Transactions in the Jungle[*]

Rachid Guerraoui
EPFL Switzerland
rachid.guerraoui@epfl.ch

Thomas A. Henzinger
IST Austria
tah@ist.ac.at

Michał Kapałka
EPFL Switzerland
michal.kapalka@epfl.ch

Vasu Singh
IST Austria
vasu.singh@ist.ac.at

ABSTRACT

Transactional memory (TM) has shown potential to simplify the task of writing concurrent programs. Inspired by classical work on databases, formal definitions of the semantics of TM executions have been proposed. Many of these definitions assumed that accesses to shared data are solely performed through transactions. In practice, due to legacy code and concurrency libraries, transactions in a TM have to share data with non-transactional operations. The semantics of such interaction, while widely discussed by practitioners, lacks a clear formal specification. Those interactions can vary, sometimes in subtle ways, between TM implementations and underlying memory models.

We propose a correctness condition for TMs, *parametrized opacity*, to formally capture the now folklore notion of strong atomicity by stipulating the two following intuitive requirements: first, every transaction appears as if it is executed instantaneously with respect to other transactions and non-transactional operations, and second, non-transactional operations conform to the given underlying memory model. We investigate the inherent cost of implementing parametrized opacity. We first prove that parametrized opacity requires either instrumenting non-transactional operations (for most memory models) or writing to memory by transactions using potentially expensive read-modify-write instructions (such as compare-and-swap). Then, we show that for a class of practical relaxed memory models, parametrized opacity can indeed be implemented with constant-time instrumentation of non-transactional writes and no instrumentation of non-transactional reads. We show that, in practice, parametrizing the notion of correctness allows developing more efficient TM implementations.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Software/Program Verification

[*]This research was supported by the Swiss National Science Foundation and by the Velox FP7 European project.

General Terms

Theory, Verification

Keywords

Transactional memory, memory models, correctness

1. INTRODUCTION

Transactional memory (TM) [15, 25] offers a promising paradigm for concurrent programming in the multi-core era. A TM allows a programmer to think in terms of coarse-grained code blocks—transactions—that appear to be executed atomically; and at the same time, a TM yields a high level of parallelism. The mutual interaction between transactions is formalized by the notion of *serializability* [22] or *opacity* [13]. Serializability, a common correctness notion in database transactions, requires that committed transactions look as if they executed sequentially. But, it was observed [8, 14] that serializability is not sufficient for memory transactions, where it is important that even aborted transactions do not observe an inconsistent state of the memory. This gave rise to a new correctness criterion called opacity. The idea of opacity is to give programmers an illusion that there is no concurrency between transactions, whether committed or aborted. That is, opacity requires that all transactions "look like" they executed in some sequential order, consistent with their real-time order. Most of the TM implementations [8, 14] satisfy opacity. The formal definition of opacity has allowed for a clear understanding of transactions [13] and also for development of model checking techniques for TM algorithms [10, 11, 12].

Ideally, a program could execute all operations on shared data within transactions, and have non-transactional operations on thread-local data. In practice, however, due to the presence of legacy code and concurrency libraries, and due to programming idioms like privatization and publication, transactions have to interact with non-transactional operations [1, 7, 9, 29, 19, 21, 26, 20]. Moreover, demanding segregation of transactional data from non-transactional data poses an additional burden on the programmer. The interaction between transactions and non-transactional operations has been expressed using a notion of *strong atomicity* [19, 17] in the literature. The intuition behind strong atomicity is that transactions execute atomically with respect to other transactions and non-transactional operations. Unfortunately, strong atomicity has not been formally defined, which has thus led to multiple interpretations [28]. Consider, for example, the execution depicted in Figure 1 (adapted from [9]). The transaction executed by thread 1 updates variables x and y. Thread 2 reads the variables x and y non-transactionally. Is it possible that thread 2 reads x as 1 and y as 0? According to the definition by Martin et al. [19], strong atom-

Thread 1	Thread 2
atomic {	
$x := 1$	
$y := 1$	$r_1 := x$
}	$r_2 := y$

Figure 1: Can $r_1 = 1$ and $r_2 = 0$? It depends on the memory model (initially $x = y = 0$).

icity allows this result. But, according to the definition of strong atomicity by Larus et al. [17], this result is not allowed. The ambiguity in this definition can be attributed to an implicit assumption on the interaction between non-transactional operations, which in turn, depends upon the underlying memory model [2]. A *memory model* specifies the set of allowed behaviors of memory accesses in a shared memory program. A relaxed memory model allows the underlying system to reorder memory instructions, while a stringent memory model like sequential consistency enforces instructions to execute in program order. While Martin et al. [19] assume a relaxed memory model which allows reordering independent reads (for example, RMO [30]), Larus et al. [17] assume a sequentially consistent memory model.

We provide a general formal framework for describing the interactions between transactions and non-transactional operations. We consider opacity as a correctness condition for transactions, and parametrize it by a memory model. We claim that while a TM can be implemented in a way to ensure opacity for transactions, there is little one can do (on a given platform or run-time environment) to change the underlying memory model. Hence, it is desirable to define opacity *parametrized* by a memory model. Moreover, we want the definition of parametrized opacity to be implementation-agnostic (like opacity), so that it allows for transactional objects with semantics richer than that of simple read-write variables.

We present a definition of a memory model which is general enough to capture a variety of memory models. Intuitively, we formalize a memory model as a function which, depending upon the sequence of operations, gives the set of possible orders of operations. Our formalism can capture common memory models like TSO, PSO, RMO [30], and Alpha [27]. Moreover, we allow different processes to observe different orders of operations, which allows us to capture memory models with non-atomic stores, like IA-32 [6]. We classify memory models on the basis of the possible reorderings of operations they allow.

Our formalization of parametrized opacity is guided by the following intuition:
Opacity for transactions. Whatever the memory model is, executions that are purely transactional must ensure opacity. Indeed, the semantics of transactions should be intuitive and strong—in the end, we want TMs to be as easy to use as coarse-grained locking. Consider Figure 2(a). Thread 2 should observe x as 0 or 2, because the intermediate state of a transaction ($x = 1$) is not visible to other transactions. Also, y can be observed as 0 or 2. Moreover, y can be observed as 2 only if x is observed as 2, because the effect of transactions is visible in real-time order. Thus, the possible values of z are 0 and 2. Note that even if thread 2 aborts, opacity requires that z is 0 or 2.
Efficiency of non-transactional operations. Executions that are purely non-transactional have to adhere to the given memory model. In particular, parametrized opacity should not strengthen the semantics of non-transactional operations. The motivation here is to avoid a framework that would inherently require non-transactional operations to be instrumented with additional memory fences or software barriers, even for very weak memory models. This supports the ideology of software memory models like Java which inherently provide weak guarantees, and require special synchronization for stronger guarantees. In Figure 2(b), a memory model may relax the order of write operations in Thread 1 or the read operations in Thread 2, resulting in $r_1 = 1$ and $r_2 = 0$.
Isolation of transactions from non-transactional operations. Transactions should appear, both to other transactions and non-transactional operations, as if they were executed instantaneously. In particular, isolation of transactions should be respected, regardless of the memory model. The intermediate computations of transactions, or updates by aborted transactions, should never be visible to non-transactional operations. Moreover, the non-transactional operations concurrent to a transaction should appear as if they happened before or after this transaction. In Figure 2(c), Thread 2 cannot observe an intermediate state of a transaction, and thus $z \neq 1$. Moreover, the effect of a non-transactional operation cannot show up in the middle of a transaction. Thus, $r_1 = r_2$.

Note that opacity parametrized by sequential consistency gives the notion of strong atomicity as proposed by Larus et al. [17]. On the other hand, parametrized opacity for a relaxed memory model like RMO matches the notion of strong atomicity given by Martin et al. [19].

In practice, TM implementations that guarantee strong atomicity require that non-transactional operations, instead of accessing memory directly, adhere to an access protocol as defined by the TM implementation. This modification of the semantics of non-transactional operations is known as *instrumentation*. For example, Tabatabai et al. [26] propose a TM implementation, where the non-transactional read and write operations follow the locking discipline as required by the transactions. The formal definition of opacity parametrized by a memory model allows us to theoretically analyze the cost of creating TM implementations that guarantee parametrized opacity. While parametrized opacity is the intuitive correctness property for transactional programs with non-transactional operations, we show that, without instrumentation, it cannot be achieved on most memory models. Even for the small class of idealized memory models, where parametrized opacity can be achieved without instrumentation, we show that a TM implementation *must* use expensive read-modify-write operations for each object read and modified by a transaction.

Next, we focus on TM implementations that instrument non-transactional write operations, without any instrumentation for non-transactional read operations. We start with a basic result that shows that for memory models that allow reordering independent reads, it is possible to achieve parametrized opacity without instrumenting the read operations, and treating every non-transactional write as a transaction in itself. Note that this might not provide a practical solution, as we do not want a non-transactional operation to carry the overhead associated with a transaction. Moreover, we want non-transactional operations to finish in bounded time, while a transaction, in general, may take arbitrarily long to finish. The next question we ask is whether we can obtain parametrized opacity for a class of memory models with constant-time instrumentation on the writes? We show that for a class of memory models that allows reordering a read of a variable following a read or write of another variable (like Alpha [27] and Java [18]), it is indeed possible to achieve parametrized opacity with constant-time instrumentation for non-transactional write operations. We also discuss how to adapt the constant-time write instrumentation solution for memory models that do not allow to reorder data-dependent reads (like RMO [30]).

Using our theoretical framework, we examine existing TM implementations that guarantee parametrized opacity. TM implemen-

Thread 1	Thread 2
atomic {	
$x := 1$	
$x := 2$	*atomic* {
}	$z := x - y$
atomic {	}
$y := 2$	
}	

(a) Can $z < 0$?

Thread 1	Thread 2
$x := 1$	$r_1 := y$
$y := 1$	$r_2 := x$

(b) Can $r_1 = 1$ and $r_2 = 0$?

Thread 1	Thread 2
atomic {	$z := x$
$x := 1$	
$x := 2$	
}	
atomic {	
$r_1 := z$	
$r_2 := z$	
}	

(c) Can $z = 1$ or $r_1 \neq r_2$?

Figure 2: Motivating examples for the definition of parametrized opacity (initially $x = y = z = 0$ in every case)

tations in the literature that satisfy strong atomicity [26], in fact, are implemented with sequential consistency of non-transactional operations in mind. We observe that a TM implementation can be designed to be more efficient, if it is to satisfy opacity parametrized by a weaker memory model. We extract the key practical ideas from our proofs which shall help to design more efficient TM implementations that guarantee opacity parametrized by relaxed memory models.

2. PRELIMINARIES

We first describe a framework of a shared memory system consisting of shared objects and operations on those objects.

Operations. We consider a shared-memory system consisting of a set P of processes that communicate by executing commands on a set Obj of shared objects. Let C be a set of commands on shared objects, where arguments and return values are treated as part of a command. For example, in a system that supports only reading and writing of shared (natural number) variables, we have $C = \{\mathsf{rd}, \mathsf{wr}\} \times \mathbb{N}$. We define *operations* $O \subseteq C \times Obj$ as the set of all allowed command-object pairs. For ease of readability, we write (rd, v, x) and (wr, v, x) instead of, respectively, $((\mathsf{rd}, v), x)$ and $((\mathsf{wr}, v), x)$.

Besides commands on shared objects, every process $p \in P$ can execute the following special operations: start to start a new transaction, commit to commit the current transaction, and abort to abort the transaction. Let $\hat{O} = O \cup \{\mathsf{start}, \mathsf{commit}, \mathsf{abort}\}$.

Object semantics. We use the concept of a *sequential specification* [16] to describe the semantics of objects. Given an object $x \in Obj$, we define the semantics $[\![x]\!] \subseteq C^*$ as the set of all sequences of commands on x that could be generated by a single process accessing x.
Example. Let x be a shared variable (with initial value 0) that supports only the commands to read and write its value. Then, $[\![x]\!]$ is a subset of $(\{\mathsf{rd}, \mathsf{wr}\} \times \mathbb{N})^*$, such that, for every sequence $c_1 \dots c_n$ in $[\![x]\!]$, and for all $i, v \in \mathbb{N}$, if $c_i = (\mathsf{rd}, v)$, then either (a) the latest write operation preceding c_i in $c_1 \dots c_n$ is (wr, v), or (b) $v = 0$ and no write operation precedes c_i in $c_1 \dots c_n$.

Histories. We define an *operation instance* as (o, p, k), where $o \in \hat{O}$ is an operation, $p \in P$ is a process that issues the operation, and $k \in \mathbb{N}$ is a natural number representing the identifier of the operation instance. A *history* $h \in (\hat{O} \times P \times \mathbb{N})^*$ is a sequence of operation instances such that, for every pair (o, p, i), (o', p', j) of operation instances in h, we have $i \neq j$. Intuitively, we want each operation instance in a history to have a unique identifier. For a natural number k, when we say "operation k", we mean "operation instance with identifier k", i.e., the element of h of the form (o, p, k), where $o \in \hat{O}$ and $p \in P$. A *transaction* of a process p is a subsequence $(o_1, p, i_1) \dots (o_n, p, i_n)$ of a history h, such that (i) o_1 is a start operation, (ii) either operation i_n is the last operation instance of p in h, or $o_n \in \{\mathsf{commit}, \mathsf{abort}\}$, and (iii) all operations $o_2, \dots, o_{n-1}$ belong to set O. A transaction T is *committed* (resp. *aborted*) in a history h if the last operation instance of T has a commit operation (resp. an abort operation). A transaction T is *completed* if T is committed or aborted. Given a history h and a natural number k, we say that operation k is *transactional* in h if operation k is part of a transaction in h. Otherwise, operation k is said to be *non-transactional*. We assume that every history h is *well-formed*: that is, every non-transactional operation in h belongs to set O. Intuitively, well-formedness of a history requires that every commit and abort of a transaction matches a corresponding start, and that there are no nested transactions. We denote by H the set of all histories.

Given a history h, we define the *real-time* partial order relation $\prec_h \subset \mathbb{N} \times \mathbb{N}$ of the operation identifiers in h, such that, for two natural numbers i and j, we have $i \prec_h j$, if:
(i) operations i and j belong to transactions T and T', respectively, where T is completed in h and the last operation instance of T precedes the first operation instance of T' in h, or
(ii) operation i precedes operation j in h, both operations are executed by the same process, and at least one of those operation instances is transactional.

Sequential histories. We say that a history h is *sequential* if, for every transaction T in h, every operation instance between the start operation instance of T and the last operation instance of T in h is a part of T. That is, intuitively, no transaction overlaps with another transaction or with any non-transactional operation in h. We say that a sequential history s *respects* a partial or a total order $\prec \subseteq \mathbb{N} \times \mathbb{N}$ if, for every pair (i, j), if $i \prec j$ then operation i precedes operation j in s.

Let s be a sequential history. We denote by $s|_x$ the subsequence of all commands invoked on object x in s. We say that s is *legal* if, for every object x, we have $s|_x \in [\![x]\!]$. We denote by *visible*(s) the longest subsequence of s that does not contain any operation instance of a non-committed transaction T, except if T is not followed in s by any other transaction or non-transactional operation instance. We say that an operation k in s is *legal* in s if history *visible*(s') is legal, where s' is the prefix of s that ends with operation k.
Example. Consider the history h depicted in Figure 3(a). The transaction of process p_1 finishes before the transaction of process p_3 starts. The precedence relation $\prec_h$ consists of elements $(1, 2)$, $(5, 7)$, $(1, 9)$, and many more. On the other hand, $(1, 6)$ and $(6, 9)$ are not in $\prec_h$. An example of a sequential history is given in Figure 3(b). Note that s respects $\prec_h$. History s is legal if $v = 0$ and $v' = 1$.

3. DEFINING PARAMETRIZED OPACITY

In this section, we first formalize the notion of a memory model. Then, we define parametrized opacity, i.e., opacity parametrized by

id	p_1	p_2	p_3
1	$\mathsf{wr}, 1, x$		
2	start		
3		$\mathsf{rd}, 1, y$	
4	$\mathsf{wr}, 1, y$		
5	commit		
6		rd, v, x	
7			start
8			commit
9			rd, v', x

(a) History h

id	p_1	p_2	p_3
6		rd, v, x	
1	$\mathsf{wr}, 1, x$		
2	start		
4	$\mathsf{wr}, 1, y$		
5	commit		
3		$\mathsf{rd}, 1, y$	
7			start
8			commit
9			rd, v', x

(b) Sequential history s

Figure 3: An example of a history and a sequential history. Every history is read top to bottom. Notation: $\mathsf{wr}, 1, x$ **in the column marked with process** p **and in the row marked with id** k **stands for the operation instance** $(((\mathsf{wr}, 1), x), p, k)$**.**

a given memory model. Finally, we give a classification of memory models based on the reorderings of operations they allow.

3.1 A Memory Model

A memory model describes the semantics of memory accesses in a shared memory system. We formalize a memory model as a per-process reordering of the history. The memory model is defined in such a way that it allows different processes to have different views of the history, similar to the formalism by Sarkar et al. [24].

We define a *process view* as a function $view : P \to 2^{\mathbb{N}\times\mathbb{N}}$ such that $view(p)$ is a partial order on the set of natural numbers for every process $p \in P$. Let V be the set of all process views. A *memory model* is given by the function $M : H \to 2^V$ that maps a history to a set of process views. Intuitively, a process view allows different processes to observe different orders of operations in a given history. This helps capturing memory models that allow non-atomic stores, e.g., IA-32 [6].

Well-formed memory models. A memory model M is *well-formed* if, for every history $h \in H$, every view $view \in M(h)$, every process $p \in P$, and every pair $(i, j) \in view(p)$: (i) operations i and j are non-transactional in h, (ii) i precedes j in h, and (iii) $(j, i) \notin view(p')$ for every process $p' \in P$. Intuitively, condition (i) requires that a memory model imposes an order only on non-transactional operations, condition (ii) requires that a view does not force a process to observe some operations out of program order, and condition (iii) requires that a view does not force two processes to observe operations in different orders.[1] All memory models we know of are indeed well-formed.

Capturing dependence of operations. Often, memory models disallow reordering two operations if the latter one is control- or data-dependent on the former one [18, 30]. To capture those memory models, we distinguish between dependent and independent operation instances. We do so by using additional commands: $\{\mathsf{cdrd}, \mathsf{ddrd}, \mathsf{cdwr}, \mathsf{ddwr}\} \times \mathbb{N} \times 2^{\mathbb{N}}$. For example, an operation instance (o, p, k) in h with $o = ((\mathsf{cdrd}, v, \{k_1, \ldots, k_n\}), x)$ denotes a read operation which reads value v from variable x, and is control-dependent on operations $k_1 \ldots k_n$ in h.

Examples. We consider histories with only read and write operations on shared variables. We say that a view $view$ is identical across processes, if $view(p) = view(p')$ for all processes $p, p' \in P$.

Sequential consistency M_{SC} requires that the order of operations of a process in a history is preserved in every view, and all processes view an identical order of operations of different processes. Formally, for all histories h, we have $view \in M_{SC}(h)$ if (a) $view$ is identical across processes, and (b) for every process $p \in P$, and every pair (o_1, p, i), (o_2, p, j) of non-transactional operations such that operation i precedes operation j in h, we have $(i, j) \in view(p)$.

Total store order M_{tso} allows a write operation to forward the value of a variable to a following read operation, and allows reordering of a write operation followed by a read operation to a different variable. Formally, for every history h, we have $view \in M_{tso}(h)$ if (a) $view$ is identical across processes, and (b) for every process p, and for every pair (o_1, p, i), (o_2, p, j) of non-transactional operations such that operation i precedes operation j in h, we have $(i, j) \in view(p)$ if one of following conditions holds:

(i) o_2 is a write operation,
(ii) o_1 and o_2 are to the same object x, or
(iii) o_1 is a read operation of the form (rd, v, x) such that (wr, v, x) is not the last preceding write operation to x by process p in h.

The intuition for the last case is to allow two read operations to different variables to be reordered if the first read obtains the value from a store buffer.

Relaxed memory order allows reordering of reads to the same variable. It also allows reordering of read and write operations to different variables, unless the first operation is a read, and the second operation is either a write control/data-dependent on the first operation, or a read data-dependent on the first operation. RMO is specified by the memory model M_{rmo} such that, for every history h, we have $view \in M_{rmo}(h)$ if (a) $view$ is identical across processes, and (b) for every process p, and for every pair (o_1, p, i), (o_2, p, j) of non-transactional operations such that operation i precedes operation j in h, we have $(i, j) \in view(p)$ if one of the following conditions holds:

(i) o_1 and o_2 are to the same object x, and one of them is a write,
(ii) o_1 is a read of a variable x and $o_2 = ((\mathsf{cdwr}, v, K), y)$ or $o_2 = ((\mathsf{ddwr}, v, K), y)$ for some $v \in \mathbb{N}$ and $K \subseteq \mathbb{N}$ such that $i \in K$, or
(iii) o_1 is a read of a variable x and $o_2 = ((\mathsf{ddrd}, v, K), y)$ for some $v \in \mathbb{N}$ and $K \subseteq \mathbb{N}$ such that $i \in K$.

3.2 Parametrized Opacity

We now define the notion of parametrized opacity, i.e., opacity parametrized by a given memory model. Recall that, intuitively, parametrized opacity requires that (1) every transaction appears as if it took place instantaneously between its first and last operation, and (2) non-transactional operations ensure the requirements specified by the given memory model.

We say that a history h ensures *opacity parametrized by a memory model* M, if there exists a total order $\ll$ on the set of transactional operations in h and a process view $view \in M(h)$, such that, for every process $p \in P$, there exists a sequential history s that satisfies the following conditions: (i) s is a permutation of h, (ii) s respects relation $\ll \cup \prec_h \cup\, view(p)$, and (iii) every operation is legal in s.

Example. The history h depicted in Figure 3(a) is parametrized opaque with respect to memory model M_{SC} if $v = 1$ and $v' = 1$. This is because: (a) operation 3 reads the value of y as 1 which is written by the transaction of process p_1, (b) operation 1 writes x as 1 before the transaction, and (c) SC requires that p_2 reads x after y. Moreover, h is parametrized opaque with respect to memory model M_{rmo} if v is either 0 or 1, and $v' = 1$. This is because RMO allows reordering of reads of different variables.

[1] Note that condition (iii) does not disallow non-atomic stores, where different processes *may* observe stores in different orders. The condition disallows memory models that *force* processes to observe different orders.

3.3 Classes of Memory Models

We now present a classification of memory models on the basis of reorderings they allow. We build the following four classes depending upon the restrictions posed by the memory model.

We define the class *read-read restrictive memory models*, denoted by M_{rr}, as the set of memory models M such that for all histories h, for all $i, j \in \mathbb{N}$, if operation i is a read operation to x and operation j is a read operation to y such that $x \neq y$, and both operations i, j are by process p, then for all process views $view \in M(h)$, for all $p' \in P$, we have $(i, j) \in view(p')$. Intuitively, M_{rr} restricts the order of read operations to different variables.

We define the class *read-write restrictive memory models*, denoted by M_{rw}, as the set of memory models M such that for all histories h, for all $i, j \in \mathbb{N}$, if operation i is a read operation to x and operation j is a write operation to y such that $x \neq y$, and both operations i, j are by process p, then for all process views $view \in M(h)$, for all $p' \in P$, we have $(i, j) \in view(p')$.

Analogously, we define the class *write-read restrictive memory models* M_{wr} and the class *write-write restrictive memory models* M_{ww}.

Examples. The SC memory model M_{SC} is in $M_{rr} \cap M_{rw} \cap M_{wr} \cap M_{ww}$. The TSO memory model M_{tso} is in $M_{rr} \cap M_{rw} \cap M_{ww}$ and $M_{tso} \notin M_{wr}$. The partial store order (PSO) memory model M_{pso} is in $M_{rr} \cap M_{rw}$ and $M_{pso} \notin M_{ww} \cup M_{wr}$. Note that these classes do not impose any restrictions on views of different processes, and thus memory models which allow non-atomic stores (like IA-32 [6]) can also be classified under these classes. For example, the IA-32 memory model has a similar classification as TSO.

4. TM IMPLEMENTATIONS

We now define a TM implementation $\mathcal{I}$, and when a given TM implementation ensures opacity parametrized by a given memory model M. Intuitively, $\mathcal{I}$ ensures opacity parametrized by M if every history *generated* by $\mathcal{I}$ ensures opacity parametrized by M. We thus need to define what a TM implementation is, and precisely which histories are generated by a given TM implementation.

Instructions. We start by defining hardware primitives that a TM implementation is allowed to use. Let *Addr* be a set of memory addresses. We define the set *In* of *instructions* as follows, where $a \in Addr$ and $v, v' \in \mathbb{N}$:

$$In ::= \langle \mathsf{load}\ a, v \rangle \mid \langle \mathsf{store}\ a, v \rangle \mid \langle \mathsf{cas}\ a, v, v' \rangle$$

We call the store and CAS instructions update instructions. An operation of a history corresponds to a sequence of instructions. To know the begin and end points of an operation at the level of hardware, we use two special instructions $\rhd, \lhd$ for each operation. For every operation $o \in \hat{O}$, we denote the *invocation* (instruction) of o as $(\rhd, o)$, and the *response* (instruction) of o as $(\lhd, o)$. Let $\hat{In} = In \cup (\{\rhd, \lhd\} \times \hat{O})$.

Traces. We define an *instruction instance* as (in, p, k), where $in \in \hat{In}$ is an instruction, $p \in P$ is the process that issues the instance, and $k \in \mathbb{N}$ is an operation identifier. A *trace* $r \in (\hat{In} \times P \times \mathbb{N})^*$ is a sequence of instruction instances. Let $k \in \mathbb{N}$ be an operation identifier. A *complete operation trace* of operation k is a sequence of the form $((\rhd, o), p, k)\ (in_1, p, k) \ldots (in_m, p, k)\ ((\lhd, o), p, k)$, where $p \in P$ is a process, $o \in \hat{O}$ is an operation, and $in_1 \ldots in_m \in In$ are instructions. An *incomplete operation trace* of operation k is a sequence of the form $((\rhd, o), p, k)\ (in_1, p, k) \ldots (in_m, p, k)$, where $p \in P$, $o \in \hat{O}$, and $in_1 \ldots in_m \in In$.

Let r be a trace. Given a process $p \in P$, we denote by $r|_p$ the longest subsequence of instruction instances in r issued by process p. We assume that every trace r satisfies the following property: for every process $p \in P$, the sequence $r|_p$ is a sequence of complete operation traces, possibly ending with an incomplete operation trace. We say that a history h *corresponds* to a trace r if:

(i) given any natural number k, an operation (o, p, k) is in h if and only if there is an instruction instance $((\rhd, o), p, k)$ in r, and
(ii) operation k occurs before operation j if some instruction instance with identifier k occurs in r before some instruction instance with identifier j.

id	p_1	p_2
1	$(\rhd, \mathsf{start})$	
1	$\langle \mathsf{cas}\ g, 0, 1 \rangle$	
2		$(\rhd, (\mathsf{rd}, 1, x))$
1	$(\lhd, \mathsf{start})$	
2		$\langle \mathsf{load}\ x, 1 \rangle$
3	$(\rhd, (\mathsf{wr}, 1, x))$	
2		$(\lhd, (\mathsf{rd}, 1, x))$
3	$\langle \mathsf{store}\ a_x, 1 \rangle$	
3	$(\lhd, (\mathsf{wr}, 1, x))$	
4	$(\rhd, \mathsf{commit})$	
4	$\langle \mathsf{store}\ g, 0 \rangle$	
4	$(\lhd, \mathsf{commit}))$	

(a) Trace r

id	p_1	p_2
1	start	
2		$\mathsf{rd}, 1, x$
3	$\mathsf{wr}, 1, x$	
4	commit	

(b) History h_1

id	p_1	p_2
2		$\mathsf{rd}, 1, x$
1	start	
3	$\mathsf{wr}, 1, x$	
4	commit	

(c) History h_2

Figure 4: A trace and corresponding histories. Notation for trace: *in* in column *p* and row marked with id *k* represents the instruction instance (in, p, k)

Intuitively, a history h that corresponds to a trace r represents the logical order of operations in r. If we assign a point in time to every instruction in r, and every operation in h, then every operation (o, p, k) in h must be somewhere in between the corresponding invocation instruction $((\rhd, o), p, k)$ and response instruction $((\lhd, o), p, k)$ in r.

Example. Consider the trace r shown in Figure 4(a). In r, the start operation issues a CAS instruction to address g to change the value from 0 to 1, and the commit instruction stores value 0 to g. Histories h_1 and h_2 are two examples of histories that correspond to r. In the trace given in Figure 4(a), the (single) invocation instance of process p_2 is non-transactional, while all invocation instances of process p_1 are transactional in r.

A trace r is *well-formed* if every history h corresponding to r is well-formed. We assume that every trace is well-formed. Let r be a trace, and p be a process. A *transaction* T of p in r is a sequence in $r|_p$ of the form $((\rhd, \mathsf{start}), p, k)\ (in_1, p, k_1) \ldots (in_m, p, k_m)$, where the following conditions are satisfied:

- $in_m \in \{(\lhd, \mathsf{commit}), (\lhd, \mathsf{abort})\}$, or (in_m, p, k_m) is the last instruction instance of r_p, and
- for all j s.t. $1 \leq j < m$, we have $in_j \notin \{(\rhd, \mathsf{start}), (\lhd, \mathsf{commit}), (\lhd, \mathsf{abort})\}$

Moreover, T is committed (resp. aborted) if the last instruction instance in T is a response of a commit (resp. abort) operation. An instance $((\rhd, o), p, k)$ of an invocation is said to be *transactional* in a trace r if it belongs to some transaction in r. Otherwise, the instance is said to be *non-transactional*.

TM implementations. A *TM implementation* $\mathcal{I} = \langle \mathcal{I}_T, \mathcal{I}_N \rangle$ is a pair, where $\mathcal{I}_T : \hat{O} \to 2^{In^*}$ is the implementation for transactional operations, and $\mathcal{I}_N : O \to 2^{In^*}$ is the implementation for non-transactional operations.

Let $\mathcal{I} = \langle \mathcal{I}_T, \mathcal{I}_N \rangle$ be a TM implementation. We say that a complete operation trace $((\rhd, o), p, k)\ (in_1, p, k) \ldots (in_m, p, k)$

$((\triangleleft, o), p, k)$ is *transactionally* (resp. *non-transactionally*) *generated* by $\mathcal{I}$, if sequence $in_1 \dots in_m$ is in $\mathcal{I}_T(o)$ (resp. in $\mathcal{I}_N(o)$). We say that an incomplete operation trace $((\triangleright, o), p, k)$ (in_1, p, k) $\dots (in_m, p, k)$ is *transactionally* (resp. *non-transactionally*) *generated* by $\mathcal{I}$, if sequence $in_1 \dots in_m$ is a prefix of some element in $\mathcal{I}_T(o)$ (resp. in $\mathcal{I}_N(o)$).

Given a trace r and a TM implementation $\mathcal{I}$, we say that r is *generated* by $\mathcal{I}$ if for every transactional (resp. non-transactional) operation k in r, the complete or incomplete operation trace of operation k in r is transactionally (resp. non-transactionally) generated by $\mathcal{I}$.

Instrumentation. We say that a TM implementation $\mathcal{I} = \langle \mathcal{I}_T, \mathcal{I}_N \rangle$ is *uninstrumented* if for every variable x, we have $\mathcal{I}_N(\mathsf{rd}, v, x) = \{\langle \mathsf{load}\ a_x, v\rangle\}$ and $\mathcal{I}_N(\mathsf{wr}, v, x) = \{\langle \mathsf{store}\ a_x, v\rangle\}$, where a_x is the address of the global copy of variable x. Otherwise, the TM implementation is *instrumented*. Note that these terms refer only to the implementation of non-transactional operations.

Languages. We define the *language* $L(\mathcal{I})$ of a TM implementation as the set of all traces generated by a TM implementation. We define that a TM implementation $\mathcal{I}$ *guarantees* opacity parametrized by a memory model M if, for every trace $r \in L(\mathcal{I})$, there is a history h that corresponds to r such that h ensures opacity parametrized by M.

Note that our definition of a trace does not require that after an instruction of the form $\langle \mathsf{store}\ a_x, v\rangle$, the subsequent load of a_x is of the form $\langle \mathsf{load}\ a_x, v\rangle$. Indeed, the underlying hardware may execute a relaxed memory model (which, in principle, may be different from the programmer's memory model at the level of operations). For example, a programmer may wish to guarantee opacity parametrized by sequential consistency on a hardware with memory model RMO. But for the sake of simplicity, in the following sections, we assume that the underlying hardware guarantees a strong memory model equivalent to linearizability, that is, every instruction is executed to completion when it is issued. Note that our impossibility results hold even when the underlying hardware executes a weaker memory model.

5. ACHIEVING PARAMETRIZED OPACITY

We use our theoretical framework to investigate the inherent cost of achieving opacity parametrized by a memory model.

5.1 Uninstrumented TM Implementations

We first study uninstrumented TM implementations, as those ones do not pose any overhead on non-transactional operations. We show that for most of the practical memory models, uninstrumented TM implementations cannot achieve parametrized opacity (Theorem 1). Moreover, we show that even to achieve opacity parametrized by very relaxed memory models, it is required that transactional write operations for variables read and modified are implemented as expensive compare-and-swap instructions (Theorem 2). Finally, we establish a complementary result to Theorem 1. We show that with an idealized memory model that relaxes the order of all operations to different variables, it is possible to obtain parametrized opacity with an uninstrumented TM implementation (Theorem 3).

We first prove an auxiliary lemma, which states that if a solo running committed transaction consists of a write operation, then the transaction must consist of a store or a compare-and-swap instruction.

Lemma 1. For every memory model M, an uninstrumented TM implementation $\mathcal{I}$ guarantees parametrized opacity only if, for all traces $r \in L(\mathcal{I})$, and for every committing transaction T in r such that there is no instruction instance of another transaction between the first and last instruction instances of T in r, if there is an operation (wr, v, x) in T, then the transaction T in r contains an update instruction to a_x with value v.

PROOF. Let the value of a_x be initially 0. Consider the trace r depicted in Figure 5(a). Let T be the committed transaction of process p in r. We observe that T consists of an operation $o_1 = (\mathsf{wr}, v, x)$. Note that the invocation of the non-transactional operation $o_2 = (\mathsf{rd}, v', x)$ occurs after the last instruction of T, that is, the response of the commit operation. As the TM implementation $\mathcal{I}$ is uninstrumented, we have $\mathcal{I}_N(\mathsf{rd}, v', x) = \{\langle \mathsf{load}\ a_x, v'\rangle\}$. Note that as r has a single process, there is only one history h corresponding to r. By definition of $\prec_h$, we know that $o_1 \prec_h o_2$. Thus, to guarantee parametrized opacity, we must have $v = v'$. Thus, T must issue an update instruction to a_x with value v. □

Theorem 1. Given a memory model M such that $M \in (M_{rr} \cup M_{rw} \cup M_{wr} \cup M_{ww})$, there does not exist any uninstrumented TM implementation that guarantees opacity parametrized by M.

PROOF. By contradiction, we assume that there exists an uninstrumented TM implementation $\mathcal{I}$ that ensures opacity parametrized by a memory model $M \in (M_{rr} \cup M_{rw} \cup M_{wr} \cup M_{ww})$. We consider four cases, each corresponding to a different class of memory models, in each case showing a trace r of $\mathcal{I}$, involving two processes p_1 and p_2, that violates opacity parametrized by M. In the following, we assume that a_x and a_y are initialized to 0.

Case 1. Let $M \in M_{rr}$. That is, M does not allow reordering two read operations to different variables. Trace r is depicted in Figure 5(b). Let T consist of operations (wr, v_1, x) and (wr, v_2, y). Let $v_1 \neq 0$ and $v_2 \neq 0$. From Lemma 1, we know that T updates addresses a_x and a_y with values v_1 and v_2, respectively. Without loss of generality, we assume that a_x is updated before a_y.[2] Let the trace r consist of two non-transactional operations (rd, v_3, x) and (rd, v_4, y) issued by process p_2 (with identifiers j and k respectively). As $\mathcal{I}$ is uninstrumented, the non-transactional read operations are implemented as load instructions. Let the two load instructions $\langle \mathsf{load}\ a_x, v_3\rangle$ and $\langle \mathsf{load}\ a_y, v_4\rangle$ of process p_2 execute between the updates of a_x and a_y. Thus, we have $v_3 = v_1$ and $v_4 = 0$. Note that if T is an aborted transaction in r, then there is no history h corresponding to r such that h satisfies opacity parametrized by M. This is because operation j observes the update to a_x. Thus, T is a committed transaction in r. Consider an arbitrary history h corresponding to r. By definition of M_{rr}, every process view $view \in M(h)$ requires that $(j, k) \in view(p)$ for all $p \in P$. On the other hand, legality requires operation j to appear after T, and operation k to appear before T. Thus, there does not exist a sequential history which satisfies conditions (ii) and (iii) of parametrized opacity at the same time.

Case 2. Let $M \in M_{wr}$. That is, M does not allow reordering a write followed by a read operation to a different variable. The trace r is shown in Figure 5(c). Let T consist of operations $o_1 = (\mathsf{rd}, v_1, x)$ and $o_2 = (\mathsf{wr}, v_2, y)$. Let $v_2 \neq 0$. From Lemma 1, we know that T updates address a_y with value v_2. Let r consist of two non-transactional operations of p_2 in the order

[2]Figure 5(b) shows the updates as part of the commit operation. In general, the updates can happen anywhere during the transaction, but always as two separate instructions.

p
$\triangleright$, start
...
$\triangleleft$, start
$\triangleright, (\mathsf{wr}, v, x)$
...
$\triangleleft, (\mathsf{wr}, v, x)$
$\triangleright$, commit
...
$\triangleleft$, commit
$\triangleright, (\mathsf{rd}, v', x)$
$\langle \mathsf{load}\ a_x, v' \rangle$
$\triangleleft, (\mathsf{rd}, v', x)$

(a) Lemma 1

p_1	p_2
$\triangleright$, start	
...	
$\triangleleft$, start	
$\triangleright, (\mathsf{wr}, v_1, x)$	
...	
$\triangleleft, (\mathsf{wr}, v_1, x)$	
$\triangleright, (\mathsf{wr}, v_2, y)$	
...	
$\triangleleft, (\mathsf{wr}, v_2, y)$	
$\triangleright$, commit	
...	
$\langle \mathsf{upd}\ a_x, v_1 \rangle$	
...	
	$\triangleright, (\mathsf{rd}, v_3, x)$
	$\langle \mathsf{load}\ a_x, v_3 \rangle$
	$\triangleleft, (\mathsf{rd}, v_3, x)$
	$\triangleright, (\mathsf{rd}, v_4, y)$
	$\langle \mathsf{load}\ a_y, v_4 \rangle$
	$\triangleleft, (\mathsf{rd}, v_4, y)$
$\langle \mathsf{upd}\ a_y, v_2 \rangle$	
...	
$\triangleleft$, commit	

(b) Theorem 1, Case 1

p_1	p_2
$\triangleright$, start	
...	
$\triangleleft$, start	
$\triangleright, (\mathsf{rd}, v_1, x)$	
...	
$\triangleleft, (\mathsf{rd}, v_1, x)$	
$\triangleright, (\mathsf{wr}, v_2, y)$	
...	
$\triangleleft, (\mathsf{wr}, v_2, y)$	
$\triangleright$, commit	
...	
	$\triangleright, (\mathsf{wr}, v_3, x)$
	$\langle \mathsf{store}\ a_x, v_3 \rangle$
	$\triangleleft, (\mathsf{wr}, v_3, x)$
	$\triangleright, (\mathsf{rd}, 0, y)$
	$\langle \mathsf{load}\ a_y, 0 \rangle$
	$\triangleleft, (\mathsf{rd}, 0, y)$
$\langle \mathsf{upd}\ a_y, v_2 \rangle$	
...	
$\triangleleft$, commit	

(c) Theorem 1, Case 2

p_1	p_2
$\triangleright$, start	
...	
$\triangleleft$, start	
$\triangleright, (\mathsf{wr}, v_1, x)$	
...	
$\triangleleft, (\mathsf{wr}, v_1, x)$	
$\triangleright, (\mathsf{wr}, v_2, y)$	
...	
$\triangleleft, (\mathsf{wr}, v_2, y)$	
$\triangleright$, commit	
...	
$\langle \mathsf{upd}\ a_x, v_1 \rangle$	
	$\triangleright, (\mathsf{rd}, v_3, x)$
	$\langle \mathsf{load}\ a_x, v_3 \rangle$
	$\triangleleft, (\mathsf{rd}, v_3, x)$
	$\triangleright, (\mathsf{wr}, v_4, y)$
	$\langle \mathsf{store}\ a_y, v_4 \rangle$
	$\triangleleft, (\mathsf{wr}, v_4, y)$
	$\triangleright, (\mathsf{wr}, 0, y)$
	$\langle \mathsf{store}\ a_y, 0 \rangle$
	$\triangleleft, (\mathsf{wr}, 0, y)$
$\langle \mathsf{upd}\ a_y, v_2 \rangle$	
...	
$\triangleleft$, commit	
	$\triangleright$, start
	...
	$\triangleleft$, commit
	$\triangleright, (\mathsf{rd}, v_5, x)$
	$\langle \mathsf{load}\ a_x, v_5 \rangle$
	$\triangleleft, (\mathsf{rd}, v_5, x)$
	$\triangleright, (\mathsf{rd}, v_6, y)$
	$\langle \mathsf{load}\ a_y, v_6 \rangle$
	$\triangleleft, (\mathsf{rd}, v_6, y)$

(d) Theorem 1, Case 3

p_1	p_2
$\triangleright$, start	
...	
$\triangleleft$, start	
$\triangleright, (\mathsf{rd}, v, x)$	
...	
$\triangleleft, (\mathsf{rd}, v, x)$	
$\triangleright, (\mathsf{wr}, v', x)$	
...	
$\triangleleft, (\mathsf{wr}, v', x)$	
$\triangleright$, commit	
...	
	$\triangleright, (\mathsf{wr}, v_1, x)$
	$\langle \mathsf{store}\ a_x, v_1 \rangle$
	$\triangleleft, (\mathsf{wr}, v_1, x)$
$\langle \mathsf{store}\ a_x, v' \rangle$	
	$\triangleright, (\mathsf{rd}, v_2, x)$
	$\langle \mathsf{load}\ a_x, v_2 \rangle$
	$\triangleleft, (\mathsf{rd}, v_2, x)$
...	
$\triangleleft$, commit	
	$\triangleright$, start
	...
	$\triangleleft$, commit
	$\triangleright, (\mathsf{rd}, v_3, x)$
	$\langle \mathsf{load}\ a_x, v_3 \rangle$
	$\triangleleft, (\mathsf{rd}, v_3, x)$

(e) Theorem 2

Figure 5: Different traces r constructed in Lemma 1, Theorem 1, and Theorem 2. An $\langle \mathsf{upd}\ a, v \rangle$ instruction denotes a store or a successful cas instruction to address a with value v. We omit the instruction identifiers. We use ... as a shorthand for a sequence of instructions.

(wr, v_3, x) (with identifier j) followed by $(\mathsf{rd}, 0, y)$ (with identifier k), such that $v_3 \neq v_1$. As $\mathcal{I}$ is uninstrumented, p_2 executes $\langle \mathsf{store}\ a_x, v_3 \rangle$ followed by $\langle \mathsf{load}\ a_y, 0 \rangle$. Let these two instructions execute after the response of o_1 and immediately before the update of a_y by process p_1.[3] Note that as p_2 does not change the value of a_y, the use of CAS instruction by p_1 to update a_y will be successful. The key point to note here is that once p_1 updates a_y with value v_2, transaction T cannot abort. This is because r can be extended to trace r', where a non-transactional read by process p_2 executes $\langle \mathsf{load}\ a_y, v_2 \rangle$. If T is an aborted transaction, then no history corresponding to r' is parametrized opaque with respect to M. Thus, T is a committed transaction. Consider an arbitrary history h corresponding to r. Legality requires that operation k occurs before T and operation j occurs after T. By definition of M_{wr}, every view $\mathit{view} \in M(h)$ requires that $(j, k) \in \mathit{view}(p)$ for all $p \in P$. Thus, h does not ensure opacity parametrized by M.

Case 3. Let $M \in M_{rw}$. That is, M does not allow reordering a read followed by a write operation to a different variable. The trace r is depicted in Figure 5(d). Let T consist of operations $o_1 = (\mathsf{wr}, v_1, x)$ and $o_2 = (\mathsf{wr}, v_2, y)$. Let $v_1 \neq 0$ and $v_2 \neq 0$. From Lemma 1, we know that T updates addresses a_x and a_y with values v_1 and v_2. Without loss of generality, we assume that a_x is updated before a_y. Process p_2 issues the following operations non-transactionally in the order: (rd, v_3, x), (wr, v_4, y), $(\mathsf{wr}, 0, y)$. Since $\mathcal{I}$ is uninstrumented, those three operations are executed as: $\langle \mathsf{load}\ a_x, v_3 \rangle$, $\langle \mathsf{store}\ a_y, v_4 \rangle$, and $\langle \mathsf{store}\ a_y, 0 \rangle$. Let r be such that those three instructions occur immediately before the update of a_y with value v_2. As p_2 changes and restores the value of a_y to 0, process p_1 does not observe the change, and thus, the update of a_y with value v_2 is successful. As in case 2, T cannot be an aborting transaction, once T updates a_y. We now extend r as follows: after the response of the commit operation of T, let p_2 execute a transaction T' followed by two non-transactional operations: (rd, v_5, x), (rd, v_6, y).

We show now that the value of a_y must remain equal to v_2 after T commits. Assume otherwise: that T stores value $v' \neq v_2$ to a_y just before committing. Consider a trace r' in which p_1 performs the same actions as in r and then reads y non-transactionally, and in which p_2 does not perform any actions. Since p_1 cannot distinguish r' from r, p_1 returns value v' in the non-transactional read after T, thus violating opacity parametrized by M—a contradiction.

Therefore, $v_6 = v_2$. Note also that $v_5 = v_1$. Consider an arbitrary history h corresponding to r. Legality requires that the first non-transactional read by process p_2 occurs after T, and the two non-transactional writes of p_2 occur before T. This contradicts the expected process view for a memory model $M \in M_{rw}$. Thus, the history h does not ensure opacity parametrized by M.

Case 4. Let $M \in M_{ww}$. That is, M does not allow reordering two write operations to different variables. The trace r is similar to the one for case 3, except that (1) T consists of four operations: (rd, v_1', x), (rd, v_2', y), (wr, v_1, x) and (wr, v_2, y), and (2) p_2 executes operation (wr, v_3, x) instead of (rd, v_3, x). Let $v_1 \neq 0$ and $v_2 \neq 0$ and $v_3 \neq 0$. As in case 3, we know that T updates addresses a_x and a_y with values v_1 and v_2, and we assume that a_x is updated before a_y. Since $\mathcal{I}$ is uninstrumented, the non-transactional write operations are implemented as store instructions. Let the three store instructions: $\langle \mathsf{store}\ a_x, v_3 \rangle$, $\langle \mathsf{store}\ a_y, v_4 \rangle$, and $\langle \mathsf{store}\ a_y, 0 \rangle$ by process p_2 occur immediately before the update of a_y with value v_2 by process p_1 in trace r. As in case 2, T cannot be an aborting transaction, after it updates a_y. Now, note that after updating

[3]For simplicity, we assume that the transaction loads the value of a_x before the response of o_1. The argument holds in general, as the value of a_x is loaded before the update of a_y.

```
On transaction start:
$lg := g$
while ($lg \neq p$) do
    if $lg = 0$
        $\langle \mathsf{cas}\ g, lg, p \rangle$
    endif
    $lg := g$
endwhile
return

On transaction write of $x$ with value $v'$:
issue a transactional read of $x$
if $\exists v \cdot (x, v) \in writeset(p)$
    update $(x, v)$ to $(x, v')$ in $writeset(p)$
    return
endif
add $(x, v)$ to $writeset(p)$
return

On transaction commit:
while $writeset(p)$ is not empty
    pick and remove $(x, v')$ from $writeset(p)$
    pick and remove $(x, v)$ from $readset(p)$
    $\langle \mathsf{cas}\ x, v, v' \rangle$
endwhile
empty $readset(p)$
$\langle \mathsf{store}\ g, 0 \rangle$
return

On transaction read of $x$:
if $\exists v \cdot (x, v) \in readset(p)$
    return $v$
endif
$\langle \mathsf{load}\ a_x, v \rangle$
add $(x, v)$ to $readset(p)$
return $v$

On transaction abort:
empty $readset(p)$
empty $writeset(p)$
$\langle \mathsf{store}\ g, 0 \rangle$
return
```

Figure 6: A global lock based TM implementation

a_y, even if T observes the non-transactional write to a_x, T cannot overwrite a_x with v_2: this requires the non-transactional write to x to appear before the transaction, but T observes x as 0, and not as v_1'. We can extend the trace r exactly as in case 3 now, and show a contradiction between the legality and the process view required by a memory model $M \in M_{ww}$. Thus, for every history h corresponding to r, we know that h does not ensure opacity parametrized by M. □

We saw in the classification of memory models (Section 3.3) that most of the practical memory models do restrict some order of operations. Thus, Theorem 1 gives an intuition that without instrumentation, it is not possible to achieve opacity parametrized by practical memory models. We now show that for an idealized memory model, which allows reordering all operations to different variables, achieving parametrized opacity is still expensive: a TM implementation must use a cas instruction within a transaction to update every variable which is read and written by the transaction.

Theorem 2. For every memory model M, an uninstrumented TM implementation $\mathcal{I}$ guarantees parametrized opacity with respect to M only if, for all traces $r \in L(\mathcal{I})$, and for every variable x, if a committed transaction T consists of operations (rd, v, x) and (wr, v', x), then T contains a $\langle \mathsf{cas}\ x, v, v' \rangle$ instruction.

PROOF. Consider a trace r with two processes p_1 and p_2 as shown in Figure 5(e). Let T be a transaction of process p_1 in r, such that T consists of operations (rd, v, x) and (wr, v', x). By Lemma 1, we know that T updates a_x with value v' using either a store or a compare-and-swap instruction. Suppose T changes the value of a_x using a store instruction. Let r consist of two non-transactional operations (wr, v_1, x) followed by (rd, v_2, x) of process p_2. As $\mathcal{I}$ is uninstrumented, we know that a read operation is implemented as a load, and a write as a store. Consider the trace r where $\langle \mathsf{store}\ a_x, v_1 \rangle$ instruction of process p_2 occurs immediately before $\langle \mathsf{store}\ a_x, v' \rangle$ instruction of process p_1 and the instruction $\langle \mathsf{load}\ a_x, v_2 \rangle$ occurs immediately after $\langle \mathsf{store}\ a_x, v' \rangle$, such that we get $v_2 = v'$. For a corresponding history of r to be parametrized opaque with respect to M, the transaction T has to be a committed transaction. After the response of the commit of T, let there be an empty transaction T' of process p_2 in r followed by a non-transactional read of x, with a load instruction $\langle \mathsf{load}\ x, v_3 \rangle$. Note that we have $v_3 = v'$. We argue that irrespective of the memory model M, there does not exist a history h corresponding to r such that h ensures opacity parametrized by M. This is because the operation (wr, v_1, x) of process p_2 can appear neither before T (as the read of v in T returns 0), nor after T (as the read after T' returns v'). Thus, an uninstrumented TM implementation has to use a cas instruction to update a variable x in a transaction T, if T consists of both read and write operations to x. □

Theorem 3. Given a memory model $M \notin (M_{rr} \cup M_{rw} \cup M_{wr} \cup M_{ww})$, there exists an uninstrumented TM implementation that guarantees opacity parametrized by M.

PROOF. Consider an uninstrumented global lock based TM implementation $\mathcal{I}$ described in pseudo code in Figure 6. $\mathcal{I}$ acquires a global lock g during the start of each transaction, and releases the lock (using $\langle \mathsf{store}\ g, 0 \rangle$) immediately before the response of the commit or abort of the transaction. Moreover, for every variable x, every transaction T loads the value of a_x only at the first read or write operation to x within T (if such an operation exists in T), and if T writes to x, then $\mathcal{I}$ uses a CAS instruction to update a_x in T. Consider an arbitrary trace $r \in L(\mathcal{I})$. Consider a history h corresponding to the trace r obtained by choosing the following logical points of execution of an operation within an operation trace: start operation at the successful cas instruction, commit and abort operations at $\langle \mathsf{store}\ g, 0 \rangle$ instruction, every non-transactional read at the load instruction, every non-transactional write at the store instruction, and every transactional read or write at its invocation. First, we note that the history h obtained is such that for every pair T, T' of transactions in h, either $T \prec_h T'$ or $T' \prec_h T$. Now, consider a variable x and a transaction T in h. Let $k_1 \ldots k_n$ be a sequence of non-transactional operation instances to x in h, which occur between the first and last operations of the transaction T. We create a sequential history s from h by repeating the following two steps for all variables:
Step 1. Consider a non-transactional write operation k_i for some $1 \leq i \leq n$. If there is a load of x in T before the store instruction, or there is no load or cas of x in T after the store instruction with identifier k_i in r, we place all operations $k_i \ldots k_n$ after the transaction T in s. For all other non-transactional write operations k_i, we place all operations $k_1 \ldots k_i$ before the transaction T in s. We now have no non-transactional write operation to x between the first and last operations of T.
Step 2. For all remaining non-transactional read operations o, we place o before T in s if the load instruction corresponding to o precedes the update to x by T in r, and after T otherwise.

Note that as the memory model freely allows reordering instructions to independent variables, we can move operations of different variables freely with respect to each other. Moreover, the two steps do not change the order of non-transactional operations of a process to a variable. □

5.2 Instrumented TM Implementations

In practice, TM implementations use instrumentation of non-transactional operations to achieve parametrized opacity. We now investigate instrumented TM implementations. Typically, a history contains more read operations than write operations. So, it is worthwhile to study whether we can achieve parametrized opacity by just instrumenting the non-transactional write operations and leaving the non-transactional read operations uninstrumented.

We first show that it is indeed possible to achieve parametrized opacity with uninstrumented reads for a class of memory models that allow to reorder read operations (Theorem 4). The construction implements non-transactional write operations as single operation transactions. Then, we show that for memory models that do not belong to set $M_{rr} \cup M_{wr}$, it is possible to implement parametrized opacity with uninstrumented reads and with constant-time instrumentation of writes (Theorem 5).

Theorem 4. There exists a TM implementation with uninstrumented reads that guarantees parametrized opacity for memory models $M \notin M_{rr}$.

PROOF. Consider a global lock based TM implementation $\mathcal{I}$ that treats transactional operations as in the proof of Theorem 3, and shown in Figure 6. Moreover, $\mathcal{I}$ implements a non-transactional write operation (wr, x, v) as: acquire the global lock g using cas (as in transaction start), followed by the instruction $\langle \mathsf{store}\ x, v\rangle$, followed by $\langle \mathsf{store}\ g, 0\rangle$. Intuitively, $\mathcal{I}$ treats every non-transactional write operation as a transaction in itself. $\mathcal{I}$ implements non-transactional read operations as load instructions (no instrumentation).

Consider an arbitrary trace $r \in L(\mathcal{I})$. Note that there exists a history h corresponding to r (obtained using the logical points as in Theorem 3) such that no non-transactional write occurs between the first and last operation of a transaction, and for every pair T, T' of transactions, either $T \prec_h T'$ or $T' \prec_h T$. Consider a variable x. Given a transaction T in h, let $k_1 \ldots k_n$ be the identifiers of non-transactional read operation instances in h, which occur between the first and last operation of the transaction T. We create a sequential history s from h as in Theorem 3: for all non-transactional read operations o to x, we place o before T in s if the load instruction corresponding to o precedes the cas instruction to x by T in r, and after T otherwise. Note that as the memory model freely allows reordering read operations to different variables, we can move reads of different variables freely with respect to each other. □

Theorem 5. There exists a TM implementation $\mathcal{I}$ with constant-time instrumentation of writes and no instrumentation of reads, such that $\mathcal{I}$ guarantees opacity parametrized by any memory model $M \notin M_{rr} \cup M_{wr}$.

PROOF. We build a TM implementation $\mathcal{I}$ which uses a global lock to execute transactions (as in Theorem 3 and 4). Moreover, $\mathcal{I}$ uses a version number per process. When a process p issues a non-transactional write operation, p increments its version number, and writes the value, the process id, and the version number using a store instruction. $\mathcal{I}$ does not use instrumentation for non-transactional read operations. Now, we prove that $\mathcal{I}$ guarantees opacity parametrized by M, where $M \notin M_{rr} \cup M_{wr}$.

Consider an arbitrary trace $r \in L(\mathcal{I})$. Consider a history h corresponding to the trace r obtained by choosing the logical points as in Theorem 3. We know that for every two transactions T, T' in h, we have $T \prec_h T'$ or $T' \prec_h T$. Consider an arbitrary transaction T in h and a variable x. Let $k_1 \ldots k_m$ be the identifiers of the non-transactional operations to x that occur between the first and last operation of T in h. Note that as $\mathcal{I}$ uses cas instruction for variables which are written in a transaction, no non-transactional write to x stores to a_x between a load and a successful update of a_x in T. We thus can obtain a sequential history from h by repeating the following for all variables x and for all transactions in h: for a non-transactional write operation k_i to x such that operation k_i occurs between the start and end of transaction T, if the corresponding store to a_x occurs after a load of a_x in T, then we place operation k_i after T in s. Otherwise we place k_i before T in s. If the load corresponding to a non-transactional read occurs after the update of a_x in T, then we place the operation after T, otherwise before T.

If a process issues non-transactional reads of x and y, such that their corresponding loads occur between the updates by a transaction, we need to reorder the non-transactional reads for legality. Thus, we want $M \notin M_{rr}$. Similarly, note that if a process issues a non-transactional write of x followed by a read of y, such that the corresponding store to a_x and load of a_y occur between the updates to a_x and a_y, we need to reorder the non-transactional accesses for legality. Thus, we want $M \notin M_{wr}$. But interestingly, we built $\mathcal{I}$ in such a way that it first adds all variables to be updated in the writeset. Only then, $\mathcal{I}$ starts updating the variables using cas instruction. Hence, if a process issues a non-transactional read which loads a value updated by T, we know that every following non-transactional write can also occur after T. Thus, we do not require that $M \notin M_{rw}$. Similarly, if a process issues two non-transactional writes, then we know that either both the writes occur before T or after T. This implies that $\mathcal{I}$ guarantees parametrized opacity even for memory models which restrict the order of a read/write followed by a write to a different variable, but allow reordering read/write followed by a read to a different variable. □

6. CONCLUDING REMARKS

Discussion. Given that a programmer expects behavior of non-transactional operations within the scope of behaviors under the given memory model, one can build a more efficient implementation which guarantees opacity with respect to the programmer's memory model. Relaxed memory models like Alpha [27] and Java [18] are neither in M_{rr}, nor in M_{wr}. So, our construction provides an inexpensive way to guarantee opacity parametrized by memory models like Alpha. Moreover, memory models like RMO [30] relax the order of reads as long as they are not data dependent. This implies that if we use special synchronization for data-dependent reads, we can use the construction of Theorem 5 for a vast class of memory models.

We now discuss the construction we used in Theorems 3 and 5. We use global locks for transactions in the construction for the sake of simplicity. But the central idea of the construction, given below, can be extended to practical lazy-versioning TM implementations which rely on some form of two-phased locking. The idea is that for many memory models, it is not necessary for a transaction to successfully update all variables it writes, if there is a concurrent non-transactional write. For example, in Theorem 5, if a transaction observes that a non-transactional operation has written a new value, then the transaction's cas operation fails, but the transaction can still commit. This sounds counterintuitive to the general belief that transactions must be atomic, and thus, appear to perform their updates completely. We suggest that the non-transactional operations, although isolated from transactions from a programmer's point of view, can be used to mask updates of transactions.

Using our theoretical framework, we also observe that no matter what the given memory model M is, if a TM implementation allows a transaction to update the value of an address a_x more than once, then the TM implementation needs to instrument read operations in order of guarantee opacity parametrized by M. This is because a load of a_x (corresponding to a non-transactional operation), if sandwiched between the two updates of a_x, can observe the intermediate state of a transaction. This is also known as *dirty reads* in the literature.

Related work. Various formalisms for memory models have been proposed in the literature [3, 4, 5, 12, 23, 24]. Most of these formalisms are tailored to capture the intricacies of specific memory models. Our reason for building a formalism for memory models was to obtain a general framework, with the focus on classification of memory models on the basis of reordering of instructions they allow. The interaction of transactions with non-transactional operations has been widely studied. The study was pioneered by Grossman et al. [9], where the authors raised several issues that need to be tackled in order to create TM implementations that handle non-transactional operations properly. The authors express the correctness property using sample executions, and thus it lacks a formal specification. Work by Scott et al. [28] focuses on providing a set of rules that should hold irrespective of the memory model. Menon et al. [20] define correctness by mapping transactions to critical sections, thus providing an intuitive definition of single global lock atomicity. Type systems and operational semantics for transactional programs with non-transactional accesses have been proposed by Abadi et al. [1] and Grossman et al. [21].

Conclusion. We formalized the correctness property of opacity parametrized by a memory model. Intuitively, parametrized opacity requires two things. Firstly, transactions are isolated from other transactions and non-transactional operations. Secondly, the behavior of non-transactional operations is governed by the underlying memory model. We used our formalism to prove several results on achieving parametrized opacity with instrumented or uninstrumented TM implementations under different memory models. In particular, we show that for most memory models, parametrized opacity cannot be achieved without instrumenting non-transactional operations. On the positive side, we show that with constant-time instrumentation for writes, and no instrumentation for reads, it is indeed possible to achieve opacity parametrized by a significant class of memory models. The practical relevance of our work lies in the fact that while creating TM implementations, one may use the relaxations provided by the memory model to create more efficient TM implementations.

7. REFERENCES

[1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, pages 63–74, 2008.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[3] Arvind and Jan-Willem Maessen. Memory Model = Instruction Reordering + Store Atomicity. In *ISCA*, pages 29–40, 2006.

[4] Gérard Boudol and Gustavo Petri. Relaxed memory models: An operational approach. In *POPL*, pages 392–403, 2009.

[5] Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. Verifying local transformations on relaxed memory models. In *CC*, pages 104–123, 2010.

[6] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A*. 2008.

[7] Luke Dalessandro and Michael Scott. Strong isolation is a weak idea. In *TRANSACT*, 2009.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, pages 194–208. Springer, 2006.

[9] Dan Grossman, Jeremy Manson, and William Pugh. What do high-level memory models mean for transactions? In *MSPC*, pages 62–69, 2006.

[10] Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. Model checking transactional memories. In *PLDI*, pages 372–382. ACM, 2008.

[11] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Nondeterminism and completeness in transactional memories. In *CONCUR*, pages 21–35. Springer, 2008.

[12] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Software transactional memory on relaxed memory models. In *CAV*, pages 321–336, 2009.

[13] Rachid Guerraoui and Michał Kapałka. On the correctness of transactional memory. In *PPoPP*, 2008.

[14] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[15] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300. ACM Press, 1993.

[16] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[17] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[18] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL*, pages 378–391. ACM, 2005.

[19] Milo M. K. Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), 2006.

[20] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for Java STM. In *SPAA*, pages 314–325, 2008.

[21] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *POPL*, pages 51–62. ACM, 2008.

[22] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, pages 631–653, 1979.

[23] Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. In *PPoPP*, pages 161–172, New York, NY, USA, 2007. ACM.

[24] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL*, pages 379–391, 2009.

[25] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[26] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *PLDI*, pages 78–88. ACM, 2007.

[27] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 2002.

[28] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. Ordering-based semantics for software transactional memory. In *OPODIS*, pages 275–294, 2008.

[29] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *PODC*, pages 338–339, 2007.

[30] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual (version 9)*. Prentice-Hall, Inc., 1994.