clocks (16.25 milliseconds), every single transaction fails. Also be aware that even though the failure rate for e.g. 100 cycles with below 0.01% is very small, failures still occur and could lead to a Lemming Effect as described in Section 2.2.

Profiling the benchmark confirms that the aborts indeed only happen due to interrupts as shown in Listing 3.8.

Listing 3.8: perf events for interrupt aborts

```
1              52.647 r01C9 (started)
2              39.516 r02C9 (committed)
3              13.130 r04C9 (aborted)
4                   0 r08C9 (misc1)
5                   0 r10C9 (misc2)
6                   0 r20C9 (misc3)
7                   0 r40C9 (misc4)
8              13.129 r80C9 (misc5)
```

For future measurements, a wait function without system calls (i.e. similar to `usleep`) will be useful. On the basis of the benchmark for transaction duration, this function uses `nops` for waiting. Since `usleep` takes microseconds as an argument, we have to multiply the amount of `nops` by a scale-factor which stands for the amount of times we have to loop the `nop` instruction to reach 1 microsecond. With our CPU clock speed of 800 MHz, we have to wait for $1\mu s \times 800MHz = 800$ cycles to achieve a wait time of 1 microsecond. Albeit this is a theoretically perfect parameter, the CPU has a dynamic clock frequency which can lead to deviating results [63]. For instance, calling our `nop_wait` function (see Listing 3.9) with the parameter 10,000 and measuring the total time with `gettimeofday` can result in different values such as 10,925 or 6612 microseconds.

Listing 3.9: nop sleep

```
1 void nop_sleep(int microseconds) {
2   for (int i = 0; i < microseconds * 800; i++) {
3     asm volatile("nop");
4   }
5 }
```

3.2.4 Transaction Nesting

It is possible to have transactions nested inside each other, which is conceptually handled by flattening the nest into a single transaction. The amount of nesting is however limited [29] - this section attempts to find the value of this limit, i.e. the maximum amount of nested transactions. This can be useful when encountering the abort reason *Too deep nesting* [49].

We analyze RTM and HLE separately since their underlying behavior differs for this setup.

RTM

We use a simple test setup where we let a function call itself recursively up to a given limit and then count the failures to determine the maximum amount of nested RTM transactions.

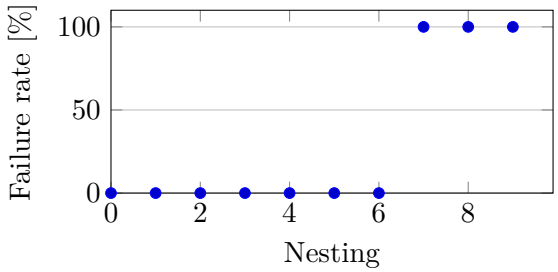


Figure 3.8: RTM Aborts per nesting amount

Figure 3.8 shows that nested functions abort as the nesting amount reaches 7, thus RTM Transactions can only be nested up to a maximum of 6 times[3].

HLE

In contrast to RTM, HLE functions that lock the same mutex can not be nested because the mutex appears locked for the thread that occupies it, even if it has been elided.

However, functions locking different HLE mutexes can be nested infinitely often. For this test, we use an array of padded mutexes where each mutex is padded to the size of a cache line to avoid false conflicts. Then, a certain amount of mutexes is acquired and by profiling the run, we can determine the abort arte.

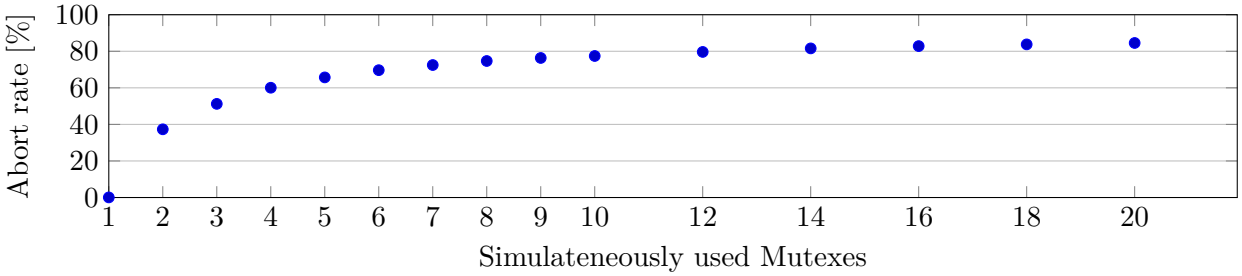


Figure 3.9: Abort rates of nested mutexes

[3] *Intel® 64 and IA-32 Architectures Optimization Reference Manual* [31, p. 12-2] states that a nesting limit of 7 is supported wherein they probably start counting with 1 for no nesting

Despite the possibility of infinite nesting, it is not useful at all as Figure 3.9 points out. The results of this benchmark are pretty unsettling because with only two different simultaneously used mutexes, over a third of the transactions fail already. Hence, global mutexes might be favorable in some cases. These results are also contrary to the statement that there is a specific limit to the number of locks that can be elided simultaneously by Kanter [29, p. 2] because the abort rate increases continuously and not abruptly.
A solution approach to this issue which leads to no more nesting aborts is a modification of the HLE lock function so that the mutex will only be loaded into the read-set if the execution is not already transactional. Therefore, we use the `_xtest` function of the RTM interface to check whether the current execution is transaction and modify the unlock function to only perform the actual HLE unlock if the mutex has been written (even if elided, the value appears as set to the eliding thread) as shown in the implementation in Listing 3.10.

Listing 3.10: Hypothetical workaround for HLE nesting aborts

```
1  void hle_lock(unsigned * mutex) {
2    if (_xtest() && *mutex != 65535) { // compare with arbitrary number to load into
         read-set
3      return;
4    } else {
5      /* actual HLE call... */
6    }
7  }
8  void hle_unlock(unsigned * mutex) {
9    if (*mutex == 0) // not set
10     return;
11   /* actual HLE call... */
12 }
```

Note that the compiler might optimize the mutex-comparison away if it finds that the value is never assigned anyway. Moreover, this implementation worked for the micro-benchmark but we did not conduct any tests in a real application environment.

Internesting

Nesting an HLE transaction inside an RTM transaction and vice-versa leads to a transactional abort [31, p. 12-19].

3.2.5 Overhead

Hardware Transactional Memory does not come for free and programs run in a single thread without locks are obviously faster than using HTM. We use a simple setup to estimate the overhead of a

traditional POSIX mutex lock, an atomic locking technique, HLE and RTM. The benchmark only calls the respective lock and unlock functions, thus there is no data involved and we focus only the control mechanism.

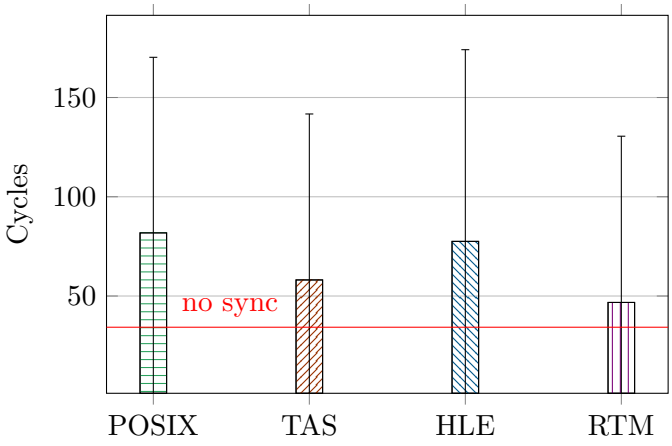


Figure 3.10: Overheads with lock-only (lower is better)

Figure 3.10 shows the overhead in cycles[4] of the different locking techniques compared to no synchonization in a single thread. In this scenario, adding synchronization with a POSIX mutex adds a 139% overhead, a similar value holds true for HLE (126% overhead), non-elided TAS locks are slightly better (69% overhead) and RTM (36% overhead) wins this comparison. Although there are close to zero aborts (except for some interrupts), the additional HTM control mechanisms seem to add a significant overhead, especially HLE. The cause for this is probably that HLE needs to keep track of the mutex additionally whereas this RTM implementation does not rely on any mutexes.

Despite the lower overhead of non-elided TAS compared to HLE in this benchmark, the difference becomes negligible when we are able to elide locks and thereby execute transactions concurrently instead. Furthermore, due to the broad range of variances, a clear statement can not be made from this test.

3.3 Isolated use cases

To further determine what scenarios are best-suited for HTM and where one should avoid using it, we analyze the following data structures each of which targets a different use case: a shared counter, banking accounts, a doubly linked list and finally a hashmap. These data structures are generic and commonly used, the two last named inter alia in MySQL.

The critical sections in the data structures in these use cases are protected using different locking implementations:

[4]Measuring the same benchmark with execution time instead of cycles produces the same result

POSIX the POSIX standard locks, used by calling the functions `pthread_mutex_lock` and `pthread_mutex_unlock`

TAS a non-elided test-and-set approach with an inner read-only loop as described in Chapter 2.4

HLE the same as the TAS implementation but with HLE prefixes

Elision only an RTM-implementation that ignores the mutex and retries the operation until it succeeds (this is an extreme that is always optimistic and does not provide a forward-guarantee - however, since we only face data conflicts in this section, all operations succeed sooner or later)

The implementation we use to allow for simple testing of different scenarios is a generalized class that functions as a proxy to all the locking methods. This class provides its own lock variables[5] and an `enum` representing the different locking implementations and an object initialized with this `enum` will only lock in the specified manner.

3.3.1 Closed banking system

The analysis in this section simulates a closed bank where the money of an account is deposited, withdrawn and read. This allows us to verify the correctness of the test-run by summing up the balances of all accounts and comparing them to the initial total money in the bank. If there is a difference between the sum of all accounts in the beginning and the end, money has been lost or created which should be impossible in a closed system, thus it implies a thread-unsafe implementation. We are however not perfectly strict in the way that the transfer between two accounts is not a transaction but only the operations on a single account, i.e. between the withdrawal of one account and the deposit in the second account, the money only exists in a variable and it is missing in the closed bank system. Hence, the sum only has to be correct after all operations have finished.

Every account instance is realized using a class which provides thread-safety for the account by locking before every balance-reading or -modifying operation and unlocking afterwards. The run function of each thread then selects a random operation, according to the given probabilities: in the case of update, money is withdrawn from a random account `A1` and deposited in another random account `A2`. If the chosen operations happens to be read-only, the value of a random account is read.
Our test is set up with 10000 accounts and a 50/50 probability for read/update.

Since we found the system `rand()` function to be quite slow when used many times, we use a

[5]Since we often need different lock variables for different locking implementations, some overhead is introduced - however, this overhead also ensures that no two locking objects are in the same cache line

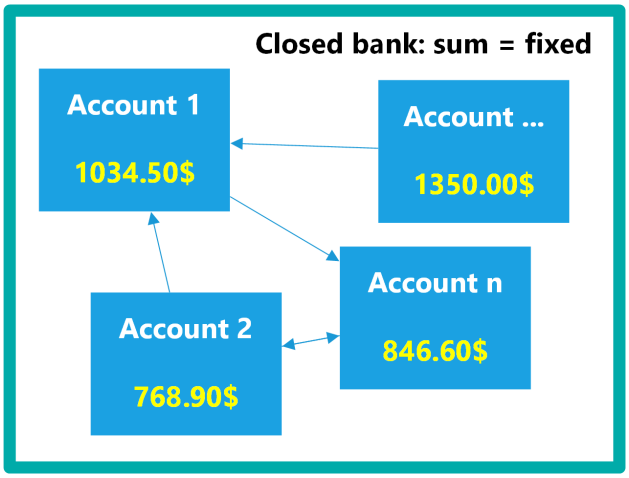


Figure 3.11: Sketch of a closed bank

custom random generator which is shown in Listing 3.11[6]. The random generator is not thread-safe, thus some conflicts might not be truly random but originate from concurrent access to the random generator. However, this benchmark mostly aims for prove of correctness and hence we take some more data conflicts into account.

Listing 3.11: Custom random generator

```
1 int customrand(int limit) {
2   static long state = 1;
3   state = (state * 32719 + 3) % 32749;
4   int res = state % limit;
5   return res;
6 }
```

Figure 3.12 shows how Hardware Transactional Memory performs slightly better than non-elided approaches with POSIX outperforming non-elided TAS a little. The graph remains roughly the same if the read/update-probabilities are modified.
In addition to the small performance-improvement, the main takeaway from this benchmark is that HTM is correct and works as a locking technique. The accumulated sum in the closed bank system remained unchanged for all locking techniques, thus the internal structures of HLE and RTM function properly.
When it comes to more complex code inside the transaction in later sections (and not just simple `ADD` or `MOV` operations as in this case), we will see a bigger difference between non-HTM and elided transactions.

[6]Other random functions have also been tested, see Appendix Section C.6

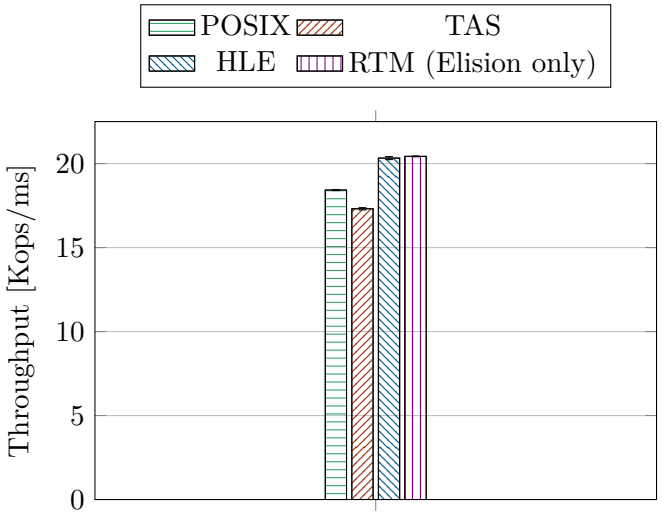


Figure 3.12: Closed Bank with 10000 accounts, 4 threads and 50% read / 50% update

3.3.2 Shared counter

This section covers the incrementation of a counter that is shared among multiple threads. Obviously, this is not an HTM-friendly setup because there are lots of conflicts.

The benchmark is inspired by a paper by Porobic, Pandis, Branco, *et al.* [64] where the latencies of hardware heterogenities were analyzed with threads that were either spread over multiple sockets or grouped together on the same socket. Our threads in contrast are always spread over the four available cores but the counter is varied between shared between all threads and shared between threads on the same core.

Single counter

To begin with, we introduce a single counter whose critical sections are protected by a mutex that will be varied. This counter is shared among all threads, each of which is pinned to a core, and thereby among multiple cores.

With one thread and one counter, there are no data conflicts, thus the very left bar chart shows overheads of the different locking approaches for incrementing an integer value. This measurement differs from the benchmark in Figure 3.10 where no data was involved in the way that HTM is now clearly worse than non-elided approaches, so apparently it is quite expensive to keep track of elided values (profiling the output confirms that there are only 0.08% aborted cycles, hence the overhead does not have its cause in aborts). Nonetheless, it should be emphasized again that this overhead is beneficial when it comes to multiple threads.

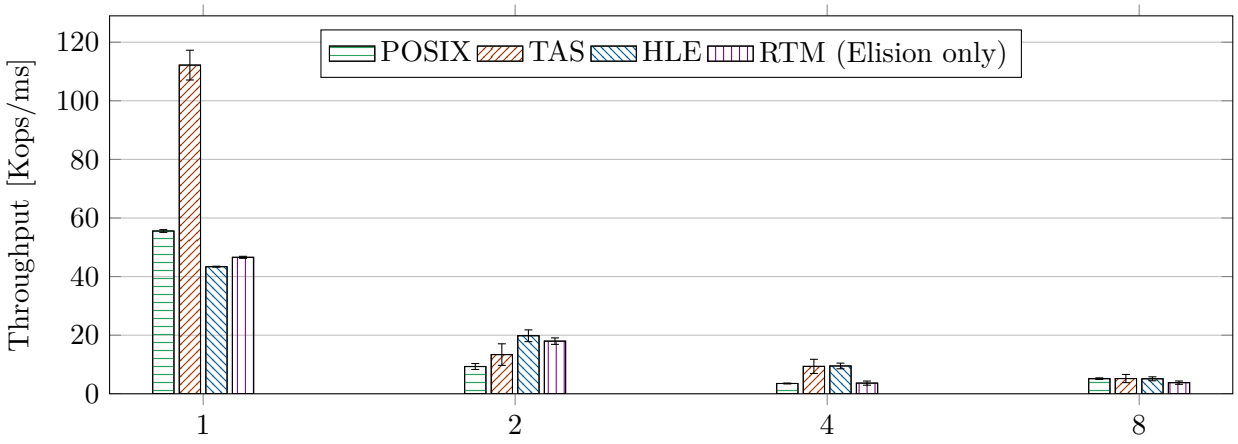


Figure 3.13: Single shared counter

Despite slight improvements with multiple threads, the HTM locks do not perform better than non-elided locks because of high contention and hence lots of aborts (e.g. 37% aborted cycles for 4 threads with RTM).

Counter per core

The single counter is now extended to four counters (equal to the amount of cores) and each thread works with one of those counters. The core-locality is guaranteed by pinning the threads to the corresponding core, so for instance if thread `i` gets assigned the counter with the number `i % CORES = 2`, the thread will also be pinned to core `2`.

Unaligned counter per core If we do not take care of the padding and keep the per-core-counters in the same cache line (i.e. by allocating an array `int counters[CORES]`), we receive the graph in Figure 3.14 where HLE and RTM perform extremely bad because the same cache line is accessed over and over again, hence there are lots of false aborts.

Aligned counter per core In the following, we make sure that the counters are located in different cache lines by padding them. The results of padded counters are shown in Figure 3.15 where there is an approximate 10× speedup compared to no alignment and the performance from two to four threads increases instead of remaining roughly the same. However, since the locks are at the most fine-grained level possible (namely per counter), HTM is still not outperforming the non-elided locks. Because there are 0% aborted cycles for e.g. RTM and 4 threads, this implies that HTM is only beneficial when the locks are not perfectly fine-grained. Obviously, implementing this kind of locks where every mutex protects e.g. a single cache line would be extremely complex and error-prone to do and moreover one would need an insane amount of mutexes which would

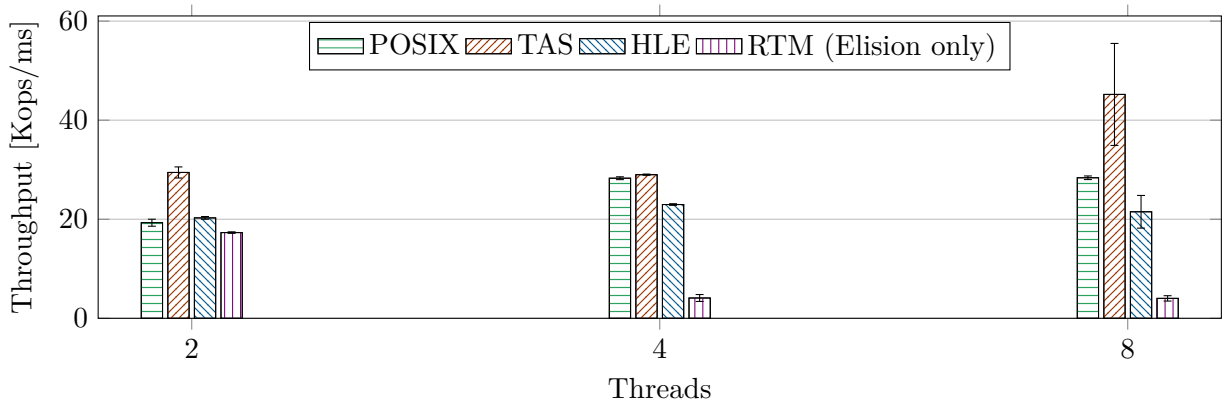


Figure 3.14: Unaligned counters per core

then lead to issues with the cache again. Hence, eliding transactions is the easier thing to do and following this argumentation, also more efficient with more data.

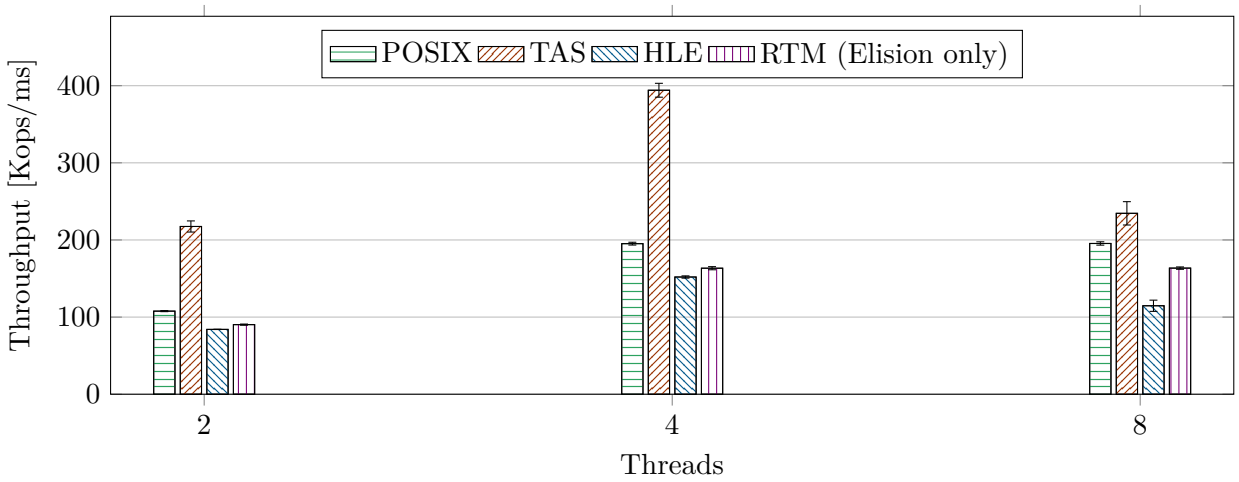


Figure 3.15: Aligned counters per core with lock per counter

Another interesting observation of the aligned counters is that not only HTM profits from alignment but also the non-elided TAS. This can be explained with caching, where the counters no longer have to be reloaded from the shared cache (which is necessary if there is only a single cache line that contains all the counters and is invalidated on every write of another thread).

Aligned Counter per core with global lock We can also illustrate how HTM is of advantage when the locks are no longer fine-grained by replacing the lock per counter with a global mutex. Thereby, each thread has to compete with the other threads even if there are no real conflicts.

Figure 3.16 illustrates how it only makes a slight difference for HLE whether the lock is acquired per counter or globally because it is elided anyway. However, we can still observe a minor performance decrease for HLE between fine-grained mutexes and a coarse-grained one, e.g. for 8 threads which

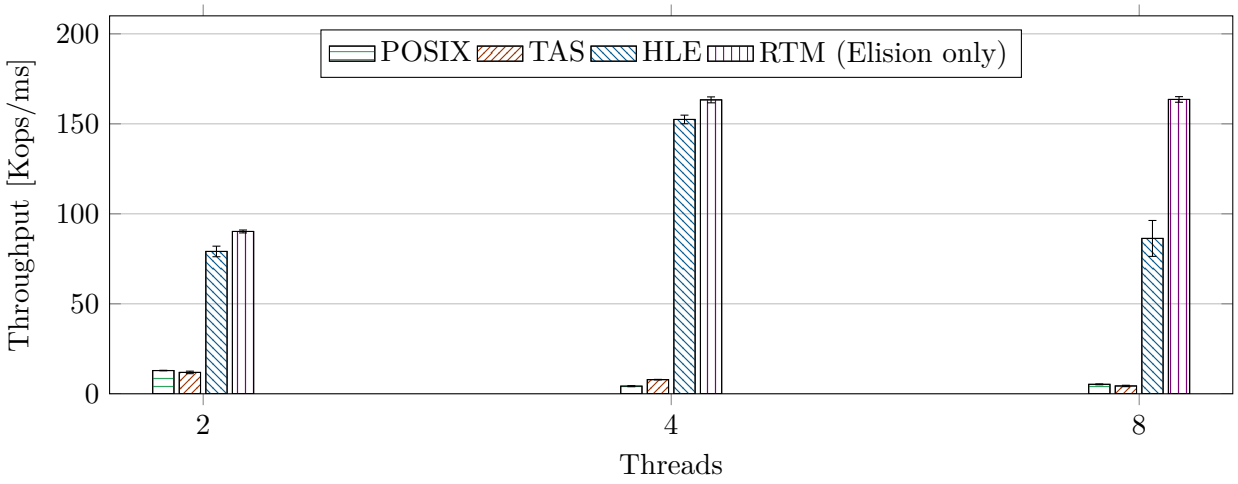


Figure 3.16: Aligned counters per core with global lock

are due to random aborts (interrupts) which have a higher impact with a global mutex. When a thread in this setup gets aborted, the global mutex is written which aborts all other threads since they speculate on the same mutex. Hence, we can argue that fine-grained mutexes can still be beneficial for HLE. In terms of performance, this only holds true as long as more mutexes do not lead to other interferences with the performance such as an increased read effort because the mutexes no longer fit in the cache.
Overall, HLE only makes small losses whereas the performance of non-elided approaches like POSIX and TAS suffers greatly because a coarse-grained mutex is used to protect the critical regions instead of fine-grained mutexes.

3.3.3 Doubly linked List

A simple data structure that finds its use in a variety of algorithms and inter alia in the MySQL read view management, is the doubly linked list. This section compares the different locking techniques on this data structure and also distinguishes between unaligned and aligned memory management.

The list, consisting of list items that have a reference to their predecessor and successor, holds references to head and tail and offers three basic operations:

insert Adds a new list item to the tail of the list

remove Searches for an item and sets the `next` pointer of the found item's `prev` item to the item's `next` pointer and vice-versa

find Traverses the list, starting from the head, and searches for an item with a certain value

Inserting at the tail usually updates only the tail of the list and not the head because the head only has to be updated if the list was empty in the first place. Thereby, the head is only read and

not written which minimizes aborts inside the find function because find only keeps the head inside the read-set and modifications of the tail do not lead to an abort as the end of the list is also determined when an item's successor is empty.

The remove function requires finding the item reference first, using the find function. Finding an item is the most expensive operation in this list implementation as it is in O(n) whereas insert is O(1). Basically, the remove function is O(1) once the reference to an item is known.

Listing 3.12: Doubly linked list find implementation

```
1 ListItem* List::find(int data) {
2   ListItem * item = this->first;
3   while (item) {
4     if (item->data == data)
5       return item;
6     item = item->next;
7   }
8   return NULL;
9 }
```

In terms of making our list thread-safe, POSIX mutex locks are easy to implement by locking the list with a global mutex before an operation and unlocking it afterwards.
When working with HTM, we need to take into account that the memory-operations `new` and `delete` will issue an abort leading to more aborts with HLE and no success at all with RTM because an elision can never be successful. Thus, to *insert* a new item, we create the item outside the critical region (since memory allocation is already thread-safe on its own [65]), lock the list afterwards, insert the created item and finally unlock the list again. One downside of this method is that the memory allocation was for nothing if the item exists in the list already - in that case, the new item has to be deleted again after the list has been unlocked. Moreover, a new list item is allocated aligned to a memory allocation that is a multiple of 64 Byte to avoid false conflicts.
The *removal* of an item is handled by locking the list, locating the item, updating the references as described above and perform the memory deallocation of the item after unlocking the list. No issues arise from deleting the item outside the critical region if the references to it are properly updated to different items or `NULL`. The same can be achieved with RTM where one has to wrap the search for items to be removed inside `_xbegin()` and `_xend()` instead of HLE locks.

The test is then run with two threads[7] and a shared list with a base size of 100 values and a value range from $0-1000$. All inserted values are stored in a per-thread-queue to make sure that we only remove items that are contained in the list. This mechanism also ensures that the list size stays roughly the same.

Figure 3.17 shows how both HTM-approaches have a solid advantage over POSIX and non-elided

[7] we encountered weird results using four threads, see Appendix Section D.2

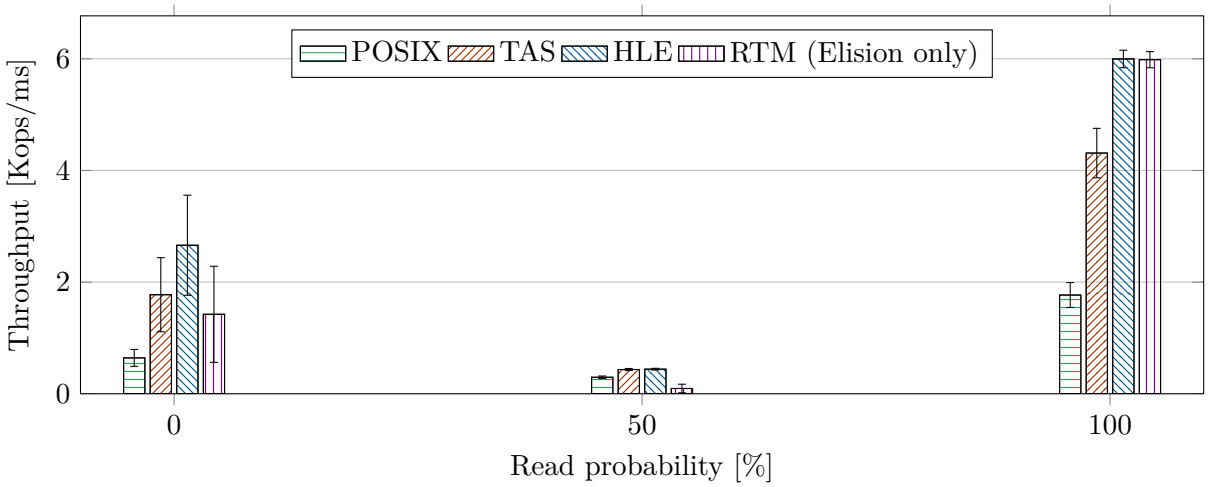


Figure 3.17: Throughputs on a list with varying read-probabilities

TAS with 100% read. HLE still outperforms other approaches with update-only (0% read), however RTM (elision only) falls behind non-elided TAS. Thus, forcefully retrying every single operation until it succeeds is not of advantage compared to the HLE approach where a pessimistic execution path is taken after an abort. However, some elisions are obviously still successful even with 100% update and eliding some of the TAS calls leads to a higher throughput than eliding none. With 50% read/50% update, the overall throughput falls to under 1000 operations per millisecond for every locking approach. In terms of HLE and RTM, this can be explained by the combination of the find method which potentially loads the whole list into the read-set with updates where a single modification is enough to abort a concurrent read-transaction and thereby the transaction itself as well. Moreover, find operations traverse the whole list which is more effort than to insert a single item at the end of the list, thus the throughput also decreases for POSIX and non-elided TAS. Yet, when all operations try to find a value in the list (read-only), no more cache-invalidations occur and the throughput increases vastly agin

It should also be noted that the list size is limited when functions are used that traverse the list (contains and remove with find in this implementation). Depending on the memory alignment of items, this size can be way smaller than the L1 DCache size. For instance, a list with an initial size of 1000 items makes elisions impossible and an RTM implementation without a fallback-path does not succeed. If we implement a List where every ListItem is allocated aligned to a memory location m with $m\%64 = 0$, false conflicts in the same cache line are avoided. However, this implementation would still only allow for as many items as there are addresses in the L1 DCache that are a multiple of 64, thus `sizeof(L1 DCache) / sizeof(Cacheline) = 32 KB / 64 B = 512` items.

3.3.4 Hashmap

This section analyzes the more complex data structure of a Hashmap which hashes each value to one of its buckets that in turn keeps a linked list of items with the same hash. However, we keep our Hashmap as simple as possible:

Fixed size The Hashmap will not be resized, thus the number of items per bucket increases with the total items

Hash modulo map size The hash-function for a value x returns `x % size`

Separate chaining with linked lists The Hashmap consists of an array where each element (*bucket*) references to the first item of a linked list that holds the values. Collisions are resolved by appending to the end of the linked list

No duplicates Duplicate inserts are ignored (thus, the list of a bucket has to be traversed on ever insert)

The Hashmap implementation offers the three basic operations insert, remove and contains. In every of these functions, the mutex is only acquired after the value's hash has been determined. The *contains* function then iterates over all items in the hashed bucket and releases the mutex either when the item has been found or when the end of the list has been reached. To *insert* an item, the new item is created at the very beginning of the function to avoid aborts that would occur if the memory allocation happened inside the critical region. Then, an iteration similar to the contains function over all list items of the bucket is performed where every value is read to check if the inset-value is already contained. If the value is found inside the Hashmap, the mutex is released and the new item is deleted again. If the value is not yet contained in the map, it becomes either the successor of the last list item or the first list item of the bucket before the mutex is released. The *remove* function also iterates over a bucket's list after hashing the value and acquiring the lock variable and unlocks the bucket afterwards. If the value to remove has been found in the map, its list item is deleted and either the item's predecessor is linked to the item's successor or the bucket's list pointer is re-referenced to the item's successor.

In the MySQL/InnoDB implementation, a global mutex is used to protect the hashmap data-structure (as shown later), hence we implement the Hashmap with a global mutex but firstly test bucket mutexes which is a more efficient approach.

The run function of each benchmark-thread randomly selects an operation according to the defined probabilities and then performs this operation on the map. We test the Hashmaps with 4 threads (equal to the number of cores) with operation-probabilities of 25% for an insert as well as a remove operation, and 50% for a contains operation. Hence, the benchmark executes 50% updates and 50% reads. Furthermore, the Hashmap is set to an initial load of 1000 values (base inserts) and values range from zero to `INT_MAX`.

Mutex per bucket

The bucket-lock implementation uses one lock variable for every bucket and its linked list. For example, to insert the value 5 into a Hashmap with 10 buckets, the `hash(5) = 5 % 10 = 5`th bucket is locked by using the mutex within the bucket. Thereby, any operation on the value 15 would have to wait for this lock to become available. This is however not the case for elisions that ignore the lock and instead check for data conflicts.

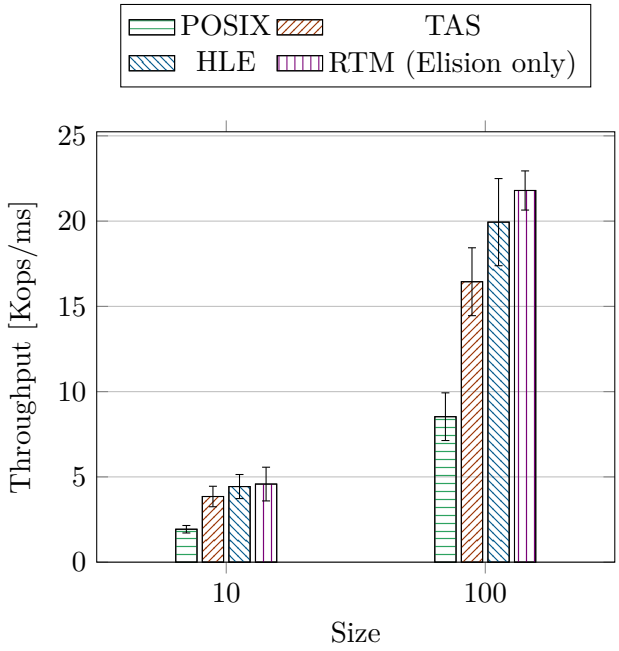


Figure 3.18: Throughputs on a bucket-locked Hashmap with 10/100 buckets

Figure 3.18 shows that HLE as well as non-elided TAS perform better than POSIX locks.

As it can be seen comparing the results of a hashmap with size 10 against a hashmap with size 100 in Figure 3.18, more buckets lead to a better throughput. Moreover, HLE achieves a performance increase of 130% and 134% with 10 and 100 buckets respectively compared to a POSIX mutex. One reason is that the values are spread on more buckets. That means that the linked list of each bucket is shorter and thus, we have to follow fewer references to reach the end of the list. For example, the Hashmap with 10 buckets has #base inserts $\div$ #buckets $= 1000 \div 10 = 100$ items per bucket in average. The Hashmap with 100 buckets only has one-tenth of that, namely $1000 \div 100 = 10$ items per bucket in average. Moreover, with more buckets also come further spread of items and a reduced likelihood of conflicts which can be shown by profiling the HLE lock function for both map sizes.

Table 3.3 shows that the aborts for a map with more buckets are significantly fewer than for a more compact map. And since the abort rate is lower, there have to be less transaction rollbacks and restarts which increases the throughput. Another interesting observation is that the share of

	10 buckets	100 buckets
HLE_RETIRED.STARTED	48×10^6	205×10^6
HLE_RETIRED.ABORTED	22×10^6	31×10^6
Transaction abort rate	45.11%	15.02%
Transactional cycles	53.06%	30.58%
Aborted cycles	34.31%	6.72%

Table 3.3: Hardware indicators for a HLE-bucket-protected Hashmap of different sizes (Figure 3.18)

transactional cycles in the total cycles are fewer when the map is bigger. This can be explained by the decrease in the aborts where only 15% of the transactions have been aborted for the Hashmap of size 100 in contrast to 45% to a map of size 10. Every abort means that cycles have to be rolled back and redone and since the re-execution is potentially executed transactionally again (if the mutex is occupied), more cycles are transactional. Hence, it is only desirable to have lots of transactional cycles if the aborted cycles are accordingly low and fewer transactional cycles might lead to better performance if the abort-rate is low in return.
The corresponding performance increases for all lock approaches when increasing the map size are shown in Table 3.4 where all data is relative to a Hashmap protected by a POSIX mutex. Therein, the HTM-approaches HLE and RTM improve slightly more than non-elided TAS, e.g. 4.5 and 4.76 times instead of 4.27.

Size	POSIX	TAS	HLE	Elision only
10	100%	199%	230%	237%
100	442%	852%	1033%	1129%
Increase	442%	427%	450%	476%

Table 3.4: Throughput increase for bucket-locked Hashmaps (Figure 3.18)

Concluding further from the thought that an increase in the map size creates a better environment for HTM, we can argue that decreasing the size to 1, thus having only a single linked list is not in favor of HLE or RTM. And indeed, a “Hashmap” with a single bucket with e.g. an RTM approach to protect the critical sections suffers a performance decrease of the factor 33 whereas the throughput of non-elided TAS is only decreased by the factor 13. Thus, a Hashmap is a way more efficient data structure to store values compared to a list in this use-case and especially Hardware Transactional Memory obviously benefits from a data structure that has a low probability of conflicts.

Global mutex

Instead of locking only a single bucket, we now use a global mutex which is acquired by the Hashmap’s function instead of using the hashed bucket’s mutex. It should be noted that the data-structures and their size remain completely the same, the only differences are an additional global mutex and the different locking-calls. Thus, the performance-differences can not be attributed to a

different memory layout, but apply only to the different locking approach.

We compare the results of a bucket-locked against a global-locked Hashmap with a size of 1000 buckets.

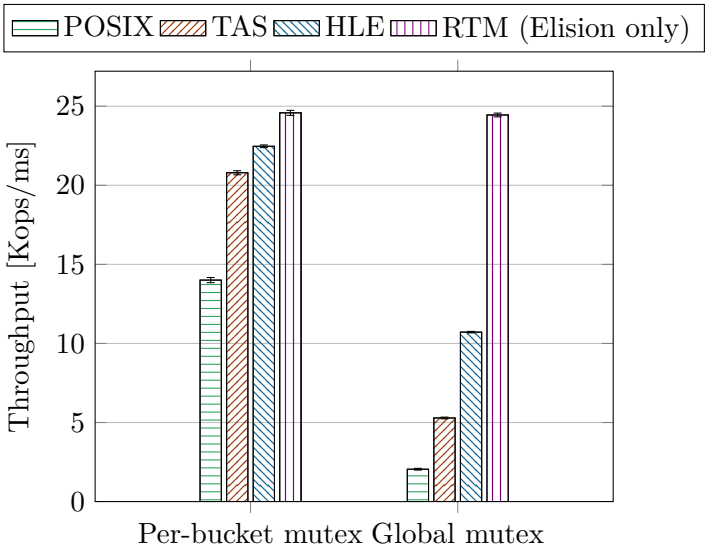


Figure 3.19: Throughputs of bucket and global locks on a Hashmap size 1000

Figure 3.19 shows that a bucket-locked Hashmap is vastly faster than a globally locked Hashmap. This can be easily explained by more spread out locks and thereby less conflicts for a bucket-lock whereas the global lock has a conflict for basically every single operation.
However, the HLE Hashmap does not seem to suffer as much from a global lock as the non-elided TAS counterpart (RTM does not use the lock at all anyway, thus there is no significant performance difference). This is because HTM does not care about the lock in the first instance but only about the data [34]. Hence, with the low conflict-probabilities of 1000 buckets, elisions are still often successful.
The reason for HLE performing worse than RTM is that if an abort occurs with a global HLE mutex (due to e.g. data conflicts or interrupts), all threads abort because they all speculated on the same lock. Furthermore, because there is only one mutex, the Lemming effect can occur where the control flow basically goes back to serial execution after an initial abort because every thread writes the mutex instead of eliding it. The condition for the Lemming effect in this case is that after one thread gets aborted and aborts all other threads by writing the mutex, the mutex has to become free quickly enough so that one of the aborted threads can write it in the non-elided TAS call. This time window is quite small but so is the operation that is executed transactionally as long as the bucket lists are not too long.
The performance decrease is in direct proportionality to the abort rate as shown in Table 3.5 where HLE with a global mutex has almost ten times (!) as many aborted cycles as with per-bucket mutexes.

	per-bucket mutex	Global mutex
Transactional cycles	22.19%	25.90%
Aborted cycles	5.81%	53.56%

Table 3.5: HLE hardware indicators of a bucket contrary to a globally locked Hashmap of size 1000

3.4 Linking with a HTM-enabled glibc

The GNU C Library (glibc) is a C library that realizes the system calls, memory allocation, printing etc. as defined in the relevant standards, such as ISO C11 and POSIX.1-2008. Glibc is used in most Unix-like operating systems, making it the de-facto standard C library [66]. Initially released by the Free Software Foundation in 1987 [67], a community-driven development process is in place since 2012 [66], [68].

Kleen *et al.* [25] provides a partly rewritten glibc that allows TSX lock elision for existing applications that use pthread mutexes and read/write locks. According to the specification, the algorithm requires RTM and stops eliding for an unspecified amount of time after an abort [25]. We assume that this means that the whole application does not start any new elisions for a small time window. This could also be a preventive measure to not execute optimistically in bad conditions and maybe even avoid the Lemming Effect by approaching pessimistically for some time before trying optimistic paths again later on.
Since the libc only comes into play in the linking phase, existing sources do not have to be rebuilt but only linked against this glibc implementation [69]. We use the most recent version 16 of the modified glibc[8] and link it with a simple test-program to demonstrate the functionality and ultimately with MySQL in the final chapter.

3.4.1 Installation

Once the library has been retrieved from git, it is advisable to compile in a dedicated folder, e.g. `build`. According to the repository readme, lock elision is enabled by default for all `PTHREAD_MUTEX_DEFAULT` mutexes and rwlock when the `--enable-lock-elision=yes` parameter is specified at configure time (otherwise one can set `GLIBC_PTHREAD_MUTEX=elision GLIBC_PTHREAD_RWLOCK=elision` before running the program). Since this is desirable for us, we call the configure command from the build directory with that parameter: `./../configure --enable-lock-elision=yes --prefix=$(pwd)`. Note that we just want to test the library and not install it as default which is the form of installation that we will assume in the following. After calling `make` and `make install`, the library can be used.

[8] https://github.com/andikleen/glibc/tree/b0399147730d478ae45160051a8a0f00f91ef965

3.4.2 Usage

Firstly, the custom glibc is linked with a naive program using a POSIX mutex.

Listing 3.13: Program using POSIX mutexes

```
1  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2  pthread_mutex_lock(&mutex);
3  /* do something... */
4  pthread_mutex_unlock(&mutex);
5  pthread_mutex_destroy(&mutex);
```

Since we did not install the glibc as standard, we need to explicitly provide the paths that lead to it. However, we were not able to do so simply by setting the `LD_LIBRARY_PATH` since we ran into relocation errors. Instead, we change the shared library search path at runtime (`rpath`) and also link to the dynamic linker of the custom glibc.

The path to the components of our glibc can be provided using the `-Wl,--rpath` flag. `-Wl,someflag` passes all arguments after `-Wl,` as a space-separated list of arguments to the linker. For instance: `gcc -Wl,a -Wl,b` eventually becomes `ld a b`.
We also need to set a different linker which can be done with `-Wl,--dynamic-linker`. Listing 3.14 shows the overall linking command for our program.

Listing 3.14: Linking with custom glibc

```
1 g++ \
2   -Wl,--rpath=/path/to/glibc-htm/build/lib \
3   -Wl,--dynamic-linker=/path/to/glibc-htm/build/lib/ld-linux-x86-64.so.2 \
4   -fgnu-tm \
5   -o program program.o
```

When we run our program and check that the pthread calls lead to elided transactions, perf confirms that lots of cycles have been executed transactionally.

When we run the sample program from Listing 3.13 with a simple addition in the critical region, perf shows us 75.20% transactional cycles with 0% aborts. If we change the mutex lock call to `while (pthread_mutex_trylock(&mutex) != 0) _mm_pause()`, the perf output shows 96.12% aborts, hence using `pthread_mutex_lock` results in less aborts although the overall throughput might still be better when implementing a spin-lock and using the TAS function due to reduced wait times.

Since we found that the case of meeting an initialization or locking method that does not work with the elided glibc happens fairly often[9], our suggestion is to test the used combination in a small

[9] Also see Appendix Section B.1 for pitfalls with this glibc

sample with perf and to consult the project homepage of the glibc implementation before using the (assumed) elided mutexes in a larger system.

This HTM implementation of the glibc does not provide all libraries, for instance the C++ standard library `libstdc++` is missing. Since the simple program we just linked does not make use of any standard components, this is not an issue but more complex programs need to be linked with components of the custom glibc as well as the standard one. This can be done by appending the path for other components to the `rpath`, separated by a colon. The final `rpath` value that we use is shown in Listing 3.15.

Listing 3.15: rpath pointing to custom and standard components

```
1  -Wl,--rpath=/path/to/glibc-htm/build/lib:/usr/lib/x86_64-linux-gnu:/lib/x86_64-
       linux-gnu
```

The successful linking can also be confirmed using the Linux command `ldd` which lists all library dependencies of a program. For our program, the output of `ldd` is shown in Listing 3.16.

Listing 3.16: Library dependencies of the linked program (without memory addresses)

```
1 $ ldd program
2         linux-vdso.so.1 =>
3         libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6
4         libm.so.6 => /path/to/glibc-htm/build/lib/libm.so.6
5         libitm.so.1 => /usr/lib/x86_64-linux-gnu/libitm.so.1
6         libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1
7         libpthread.so.0 => /path/to/glibc-htm/build/lib/libpthread.so.0
8         libc.so.6 => /path/to/glibc-htm/build/lib/libc.so.6
9         /path/to/glibc-htm/build/lib/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86
              -64.so.2
```

The last line refers to the linker that is used. This output also shows that not every library is linked with the custom glibc, but e.g. `libstdc++.so.6` and `libitm.so.1` still point to the standard components.

CHAPTER 4

Database Concurrency Control using Intel TSX

While the focus in the previous chapters has been put on micro benchmarks to understand the application area of Hardware Transactional Memory, this chapter explains crucial concepts and structures in MySQL's storage-component InnoDB. Finally, we suggest different modifications to integrate HTM in InnoDB's Concurrency Control with the goal of providing improved scalability to concurrent clients and to solve the issue of a performance loss with too many parallel queries [70]. These modifications will then be evaluated in the next chapter.

There are many database management systems (DBMS) that one can choose from, such as the relational implementations Oracle Database, MySQL, Microsoft SQL Server, PostgreSQL, IBM DB2, Microsoft Access, SQLite, SAP Hana, MariaDB and also NoSQL systems including MongoDB and CouchDB (document stores), Neo4J (Graph DBMS), Apache Cassandra and HBase (wide column stores), among others [71]. Despite the long list of options, most databases support ACID transactions and all of them need to have some form of concurrency control [72], [73]. The informal term "ACID transactions" has been created by Haerder and Reuter [74] in 1983 and stands for:

Atomicity "all or nothing" guarantee for transactions (either all changes are committed or none)

Consistency a transaction must leave the database in a consistent state, thus be valid according to all defined rules

Isolation describes how two concurrent transactions do not see each others updates until they are committed

Durability guarantees that the changes made in a transaction will be visible to subsequent transactions

The parts Atomicity and Isolation are typically implemented by a locking protocol in combination with logging [72, p. 220].
To be able to fulfill the "I" in ACID and thus execute a transaction in isolation without it seeing

any concurrency anomalies, three broad techniques exist:

Two-phase locking (2PL) shared locks on every data record are acquired before reading it, and exlusive locks before writing it. The locks are only released atomically at the end of the transaction and transactions block in a wait-queue until the lock can be acquired [73], [75]

Multi-Version Concurrency Control (MVCC) transactions do not acquire locks but are assigned a consistent view of the database in the past to instead. On an update-commit, the new data is added and the old data is marked as obsolete (thus, data is not really updated but newer versions of it are inserted)

Optimistic Concurrency Control (OCC) records can be read and written by multiple transactions without blocking but transactions maintain a read- and update-history and check for isolation conflicts before committing. If conflicts occurred, the conflicting transaction is rolled back (STM technique!)

As in most Computer Science topics, each technique has its pros and cons: OCC, for instance, is advantageous with read-heavy workloads because transactions do not block and can execute concurrently whereas 2PL performs better in high-contention environments since synchronization needs to happen anyway [72, p. 221]. On the other hand, 2PL does often not scale well with an increased multi-programming level due to the introduced blocking overhead [76]. While MVCC generally allows transaction concurrency, it introduces the overhead of many different versions that grows with each concurrent update and the obsolete dead rows have to be removed when no transactions refer to them anymore [77]. However, the techniques can also be combined: one example is multiversion two-phase locking (MV2PL) where transactions are initially categorized as either read-only or as involving write. Read-only transactions are executed right away by assigning them to the current version of the database and a transaction modifying data waits for transactions modifying the same data [78].
When a database guarantees strict serializability, this usually comes at the cost of decreased concurrency. Hence, attempts to increase the concurrency instead provide weaker semantics than serializability which might not be necessary for all applications [72, pp. 224–225]. Therefore, the ANSI SQL standard defines four Isolation levels [72], [79]:

1. **READ UNCOMMITTED** transactions can read any version of data, even if it has not been committed

2. **READ COMMITTED** transactions can read any *committed* data, different values are potentially read for repeated reads

3. **REPEATABLE READ** only one version of committed data will be read but newly inserted tuples, so-called "phantom rows", might be read during a consecutive read

4. **SERIALIZABLE** fully serializable access

This list goes from weak isolation levels to strong ones and does not include additional levels defines by various vendors, such as SNAPSHOT ISOLATION where a transaction operates on a snapshot of the database at the time the transaction began and conflicts are detected by timestamps [72, pp. 226–227].

In this context, it is further important to distinguish between locks and latches: in database terminology, a latch is a rather fast-executed mutex that is often implemented using hardware instructions and usually resides in memory near the resources the protect. To acquire a lock in contrast, usually takes some time because the lock request if first placed into a queue and only dequeued in a particular order after the lock is given [72], [80]. Furthermore, there is a difference in deadlocks where locks are allowed to produce deadlocks that are then detected whereas latch deadlocks must be avoided.

So far, the described techniques have a broad scope of application and are generally used in a range of different database implementations. In the following, we want to perform concrete modifications and it is therefore important to narrow our inspection to a single implementation.
We chose to go with MySQL, "the world's most popular open source database" according to the official website `www.mysql.com` (visited on 06/10/2014). One of the main reasons next to its popularity and the fact that it is open source was for us that a new approach to implement Serializable Snapshot Isolation has been proposed in a previous paper [81] and we had the source code for MySQL 5.6.10 available as well as some prior knowledge about internal data structures.

MySQL is quite unique in the way that it supports a variety of storage managers [72, p. 238]. Since version 5.5.5, the storage engine InnoDB has replaced the previous MyISAM implementation [82] due to several reasons one of which is that InnoDB is capable of row-level locking whereas MyISAM implements table-level locks. Furthermore, InnoDB provides some useful features such as *transactions*, foreign keys and relationship constraints and an enhanced crash recovery [83]–[85].

4.1 InnoDB internals

This section narrows the elaboration down to concrete concepts and implementations used in the InnoDB storage engine that are of relevance to us and that are a key for the following modifications.

InnoDB uses most of the concepts previously explained, such as a combination of two-phase locking with a multi-versioning database to ensure ACID transactions and exactly the four standardized transaction levels with the default being REPEATABLE READ [86].

4.1.1 Multi-granularity locking

Two general types of locks are used in InnoDB: shared (S) locks and exclusive (X) locks where the shared lock is used to read a row and the exclusive lock to update or delete a row. Obviously, shared locks can be granted to multiple transactions at once whereas the exclusive lock does not allow such. Additionally, coexistence of locks is permitted by multi granularity locking which introduces *intention locks*. Intention locks are table locks that indicate which type of lock will be required later on for a row in that table by a transaction [87]. These intention locks again exist in the two types shared and exlusive which leaves us at 4 kinds of locks total. Table 4.1 shows their compatibility. A lock is then only granted if it is compatible with existing locks. This is also the context in which

	X	IX	S	IS
X	Conflict	Conflict	Conflict	Conflict
IX	Conflict	Compatible	Conflict	Compatible
S	Conflict	Conflict	Compatible	Compatible
IS	Conflict	Compatible	Compatible	Compatible

Table 4.1: Compatibility of InnoDB locks [87]

deadlock detection is performed, e.g. when transactions introduce circular dependency [87]. In some descriptions of multi granularity locking, an additional null lock (NL) is introduced to indicate that no lock is requested. The null lock is compatible with everything [32].

4.1.2 Transaction locks

InnoDB uses two kinds of data locks to ensure correct concurrent execution: table locks and row locks. Whereas row locks are at a fine granularity and allow different threads to access different parts of the table without interfering with each other, table locks are necessary to avoid the alteration of a table by one transaction when another transaction is using it [88].

Table locks When a table lock is acquired to ensure that no other transactions modify the structure of the table when the table or rows in this table are accessed or modified, the lock is usually intentional (either intentional shared with `LOCK_IS` or intentional exclusive `LOCK_IX`). Intentional locks are converted to explicit locks only when required [89, pp. 736–737]. The shared lock is used for read-only transactions whereas the exclusive lock would be used for e.g. inserts. Table locks are only taken if the transaction is not already holding an equal or stronger lock on the table and there is not another transaction that already holds an incompatible lock on the table. If the latter is the case, we need to wait for this lock [88]. These steps to acquire the table lock obviously have to be protected by mutexes to work in a multithreaded environment. In this case, this is done by the system wide `lock_sys->mutex` that protects the search for compatibility with other tranactions' locks and the creation of the new lock. Additionally, the transaction's mutex is

acquired before the query thread is enqueued for waiting or the table lock is created. After the table lock has been created, it is added to the list of transaction locks of the transaction's requested locks as well as the list of locks in the table. The table's locks list is then checked when another transaction searches for incompatible locks.

Row Locks A row is uniquely identified by its space identifier, the page number within the space and the heap number of the record within the page [88], [90]. Row locks have implicit and explicit locks as well: explicit row locks make use of the global row lock hash table whereas implicit row locks "are logically arrived at based on the transaction information in the clustered index or secondary index record" [88]. For instance, after a transaction has just modified or inserted an index record, it still owns an implicit lock on the record and therefore does not have to acquire an exclusive lock. This means that another transaction needs to determine whether the record is already locked implicitly by another transaction before it can acquire a row lock.
Explicit locks (`LOCK_X`) are managed in a hash table within the global `lock_sys` object. The hashing is based on the record's page address, hence all locks on the same page are in the same hash bucket and since there are multiple records in a page, locks of different records are possibly mapped to the same hash bucket [88], [90] where they are handled in a single linked list. When acquiring an explicit lock, the first step is to search the hash table for an explicit row lock of the transaction with equal or stronger lock mode. If such a lock is not found, the hash table is used again to check if other transactions have a conflicting lock on the row (if yes: wait). Afterwards, the lock is created by setting the appropriate bit in the lock bitmap based on the heap number of the row, then inserted in the global hash table of row locks and added to the list of the transaction's locks. All locks of a transaction are released at the end of the transaction, so either on commit or rollback. With an isolation evel of `READ COMMITTED`, the SQL layer can even release the locks before the transaction ends [88].
Obviously, the hash table managing the row locks is heavily accessed: for instance, a run of the later introduced txbench led to approximately 2.5 million retrieved hash cells per second (without network overhead).
It also needs to be protected by mutexes again which is done with the global `lock_sys->mutex` and transaction level mutexes again.
Furthermore, a lock mutex (`lock_sys->wait_mutex`) is used to protect operations on the lock wait table which manages threads that are waiting on a lock.

4.1.3 Function hierarchy

The source code of InnoDB is organized in the manner of putting most logical operations in their own methods (or macros) even if it is just a simple function such as a test-and-set. This allows to adjust to different operating systems with more ease and we are also in the position to change certain code parts quickly. Figure 4.1 shows the call-graph for the relevant function in the InnoDB

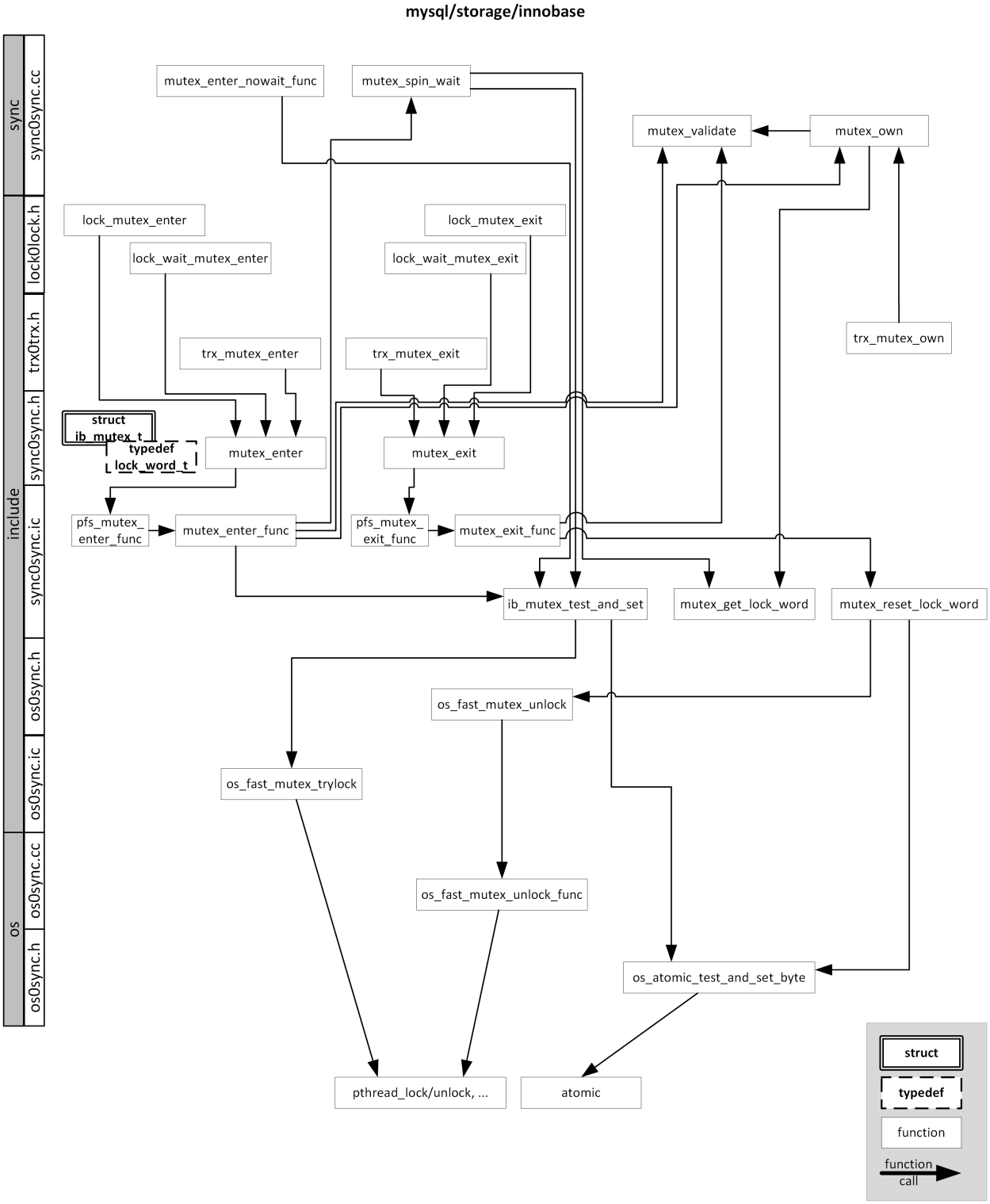


Figure 4.1: InnoDB Concurrency Control call-graph

Concurrency Control where the gray rectangles on the very left denote the folders (packages) relative to the `storage/innobase` directory inside MySQL and the white rectangles slightly to the right of the directories the file names. For instance, the function `mutex_enter_nowait_func` in the top left corner resides in the file `sync0sync.cc` in the directory `storage/innobase/sync`.

While we already discussed the syslock (`lock_mutex_enter`) and transaction mutexes (`trx_muter_enter`), the `lock_wait_mutex` is acquired when threads are put to wait for a lock to be released.
Generally, a mutex does not only hold the actual mutex value in its `lock_word` member where `lock_word_t` is defined as `LONG` on Windows and as `byte` otherwise but also additional information such as the PID of the thread that acquired the mutex or a performance schema instruction hook.
It is further shown that the lowest functions above OS-specific functions are `ib_mutex_test_and_set`, `mutex_get_lock_word` and `mutex_reset_lock_word` - they then call specific OS functions depending on availability. If the system has atomic builtins for example, `os_atomic_test_and_set_byte` will be called, otherwise `os_fast_mutex_trylock`. These OS functions are then defined in the InnoDB directories and files containing `os`.
We also observe that the actual `mutex_enter_func` and `mutex_exit_func` functions are wrapped in a `pfs` function which is responsible for performance analysis in the context of the MySQL Performance Schema[1] [91].
The functions `mutex_own` and `mutex_validate` are mostly used for assertions, e.g. when a function relies on a mutex that it does not acquire itself but that it should own according to the call-order where a previous function already acquired the latch.
This call-graph is not complete but it outlines the most important dependencies and call-hierarchies.

Moreover, we find that a central method of InnoDB's Concurrency Control, the `mutex_spin_wait` function, is implemented in a similar manner to the TAS-T spin-lock.

Listing 4.1: Excerpt of InnoDB mutex_spin_wait

```
1  UNIV_INTERN
2  void
3  mutex_spin_wait(
4    ib_mutex_t* mutex,    /*!< in: pointer to mutex */
5    // ...
6  {
7    ulint   i;    /* spin round count */
8    i = 0;
9
10 spin_loop:
11   while (mutex_get_lock_word(mutex) != 0 && i < SYNC_SPIN_ROUNDS) {
12     if (srv_spin_wait_delay) { // srv_spin_wait_delay is set to 6
13       // ut_delay delays for the argument in microseconds on 100 MHz Pentium
```

[1] the pfs wrapper is only be used if `UNIV_PFS_MUTEX` is defined which it is by default

```
14          ut_delay(ut_rnd_interval(0, srv_spin_wait_delay));
15        }
16        i++;
17      }
18
19      if (ib_mutex_test_and_set(mutex) == 0) {
20        return;
21      }
22      i++;
23      if (i < SYNC_SPIN_ROUNDS) {
24        goto spin_loop;
25      }
26    }
```

Listing 4.1 shows that the `spin_loop` begins with a `while-loop` that delays until the mutex is no longer occupied which is the test part of the function. After the mutex is free, a test-and-set of the mutex is attempted which returns on success. If the TAS call fails, we go back to the spin-loop and test again until the mutex is free. Thus, the implementation only differs in that it tests first and tries to set afterwards (one could name this T-TAS) whereas the TAS-T implementation tries to set straight away and only tests if setting the mutex fails in the first place.

4.2 Modifications to apply HTM

Since earlier benchmarks have shown that Hardware Transactional Memory is inter alia limited in its size and duration, using it for a complete transaction or query would exceed these limitations. Instead, we aim for the latches (mutexes) protecting these transaction locks which should mostly be within the constraints.

Based on the previous section that analyzed the InnoDB internals, we identify several approaches to include HTM in the InnoDB Concurrency Control:

low-level HLE modify the lowest-hierarchy calls that actually write the mutex, therefore replace *all* writes to the mutex with HLE

low-level retried modify the lowest-hierarchy calls with an RTM version that retries the elision 100 times after an abort before using the latch

glibc compile MySQL with a custom HTM glibc - similar to the low-level calls but without implementing HTM on our own, only with different linking

lock-mutex use the TAS-T implementation with HLE for the system-wide lock mutex that protects inter alia the hash map of all row locks

transaction latches use the TAS-T implementation with HLE for the latches protecting the locks of a transaction

It is worth to note that the versions *low-level HLE*, *low-level retried* and *glibc* all target *all* mutex writing functions in the application, only with different behavior: *low-level HLE* applies HLE and thus uses an implicit latch-fallback after an abort whereas *low-level retried* doesn't fall back to write the latches before retrying the elision several times. With the *glibc* version, a system-wide timeout stops elisions for some time, thus potentially avoiding the Lemming Effect.

Another mutex that could be targeted is the `wait_mutex` which we will not modify in particular because the operations inside the critical region defined by this mutex often involve waiting operations and system calls when a thread is queued to wait for a lock to become free. As discussed earlier, operations with a long duration and system calls are not advantageous for HTM.

4.2.1 Targeted functions

To implement the version targeting the most low-level calls to the mutex, we adjust `ib_mutex_test_and_set` and `mutex_reset_lock_word` to use HLE with the calls `__hle_acquire_test_and_set4` and `__hle_release_clear4` from the `hle-emulation` header as illustrated in Figure 4.2. Therefore, we also need to modify the `ib_mutex_t` struct by changing the definition of the `lock_word` to `unsigned`. No modifications are necessary in the function `mutex_get_lock_word` since it only returns the value of the `lock_word`.

When we target only specific mutexes, e.g. the lock-mutex, we need to start higher in the call-graph as shown in Figure 4.3. Here, we leave most calls as they are and add new macro- and function-definitions, `hle_mutex_enter` and `hle_mutex_enter_func` as well as `hle_mutex_exit` and `hle_mutex_exit_func` that in turn use the HLE functions defined in the `hle-emulation` header. The `hle_mutex_enter_func` is implemented with the TAS-T lock algorithm in Section 2.4.3. After applying these changes, we only need to change the respective macro definitions, in this case `lock_mutex_enter` and `lock_mutex_exit` to point to the new functions. The trx-locks can be changed accordingly by modifying the macros `trx_mutex_enter` and `trx_mutex_exit`.

4.2.2 RTM implementation with fallback path

To apply RTM to the low-level mutex calls, we proceed in a similar manner as with HLE and introduce the procedures `rtm_mutex_enter`, `rtm_mutex_enter_func`, `rtm_mutex_exit`, `rtm_mutex_exit_func`. However, instead of the `hle-emulation` header, we add our own file `myrtm.h` and `myrtm.cc` to define and implement the RTM lock function. To get MySQL running, the implementation file `myrtm.cc` has to be linked during compilation which can be achieved

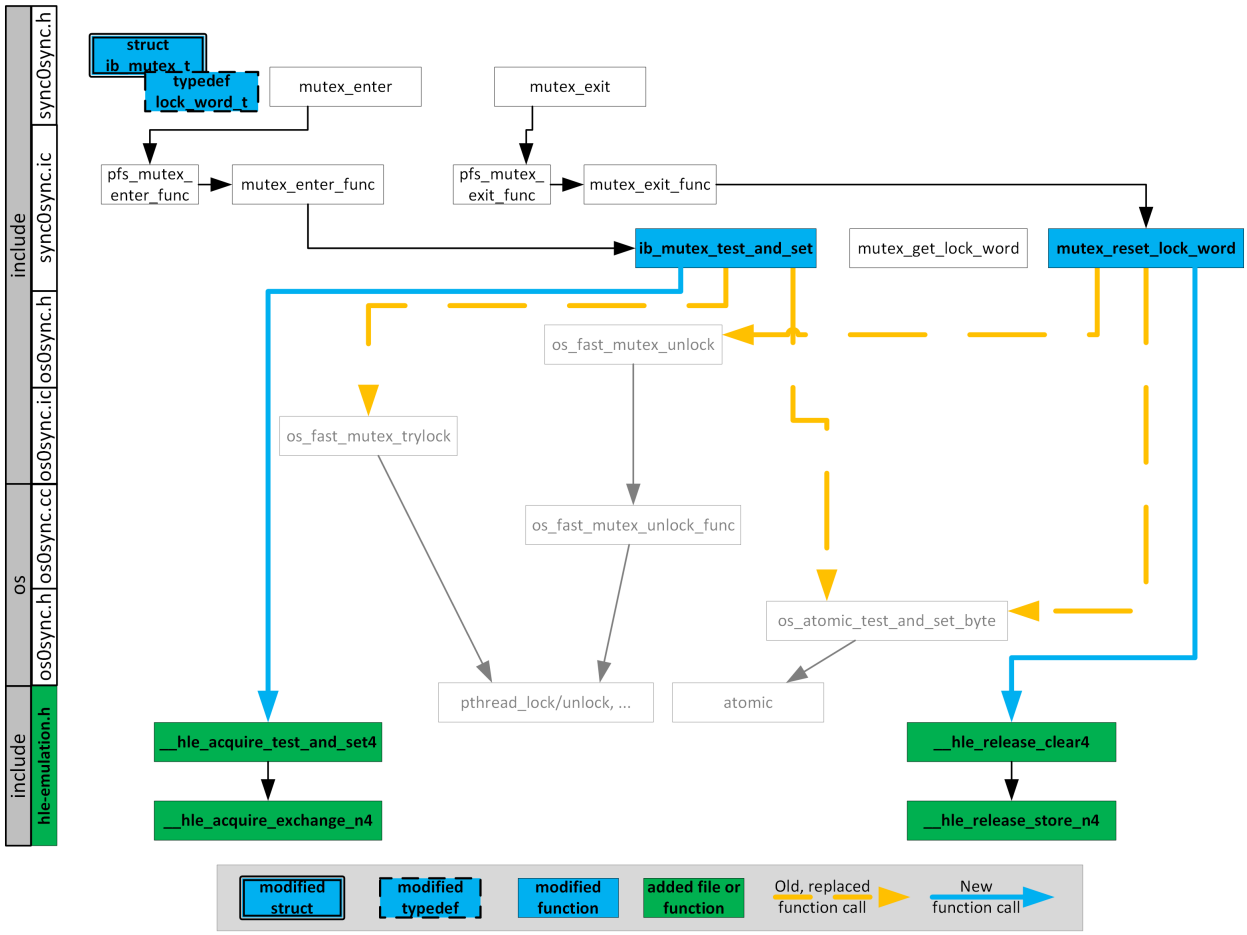


Figure 4.2: Modifications to make all low-level mutex calls use HLE

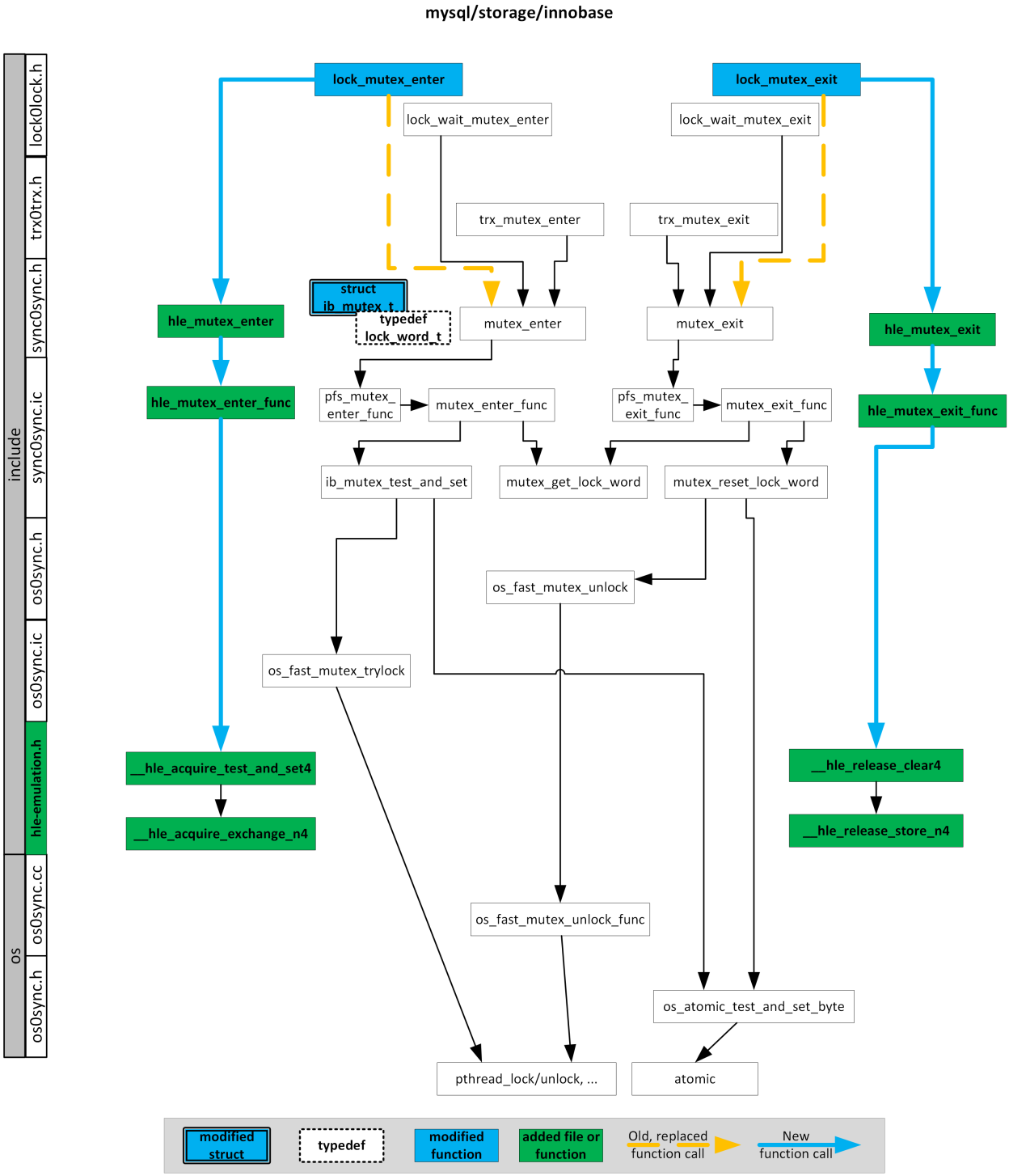


Figure 4.3: Modifications to use HLE for the `lock_mutex`

by adding it to the list of `INNOBASE_SOURCES` in `storage/innobase/CMakeLists.txt`. Furthermore, the `cmake CXXFLAGS` need to be extended by `-fgnu-tm -mrtm` to allow RTM. This can be done by passing the option `-DCMAKE_CXX_FLAGS="-fgnu-tm␣-mrtm"` to `cmake`. The actual RTM functions are implemented as described in Section 2.1 with 100 retries and a fallback path if the elisions haven't succeeded after the maximum amount of retries. Moreover, Section 3.1 has shown that it is necessary to keep the mutex in the read-set, thus we check the value of the mutex and abort if the mutex is set already. With the fallback path in mind, the unlock function distinguishes between elided and non-elided execution and either commits the transaction or unlocks the mutex.

4.2.3 Relinking with the HTM-enabled glibc

A special case is the version utilizing the HTM-enabled glibc.
To link MySQL with this glibc, we modify the `mysql-5.6.10/CMakeLists.txt` in the MySQL root directory and extend the flags `DCMAKE_C_FLAGS` and `DCMAKE_CXX_FLAGS` with our glibc as shown in Listing 4.2.

Listing 4.2: Define CMakeLists.txt to link MySQL with glibc

```
1 SET(HTM_GLIBC_FLAGS "-Wl,--rpath=/path/to/glibc-htm/build/lib:/usr/lib/x86_64-linux
    -gnu:/lib/x86_64-linux-gnu -Wl,--dynamic-linker=/path/to/glibc-htm/build/lib/ld-
    linux-x86-64.so.2")
2 SET(CMAKE_C_FLAGS "${HTM_GLIBC_FLAGS} ${CMAKE_C_FLAGS}")
3 SET(CMAKE_CXX_FLAGS "${HTM_GLIBC_FLAGS} ${CMAKE_CXX_FLAGS}")
```

Although this passes the flags to all components, only $\approx 0.53\%$ transactional cycles can be measured when excluding InnoDB caused by components such as the metadata lock (MDL [92]).

Since atomic test-and-set functions of the operating system are used by default and not the POSIX implementations, we disable all `#define HAVE_ATOMIC_BUILTINS` in the innobase container. This way, the `pthread_mutex_trylock` is used which is the POSIX implementation of a TAS function[2]. Furthermore, the default type of the `pthread_mutex_attr_t my_fast_mutexattr` is set to `PTHREAD_MUTEX_ADAPTIVE_NP` which does not allow for elisions as mentioned previously. Hence, we introduce the `pthread_mutexattr_t my_mutexattr_htm` as an attribute that is not set to any type. This attribute is then used in the mutex initializations in the `os_fast_mutex_init_func` function (`storage/innobase/os/os0sync.cc`).

[2]actually, the OS lock function is used, hence if the distribution is built on Windows, no POSIX is used at all

CHAPTER 5

Evaluation of HTM in MySQL/InnoDB

After applying all modifications of the previous chapter, we can now analyze the results. Therefore, we use the tx-benchmark with one client-server and one MySQL-server[1].

5.1 MySQL configuration

Our MySQL version 5.6.10 is configured with the following parameters which have been adopted from the configuration in a previous paper targeting similar components within the database [81]:

max_connections	1000
performance_schema	OFF
table_open_cache_instances	32
query_cache_type	0
query_cache_size	0
innodb_buffer_pool_size	2.5 GB
innodb_log_file_size	256 MB
innodb_log_buffer_size	16 MB
innodb_flush_method	fsync
innodb_flush_log_at_trx_commit	2
Indexing	ON

Table 5.1: MySQL configuration

To make sure that unnecessary IO-overhead by writing the log-state on the hard-drive is avoided, we use the temporary filesystem (`tmpfs`) for MySQL's tmpdir. This can be achieved by creating an according directory e.g. in the `/tmp` directory, specifying it as a temporary filesystem in `/etc/fstab` with the line `tmpfs /tmp/mytmpfs tmpfs rw,uid=123,gid=456,size=2G,nr_inodes=10k,`

[1]The hardware and environment of the server running MySQL can be found in Appendix Chapter A

`mode=0700 0 0`, mounting it and either appending the line `tmpdir=/tmp/mytmpfs` below the `[mysqld]` section to the `my.cnf` file [93] or passing the option `--tmpdir=/tmp/mytmpfs` to the `mysqld` process. The query `SHOW VARIABLES LIKE 'tmpdir'` should then show the updated directory after a MySQL restart [94].

5.2 Benchmarking with the txbench

To measure the performance of the different MySQL versions, we use the txbench presented by Jung, Han, Fekete, *et al.* [81].

This benchmark uses three tables `txbench-{1, 2, 3}` containing two non-null integer columns (one of them is the primary key) and ten variable sized TEXT columns `b_value-{1, 2, ..., 10}`. The tables are then populated randomly with 100K items and can be queried by either read-only transactions or read-update transactions where an update is issued after the read. With an isolation level of `READ COMMITTED`, transactions can only read committed data but read different values for repeated reads potentially.
The *read-only transaction* consists of a single Select-From-Where query and reads 100 rows:

```
1 SELECT sum(b int value) FROM txbench-i
2 WHERE b int key > :id and b int key <= :id+100
```

The *read-update transaction* reads items from `txbench-i` and updates rows from `txbench-j` where $j = i + 1 \mod 3$. A read-update transaction reads 100 rows and modifies 20 rows.

```
1 UPDATE txbench-((i+1)%3) SET b value-k = :rand str
2 WHERE b int key = :id1
3 OR b int key = :id2
4 OR ... b int key = :id20
```

The benchmark then outputs the times for (1) the query without the commit, (2) the commit only and (3) the composition of both, each of which is measured per thread. We further extended our version to provide standard deviations and a total throughput. Figure 5.1 illustrates the composition of the two queries.

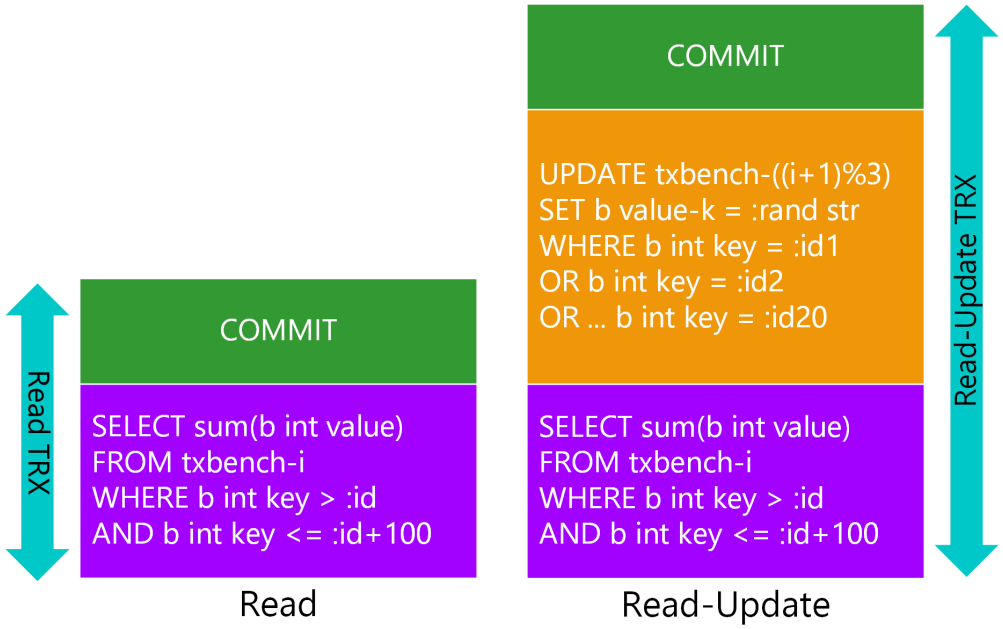


Figure 5.1: txbench composition

5.3 Comparison and analysis of the applied changes

After applying the modifications described in Section 4.2, we can now compare the performances of the different versions using the txbench with 75% read-only and 25% read-update transactions.

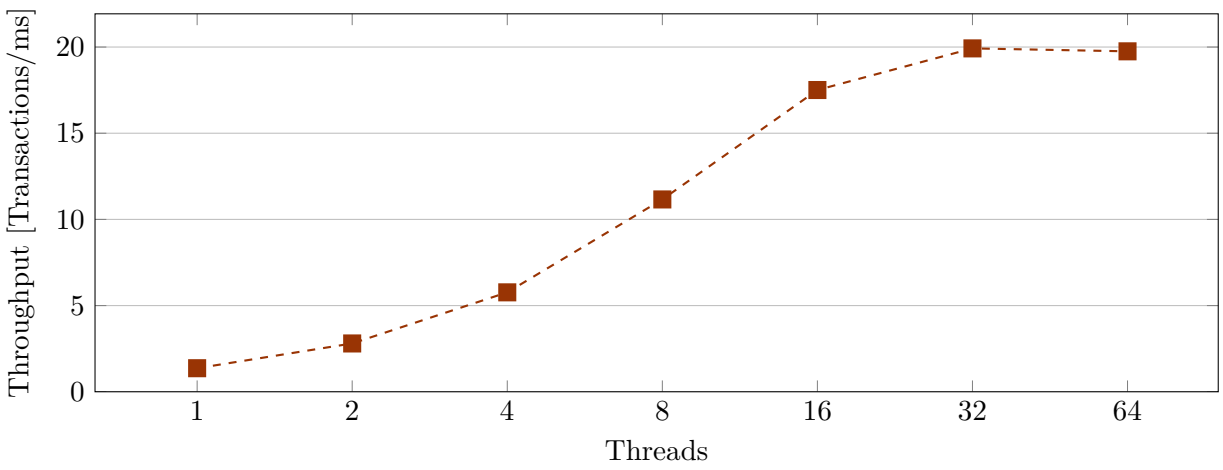


Figure 5.2: Unmodified MySQL 5.6.10 throughput as measured by the txbench

Figure 5.2 shows the throughput of the unmodified MySQL 5.6.10 version subject to the multi-programming level or the amount of client-threads. It can be observed how up to 16 threads, the slope of the graph is almost linear (twice as many threads would ideally lead to two times the throughput). However, the transition from 16 to 32 connections already shows a decreased slope

and from 32 to 64 threads, the throughput almost remains unchanged.

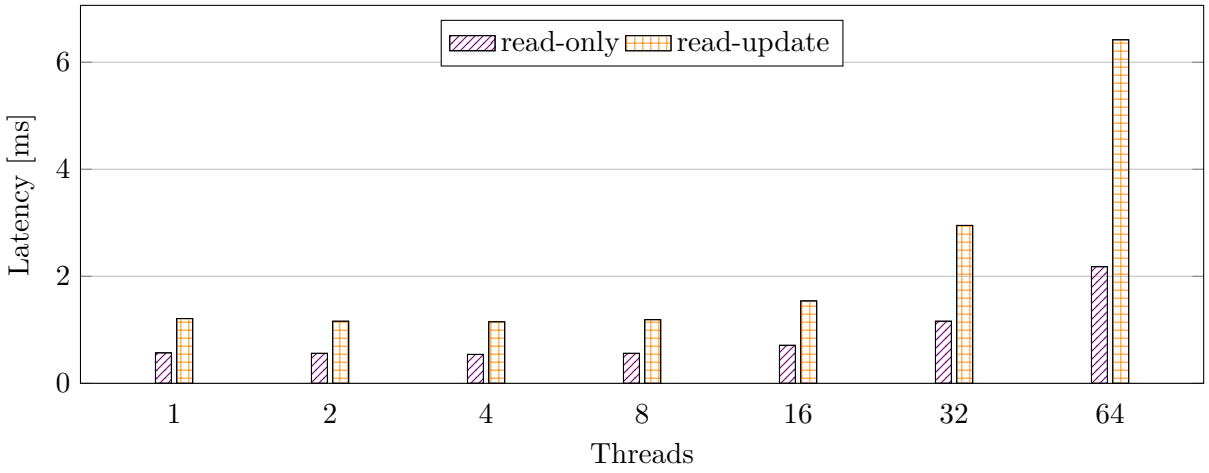


Figure 5.3: Transaction latencies (unmodified version)

This per-thread throughput decrease can be shown especially well by analyzing the transaction-latencies displayed in Figure 5.3 where the read-update latencies stay roughly the same up to 16 threads and then double on the transition from 32 to 64 threads.

The read-update transaction performs more work than the read-only transaction and thus has a higher latency. From one to eight threads, there is no noticeable difference in the transaction latencies which indicates perfect scaling. The reason for the good scaling even with eight threads - which is more than the four cores on our machine - is that the cores are not fully busy due to network latencies. Thus, with eight client-threads firing requests and waiting for responses, the delay between receiving the response and firing a new request is apparently large enough to have only four threads effectively running concurrently on the MySQL-server. Analyzing the CPU utilization for four client connections with the Linux processor activity tool `top`, the load of each core is approximately 50%. However, when it comes to 16 threads, the cores are under a 100% workload and we can observe a first increase in the transaction latency. This increased waiting time is even more visible when it comes to 32 and above all 64 threads where the latency is twelve times as big as initially. Furthermore, the read-update latency increases more than the latency of the read-only transaction which indicates that a fair amount of the performance decrease has its cause in the locking system. Whereas a read-only query will only be assigned a temporary read-view, read-update threads requires exclusive access on data that will be modified.

Ultimately, when running the benchmark with 256 concurrent threads (not shown in the figure), the performance even drops below the throughput of 16 threads and the read-update latency is ten times as much as with a fourth of the threads. In a different benchmark, MySQL was queried with up to 8192 simultaneous database connections and the performance measured in transactions per second falls close to zero [70]. This illustrates how serious scaling issues arise when there are too many concurrent requests to a database and a single client's request takes increasingly more time the more concurrent requests are processed.

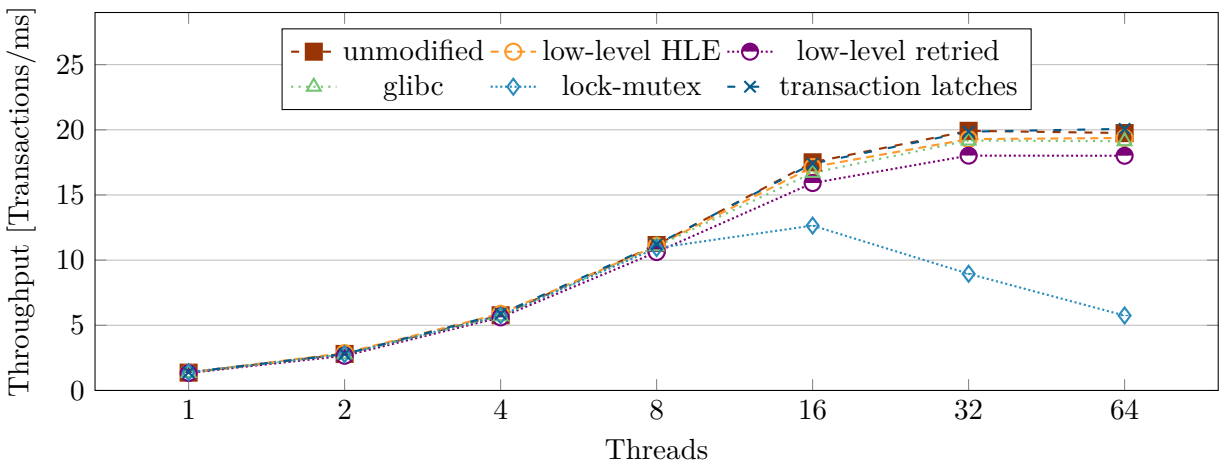


Figure 5.4: Comparison of different MySQL versions

When we compare the unmodified version with the modified versions in Figure 5.4, we observe that the modifications do not increase the throughput. Even worse, some versions show a throughput-decrease compared to the unmodified version, especially the modified system lock-mutex collapses with more than eight threads and decreases the performance to less than a third (29%) of the unmodified version with 64 threads. The version replacing all low-level calls with HLE is only slightly worse and has a throughput fairly similar to the glibc which also replaces all POSIX mutex calls but with a different technique that does not elide for some time after an abort. When we retry the low-level calls more often, the throughput becomes even worse. By replacing the accesses to the transaction latches with elided calls, nothing really changes; the throughput remains unchanged to the unmodified version.

We can elaborate further on these observations by using the perf tool to analyze the transactional and aborted cycles per amount of connections and per version (see Figure 5.5). First of all, an issue that can be observed is that more transactional cycles often come with more aborts. For instance, the low-level HLE version has the most transactional cycles with approximately five percent of all cycles compared to the other versions but with over a third of these cycles aborted again, it also has the most aborts. In contrast, the lock-mutex version has only a share of 0.04% aborted cycles compared to the total cycles or 4% of the transactional cycles but there are also only 1% cycles that are transactional. A similar effect can be observed with the version with modified transaction latches and the glibc has the highest abort rate relative to transactional cycles, with almost half of them aborted.

The modification of the Transaction latches has almost no transactional cycles because there is only a small number of elements in the transaction's list of acquired locks - our measurements showed a mean value of 1.5 items.

When we compare the glibc version with the low-level HLE version, we observe that the glibc version has less transactional cycles. Because no new elisions are started after an abort, less transactional cycles are obvious. However, the aborted cycles decrease less than the transactional ones compared

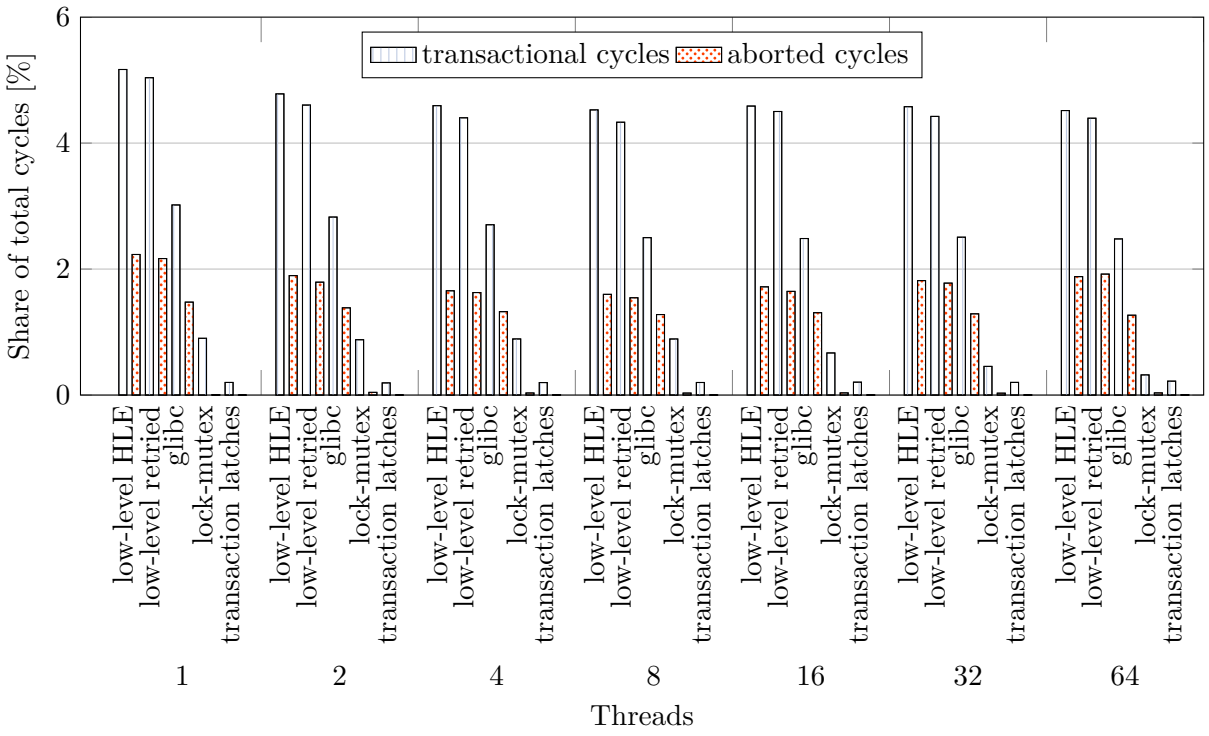


Figure 5.5: Transactional and aborted cycles of the modified MySQL versions

to the low-level HLE version, leading to some slight performance deficits. Adding the retried low-level version to this comparison, we can observe that it has generally marginally less transactional cycles than the non-retried version and roughly the same amount of aborted cycles. We therefore conclude that the conflicts that lead to aborts in the low-level HLE version can not easily be resolved by simply retrying the operation and more sophisticated modifications in the data structures and algorithms would be necessary to decrease the aborts.

Ultimately, the lock-mutex version has only 1% transactional cycles in the beginning and this number decreases once 16 concurrent threads are using the capacity of the cores. This provides an explanation for the sudden performance collapse in Figure 5.4 and with 64 threads, only 0.3% of the cycles are transactional. Since the aborted cycles remain the same compared to total cycles, their share of transactional cycles obviously rises. For instance, with eight threads, 4% of the transactional cycles are aborted whereas with 64 threads, this value grows to 11 percent points. This indicates that transactions that could be committed transactionally with up to eight threads no longer succeed and are aborted during their execution. However, we assume that they are not aborted in their very beginning due to the massive performance decrease. After an abort, the transaction is executed non-transactionally and following transactions might be executed non-transactionally as well due to the Lemming effect. These two effects explain the decrease of transactional cycles in this version. Furthermore, we do not think that the TAS-T implementation for the lock-mutex contributed to the performance-decrease since it did no harm to the transaction latches. Hence, the reasons are solely in the MySQL/InnoDB structures which are apparently not immediately

advantageous for Hardware Transactional Memory.
To find the reasons for the aborts, we collect event statistics using perf again.

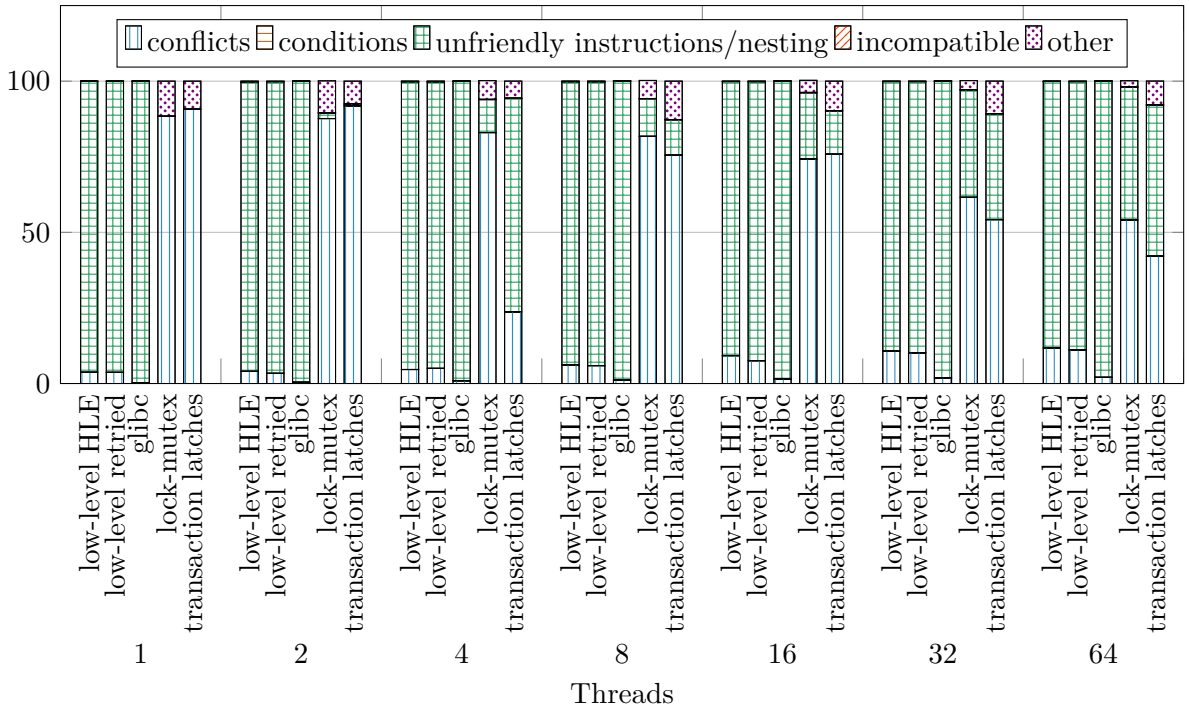


Figure 5.6: Composition of abort events

The labels in Figure 5.6 are short-hand for the abort codes `MISC1-5` from Section 2.3.3, using either a short summarizing description.
Firstly, the events analysis tells us that no aborts happened due to uncommon conditions or incompatible memory types (`MISC2` and `4`).
Then, a lot of aborts with the low-level versions (including glibc) occur due to unfriendly instructions or a transaction nesting limit overflow. Although, in terms of the low-level HLE version, we need to keep in mind that MySQL's mutex spin implementation waits in between spins if the mutex is occupied, similar to our implementation with a system pause call. Hence, the initial assumption is that many of these aborts actually have their cause in a mutex which has been occupied earlier by another thread and fail the transaction early due to a non-zero value of the lock word. However, the versions low-level retried and glibc show a similar distribution of abort causes and they provide a different RTM implementation that does not use system calls to abort transactions that are preprogrammed to abort. Analyzing the assemblies of the three different versions, we find that with 64 threads, over two thirds of the aborts occur in the function `buf_page_get_gen` which is used to get access to a database page. This function uses multiple mutexes to protect the memory blocks and it also changes the value of the mutex lock word to indicate the amount of readers which obviously leads to aborts as well because the read-set is invalidated. Many of these mutexes are nested inside each other and moreover, pages are read from

file when they are not cached. This involves system calls that abort any speculative execution.
In addition, the functions `lock_rec_convert_impl_to_expl` and `lock_rec_create` account for approximately 6 percent of the aborts each. Following its naming, `lock_rec_convert_impl_to_expl` converts an implicit record lock to an explicit one and since it uses the global lock-mutex, a high contention is natural. Similarly, the function to create a new record lock, `lock_rec_create`, uses the global lock-mutex as well which increases the abort rate even more. Another function that accounts for 6% of the aborts is the actual `mutex_spin_wait` implementation. One might wonder why this function does not have a bigger share in the total aborts despite the many conflicts. The reason is that the compiler aggressively inlines functions in our assembly to reduce overhead due to function calls and the resulting assembly code will often not call the spin-function because it is inlined instead. In return, this leads to only the top-level functions that have not been inlined showing up in the perf assembly report. Since not all functions are inlined however, we see a small appearance of the spin-wait function in the report as well.
Despite the visible occurrence of the `MISC5` event for aborts that are not due to any of the other abort cases and thus were mostly caused by interrupts in previous test cases, this category of events is not visible in the low-level versions. From the raw data, we can however tell you that interrupts occur in the low-level versions as well but since these versions have way more transactional cycles, other abort events are way more common and `MISC5` becomes vanishingly small.
It is further interesting to see how the share of actual data conflicts rises the more concurrent clients access the database. In the low-level HLE version for instance, the share of `MISC1` events grows from 4% between one and eight threads up to 12% with 64 threads.

In terms of the lock mutex, it seems to start off well with a single thread where aborts are only due to conflicts and not to unfriendly instructions. Since there is only one thread, it is safe to assume that the conflicts are not data conflicts but rather capacity conflicts. Taking into account that no other thread will ever occupy the mutex, it is also obvious that no aborts occur due to an early abort with a system call (`MISC3`/unfriendly instructions) when the mutex is occupied by a different thread. With more concurrent clients, the lock-mutex modification performs worse and worse which goes along with an increased abort rate due to unfriendly instructions. Considering that the modifications in this version target only a single global mutex, an abort in one thread will write the mutex, thereby abort all other threads speculating on the same global mutex and cause a Lemming Effect. With more and more threads, the likelihood of this happening increases as shown earlier in Paragraph 2.2, thus less transactional cycles are started and the code is executed increasingly pessimistically as shown in Figure 5.5. Where the aborts mostly occur in the record lock functions with a single thread, namely `lock_clust_rec_read_check_and_lock` which checks if locks of other transactions prevent an immediate read of data and potentially puts the transaction in the wait state as well as `lock_rec_lock` and `lock_rec_convert_impl_to_expl`, their hotspot shifts towards functions that put the thread in the wait state such as `ut_delay` with 64 threads. Moreover, the function `lock_clust_rec_read_check_and_lock` also faces

a decreased chance of being able to read a record immediately with more concurrent transactions and more transactions have to wait which in turn leads to more aborts.

With the transaction latches, we observe a similar distribution as with the lock-mutex: only data or capacity conflicts occur with a single thread but this number decreases continuously until half of the aborts have their cause in the `MISC3` event, thus unfriendly instructions or nesting limit overflow. However, the aborts occur almost exclusively in the function `lock_rec_create` which does not put the transaction in the wait state and supported by a perf analysis, this means that the aborts arise when the mutex is occupied and a system call is issued. In contrast to the lock-mutex where the aborts happen late in the execution path when the transaction is put in the wait state, the aborts in the transaction-latches happen early and do not lead to a decreased throughput.

Finally, the observation that there is a trade-off between a high share of transactional cycles and a low abort rate, which has already been made in the analysis of the transactional and aborted cycles, manifests itself in the analysis of the abort causes again. When we limit the modifications to a well-defined scope, for instance by targeting only the transaction latches, we achieve a low abort rate and the aborts effectively occur only due to data conflicts. Unlike with the modification of all low-level mutex writes where many cycles are transactional but also a significant amount of them aborts and it is hard to pin down the abort causes.

CHAPTER 6

Conclusion

6.1 Conclusions on Intel TSX and its application in a Database Concurrency Control

This work elaborated on the usage of Intel's Transactional Synchronization Extensions wherein Hardware Lock Elision has been analyzed in depth and we proposed a general RTM as well as a HLE locking implementation that out-performed other implementations, such as POSIX. The observations in this analysis can contribute to future work, especially by providing critical points that need to be paid attention to. We defined the application scope of Hardware Transactional Memory wherein transaction size (inter alia with regards to data associativity and alignment), duration and nesting is limited which narrows applicable use-cases down to environments that fulfill these conditions. Moreover, we found that HTM imposes a significant overhead for use cases with little data, such as a shared counter, which makes this new technique more applicable in cases where the data structure is more complex and one mutex protects a lot of data, such as a Hashmap. It is also important that concurrent threads can modify data without interrupting each other, which can very well be done in a Hashmap in contrast to a doubly linked list.
After analyzing single use cases and data structures, we showed how these fit into MySQL InnoDB's Concurrency Control and proposed several modifications based on the previous investigations. In particular, we pointed out how isolated latches protecting the actual locks can be targeted and which components within the Concurrency Control seem promising to be provided with HTM. Although an evaluation of these modifications has not shown a performance increase on our four core server, we identified the bottlenecks as a lack of cores, global data and system calls within MySQL that are often hard to exclude. Furthermore, the evaluation has shown that it is fairly easy to add HLE prefixes to the mutex calls or to link the application with an HTM-enabled glibc but this by itself will often not work - one needs to debug the data structures and environment with the goal of maximizing transactional cycles and minimizing aborted cyles. In a future paper, we

will further improve the InnoDB modifications and for instance only provide Hashmap latches with Hardware Transactional Memory as well as carry out the benchmark on a server with 64 Haswell cores.

Overall, we hope that the micro-analyses can serve as a reference for future papers and that an advanced application of HTM to MySQL can improve its scalability, especially with more cores.
In addition, we expect Intel to improve the underlying technique of this optimistic concurrency technique so that it becomes even more useful. For instance, there is no particular reason to keep the mutex in the read-set. Nonetheless, Hardware Transactional Memory has a great potential to make programming easier which would already be a huge contribution without any performance increases to the nowadays complex locking algorithms. HTM could be particularly useful for critical sections in programming languages such as Java where critical sections are often marked as such without specifying further how the thread-safety should be achieved. Therein, an optimistic technique might be superior to a global mutex for the whole critical region. This speculative locking technique is also perfectly suited for complex data structures that often use a global mutex to deal with concurrency. Using e.g. HLE prefixes within such structures could provide a significant performance speedup at a low implementation effort.

The source code of this work is publicly available in a Github repository: https://github.com/Measch/htmsql/tree/46afa3d1dd151831ed6b364de0ed8aa415aa5977.

6.2 Lessons learned

Apart from the implications concerning the content covered in the previous section, I also learned a lot for myself - ranging from implementation details over LaTeX structure to general life strategies.

After some of the first tests produced results that were hard to explain due to multiple data in the same cache line, I tended to pad everything in the end to avoid false conflicts and memory alignment that was sometimes randomly beneficial.
Moreover, I have learnt to think about what I actually want to measure and what information I want to include in the plot that illustrates a benchmark before writing the actual benchmark. The reasoning behind this is that, for example, including standard deviation later on was already a burden by itself but for some benchmarks I had to change the actual benchmarking architecture from fixed operations per thread and measured time to fixed time for the whole benchmark and measured total operations. If one was to define a fixed amount of operations that every thread executes, then starts all threads and measures the start time, waits until all threads are finished and measured the end time, only the time of the slowest thread would be measured. Therefore, it makes sense to define the time that the benchmark should run and see how many operations all threads together achieve - even better, a warmup time guarantees that no time within the measurement is wasted on thread creation.

Related to this is that I now tend to question unexpected results more, as minor as they may be. This often helps to reveal issues in the big picture and the most minor anomaly can bring new insights.
In programming in general, every software engineer should learn at some point that naming variables, functions etc. properly is absolutely necessary for others to understand the code as well as for oneself when looking at the code again at a later point in time. I found that this applies to naming TeX-files and especially benchmark results as well. Therefore, my final system of naming the measured raw data consisted of the benchmark name and all variables that were measured differently. For instance, I would put a run of the shared counter with 1 counter and 4 pinned threads per core in the `shared_counter` directory and name it `1counter-4threads-pin1` which expresses the whole configuration.
Using a Version Control System (VCS) is obviously recommended and it helped me solely by giving me a certain ease at refactoring because even if something goes wrong, it won't destroy all my work. Going back to a previous revision or comparing the changed file contents is another aspect of a VCS that came in handy during this work.
I also improved my skills in a variety of programming languages. The git repository with the source code (excludes LaTeX) contains over 130,000 lines of code and according to git's statistics, these mainly consist of 48% C++, 28% C, 17% Shell, 5% Assembly and some files in Java and Python.

In terms of LaTeX, I learned how useful it is to properly structure the code from the beginning (which I luckily did). I suggest dividing the structure as deep as into sections - more than often people use one directory per chapter which I did not do because I found a flat hierarchy to be more efficient due to less folder navigation and it makes it easier to move sections between chapters. In that sense, it might also be beneficial to use the `include` command which allows partial compilation and to exclude all but one file. However, it always adds page breaks before and after an included file, so I found it to only be applicable for chapters and to use `input` for sections instead. Furthermore, I really got into the `pgfplots` package for the plotting of graphs. It is code-based (thus is well-suited for VCSs) and allows you in-LaTeX compilation of benchmarks, hence removes the extra step of compiling the data to an image with, for example, Excel first. The package also allows to cache compiled plots that will only be changed when the plotting code is modified. Also in terms of `tex`, I found the `tex.stackexchange.com` community particularly useful and there was no issue that could not be solved with the help of the expertise on this site.

I also felt that the thesis changed how I approach situations in life generally. First of all, I find the divide-and-conquer to be a particularly well applicable problem-solving technique. Dividing problems into smaller ones until the problem is trivial to solve is especially well-suited in the IT-world but can be applied to any area. The whole thesis actually follows this technique: we got an understanding of the basic HTM-technique first, then applied it to isolated use cases and finally integrated everything in the MySQL modifications. Additionally, I now invest more time in understanding the tools and techniques I am using first instead of jumping right into implementation

details. Reading related work and documentation as a start definitely supports acquiring an idea of the used technique.
In addition, I also learned to question statements and ask for credible sources and in the same sense, I reference the sources of my own statements.

APPENDIX A

Environment

A.1 Server Hardware

Our server has a total of 4 cores.

Listing A.1: Core specification

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 60
model name      : Intel(R) Core(TM) i5-4670T CPU @ 2.30GHz
stepping        : 3
microcode       : 0x16
cpu MHz         : 800.000
cache size      : 6144 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
```

```
                    mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2
                    ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc
                    arch_perfmon pebs bts rep_good nopl xtopology
                    nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64
                    monitor ds_cpl vmx smx est tm2 ssse3 fma cx16 xtpr
                    pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt
                    tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm
                    abm ida arat epb xsaveopt pln pts dtherm tpr_shadow
                    vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1
                    hle avx2 smep bmi2 erms invpcid rtm
bogomips        : 4589.52
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

The cache sizes can be found in the processor specifications[1] and by querying `/sys/devices/system/cpu/cpu0/cache/index0/size` for the data cache size, `/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size` for the size of a cache line and `/sys/devices/system/cpu/cpu0/cache/index0/ways_of_associativity` for the set associativity.

Listing A.2: Cache sizes

```
L1      4 x 32 KB Instruction Caches
        4 x 32 KB Data Caches
L2      4 x 256 KB
L3      6 MB shared cache

Level1 DCache size: 32 K
LEVEL1_DCACHE_LINESIZE: 64
Ways of associativity: 8
```

[1] http://ark.intel.com/products/75050/Intel-Core-i5-4670T-Processor-6M-Cache-up-to-3_30-GHz and http://www.cpu-world.com/CPUs/Core_i5/Intel-Core%20i5-4670T.html#specs

A.2 Operating system

Listing A.3: Linux and gcc version

```
$ cat /proc/version
Linux version 3.11.0-custom+
(gcc version 4.8.1 (Ubuntu/Linaro 4.8.1-10ubuntu8) )
```

Furthermore, Hyper-Threading is not used on our machine.

APPENDIX B

Technical pitfalls

B.1 HTM-enabled glibc

One needs to pay attention to the initialization of the mutex: tests have shown that initializing a mutex with a mutex attribute that itself is not initialized will have zero transactional cycles as a consequence. The same holds true for mutex attributes where the type is set to any of the four available ones [95], [96], such as `PTHREAD_MUTEX_ADAPTIVE_NP` (used in MySQL), thus we recommend to use the `PTHREAD_MUTEX_INITIALIZER`.
Releasing mutexes linked with this glibc with `pthread_mutex_destroy` results in the error code 16 (EBUSY) regardless of how the mutex has been initialized. This might result in disruptions when surrounding software, such as MySQL, checks for an erroneous return value (as it should do obviously).

B.2 Linking MySQL with custom glibc

Originally, we were able to modify the linker flags with the cmake options shown in Listing B.1 [97].

Listing B.1: Outdated linking of MySQL with glibc

```
1 GLIBC_FLAGS="-Wl,--rpath=/path/to/glibc-htm/build/lib:/usr/lib/x86_64-linux-gnu:/
      lib/x86_64-linux-gnu \
2       -Wl,--dynamic-linker=/path/to/glibc-htm/build/lib/ld-linux-x86-64.so.2"
3
4 cmake \
5   -DCMAKE_INSTALL_PREFIX=../install .. \
6   -DCMAKE_C_FLAGS="$GLIBC_FLAGS" \
7   -DCMAKE_CXX_FLAGS="$GLIBC_FLAGS"
```

After a system-update however, this did not work anymore since cmake now ignored the `MAKE_C_FLAGS` and `MAKE_CXX_FLAGS` options if the paths were colon-separated.

APPENDIX C

Omitted Benchmarks and figures

C.1 Memory alloation

Listing C.1: Memory (de-) allocation inside RTM

```
#include <stdlib.h>
#include <stdio.h>

#include <immintrin.h> // rtm
#include "../Stats.h"

/**
 * @return 0 if success, 1 otherwise
 */
int stack(int size) {
  if (_xbegin() == _XBEGIN_STARTED) {
    unsigned char buf[size];
    _xend();
    return 0;
  } else {
    return 1;
  }
}

/**
 * @return 0 if success, 1 otherwise
 */
int freeStore(int size) {
  if (_xbegin() == _XBEGIN_STARTED) {
    unsigned char *buf = new unsigned char[size];
    delete[] buf;
    _xend();
```

```
28      return 0;
29    } else {
30      return 1;
31    }
32  }
33
34  /**
35   * @return 0 if success, 1 otherwise
36   */
37  int heap(int size) {
38    if (_xbegin() == _XBEGIN_STARTED) {
39      unsigned char *buf = (unsigned char*) malloc(
40          sizeof(unsigned char) * size);
41      free(buf);
42      _xend();
43      return 0;
44    } else {
45      return 1;
46    }
47  }
```

The code in Listing C.1 leads to 0 failures for stack-allocation and 100% failures for heap-allocation (`freeStore` and `heap`).

C.2 Transaction size with randomized access

C.2.1 Access range

We vary the array range that is accessed. The size of the integer-array is fixed.

```
1  if (_xbegin() == _XBEGIN_STARTED) {
2    for (int i = 0; i < accesses; i++) {
3      a[rand() % modulo]++;
4    }
5    _xend();
6  }
```

Figure C.1 shows once more how too much data leads to aborts. In particular, an increase in the array size can lead to pre-fetching data[98] and aborts occur even if they are not necessary.

C.2.2 Comparison of sequential and random array access

Instead of accessing the array sequentially from front to end, we write and read data randomly within the whole array. We further distinguish between initialized and uninitialized arrays as defined

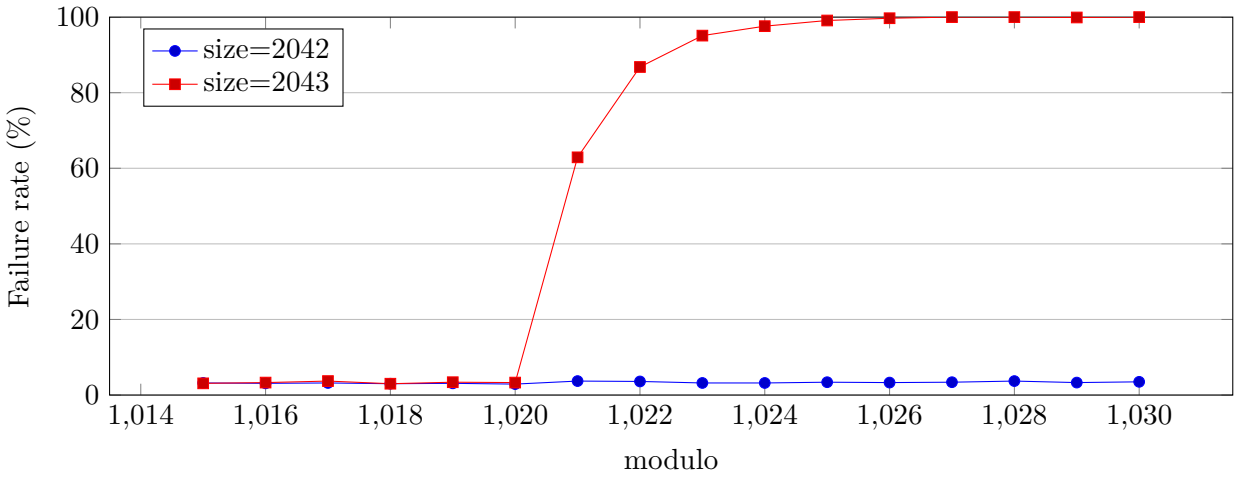


Figure C.1: Aborts with different array sizes and the same access range

in Section 3.2.1.

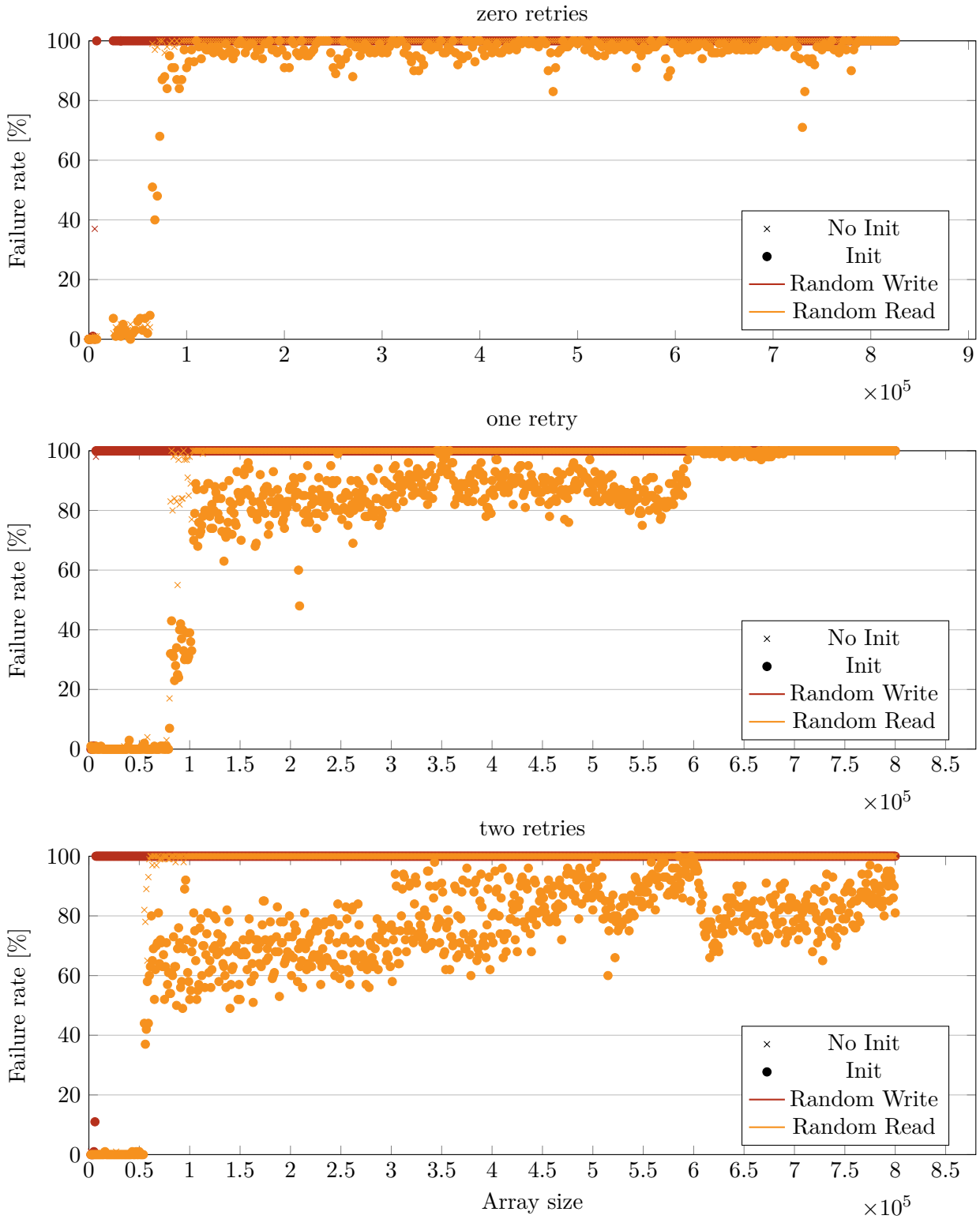


Figure C.2: Failure rate for modified array access

C.3 Locking Overhead with data

The benchmark uses an array of *unsignedchar*s (1 B) with a total size of 1 KB and reads all of its elements, counting the elapsed time in nanoseconds. The whole array access is protected by either a POSIX or atomic/HLE mutex or RTM.

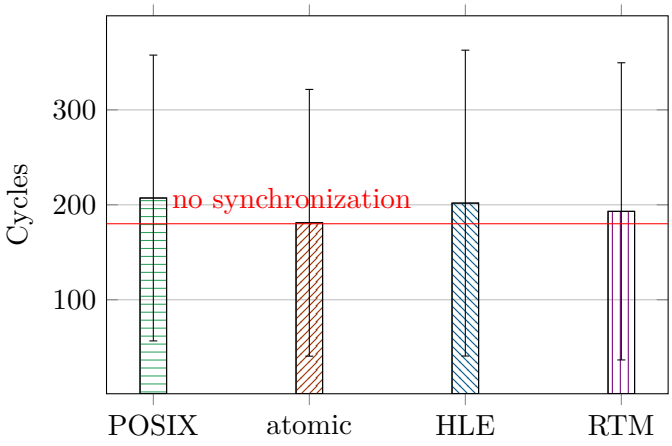


Figure C.3: Overheads with locks protecting an array (lower is better)

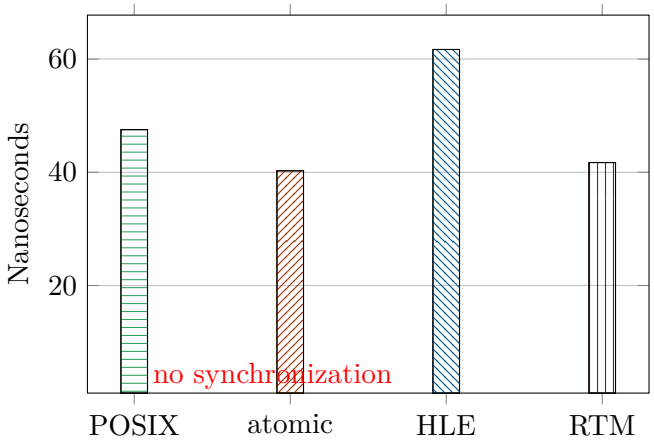


Figure C.4: Overheads with lock-only (lower is better)

C.4 Partitioning the data access range between threads

All lock variables are stored in an array and the `lock_accesses` array decides where to access them. Every thread gets its own distinct region of locks that it can access (Figure C.6).

Figure C.5: Non-partitioned (hotspot) lock access

Figure C.6: Partitioned lock access

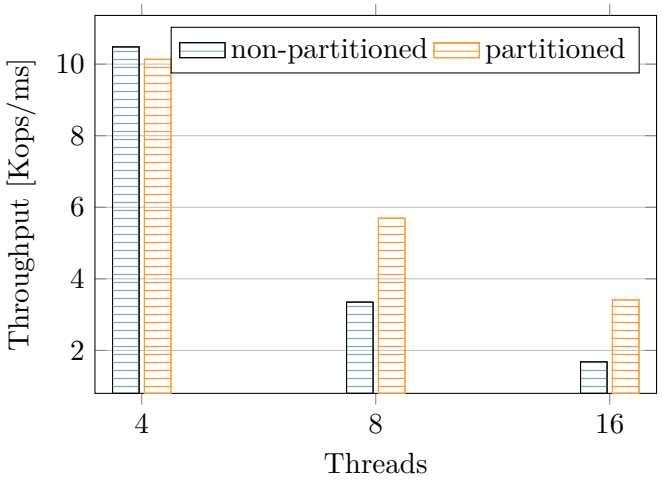


Figure C.7: HLE locking of 100 (non-/) partitioned lock variables with immediate unlock

C.5 Notes on functions defined in the hle-emulation header

Listing C.2: Operation or fetch first

```
__hle_{acquire,release}_add_fetch{1,2,4,8} // first adds the value, returns the
    lock value with the added value
__hle_{acquire,release}_fetch_add{1,2,4,8} // first fetches the lock value then
    adds the value to it, returns the result without the added value

/* Example */
static unsigned lock;
unsigned res;
res = __hle_acquire_add_fetch4(&lock, 1); // OP_FETCH (operation first, fetch
    afterwards)
// res == 1, lock == 1

res = __hle_acquire_fetch_add4(&lock, 1); // FETCH_OP
// res == 1, lock == 2
```

Equivalence of operations Some operations are shorthand forms of others:

Listing C.3: Symbiosis between TAS and EXCH

```
__hle_acquire_test_and_set4(&lock); == __hle_acquire_exchange_n4(&lock, 1);
```

TEST_AND_SET After calling `__hle_acquire_test_and_set4(&lock)`, lock will always be `true (1)`. The return value of the function is the previous value of `lock`.

Listing C.4: TAS evaluation

```
static unsigned lock;
unsigned res;
res = __hle_acquire_test_and_set4(&lock);
// res = 0, lock = 1
res = __hle_acquire_test_and_set4(&lock);
// res = 1, lock = 1
__hle_release_clear4(&lock);
//      lock = 0
res = __hle_acquire_test_and_set4(&lock);
// res = 0, lock = 1
```

We observe that test_and_set behaves the same as calling exchange with a value of one and checking the result for equality with one. In fact, `__hle_acquire_test_and_set4(&lock)` internally does exactly that as defined in `hle-emulation.h`:

Listing C.5: Implementation of ___hle_acquire_test_and_set4

```
1 static __hle_force_inline int         \
2 __hle_##prefix##_test_and_set##size(type *ptr)      \
3 {               \
4   return __hle_##prefix##_exchange_n##size(ptr, 1) == 1;  \
5 }
```

As can be seen in this code-part, the whole `hle-emulation.h` file is a header file filled with macros that go down to assembler calls. In this extract for example, prefix will be replaced with either `acquire` or `require` and size with one of `{2, 4, 8}`. The `ptr` variable is our lock variable and `__hle_force_inline` is one of `{__HLE_ACQUIRE, __HLE_RELEASE}`.

lock	oldv	newv	lock	oldv	newv	res
0	0	0	0	0	0	1
0	0	1	1	0	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	0
1	0	0	1	1	0	0
1	0	1	1	1	1	0
1	1	0	0	1	0	1
1	1	1	1	1	1	1

Table C.1: Table of value combinations of HLE `ACQUIRE_COMPARE_EXCHANGE`

ACQUIRE_COMPARE_EXCHANGE Using Table C.1, we can conclude the way the values are calculated:

Listing C.6: Pseudo compare_exchange implementation

```
1 res = lock == oldv;
2 if(res) lock = newv;
3 else oldv = lock;
```

C.6 Different random generators within the Closed Bank

rand() System built-in `rand` function

Partitioned This case should play into the hands of HLE: no collisions happening at all. This is achieved through partitioning the account-pool to n partitions where n is the number of threads. We then assign one partition to each thread and let it modulo-iterate over its accounts.

Custom rand() Assuming that the system built-in `rand()`-function is responsible for discrepant results, we implement our own function [1].

Saved rand Generate and save random values before running the benchmark.

[1] on the basis of http://www.daniweb.com/software-development/c/code/216329/construct-your-own-random-number-generator

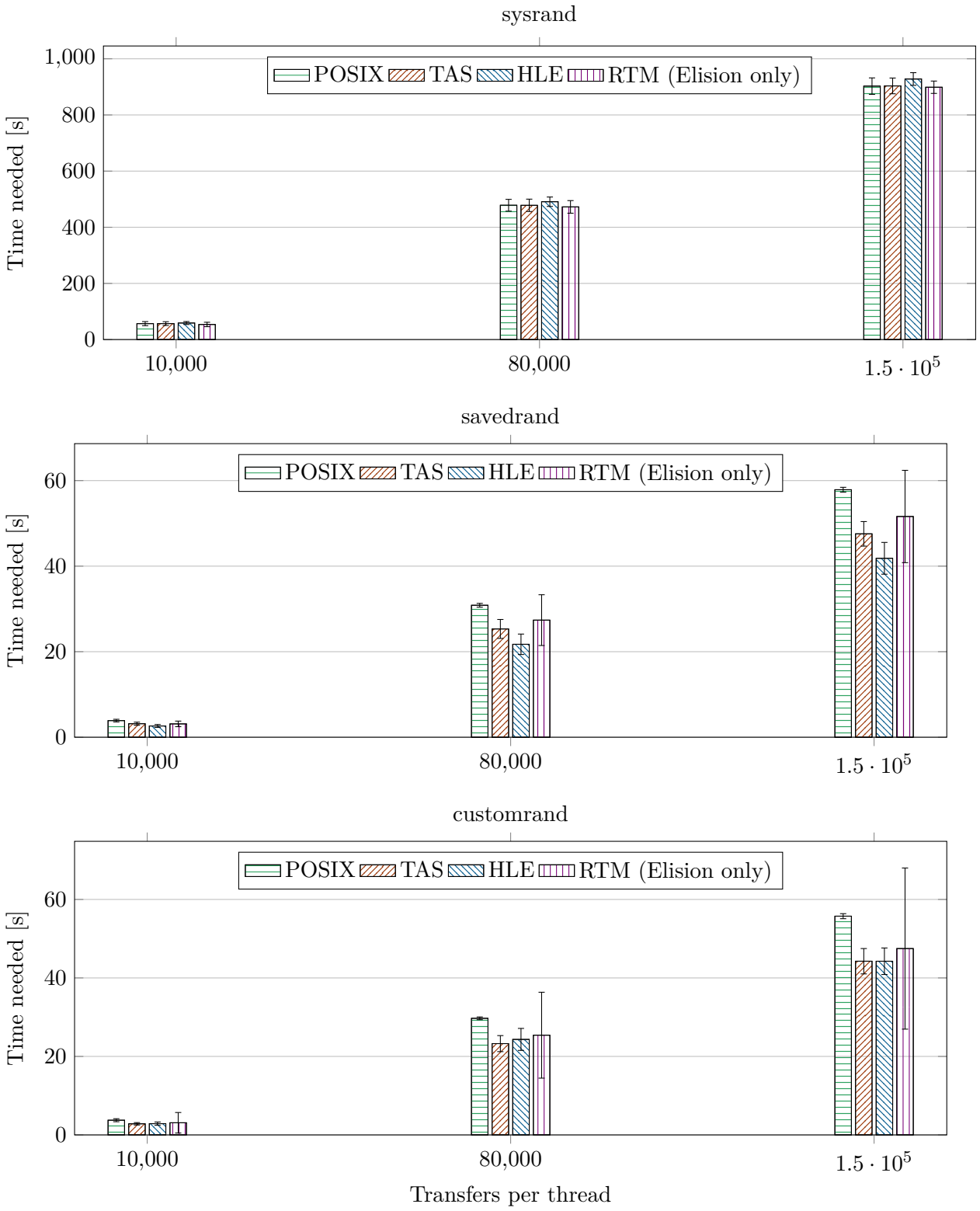


Figure C.8: Mutex per account in closed bank with 100 accounts and 4 threads

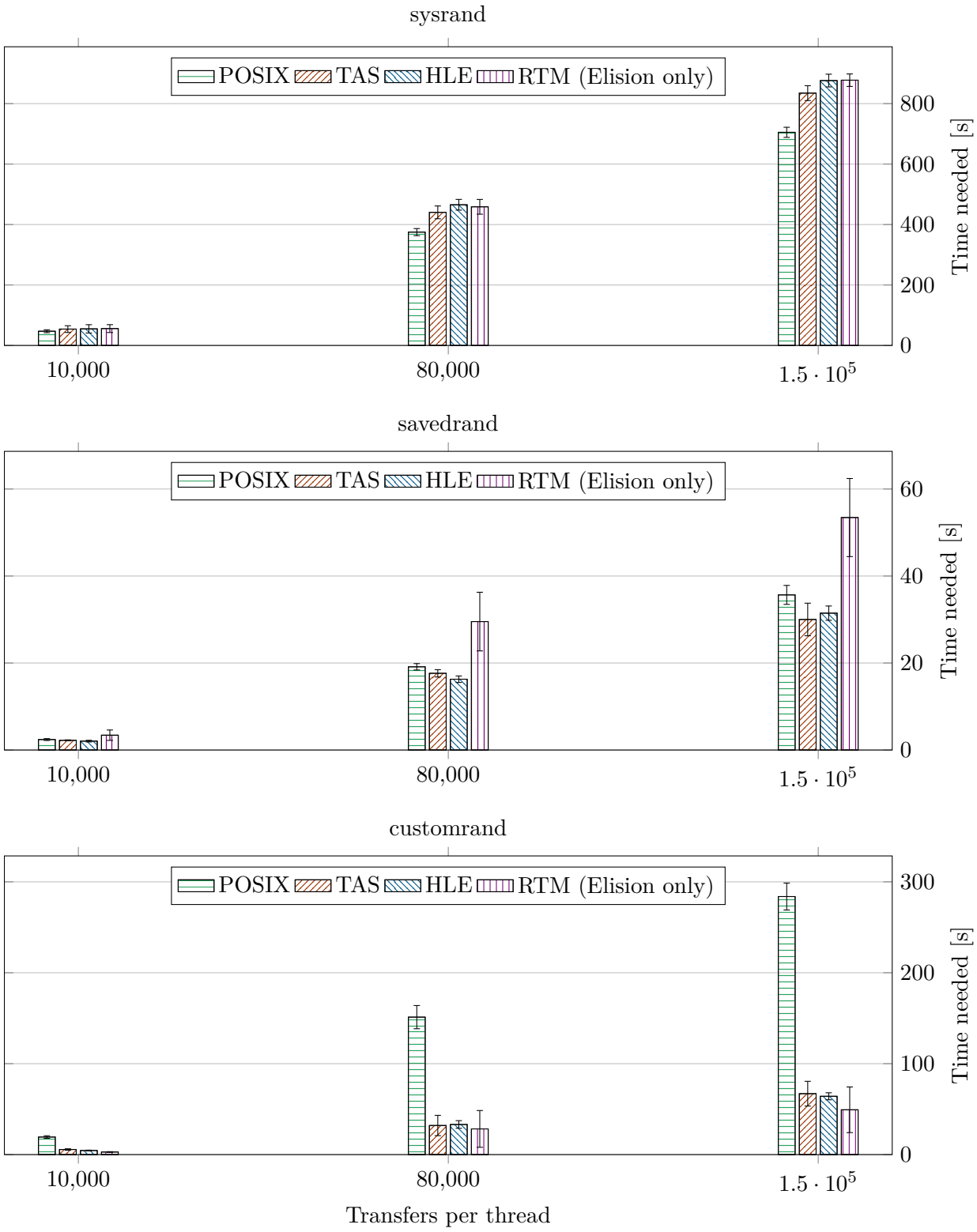


Figure C.9: Globally locked Bank with 100 accounts and 4 threads

C.7 Hashmap

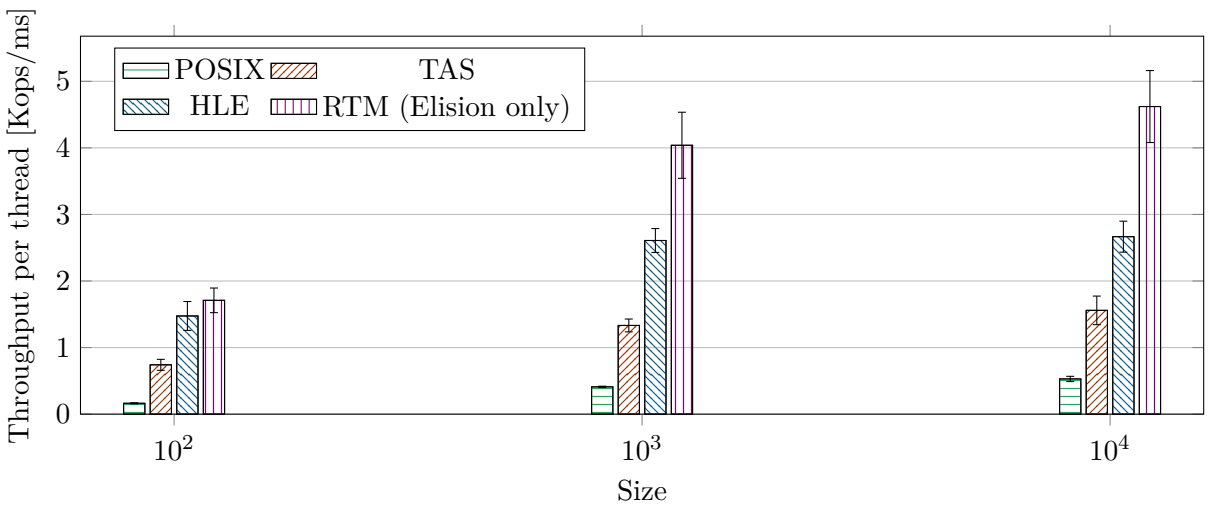
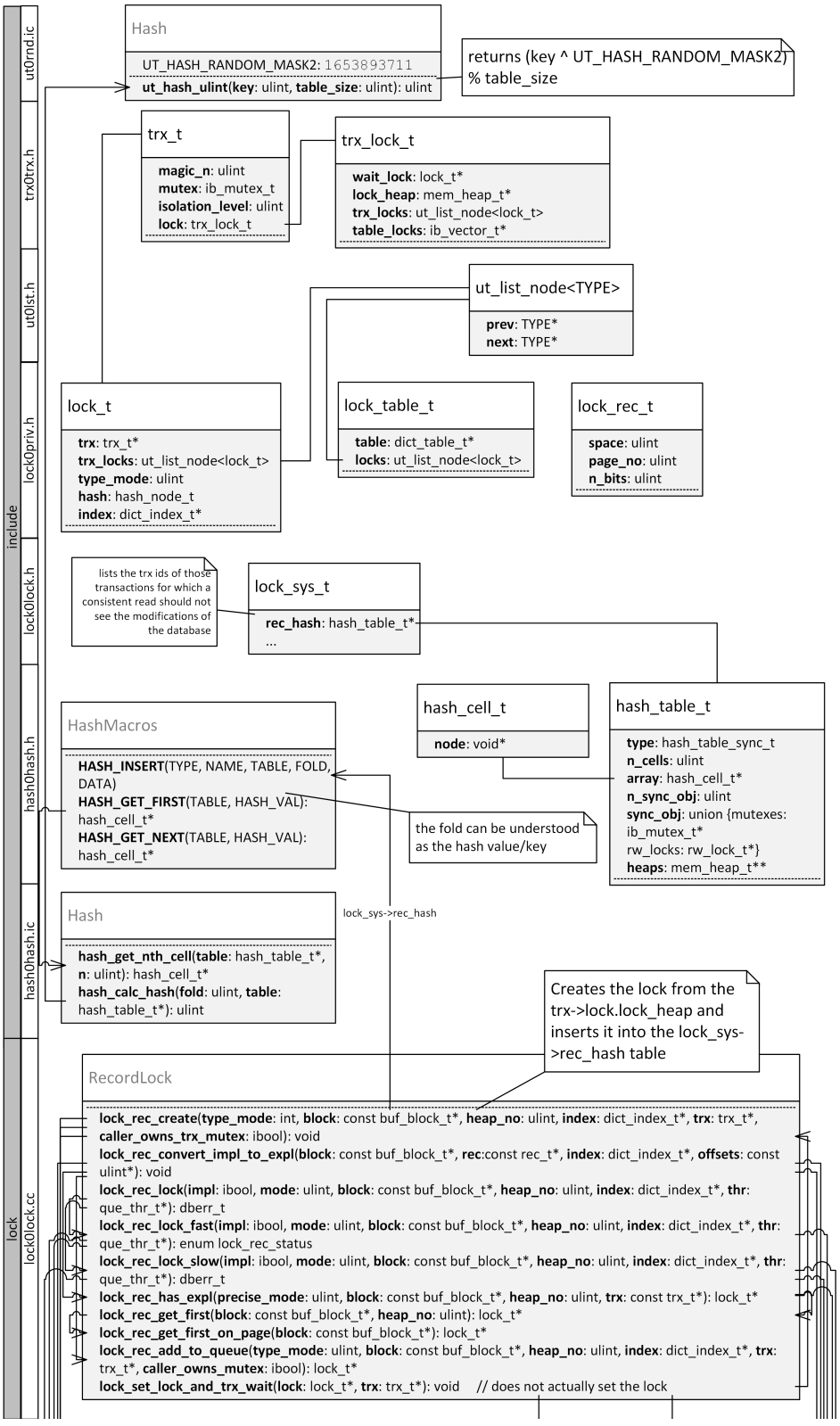


Figure C.10: Throughputs on a globally locked HashMap with various sizes ($pr_i = pr_r = 25\%, pr_c = 50\%, 10000$base inserts)

C.8 Logically-grouped call-graph of InnoDB entities



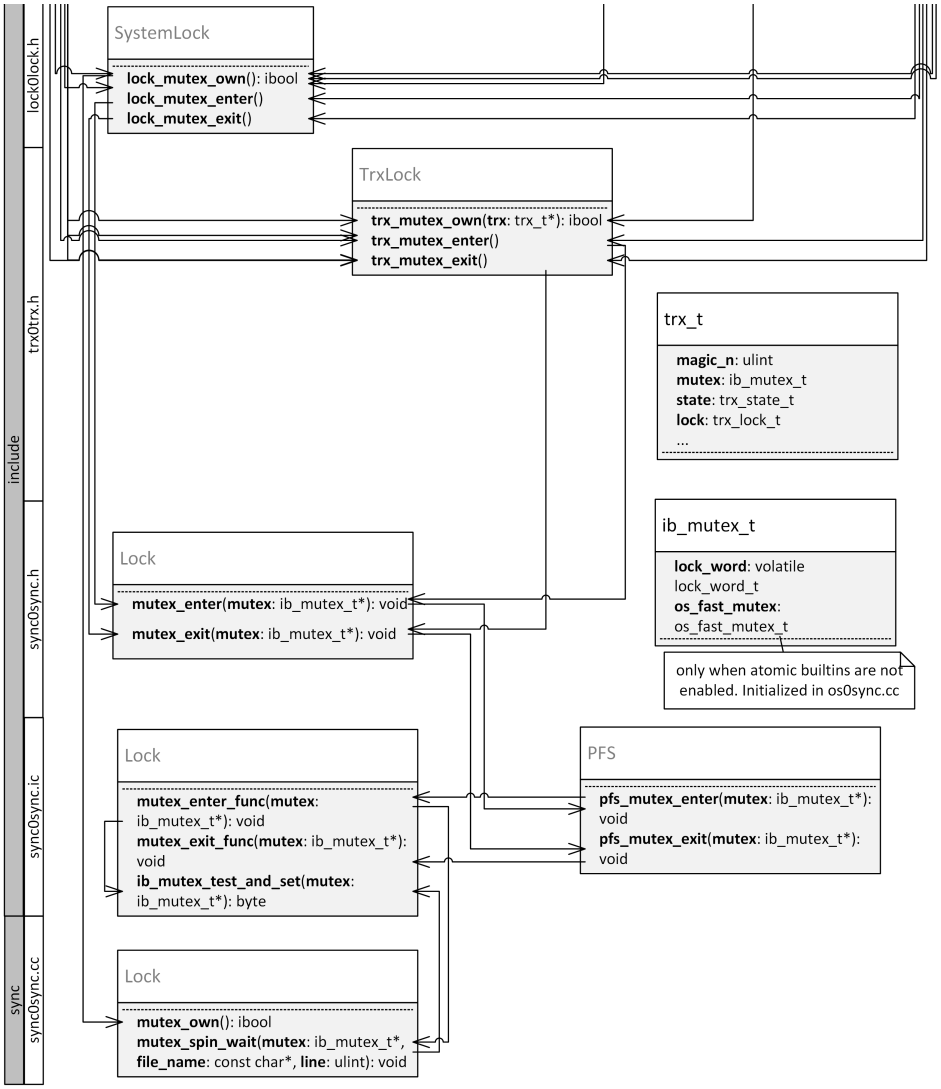


Figure C.11: Logically grouped and labeled call-graph of Concurrency and Hashmap functions in MySQL InnoDB

APPENDIX D

Open Questions

D.1 Abort costs

We already argued that cycles have a greater impact on the performance than transactions. When taking a closer look at the functionality of HTM and especially what happens after a rollback, Intel states that a transactional abort leads to the discard of all updates in the transactional region and the restoration of the architectural state to appear as if the optimistic execution never occured [31, p. 12-1]. We assume that the mentioned discard of all updates means that all cache lines that have been used (either read or written) are invalidated and will be reloaded from the next cache level in the next execution.

The reloading should then be observable in the execution time measured in cycles where the second execution after an abort on a practically cold cache takes longer than the first execution on a warm cache.

The cycles can be counted using the `rdtsc` instruction described in Listing D.1 at the beginning and the end of the program and subtracting the two values [99], [100]:

Listing D.1: counting cycles with rdtsc (AMD64)

```
1 __inline__ uint64_t rdtsc() {
2   uint64_t a, d;
3   __asm__ volatile ("rdtsc" : "=a" (a), "=d" (d));
4   return (d<<32) | a;
5 }
```

This difference should further be equal to the difference of writing to an array that has been loaded into the cache before and writing to an array with a practically cold cache which can be seen in Figure D.1. The setup for this graph is to clear the cache first by creating and writing an array

that exceeds the cache size. Then, another array is created and written but in case of the warm cache it is loaded in the cache beforehand by accessing all its entries.

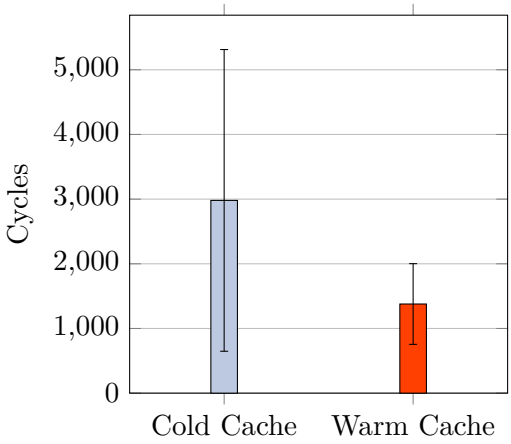


Figure D.1: Cycles required to access an array of 1 KB

The difference between a cold and warm cache amounts to approximately 100 cycles in this test.

To verify our assumptions on the effect on aborts, we create an array of size `n`, write once to all its `n` elements to make sure all the data is in L1 cache and then write in two different ways

1. write to all `n` elements of the array
2. write to only one element of the array (`n` times to have the same amount of write-cycles as the first method)

The both described functions are then executed inside a HTM region marked by RTM. After each function has accessed all or a single element of the array, the transaction is aborted and we compare the cycle count per execution.

Figures D.2 and D.3 outline how after an abort, the function writing to all array elements of the array has to reload everything in the cache whereas the function that writes to a single element only has to reload a single cache line of 64 Byte.

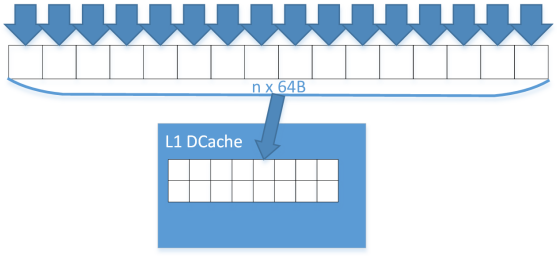


Figure D.2: Access on all array elements

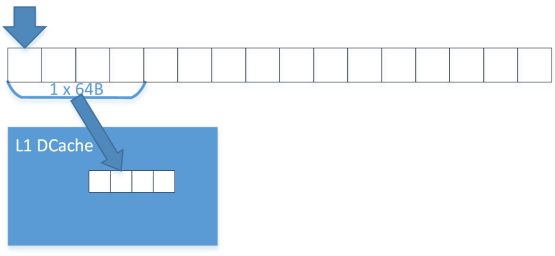


Figure D.3: Access on a single array element

However, we were unable to produce this effect as the cycles for accessing the whole array and only

a single element of it required the same amount of cycles as shown in Figure D.4. One explanation could be pre-fetching although we accessed the array in a pre-fetching unfriendly manner by keeping an offset of over four cache lines between two consecutive array accessed.

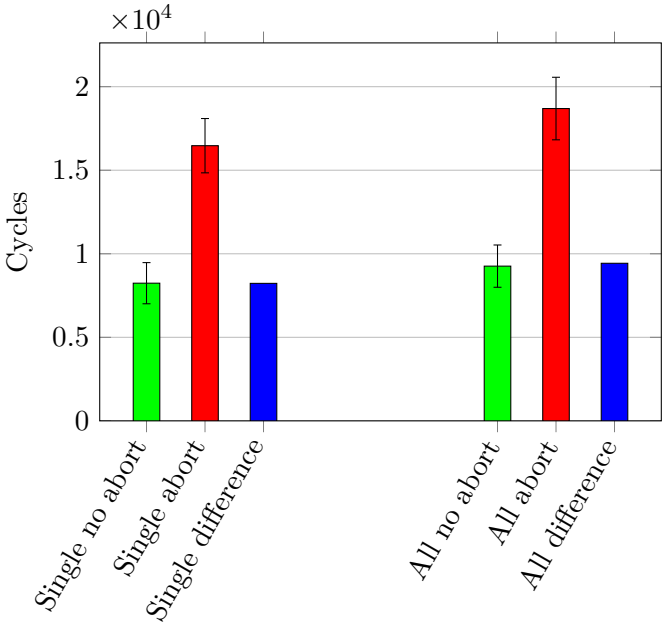


Figure D.4: Cycles of different array accesses (array size 1 KB / 16 cache lines)

D.2 List with four threads

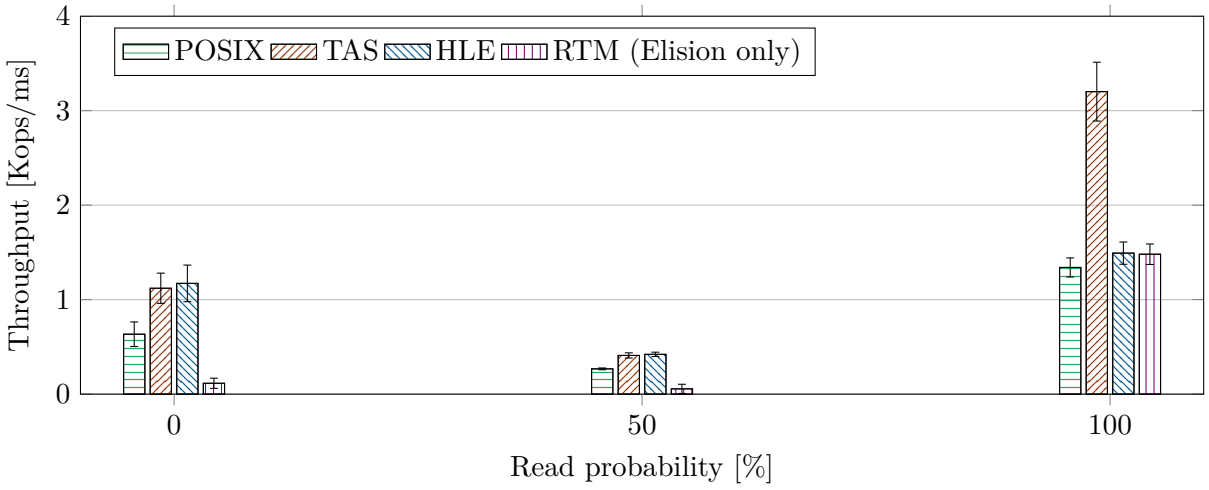


Figure D.5: Throughputs subject to read/update probabilities on an unaligned List (4 Threads)

Figure D.5 shows a TAS lock which behaves as expected: the more reads instead of writes, the better. Because we work on 4 threads, a write of one threads leads to the invalidation of cache lines

Transactional cycles	13.31%
Aborted cycles	64.61%

Table D.1: Unaligned List HLE transactional indicators

of other threads, thus requiring more cycles to reload the cache lines from a higher level cache. Read instead does not invalidate cache lines which is a performance gain as shown in this figure.

As shown in Table D.1, we experience lots of aborts. `perf report` tells us that these mostly occur in the find function where an item's value is compared to the search-value. Because the find function traverses the whole list, it is quite likely that another thread will issue a data conflict by removing a list item.

Memory management can play a role as Section 3.2.2 has shown. Therefore, we use the `aligned_alloc` function to align all our list items to multiples of 64 B (cache line size). The results of this modified test are shown in Figure D.6 where e.g. for update-only (0% read), RTM's performance is three times better than in the unaligned case. However, the improvements are not nearly as significant otherwise.

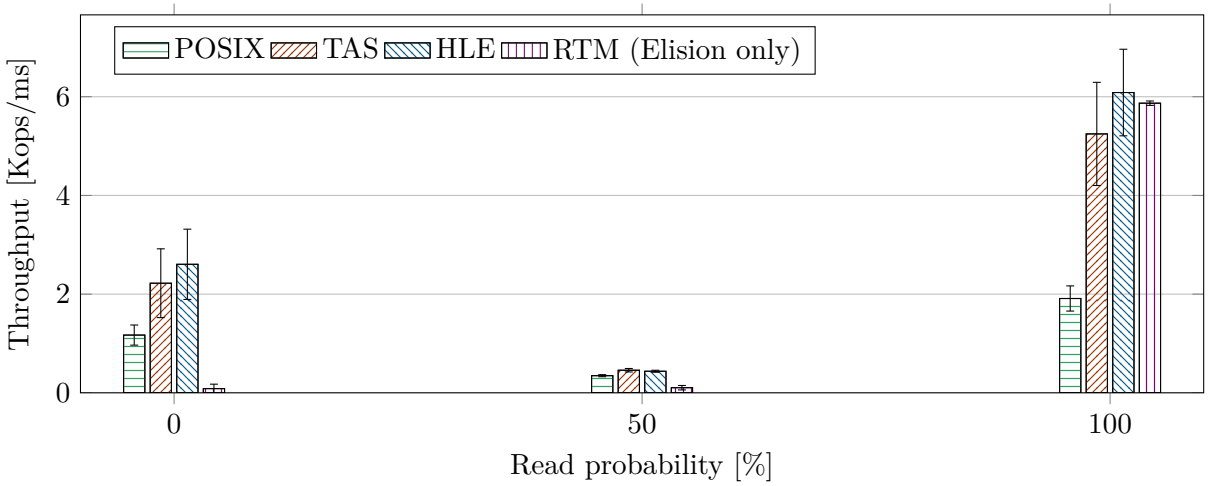


Figure D.6: Throughputs on an unaligned list (2 Threads)

Having fun with MySQL

Although this section is supposed to be sarcastic and maybe even a bit funny, it also illustrates that working with the often old (2000's) code inside MySQL can be a pain. Therefore, I have full understanding for approaches that just start over and write a whole new database instead of continuously improving a legacy system.

Listing D.2: Macro hash insert with useful variable names and clear function behavior

```
1  // in innobase/include/hash0hash.h:122
2  #define HASH_INSERT(TYPE, NAME, TABLE, FOLD, DATA)\
3  do {\
4    hash_cell_t*  cell3333;\
5    TYPE*   struct3333;\
6  \
7    HASH_ASSERT_OWN(TABLE, FOLD)\
8  \
9    (DATA)->NAME = NULL;\
10 \
11   cell3333 = hash_get_nth_cell(TABLE, hash_calc_hash(FOLD, TABLE));\
12 \
13   if (cell3333->node == NULL) {\
14     cell3333->node = DATA;\
15   } else {\
16     struct3333 = (TYPE*) cell3333->node;\
17 \
18     while (struct3333->NAME != NULL) {\
19 \
20       struct3333 = (TYPE*) struct3333->NAME;\
21     }\
22 \
23     struct3333->NAME = DATA;\
24   }\
25 } while (0)
```

Despite many parts of the code being well-documented, some crucial functions lack an explanation such as the hash function.

Listing D.3: Hash function `ut_hash_ulint`

```
1 key = key ^ UT_HASH_RANDOM_MASK2; // UT_HASH_RANDOM_MASK2 = 1653893711
2 return(key % table_size);
```

Contents of mysql/storage/innobase/include:

1. `ut0list.h`
2. `ut0lst.h`

Because who needs proper naming anyway.

Apparently, MySQL isn't even finished completely.

Listing D.4: Fix me!

```
1 #define SYNC_FTS_OPTIMIZE       164     // FIXME: is this correct number, test
```

You guessed correctly, this is not the only `FIXME` in the code.

To be fair, MySQL has been written when compiler support and optimizations were far from today's standard (most header files show a creation date in the late two thousands). However, the code often made me laugh and not too seldom cry.

Bibliography

[1] M. J. Flynn and K. W. Rudd, "Parallel architectures", *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 67–70, 1996.

[2] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[3] D. Geer, "Chip makers turn to multicore processors", *Computer*, vol. 38, no. 5, pp. 11–13, May 2005, ISSN: 0018-9162. DOI: 10.1109/MC.2005.160.

[4] H. Sutter, "The free lunch is over: a fundamental turn toward concurrency in software", *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005.

[5] J. D. Ruiz, *Overcoming the embedded cpu performance wall*, Jan. 2013. [Online]. Available: http://www.embedded.com/design/mcus-processors-and-socs/4405280/Overcoming-the-embedded-CPU-performance-wall- (visited on 05/28/2014).

[6] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, A. J. Rebón, and P. W. Trinder, "Comparing parallel functional languages: programming and performance", *Higher Order Symbol. Comput.*, vol. 16, no. 3, pp. 203–251, Sep. 2003, ISSN: 1388-3690. DOI: 10.1023/A:1025641323400. [Online]. Available: http://dx.doi.org/10.1023/A:1025641323400.

[7] E. W. Dijkstra, "Solution of a problem in concurrent programming control", *Commun. ACM*, vol. 8, no. 9, pp. 569–, Sep. 1965, ISSN: 0001-0782. DOI: 10.1145/365559.365617. [Online]. Available: http://doi.acm.org/10.1145/365559.365617.

[8] IBM, *Synchronization techniques among threads*, 2005. [Online]. Available: http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=%2Frzahw%2Frzahwsynco.htm (visited on 05/28/2014).

[9] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs", *SIGPLAN Not.*, vol. 37, no. 10, pp. 5–17, Oct. 2002, ISSN: 0362-1340. DOI: 10.1145/605432.605399. [Online]. Available: http://doi.acm.org/10.1145/605432.605399.

[10] Oracle, *Concurrency. synchronized methods.* [Online]. Available: http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html (visited on 05/28/2014).

[11] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures", *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, May 1993, ISSN: 0163-5964. DOI: 10.1145/173682.165164. [Online]. Available: http://doi.acm.org/10.1145/173682.165164.

[12] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency", *SIGARCH Comput. Archit. News*, vol. 32, no. 2, pp. 102–, Mar. 2004, ISSN:

0163-5964. DOI: 10.1145/1028176.1006711. [Online]. Available: http://doi.acm.org/10.1145/1028176.1006711.

[13] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III, "Software transactional memory for dynamic-sized data structures", in *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, ser. PODC '03, Boston, Massachusetts: ACM, 2003, pp. 92–101, ISBN: 1-58113-708-7. DOI: 10.1145/872035.872048. [Online]. Available: http://doi.acm.org/10.1145/872035.872048.

[14] N. Shavit and D. Touitou, "Software transactional memory", English, *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997, ISSN: 0178-2770. DOI: 10.1007/s004460050028. [Online]. Available: http://dx.doi.org/10.1007/s004460050028.

[15] V. Gramoli, R. Guerraoui, and V. Trigonakis, "TM^{2}C: a software transactional memory for many-cores", in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12, Bern, Switzerland: ACM, 2012, pp. 351–364, ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168872. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168872.

[16] S. Dabdoub and S. Tu, "Supporting intel transactional synchronization extensions in qemu", [Online]. Available: http://people.csail.mit.edu/stephentu/papers/tsx.pdf (visited on 05/24/2014).

[17] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing", in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2013, p. 19.

[18] *Intel 64 and ia-32 architectures software developer's manual - volume 3 (3a, 3b & 3c): system programming guide*, 325384-050US, Intel Corporation, Feb. 2007. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf.

[19] V. Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases", in *Proc. of ICDE*, 2014.

[20] A. Matveev and N. Shavit, *Reduced hardware norec: an opaque obstruction-free and privatizing hytm*.

[21] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii", in *Distributed Computing*, Springer, 2006, pp. 194–208.

[22] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: log-based transactional memory.", in *HPCA*, vol. 6, 2006, pp. 254–265.

[23] ——, "Logtm: log-based transactional memory", in *Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture (HPCA-12)*, Austin, TX, Feb. 2006, pp. 258–269.

[24] A. Levy, "Programming with hardware lock elision", PhD thesis, Tel-Aviv University, 2013. [Online]. Available: http://mcg.cs.tau.ac.il/papers/amir-levy-msc.pdf.

[25] A. Kleen *et al.*, *Glibc*, 2013. [Online]. Available: https://github.com/andikleen/glibc.

[26] J. Reindeirs, *Coarse-grained locks and transactional synchronization explained*, Jul. 2012. [Online]. Available: https://software.intel.com/en-us/blogs/2012/02/07/coarse-grained-locks-and-transactional-synchronization-explained (visited on 05/29/2014).

[27] A. Kleen, "Adding lock elision to linux", in *Proceedings of the Linux Plumbers Conference*, Aug. 2012. [Online]. Available: http://halobates.de/adding-lock-elision-to-linux.pdf.

[28] D. Schwartz-Narbonne, "Hardware transactional memory", *Programming Paradigms for Concurrency*, 2014. [Online]. Available: http://www.cs.nyu.edu/wies/teaching/ppc-14/material/lecture09.pdf (visited on 05/29/2014).

[29] D. Kanter, "Analysis of haswell's transactional memory", *Real World Tech*, Feb. 2012. [Online]. Available: http://www.realworldtech.com/haswell-tm/2/.

[30] *Intel® c++ compiler xe 13.1 user and reference guides*, 323273-131US, Intel Corporation, 2013. [Online]. Available: http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-win/GUID-A462FBC8-37F2-490F-A68B-2FFA8010DEBC.htm.

[31] *Intel® 64 and ia-32 architectures optimization reference manual*, 248966-029, Intel Corporation, Mar. 2014. [Online]. Available: http://www.intel.pl/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf (visited on 04/23/2014).

[32] A. Kemper and A. Eickler, *Datenbanksysteme: Eine Einführung*. Oldenbourg Verlag, 2011.

[33] *Atomic transactions in distributed systems*. [Online]. Available: http://www.scalus.eu/files/2012/04/Atomic_Transactions.pdf (visited on 05/29/2014).

[34] A. Kleen, "Scaling existing lock-based applications with lock elision", *ACM Queue*, vol. 12, no. 1, 20:20–20:27, Jan. 2014, ISSN: 1542-7730. DOI: 10.1145/2576966.2579227. [Online]. Available: http://doi.acm.org/10.1145/2576966.2579227.

[35] ——, "Intel® transactional synchronization extensions (intel® tsx) linux update", in *Proceedings of the Linux Plumbers Conference*, Aug. 2012. [Online]. Available: http://www.halobates.de/tsx-plumbers13.pdf.

[36] M. E. Thomadakis, "The architecture of the nehalem processor and nehalem-ep smp platforms", *Resource*, vol. 3, p. 2, 2011. [Online]. Available: http://elastos.org/redmine/attachments/download/457/nehalem.pdf.

[37] J. D. Gelas. (Sep. 2012). Making sense of the intel haswell transactional synchronization extensions, [Online]. Available: http://www.anandtech.com/show/6290/making-sense-of-intel-haswell-transactional-synchronization-extensions/4 (visited on 04/23/2014).

[38] Intel, *Intel® c++ compiler xe 13.1 user and reference guide*, 323273-131US. [Online]. Available: https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-54A84479-DEC6-4D2F-9895-46D278EDA820.htm (visited on 05/24/2014).

[39] M. D. Wang and M. Burcea, "Intel tsx (transactional synchronization extensions)", [Online]. Available: http://individual.utoronto.ca/mikedaiwang/tm/Intel_TSX_Overview.pdf (visited on 05/24/2014).

[40] R. Dementiev, *Exploring intel® transactional synchronization extensions with intel® software development emulator*, Intel. [Online]. Available: https://software.intel.com/en-us/blogs/2012/11/06/exploring-intel-transactional-synchronization-extensions-with-intel-software (visited on 05/24/2014).

[41] J. Reinders. (Feb. 2012). Transactional synchronization in haswell, Intel Corporation, [Online]. Available: http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell (visited on 10/11/2013).

[42] A. Kleen, *Tsx fallback paths*, Jun. 2013. [Online]. Available: https://software.intel.com/en-us/blogs/2013/06/23/tsx-fallback-paths (visited on 06/05/2014).

[43] M. Brooker, *Hardware lock elision on haswell*, Dec. 2013. [Online]. Available: http://brooker.co.za/blog/2013/12/14/intel-hle.html (visited on 06/02/2014).

[44] "Intel pentium 4 3.06ghz cpu with hyper-threading technology, Killing two birds with a stone...", *X-bit labs*, 2002. [Online]. Available: http://www.xbitlabs.com/articles/cpu/display/pentium4-3066.html (visited on 03/22/2014).

[45] A. Kleen, private communication, Mar. 2014.

[46] A. C. de Melo, "The new linux'perf'tools", in *Slides from Linux Kongress*, 2010. [Online]. Available: http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf (visited on 06/18/2014).

[47] R. A. Vitillo, "Performance tools developments", in *Future computing in particly physics*, Jun. 2011. [Online]. Available: http://indico.cern.ch/event/141309/session/4/contribution/20/material/slides/0.pdf (visited on 05/19/2014).

[48] *Perf: linux profiling with performance counters*, 2013. [Online]. Available: https://perf.wiki.kernel.org/index.php/Tutorial (visited on 02/25/2014).

[49] A. Kleen. (Mar. 2013). Intel(r) transactional synchronization extensions (intel(r) tsx) profiling with linux perf, [Online]. Available: http://software.intel.com/en-us/blogs/2013/05/03/intelr-transactional-synchronization-extensions-intelr-tsx-profiling-with-linux-0 (visited on 03/04/2014).

[50] K. Lai, private communication, Feb. 2014.

[51] A. Kleen. (May 2013). Using hle and rtm with older compilers with tsx-tools, Intel Corporation, [Online]. Available: https://software.intel.com/en-us/blogs/2013/05/20/using-hle-and-rtm-with-older-compilers-with-tsx-tools (visited on 10/11/2013).

[52] *X86 assembly/data transfer*, Wikibooks, Dec. 2013. [Online]. Available: http://en.wikibooks.org/w/index.php?title=X86_Assembly/Data_Transfer&oldid=2596434 (visited on 03/05/2014).

[53] C. L. Coleman, *Using inline assembly with gcc*, 1988.

[54] O. S. (Intel), *The inline asm containing macro for semicolon causes the intel c++ compiler to hang*, Jan. 2012. [Online]. Available: https://software.intel.com/de-de/articles/the-inline-asm-containing-macro-for-semicolon-causes-the-intel-c-compiler-to-hang (visited on 05/20/2014).

[55] F. S. Foundation, *Gnu gcc manual. extended asm - assembler instructions with c expression operands*, 2005. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html (visited on 05/20/2014).

[56] L. S., Intel Corporation, Feb. 2005. [Online]. Available: http://software.intel.com/en-us/forums/topic/309231 (visited on 03/07/2014).

[57] J. Pierce and T. Mudge, "Wrong-path instruction prefetching", in *Microarchitecture, 1996. MICRO-29.Proceedings of the 29th Annual IEEE/ACM International Symposium on*, Dec. 1996, pp. 165–175. DOI: 10.1109/MICRO.1996.566459.

[58] A. Kleen. (Mar. 2013). Intel(r) transactional synchronization extensions (intel(r) tsx) profiling with linux perf, Intel Corporation, [Online]. Available: https://software.intel.com/en-us/blogs/2013/05/03/intelr-transactional-synchronization-extensions-intelr-tsx-profiling-with-linux-0 (visited on 04/21/2014).

[59] C. Lin, "Fully associative cache", 2003, [Online]. Available: http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Memory/fully.html (visited on 05/02/2014).

[60] ——, "Set associative cache", 2003, [Online]. Available: http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Memory/set.html (visited on 05/02/2014).

[61] J. Schlichter, *Grundlagen: betriebssysteme und systemsoftware (gbs)*, Sep. 2007. [Online]. Available: http://www11.in.tum.de/dokument.php?id_dokument=329.

[62] *Intel® transactional synchronization extensions (intel® tsx) programming considerations*, Intel Corporation, 2013. [Online]. Available: http://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-54A84479-DEC6-4D2F-9895-46D278EDA820.htm.

[63] K. Moiseev, A. Kolodny, and S. Wimer, "Timing-aware power-optimal ordering of signals", *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 4, 65:1–65:17, Oct. 2008, ISSN: 1084-4309. DOI: 10.1145/1391962.1391973. [Online]. Available: http://doi.acm.org/10.1145/1391962.1391973.

[64] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki, "Oltp on hardware islands", *CoRR*, vol. abs/1208.0227, 2012. [Online]. Available: http://arxiv.org/abs/1208.0227.

[65] R. M. Stallman and the GCC Developer Community, *Using concurrency*, GNU press, 2014. [Online]. Available: http://gcc.gnu.org/onlinedocs/libstdc++/manual/using_concurrency.html (visited on 05/12/2014).

[66] *The gnu c library (glibc)*, Aug. 2013. [Online]. Available: http://www.gnu.org/software/libc/libc.html (visited on 07/01/2014).

[67] J. Corbet, *A turning point for gnu libc*, Mar. 2012. [Online]. Available: https://lwn.net/Articles/488847/ (visited on 07/01/2014).

[68] J. S. Myers, *Gnu c library development and maintainers*, Mar. 2012. [Online]. Available: http://sourceware.org/ml/libc-alpha/2012-03/msg01040.html (visited on 07/01/2014).

[69] K. H. Hyouck, "How main() is executed on linux", *Linux Gazette*, vol. 84, Nov. 2012. [Online]. Available: http://linuxgazette.net/84/hawk.html (visited on 10/08/2013).

[70] O. Corporation, *Mysql enterprise scalability*, 2014. [Online]. Available: http://www.mysql.com/products/enterprise/scalability.html (visited on 07/03/2014).

[71] solidIT consulting & software development GmbH, *Db-engines ranking*. [Online]. Available: http://db-engines.com/en/ranking (visited on 06/10/2014).

[72] J. M. Hellerstein, M. Stonebraker, and J. Hamilton, *Architecture of a database system*. Now Publishers Inc, 2007, vol. 1.

[73] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-wesley New York, 1987, vol. 370.

[74] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery", *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, Dec. 1983, ISSN: 0360-0300. DOI: 10.1145/289.291. [Online]. Available: http://doi.acm.org/10.1145/289.291.

[75] A. Thomasian and I. K. Ryu, "Performance analysis of two-phase locking", *Software Engineering, IEEE Transactions on*, vol. 17, no. 5, pp. 386–402, May 1991, ISSN: 0098-5589. DOI: 10.1109/32.90443.

[76] A. Thomasian and E. Rahm, "A new distributed optimistic concurrency control method and a comparison of its performance with two-phase locking", in *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, 1990, pp. 294–301. DOI: 10.1109/ICDCS.1990.89296.

[77] eXcelon Corporation, *Controlling concurrency*, 2000. [Online]. Available: http://www.cslab.uky.edu/apps/odocs/osji/apiug/6n_mvcc.htm (visited on 06/14/2014).

[78] S. Faller, "Multiversion concurrency control", 2009. [Online]. Available: http://www.inf.uni-konstanz.de/dbis/teaching/ss09/tx/Sebastian.pdf (visited on 06/14/2014).

[79] J. Thijssen, *Innodb isolation levels*, Dec. 2010. [Online]. Available: https://www.adayinthelifeof.nl/2010/12/20/innodb-isolation-levels/ (visited on 06/10/2014).

[80] C. Shallahamer, *Locks and latches...what a difference!*, Oct. 2010. [Online]. Available: http://shallahamer-orapub.blogspot.com.au/2010/10/locks-and-latcheswhat-difference.html (visited on 06/10/2014).

[81] H. Jung, H. Han, A. Fekete, U. Röhm, and H. Y. Yeom, "Performance of serializable snapshot isolation on multicore servers", in *DASFAA (2)*, W. Meng, L. Feng, S. Bressan, W. Winiwarter, and W. Song, Eds., ser. Lecture Notes in Computer Science, Springer, 2013, pp. 416–430, ISBN: 978-3-642-37449-4, 978-3-642-37450-0. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37450-0_31.

[82] Oracle, *Mysql 5.5 reference manual. the myisam storage engine*, 2010. [Online]. Available: http://dev.mysql.com/doc/refman/5.5/en/myisam-storage-engine.html (visited on 06/10/2014).

[83] Rackspace, *Mysql engines - myisam vs innodb*, May 2013. [Online]. Available: http://www.rackspace.com/knowledge_center/article/mysql-engines-myisam-vs-innodb (visited on 06/10/2014).

[84] Y. Yang, *Mysql engines: innodb vs. myisam – a comparison of pros and cons*, Sep. 2009. [Online]. Available: http://www.kavoir.com/2009/09/mysql-engines-innodb-vs-myisam-a-comparison-of-pros-and-cons.html (visited on 06/10/2014).

[85] P. Zaitsev, *Should you move from myisam to innodb ?*, Jan. 2009. [Online]. Available: http://www.mysqlperformanceblog.com/2009/01/12/should-you-move-from-myisam-to-innodb/ (visited on 06/10/2014).

[86] Oracle, *Mysql 5.6 reference manual. the innodb transaction model and locking*, 2012. [Online]. Available: http://dev.mysql.com/doc/refman/5.6/en/innodb-transaction-model.html (visited on 06/10/2014).

[87] ——, *Mysql 5.6 reference manual. innodb lock modes*, 2012. [Online]. Available: http://dev.mysql.com/doc/refman/5.6/en/innodb-lock-modes.html (visited on 06/11/2014).

[88] A. Gurusami, *Introduction to transaction locks in innodb storage engine*, May 2013. [Online]. Available: https://blogs.oracle.com/mysqlinnodb/entry/introduction_to_transaction_locks_in (visited on 05/22/2014).

[89] B. Schwartz, P. Zaitsev, and V. Tkachenko, *High Performance MySQL: Optimization, Backups, and Replication*. " O'Reilly Media, Inc.", 2012.

[90] Oracle, *Mysql internals manual. innodb page structure*. [Online]. Available: http://dev.mysql.com/doc/internals/en/innodb-page-structure.html (visited on 05/22/2014).

[91] ——, *Innodb 1.1 for mysql 5.5 user's guide. integration with the mysql performance schema*, 2012. [Online]. Available: http://dev.mysql.com/doc/innodb/1.1/en/innodb-performance-schema.html (visited on 06/11/2014).

[92] C. Calender, *Mitigating the effect of metadata lock (mdl) contention*, Jul. 2013. [Online]. Available: http://www.chriscalender.com/?p=1323 (visited on 05/28/2014).

[93] C. Schneider, *Using tmpfs for mysql's tmpdir*, Sep. 2009. [Online]. Available: http://everythingmysql.ning.com/profiles/blogs/using-tmpfs-for-mysqls-tmpdir (visited on 06/11/2014).

[94] 2bits.com, Inc., *Reduce your server's resource usage by moving mysql temporary directory to tmpfs*, Mar. 2013. [Online]. Available: http://2bits.com/articles/reduce-

your-servers-resource-usage-moving-mysql-temporary-directory-ram-disk.html (visited on 06/11/2014).

[95] T. O. Group, *Pthread_mutexattr_settype(3) - linux man page*. [Online]. Available: http://linux.die.net/man/3/pthread_mutexattr_settype (visited on 05/28/2014).

[96] "Ieee standard for information technology - portable operating system interface (posix). shell and utilities", *IEEE Std 1003.1, 2004 Edition The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Shell*, 2004. DOI: 10.1109/IEEESTD.2004.94572.

[97] Oracle, *Mysql internals manual. 4.4.6.2 how to control compiler flags*. [Online]. Available: http://dev.mysql.com/doc/internals/en/controlling-compiler-flags.html (visited on 05/28/2014).

[98] M. Snir and J. Yu, "On the theory of spatial and temporal locality", University of Illinois ar Urbana-Champaign, Tech. Rep., Jul. 2005. [Online]. Available: https://www.ideals.illinois.edu/bitstream/handle/2142/11077/On%20the%20Theory%20of%20Spatial%20and%20Temporal%20Locality.pdf (visited on 10/11/2013).

[99] *Using the rdtsc instruction for performance monitoring*, Intel Corporation, 1997. [Online]. Available: http://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf (visited on 04/29/2014).

[100] G. Paoloni, *How to benchmark code execution times on intel® ia-32 and ia-64 instruction set architectures*, Intel Corporation, Sep. 2010. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf (visited on 04/29/2014).

[101] *Proceedings of the Linux Plumbers Conference*, Aug. 2012.