A Composable Mixed Mode Concurrency Control Semantics for Transactional Programs

Granville Barnett[1] and Shengchao Qin[2,3]

[1] School of Engineering and Computing Sciences, Durham University
[2] School of Computing, Teesside University
[3] State Key Lab. for Novel Software Technology, Nanjing University
granville.barnett@durham.ac.uk, s.qin@tees.ac.uk

Abstract. Most software transactional memories employ optimistic concurrency control. A pessimistic semantics, however, is not without its benefits: its programming model is often much simpler to reason about and supports the execution of irreversible operations. We present a programming model that supports both optimistic and pessimistic concurrency control semantics. Our pessimistic transactions, guaranteed transactions (gatomics), afford a stronger semantics than that typically employed by pessimistic transactions by guaranteeing run once execution and safe encapsulation of the privatisation and publication idioms. We describe our mixed mode transactional programming language by giving a small step operational semantics. Using our semantics and their derived schedules of actions (reads and writes) we show that conflicting transactions (atomics) and gatomics are serialisable. We then go on to define schedules of actions in the form of Java's memory model (JMM) and show that the same properties that held under our restrictive memory model also hold under our modified JMM.

1 Introduction

Software transactional memory (STM) [27] has gained considerable traction in recent years and has subsequently been adopted by a number of languages [10, 13]. STM is only an *alternative* to locks. One cannot safely substitute every occurrence of a lock with a transaction and guarantee the original program semantics. This is mainly due to the optimistic concurrency control traditionally employed by STMs. For example, one cannot optimistically execute an irreversible operation and still guarantee consistency. Similarly, optimistic concurrency control is not ideal for executing expensive operations, catering for "hot" regions of memory [29], or for systems with finite resources [4].

One approach of addressing these issues is to permit pessimistic and optimistic transactions to co-exist. Previous literature such as that by McCloskey et al. [21], Ziarek et al. [32], Ni et al. [23], Welc et al. [31] and Sonmez et al. [29] have investigated such approaches, with each providing a different take on how and why pessimism should be introduced into systems already exposing optimistic STM. However, each lack a formal underpinning when addressing

two problems: *when* pessimistic semantics are necessary due to the semantics of the operations being performed, and *how* pessimistic and optimistic modes of concurrency control safely co-exist. The closest work on providing a formal foundation for pessimistic and optimistic transactions was by Koskinen et al. [16] but they treat each in isolation. Other work also exists on the semantics of STM such as that by Abadi et al. [1] but again does not provide a model of co-existence for transactions of differing concurrency control semantics.

The focus of this paper is on presenting an operational semantics for a programming language that supports both optimistic and pessimistic transactions. We partition transactional mediation of accesses to memory into two types: transactions (atomic) which are optimistic, and *guaranteed transactions* (gatomic) which are pessimistic. Atomics under our system have the following properties (in addition to the ACI properties [8]): (i) *word-based*: conflicts are detected at the granularity of memory locations; (ii) *out-of-place*: atomics operate upon a thread-local copy of their dataset which is only observed by other threads should the atomic commit; (iii) *optimistic*: a contention manager [12] determines, at the point when all constituent commands of an atomic have been executed, whether or not the atomic should commit or abort; and (iv) *weakly isolated* [6, 7]: atomic accesses are only isolated w.r.t. other atomic and gatomic accesses. The best comparison of a gatomic w.r.t. the current literature is that a gatomic is an obstinate transaction [23] but is guaranteed to never abort, either prior to, or during its execution, and infers a stronger notion of its dataset. Multiple gatomics can run at any given time provided consistency invariants are maintained, unlike single owner read locks presented by Welc et al. [31]. A constituent operation of a gatomic is guaranteed to only ever run once. Furthermore, gatomics offer a sensible and intuitive encapsulation of the privatisation and publication idioms which are erroneous under some STMs [30]. In addition to being word-based and out-of-place, gatomics entail the following properties: (i) *pessimistic*: contention is resolved at the point of execution; and (ii) *dataset inference*: the transitive closure of reachable objects from those referenced within the gatomic form the dataset of the gatomic. Atomics and gatomics can be freely composed w.r.t. one another.

The algorithm in Fig. 1 uses a gatomic to privatise a list suffix to the invoking thread. Under an atomic semantics the privatisation of the list suffix may not be consistent [30]. Executing `privatiseListSuffix` under a gatomic semantics always maintains memory consistency. For example, given two threads T_1 and T_2, where T_1 and T_2 invoke `list.privatiseListSuffix(5)` and resp. `list.privatiseListSuffix(3)` on a shared list object `list` (a singly linked list) which comprises of the values [1..10], we have either T_1 and T_2 printing [5..10] and resp. [3, 4], or [3..10] and resp. [ ]. Under other STMs [30] this example would require programmer specified logic to explicitly transfer ownership of heap locations to the invoking thread [30], however under a gatomic semantics this process is managed entirely by the underlying system.

Our guiding philosophy can be summarised as follows: optimistic concurrency control (atomic) should be used in *most* cases, but for operations that

```
class LinkedList {
  // ...
  gatomic privatiseListSuffix(Value v) {
    prev := head;
    curr := prev.next;
    while (curr.value != v) {
      prev := curr; curr := curr.next;
      if (curr == null) goto 9;
    }
    prev.next = null;
    while (curr != null) {
      print(curr.value);
      curr = curr.next;
    } }
  // ...
}
```

Fig. 1. Gatomics guarantee the safety of privatising operations.

access highly contended memory, execute irreversible operations, demand run once semantics or perform expensive computations, then *on-occasion* pessimistic concurrency control (gatomic) may be preferable [26].

To summarise, we make the following contributions:

- We give a small-step operational semantics (Sect. 2) for an object-oriented programming language that supports atomics and gatomics.
- We define legal schedules of reads and writes issued by atomics and gatomics, first in the form of sequential consistency (SC, Sect. 2.5), and then as part of a modified definition of the Java memory model (JMM, Sect.3).
- We show that the actions issued by conflicting atomics and gatomics are serialisable both under SC and the JMM. (Sects. 2.6 and 3.)

2 Programming Model

2.1 Programming Language

We present a minimal object-oriented language that supports atomics and gatomics for mediating accesses to memory locations. Atomic and gatomic regions of code can be defined at the granularity of a method or be explicitly scoped.

prog ::= cdecl* $(t\ v)^*$ (S || ... || S)
cdecl ::= class cn { $(t\ v)^*$ meth* }
t ::= cn | primitive
meth ::= [atomic | gatomic] $t\ m((t\ p)^*)$ {C}
S ::= $(t\ v)^*$ C
C ::= $v := x$ | $v.f := x$ | $v.m(p^*)$ | atomic{C} | gatomic{C} | C;C

Where v, p and x range over variables, t over types and f over the fields defined by the variable's type. Notable features of our language include the use of atomics ($\mathsf{atomic}\{C\}$) and gatomics ($\mathsf{gatomic}\{C\}$) as commands. Classes, methods and method calls are also permitted. Methods can be defined to execute under an atomic, gatomic or non-synchronised semantics. For simplicity of presentation, the above language does not include conditional commands and while loops. Conditional commands cause no extra difficulty in our semantic definitions. While loops can be dealt with via their corresponding tail-recursive methods.

2.2 Program Text Preprocessing

Each invocation of an atomic or gatomic method is wrapped with the synchronisation action (SA, either an atomic or gatomic) defined by the method's signature. A method invocation $v.m(p^*)$ is transformed into $\mathsf{atomic}\{v.m(p^*)\}$ if m is defined as an atomic method. Similarly, the invocation $v.m(p^*)$ is transformed into $\mathsf{gatomic}\{v.m(p^*)\}$ if m is defined to execute under a gatomic semantics.

Nested SAs are flattened by applying the following sequence of phases:

1. *Discovery:* determines the strongest semantics used within the nested SAs. The semantics afforded by a gatomic are stronger than that of an atomic.
2. *Semantic Boosting:* uses the semantics yielded by the discovery phase as the new semantics of the outermost SA.
3. *Flattening:* removes the nested SA semantics from the outermost SA resulting in a single monolithic SA.

To illustrate the use of these phases we now apply them to the command $\mathsf{atomic}\{c_i; \mathsf{gatomic}\{c_j\}\}$:

- *Discovery:* the gatomic is selected as its semantics are stronger than that afforded by an atomic.
- *Semantic Boosting:* the outermost SA (atomic) adopts the semantics identified by the discovery phase resulting in $\mathsf{gatomic}\{c_i; \mathsf{gatomic}\{c_j\}\}$.
- *Flattening:* the nested gatomic is removed resulting in $\mathsf{gatomic}\{c_i; c_j\}$.

The last stage of our preprocessing is to associate a unique identifier, id, with each instance of an atomic and gatomic.

2.3 Preliminaries

State Our model of state is defined as follows:

$$\mathsf{s} \in \mathsf{Stores} \stackrel{\mathsf{def}}{=} \mathsf{Variables} \to \mathsf{Addresses}$$

$$\mathsf{h} \in \mathsf{Heaps} \stackrel{\mathsf{def}}{=} \mathsf{Addresses} \rightharpoonup_{\mathrm{fin}} \mathsf{ObjVal}$$

$$\mathsf{Variables} \stackrel{\mathsf{def}}{=} \{x, y, ...\} \qquad \mathsf{Addresses} \stackrel{\mathsf{def}}{=} \{0, 1, 2, ...\}$$

$$(\mathsf{s}, \mathsf{h}) \in \mathsf{States} \stackrel{\mathsf{def}}{=} \mathsf{Stores} \times \mathsf{Heaps}$$

Note that s is a function that maps a variable to its address on the heap. A heap h is a partial function that maps an address to an object value $cn[f_1 \mapsto \nu_1, \ldots, f_n \mapsto \nu_n]$ where cn is the type of the object and $\nu_1 \ldots \nu_n$ are the values of the fields $f_1 \ldots f_n$. We use $\mathsf{h}(\mathsf{s}(x)).f$ to access the value of a field f of the object pointed-to by x. We often use σ or δ to denote a state, (s, h).

Configurations Given a parallel composition $C_1 \| \ldots \| C_n$, the machine configuration P is of the form $\langle T_1 \| \ldots \| T_n, \sigma, \Gamma \rangle$, where $T_1, .., T_n$ are thread configurations, σ is the program state, and Γ records all current instances of SAs.

A thread configuration T is of the form $\langle \tau, C_\tau, \mathsf{s}_\tau, \delta \rangle$, where $\tau \in \mathcal{T} = \{1, .., n\}$ is the thread identifier, C_τ is the command to execute, s_τ is the thread's store mapping, and δ is a copy of the program state. δ entails the thread store and the program state.

Every active SA instance is associated with some metadata of the form $(\mathsf{beg}, \mathsf{cmt}, \mathsf{rs}, \mathsf{ws}, \mathsf{ds}, \mathsf{type})$, where beg and cmt are time stamps representing begin and commit times, rs and ws are read and write sets (sets of addresses), $\mathsf{ds} \stackrel{\mathsf{def}}{=} \mathsf{rs} \cup \mathsf{ws}$ is the dataset, and type is the type of the SA instance, defined as Ψ if the SA is a gatomic or undefined ($\perp$) otherwise. Each component of an SA entry is initially $\perp$. Γ in the program configuration is a mapping that takes a thread identifier and an SA label and returns the metadata associated with the specified SA instance. Metadata facilitates the safe execution of SAs; specifically, the checking of which SAs conflict and which do not. Given two distinct SAs x and y, x and y conflict if x's write set (ws) intersected with y's dataset (ds) yields a non-empty set.

Transition relations We model the execution of each thread command using the following transition relation: $T, \sigma, \Gamma \xrightarrow{\lambda} T', \sigma', \Gamma'$, as in Fig. 3. To facilitate the presentation, we also define a set of auxiliary rules of the form $C, \sigma, \gamma_R, \gamma_W \xrightarrow{\lambda_\mathsf{R} \mid \lambda_\mathsf{W}} C', \sigma', \gamma'_R, ' \gamma'_W$ in Fig. 2. Where γ_R and γ_W are incrementally accumulated read and write sets. A transition in the auxiliary rules generates a sequence of reads (λ_R) and/or writes (λ_W). An action that is generated due to a reduction in either the thread or auxiliary rules is termed a *fine-grained action*. λ is used as a metavariable that ranges over fine-grained actions. We model the execution of a parallel composition (Fig. 5) using the following transition relation $P \xrightarrow{\|_{\mathsf{mv} \in I \cup M} \Lambda_\mathsf{mv}} P'$, where P, P' are of the form $\langle T_1 \| \ldots \| T_n, \sigma, \Gamma \rangle$ as defined earlier. The label $\|_{\mathsf{mv} \in I \cup M}\ \Lambda_\mathsf{mv}$ denotes that during the transition from P to P', all actions Λ_mv ($\mathsf{mv} \in I \cup M$) take place concurrently in some arbitrary order. Each action Λ_mv entails a sequence of finer-grained actions, λ, either of the form $\mathsf{tbeg}^\frown \lambda^* {}^\frown \mathsf{tcmt}$ (for atomics) or $\mathsf{gbeg}^\frown \lambda^* {}^\frown \mathsf{gcmt}$ (for gatomics). Where λ^* is a sequence of reads and/or writes.

2.4 Thread Command Semantics

We extend the commands of our language to include the following intermediate constructs to facilitate the presentation of the semantics:

$$\text{C} ::= \ \ldots \mid \mathsf{ablk}(\text{C, C}) \mid \mathsf{gablk}(\text{C}) \mid \mathsf{ret}(\{v_1, \ldots, v_n\}, \text{C})$$

$$\frac{\mathsf{s}' = \mathsf{s}[v \mapsto \mathsf{s}(x)] \quad \gamma'_R = \gamma_R \cup \{x\} \quad \gamma'_W = \gamma_W \cup \{v\}}{v{:=}x, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda_R \lambda_W} \epsilon, (\mathsf{s}',\mathsf{h}), \gamma'_R, \gamma'_W} \; [\textbf{ASSIGN}]$$

$$\frac{\mathsf{h}' = \mathsf{h}[\mathsf{s}(v) \mapsto (\mathsf{h}(\mathsf{s}(v))[f \mapsto \mathsf{s}(x)])] \quad \gamma'_R = \gamma_R \cup \{x\} \quad \gamma'_W = \gamma_W \cup \{\mathsf{s}(v)\}}{v.f{:=}x, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda_R \lambda_W} \epsilon, (\mathsf{s},\mathsf{h}'), \gamma'_R, \gamma'_W} \; [\textbf{FLD\_UPDATE}]$$

$$\frac{t\ mn(t_1\ p_1, .., t_n\ p_n)\ \{c\} \in \mathsf{methods}(cn) \quad \mathsf{s}_0 = [\mathsf{this} \mapsto \mathsf{s}_\tau(u_0), p_1 \mapsto \mathsf{s}_\tau(u_1), .., p_n \mapsto \mathsf{s}_\tau(u_n)] \quad \mathsf{s}' = \mathsf{push\_frame}(\mathsf{s}_0, \mathsf{s}) \quad \gamma'_R = \gamma_R \cup \{u_1, \ldots, u_n\}}{u_0.mn(u_1..u_n), (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda_{R1} \ldots \lambda_{Rn}} \mathsf{ret}(\{\mathsf{this}, p_1..p_n\}, c), (\mathsf{s}',\mathsf{h}), \gamma'_R, \gamma_W} \; [\textbf{INV1}]$$

$$\frac{c, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda} c', (\mathsf{s}',\mathsf{h}'), \gamma'_R, \gamma'_W}{\mathsf{ret}(V, c), (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda} \mathsf{ret}(V, c'), (\mathsf{s}',\mathsf{h}'), \gamma'_R, \gamma'_W} \; [\textbf{INV2}]$$

$$\frac{\mathsf{s}' = \mathsf{pop\_frame}(V, \mathsf{s})}{\mathsf{ret}(V, \epsilon), (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\epsilon} \epsilon, (\mathsf{s}',\mathsf{h}), \gamma_R, \gamma_W} \; [\textbf{INV3}]$$

$$\frac{c_1, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda} c'_1, (\mathsf{s}',\mathsf{h}'), \gamma'_R, \gamma'_W}{c_1; c_2, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda} c'_1; c_2, (\mathsf{s}',\mathsf{h}'), \gamma'_R, \gamma'_W} \; [\textbf{SEQ1}]$$

$$\frac{c_1, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda} \epsilon, (\mathsf{s}',\mathsf{h}'), \gamma'_R, \gamma'_W}{c_1; c_2, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda} c_2, (\mathsf{s}',\mathsf{h}'), \gamma'_R, \gamma'_W} \; [\textbf{SEQ2}]$$

Fig. 2. Auxiliary Sequential Rules instrumented with Datasets.

where ablk(C, C) and gablk(C) are intermediate representations of atomic$\{C\}$ and resp. gatomic$\{C\}$ within the program source text. The second component of ablk is the backup program command and is used as the point to rollback to should the atomic abort. The construct ret($\{p_1, \ldots, p_n\}$, C) is used as a mark when executing methods.

The rest of this section covers the operational semantics of our language. During our commentary we give only brief descriptions of the auxiliary functions our rules reference. We refer the reader to Fig. 4 for their formal definitions.

Auxiliary rules [ASSIGN] and [FLD_UPDATE] (Fig. 2) are employed when executing either an assignment or field update within an SA. Each command registers the memory locations it reads and writes and stores them in its read (γ_R) and/or resp. write (γ_W) set. The read and write sets of a command aid conflict detection of atomics and gatomics (see Sects. 2.4 and 2.4). The rules [INV1], [INV2] and [INV3] facilitate method calls. [INV1] is applied when invoking a method. Note that the store s is viewed as a "stackable" mapping, where a variable p may occur several times, and s(p) always refers to the value of the variables p that were pushed in most recently. We use the operation push_frame(s_0, s) to "push" the frame s_0 to s, push_frame($[p \mapsto \nu]$, s)(p) = ν. [INV2] is applied when executing the constituent commands of a method and [INV3] is applied when a method completes. In [INV3] pop_frame(V, s) is used to "pop out" the variables

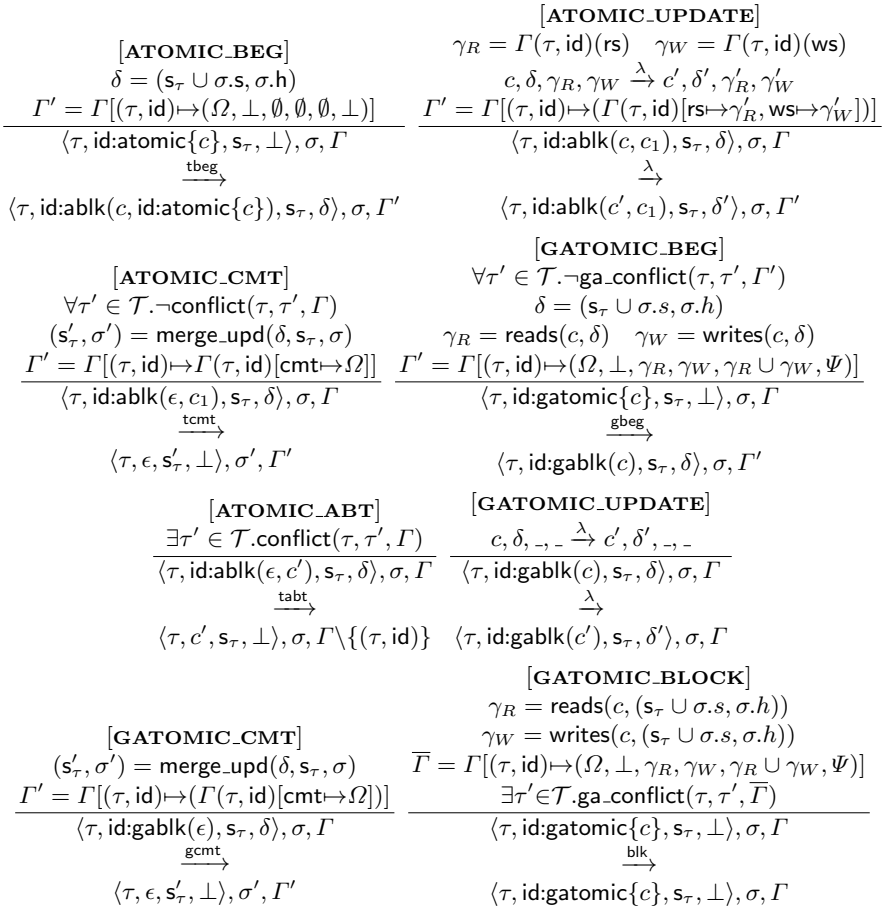

Fig. 3. Synchronisation Action Command Semantics.

in V from the stack s, and $\mathsf{s}[p\mapsto\nu]$ changes the value of the most recent p in stack s to ν. The ϵ in [SEQ2] denotes an empty command.

Transactions [ATOMIC_BEG] (Fig. 3) is applied when an $\mathsf{atomic}\{\mathrm{C}\}$ is encountered in the source text. An atomic is relatively simple to setup: a local copy of the state, δ, is made which comprises of a store heap pair where the store entails both the thread-local and global store mappings and some default metadata is associated with the SA instance. We use Ω as a meta-timestamp that returns the current time. [ATOMIC_UPDATE] is applied for each constituent command within an atomic. When all the constituent commands of an atomic have been executed either [ATOMIC_CMT] or [ATOMIC_ABT] is applied. [ATOMIC_CMT] applies should the dataset of the atomic not conflict with any other running or recently committed SA. Committing an atomic entails the merging of its effect

$$\mathsf{merge}(\sigma_1,\sigma_2) \stackrel{\mathsf{def}}{=} (\mathsf{mergefun}(\sigma_1.\mathsf{s},\sigma_2.\mathsf{s}),\mathsf{mergefun}(\sigma_1.\mathsf{h},\sigma_2.\mathsf{h}))$$

$$\mathsf{mergefun}(f_1,f_2)(x) \stackrel{\mathsf{def}}{=} \begin{cases} f_2(x) & x \in \mathsf{dom}(f_2) \\ f_1(x) & x \in \mathsf{dom}(f_1) \backslash \mathsf{dom}(f_2) \end{cases}$$

$$\mathsf{merge\_sets}(\sigma,\{\sigma_1,..,\sigma_n\}) \stackrel{\mathsf{def}}{=} \begin{cases} \mathsf{merge}(\sigma,\sigma_1) & n=1 \\ \mathsf{merge\_sets}(\mathsf{merge}(\sigma,\sigma_1),\{\sigma_2,..,\sigma_n\}) & n \geq 2 \end{cases}$$

$$\mathsf{merge\_upd}((\mathsf{s}_1,\mathsf{h}_1),\mathsf{s},\sigma) \stackrel{\mathsf{def}}{=} (\mathsf{mergefun}(\mathsf{s} \cup \sigma.\mathsf{s},\mathsf{s}_1),\mathsf{mergefun}(\sigma.\mathsf{h},\mathsf{h}_1))$$

$$\begin{aligned} \mathsf{conflict}(\tau_1,\tau_2,\Gamma) \stackrel{\mathsf{def}}{=}\ & \tau_1 \neq \tau_2 \wedge \exists \mathsf{id}_1,\mathsf{id}_2 \in \mathsf{Labels}.\Gamma(\tau_1,\mathsf{id}_1).\mathsf{ws} \cap \Gamma(\tau_2,\mathsf{id}_2).\mathsf{ds} \neq \emptyset \\ & \wedge\ (\Gamma(\tau_1,\mathsf{id}_1).\mathsf{beg} \leq \Gamma(\tau_2,\mathsf{id}_2).\mathsf{cmt} \leq \Omega \\ & \quad \vee\ \Gamma(\tau_2,\mathsf{id}_2).\mathsf{type} = \Psi \wedge \Gamma(\tau_2,\mathsf{id}_2).\mathsf{cmt} = \bot) \end{aligned}$$

$$\begin{aligned} \mathsf{ga\_conflict}(\tau_1,\tau_2,\Gamma) \stackrel{\mathsf{def}}{=}\ & \tau_1 \neq \tau_2 \wedge \exists \mathsf{id}_1,\mathsf{id}_2 \in \mathsf{Labels}.\Gamma(\tau_1,\mathsf{id}_1).\mathsf{type} = \Psi \\ & \wedge \Gamma(\tau_2,\mathsf{id}_2).\mathsf{type} = \Psi \\ & \wedge \Gamma(\tau_1,\mathsf{id}_1).\mathsf{ws} \cap \Gamma(\tau_2,\mathsf{id}_2).\mathsf{ds} \neq \emptyset \wedge \Gamma(\tau_2,\mathsf{id}_2).\mathsf{cmt} = \bot \end{aligned}$$

$$\begin{aligned} \mathsf{T}_i,\sigma,\Gamma\ (\xrightarrow{\lambda_1} \circ \xrightarrow{\lambda_2})\ \mathsf{T}_i',\sigma',\Gamma' \stackrel{\mathsf{def}}{=}\ & \exists \mathsf{T}_i'',\sigma'',\Gamma'' \cdot \mathsf{T}_i,\sigma,\Gamma \xrightarrow{\lambda_1} \mathsf{T}_i'',\sigma'',\Gamma'' \\ & \wedge\ \mathsf{T}_i'',\sigma'',\Gamma'' \xrightarrow{\lambda_2} \mathsf{T}_i',\sigma',\Gamma' \end{aligned}$$

$$(\xrightarrow{\lambda})^* \stackrel{\mathsf{def}}{=} \bigcup_{r \geq 1} (\xrightarrow{\lambda})^r \qquad (\xrightarrow{\lambda})^{r+1} \stackrel{\mathsf{def}}{=} (\xrightarrow{\lambda}) \circ (\xrightarrow{\lambda})^r$$

$$\mathsf{merge\_sa}(\Gamma,\{\Gamma^1,..,\Gamma^n\}) \stackrel{\mathsf{def}}{=} \Gamma \cup \bigcup_{1 \leq i \leq n} (\Gamma^i - \Gamma)$$

Fig. 4. Auxiliary Definitions.

(δ) into s_τ and σ via the function merge_upd, and the updating of its SA entry. The dataset of an atomic is validated only when all its constituent commands have been executed. [ATOMIC_ABT] is applied if the atomic's dataset has been invalidated due to a conflict with another running or recently committed SA. Aborting an atomic is trivial: its SA entry is removed from Γ and the program counter of τ is set to c'. The predicate conflict in Fig. 4 determines whether or not an atomic conflicts with another SA. Informally, if a conflicting atomic or gatomic committed after or at the same time as the atomic under investigation began then the atomic is aborted.

Guaranteed Transactions [GATOMIC_BEG] is applied when gatomic{C} is encountered in the program source text. Due to the run once semantics of gatomics [GATOMIC_BEG] performs a check to see if the gatomic conflicts with any other currently running gatomic. Should a conflict exist then the gatomic cannot be immediately scheduled to run and [GATOMIC_BLOCK] is applied; otherwise, it begins execution. The functions reads and writes return the transitive closure of all locations reachable by the objects referenced within the gatomic that are read and resp. written. We resort to existing analyses (e.g., [5, 15, 20, 25]) to compute this information. The predicate ga_conflict (Fig. 4) encapsulates the pessimistic scheduling check that gatomics entail. Conceptually gatomics

use two-phase locking: locks associated with the referenced objects within the gatomic are acquired before entering the gatomic and only released upon the completion of the gatomic. The temporary mapping $\overline{\Gamma}$ in [GATOMIC_BLOCK] is used to determine if the gatomic conflicts with any other running gatomic. [GATOMIC_UPDATE] is applied per each constituent command executed by the gatomic. The commands that a gatomic executes are non-instrumented versions of the commands presented in Fig. 2. [GATOMIC_CMT] is applied when the gatomic has executed all of its constituent commands.

Note that in Fig. 3, we present only semantics for atomics and gatomics, and ignore semantics for other commands (which are straightforward to define).

$$
\frac{
\begin{array}{c}
[\textbf{PCOMP}] \\
\textbf{1}\ \mathcal{T} = I \cup J \cup K \cup M \quad I \cup M \neq \emptyset \\
\textbf{2}\left\{\begin{array}{c}
\forall i \in I \cdot \mathsf{T}_i{=}\langle i, \mathsf{id}_i{:}\mathsf{gatomic}\{c_i\}; c_i', \mathsf{s}_i, \bot\rangle \wedge \mathsf{T}_i'{=}\langle i, c_i', \mathsf{s}_i', \bot\rangle \wedge \mathsf{T}_i''{=}\langle i, \mathsf{id}_i{:}\mathsf{gablk}(\epsilon); c_i', \mathsf{s}_i, \delta_i\rangle \\
\forall j \in J \cdot \mathsf{T}_j{=}\langle j, \mathsf{id}_j{:}\mathsf{gatomic}\{c_j\}; c_j', \mathsf{s}_j, \bot\rangle \\
\forall k \in K \cdot \mathsf{T}_k{=}\langle k, \mathsf{id}_k{:}\mathsf{atomic}\{c_k\}; c_k', \mathsf{s}_k, \bot\rangle \wedge \mathsf{T}_k''{=}\langle k, \mathsf{id}_k{:}\mathsf{ablk}(\epsilon, c_k); c_k', \mathsf{s}_k, \delta_k\rangle \\
\forall m \in M \cdot \mathsf{T}_m{=}\langle m, \mathsf{id}_m{:}\mathsf{atomic}\{c_m\}; c_m', \mathsf{s}_m, \bot\rangle \ \wedge\ \mathsf{T}_m' = \langle m, c_m', \mathsf{s}_m', \bot\rangle \\
\wedge\ \mathsf{T}_m'' = \langle m, \mathsf{id}_m{:}\mathsf{ablk}(\epsilon, c_m); c_m', \mathsf{s}_m, \delta_m\rangle
\end{array}\right. \\
\textbf{3}\quad \forall i, j \in I \cdot i{\neq}j \Rightarrow \Gamma(i, \mathsf{id}_i)(\mathsf{ws}) \cap \Gamma(j, \mathsf{id}_j)(\mathsf{ds}) = \emptyset \\
\textbf{4}\quad \forall j \in J \cdot (\exists i \in I \cdot \Gamma(i, \mathsf{id}_i)(\mathsf{ws}) \cap \Gamma(j, \mathsf{id}_j)(\mathsf{ds}) \neq \emptyset) \\
\textbf{5}\left\{\begin{array}{c}
\forall i \in I \cdot \mathsf{T}_i, \sigma, \Gamma \xrightarrow{\mathsf{gbeg}} \circ(\xrightarrow{\lambda})^*\ \mathsf{T}_i'', \sigma_i', \Gamma_i' \xrightarrow{\mathsf{gcmt}} \mathsf{T}_i', \sigma_i, \Gamma_i \quad \wedge \quad \Lambda_i = \mathsf{gbeg}^\frown\lambda^*{}^\frown\mathsf{gcmt} \\
\forall k \in K \cdot \mathsf{T}_k, \sigma, \Gamma \xrightarrow{\mathsf{tbeg}} \circ(\xrightarrow{\lambda})^*\ \mathsf{T}_k'', \sigma, \Gamma_k' \xrightarrow{\mathsf{tabt}} \mathsf{T}_k, \sigma, \Gamma \\
\forall m \in M \cdot \mathsf{T}_m, \sigma, \Gamma \xrightarrow{\mathsf{tbeg}} \circ(\xrightarrow{\lambda})^*\ \mathsf{T}_m'', \sigma, \Gamma_m' \xrightarrow{\mathsf{tcmt}} \mathsf{T}_m', \sigma_m, \Gamma_m \\
\wedge \quad \Lambda_m = \mathsf{tbeg}^\frown\lambda^*{}^\frown\mathsf{tcmt}
\end{array}\right. \\
\textbf{6}\left\{\begin{array}{c}
\forall k \in K \cdot (\exists i \in I . \Gamma_k'(k, \mathsf{id}_k)(\mathsf{ws}) \cap \Gamma(i, \mathsf{id}_i)(\mathsf{ds}) \neq \emptyset \\
\vee\ \exists m \in M . \Gamma_k'(k, \mathsf{id}_k)(\mathsf{ws}) \cap \Gamma_m'(m, \mathsf{id}_m)(\mathsf{ds}) \neq \emptyset)
\end{array}\right. \\
\textbf{7}\left\{\begin{array}{c}
\forall m \in M \cdot (\forall i \in I . \Gamma_m'(m, \mathsf{id}_m)(\mathsf{ws}) \cap \Gamma(i, \mathsf{id}_i)(\mathsf{ds}) = \emptyset \ \wedge \\
\forall m' \in M \backslash \{m\} . \neg\exists \mathsf{id}_{m'} . \Gamma_m'(m, \mathsf{id}_m)(\mathsf{ws}) \cap \Gamma_{m'}'(m', \mathsf{id}_{m'})(\mathsf{ds}) \neq \emptyset)
\end{array}\right. \\
\textbf{8}\ \sigma' = \mathsf{merge\_sets}(\sigma, \{\sigma_\tau \mid \tau \in I \cup M\}) \qquad \Gamma' = \mathsf{merge\_sa}(\Gamma, \{\Gamma_\tau \mid \tau \in I \cup M\})\ \textbf{9}
\end{array}
}{
\langle \ldots || \mathsf{T}_i || .. || \mathsf{T}_j || .. || \mathsf{T}_k || .. || \mathsf{T}_m || .., \sigma, \Gamma\rangle \xrightarrow{\|_{\mathsf{mv} \in I \cup M} \Lambda_{\mathsf{mv}}} \langle .. || \mathsf{T}_i' || .. || \mathsf{T}_j || .. || \mathsf{T}_k || .. || \mathsf{T}_m' || .., \sigma', \Gamma'\rangle
}
$$

Fig. 5. Big-Step Program Move Semantics.

2.5 Program Move Semantics

The orchestration of concurrently executing threads is handled by the rule [PCOMP] (Fig. 5). [PCOMP] caters for the most interesting scenario where each thread has an atomic or gatomic to run (Other scenarios are not included). It distinguishes the set of threads which make progress in their respective transition systems (*moving* threads) from those that do not (*non-moving* threads). Moving threads are executing either atomics that are to be committed or gatomics that are safe to run. To facilitate our reasoning we assume the set of concurrently executing threads $\mathcal{T}$ are split into four sets (Label 1 in Fig. 5):

- I is the set of threads that are executing code under a gatomic semantics. Every thread in I has satisfied the predicate $\neg$ga_conflict for its respective gatomic.
- J represents the set of threads that are currently blocking due to their resp. gatomics conflicting with some threads already running in I.
- M is the set of threads that can commit their atomics.
- K is the set of threads whose atomics are to be aborted.

Label 2 in Fig. 5 defines the configurations that each of the partitioned threads in I, J, M and K move through. Label 3 requires that each of the threads in I running a gatomic do not conflict w.r.t. each other. Label 4 denotes that the gatomics in threads J are currently blocking due to them conflicting with some threads currently running in I. Label 5 illustrates the transitions undertaken by each of the threads in I, J, M and K. Threads in I apply an instance of [GATOMIC_BEG], a number of [GATOMIC_UPDATE] instances and an instance of [GATOMIC_CMT]. The blocking threads in J apply an instance of [GATOMIC_BLOCK] and as such do not make any progress in their resp. transition systems. Threads in K and M differ only in their final reduction: threads in K apply instances of [ATOMIC_ABT] and those in M apply instances of [ATOMIC_CMT]. Label 6 states that the threads in K are due to abort if they conflict with either a gatomic or a committing atomic. Label 7 requires that the committing atomics do not conflict with any gatomics nor any other committing atomic. Moving threads merge their entailed effects with the program state via merge_sets (Label 8) and also merge the updates made to their respective SA entries courtesy of merge_sa (Label 9).

Each SA being executed by the threads in I, J, M and K generates a sequence of fine grained actions, λs, due to the reductions taken in their resp. transition systems. The sequence of fine grained actions generated by each SA form an action, Λ. Λ_{mv} is used to denote the actions associated with the SAs being executed by the moving threads in I and M. The fine grained actions (λs) of the actions in Λ_{mv} can be arbitrarily concurrently interleaved due to the SAs executed by threads in I and M not conflicting. This interleaving is denoted in the reduction of [PCOMP]. The resulting interleaving is governed by the same restrictions imposed by sequential consistency (SC) [17].

2.6 Properties

We show that the semantics of atomics and gatomics given in Figs. 3 and 5 are serialisable [24] and that correctly isolated programs are data-race-free.

Definition 1. *Ordered-Before ($<$). Defined over actions (Λs). Total ordering. If each fine grained action λ_i of Λ_i takes effect before the first fine grained action λ_j of Λ_j, then $\Lambda_i < \Lambda_j$.*

Intuitively if $\Lambda_i < \Lambda_j$ then we say that the effect of Λ_i *serialises-before* Λ_j. The ordered-before relation is constructed during the reduction of [PCOMP]. We show for any given execution that conflicting atomics and gatomics are serialised.

Theorem 1. *There exists a total (serialisable) order over conflicting atomics.*

Proof. Let Λ_i and Λ_j be the actions associated with the conflicting atomics T_i and resp. T_j. By definition of conflict we apply instances of [ATOMIC_CMT] and resp. [ATOMIC_ABT]. Therefore, either $\Lambda_i < \Lambda_j \vee \Lambda_j < \Lambda_i$.

Theorem 2. *There exists a total (serialisable) order over conflicting gatomics.*

Proof. Let Λ_i and Λ_j be the actions associated with the conflicting gatomics T_i and resp. T_j. By definition of ga_conflict we apply instances of [GATOMIC_BEG] and resp. [GATOMIC_BLOCK]. Therefore, either $\Lambda_i < \Lambda_j \vee \Lambda_j < \Lambda_i$.

Theorem 3. *There exists a total (serialisable) order over conflicting SAs of mixed type.*

Proof. Let Λ_i and Λ_j be the actions associated with the conflicting atomic T_i and resp. gatomic T_j. By definition of conflict we apply instances of [ATOMIC_ABT] and resp. [GATOMIC_CMT]. Therefore, $\Lambda_j < \Lambda_i$.

Intuitively gatomics have a serialisation priority over transactions due to a gatomic guaranteeing run once semantics.

A correctly isolated program is one that encapsulates every access to a memory region that is accessed by multiple threads with either an atomic or gatomic.

Theorem 4. *Correctly isolated programs are data-race-free.*

Proof. Follows from Thms. 1, 2 and 3.

3 Java Memory Model

In Sect. 2.5 we presented actions, λs, and described via [PCOMP] (Fig. 5) how these actions were permitted to be interleaved. For each execution this interleaving forms a *schedule*. In the literature a schedule is governed by a memory model [2]. Most importantly a memory model specifies the set of values a read may observe. The schedules of actions constructed in Fig. 5 were due to SC. Under SC actions from all threads appear in a totally ordered sequence, with each action respecting the program order of its issuing thread. The goal of this section is to define legal schedules of actions in terms of the Java memory model (JMM) [19]. In particular, we wish to show how *happens-before* relationships are established between atomics and gatomics. Having defined our SAs under the JMM we show that they satisfy the properties given in Sect. 2.6.

3.1 Correctly Synchronised Programs

The JMM takes a rather simple approach when defining what constitutes a correctly synchronised program, informally: any program that is *data-race-free (DRF)* [3] is guaranteed to observe an SC semantics. Before describing what it means for a program to be DRF we must cover the terminology outlined by the JMM.

- **Conflicting Accesses:** a read or write to a variable x is an *access* of x. Two accesses to x are *conflicting* if at least one of the accesses is a write.
- **Synchronisation Actions:** includes locks, unlocks, reads of volatile variables and writes to volatile variables.
- **Program Order:** the actions issued by a thread τ form a total ordering known as the program order of τ.
- **Synchronisation Order:** every execution is associated with a synchronisation order which is a *total ordering* over all SAs. Only synchronisation orders that are consistent with program order can be considered. For example, a read r of a volatile field v *must* observe the value written to v by the write w such that w occurs before r in the synchronisation order, $w \xrightarrow{so} r$. Every execution is associated with a synchronisation order.
- **Synchronises-With Order:** an unlock action a on a monitor M "synchronises-with" a *subsequent*, as defined by the synchronisation order, lock action b on M, $a \xrightarrow{sw} b$. $\xrightarrow{sw}$ is a partial order. Actions within the synchronises-with order can be issued by different threads.
- **Happens-Before Order:** is the transitive closure of the program order and synchronises-with order. An action a "happens-before" [18] another action b if a occurs before b in the happens-before ordering, written $a \xrightarrow{hb} b$.
- **Data Race:** two accesses a and b to a variable v form a data race if a and b conflict, are issued by separate threads and are not ordered by the happens-before relation.
- **Data-Race-Free Program:** a program is DRF if and only if all sequentially consistent executions of the program are free of data races.

The happens-before relation in the JMM defines the set of values a read can observe. Establishing edges in this relation requires the use of an SA. In Java such actions are generated via the use of either `volatile` variables or `synchronized` methods/blocks. As defined by the synchronises-with order there exists a pair of matching unlock and lock actions on the same monitor object M. Conceptually every monitor M is associated with an unlock action on M before any actions of a program are executed. Before we proceed further we must update the definitions of synchronisation actions and the synchronises-with order to be the following:

- **Synchronisation Actions:** includes the beginning of an atomic and gatomic (tbeg and resp. gbeg) and the end of an atomic and gatomic (tcmt and resp. gcmt).
- **Synchronises-With Order:** a tcmt (and resp. gcmt) action a on a dataset d_a "synchronises-with" a *subsequent*, as defined by the synchronisation order, tbeg (and resp. gbeg) action b on a dataset d_b when $d_a \cap d_b \neq \emptyset$, written $a \xrightarrow{sw} b$.

Our begin and end actions for atomics and gatomics are abstractions. We give their semantics in the form of the JMMs acquire and release actions in the next section.

3.2 Execution Semantics

Actions An action $A = \langle \tau, k, v, u \rangle$ where τ is the thread performing the action, k is the kind of action being performed (discussed later), v is the variable involved in the action and $u \in \mathsf{Integers}$ is the unique identifier of the action. The kind k of an action can be one of the following: (i) write (W); (ii) read (R); (iii) acquire (Acq); or (iv) release (Rel). We do not cater for volatiles. To keep things simple we permit a write to be performed directly on a field, e.g. $\langle \tau, \mathsf{W}, v.f, 1 \rangle$. The semantics of tcmt, gbeg and gcmt are defined as follows:

$$\begin{aligned}
\mathsf{tcmt} &\stackrel{\mathsf{def}}{=} \mathsf{Acq}\ \ell_i \ldots \mathsf{Acq}\ \ell_n\ \mathsf{Rel}\ \ell_n \ldots \mathsf{Rel}\ \ell_i \\
\mathsf{gbeg} &\stackrel{\mathsf{def}}{=} \mathsf{Acq}\ \ell_i \ldots \mathsf{Acq}\ \ell_n \\
\mathsf{gcmt} &\stackrel{\mathsf{def}}{=} \mathsf{Rel}\ \ell_n \ldots \mathsf{Rel}\ \ell_i
\end{aligned}$$

Where each ℓ can be a variable or an object value and forms the dataset of an atomic or gatomic. We require that the contention manager has made this schedule safe and that the acquisition and release orders are topologically sorted as in McCloskey et al. [21].[4] blk is ignored (it is a sink action) and tbeg simply acts as a mark to delimit the beginning of an atomic's constituent actions. tabt does not feature in any execution as its subsequence of actions will not be observed. A valid sequence of actions for atomics and gatomics is a composition of the above with a number of reads and writes:

$$\mathsf{tbeg}\ (\mathsf{W} \mid \mathsf{R})^*\ \mathsf{tcmt} \qquad \mathsf{gbeg}\ (\mathsf{W} \mid \mathsf{R})^*\ \mathsf{gcmt}$$

Note that these sequences correspond to those of the moving threads I and M in Fig. 5.

Executions An execution $E = \langle P, \mathsf{A}, \xrightarrow{po}, \xrightarrow{so}, \mathsf{Ws}, \mathsf{Vs}, \xrightarrow{sw}, \xrightarrow{hb} \rangle$ where P is as defined in Sect. 2, A is a set of actions, $\xrightarrow{po}$ is the program order of the actions performed by each $\tau \in \mathcal{T}$, $\xrightarrow{so}$ is the synchronisation order, Ws is a write-seen function, Vs is a value-seen function and $\xrightarrow{sw}$ and $\xrightarrow{hb}$ are as defined previously.

4 Related Work

Shavit and Touitou [27] introduced software transactional memory (STM). The isolation semantics afforded by an STM are either weak or strong [6, 7, 11]. Literature on the semantics of STM includes that by Abadi et al. [1] which is based on the automatic mutual exclusion (AME) [14] concurrent programming language. Koskinen et al. [16] have also studied the semantics of STM but their work does not entail the mixing of pessimistic and optimistic concurrency control.

[4] This model is used only to illustrate a projection onto the JMMs existing SAs.

Ziarek et al. [32] described a dynamic approach for selecting a stronger semantics when an atomic attempted to execute an operation which seems (determined by a magic analysis) to require stronger guarantees than that afforded by an atomic. Unfortunately, such a semantics reverts to using programmer specified lock invariants which are error prone. Smaragdakis et al. [28] presented a set of language extensions to temporarily "suspend" an atomic's isolation in order to support irreversible operations, however they rely heavily on the specification of isolation invariants, which are again, error prone. Privatisation and publication [30] can be used to emulate a stronger semantics within STM but requires the programmer to correctly transfer ownership of memory regions between threads.

Ni et al. [23] championed *obstinate* transactions but are a product of a prior abort. Welc et al. [31] use single owner read locks to transition to a guaranteed semantics but permit only a single such atomic to run at any given time. Sonmez et al. [29] present a model built on Haskell STM that turns atomics that access "hot" regions of memory into pessimistic atomics, however this approach again is dynamic and does not afford dataset guarantees. Autolocker [21] presents a model of pessimistic atomics by using a type system that uses programmer specified lock protection annotations to convert atomics into lock-based equivalents statically. Recent literature such as that by McCloskey et al. [21], Ni et al. [23], Shavit and Matveev [26] and Welc et al. [31] have, via empirical evidence, more than justified not only the practical feasibility of pessimistic concurrency control for STM but also its importance in simplifying the programming model.

Adding atomics to a language such as Java impacts the underlying memory model [2] as outlined by Ziarek et al. [32] and Grossman et al. [9]. Menon et al. [22] provide a number of properties that memory models must take into consideration for supporting atomics, such as the Java memory model (JMM) [19].

5 Summary

We have presented a small-step operational semantics for a programming language that supports compositional mixed mode concurrency control for transactional programs. Our language partitions transactions into two types: atomics (optimistic) and gatomics (pessimistic). Gatomics guarantee run once semantics and the safe use of the privatisation and publication idioms. We also showed that the reads and writes issued by atomics and gatomics are serialisable under both a sequential consistency and Java memory model semantics.

Acknowledgment

Granville Barnett is supported by EPSRC Doctoral Training Award. Shengchao Qin is supported in part by EPSRC project EP/G042322.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. Principles of Programming Languages, 2008.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 1996.

[3] Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. International Symposium on Computer Architecture, 1990.

[4] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: alternatives and implications. *ACM Trans. Database Syst.*, 1987.

[5] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, University of Copenhagen, 1994.

[6] Colin Blundell, E Christopher, Lewis Milo, and M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. Workshop on Duplicating, Deconstructing, and Debunking, 2005.

[7] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. International Symposium on Computer Architecture, 2007.

[8] Jim Gray. The transaction concept: virtues and limitations. Very Large Data Bases, 1981.

[9] Dan Grossman, Jeremy Manson, and William Pugh. What do high-level memory models mean for transactions? Memory System Performance and Correctness, 2006.

[10] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. Principles and Practice of Parallel Programming, 2005.

[11] Tim Harris, James Larus, and Ravi Rajwar. Transactional memory, 2nd edition. 2010.

[12] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. Principles of Distributed Computing, 2003.

[13] Rich Hickey. The clojure programming language. Dynamic Languages Symposium, 2008.

[14] Michael Isard and Andrew Birrell. Automatic mutual exclusion. USENIX, 2007.

[15] James Jenista and Brian Demsky. Disjointness analysis for java-like languages. Technical report, University of California, Irvine, 2009.

[16] Eric Koskinen, Matthew Parkinson, and Maurice Herlihy. Coarse-grained transactions. Principles of Programming Languages, 2010.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Transactions on Computers*, 1979.

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 1978.

[19] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. Principles of Programming Languages, 2005.

[20] Mark Marron, Mario Méndez-Lojo, Manuel Hermenegildo, Darko Stefanovic, and Deepak Kapur. Sharing analysis of arrays, collections, and recursive structures. Program Analysis for Software Tools and Engineering, 2008.

[21] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. Principles of Programming Languages, 2006.

[22] 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. Symposium on Parallelism in Algorithms and Architectures, 2008.

[23] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for c/c++. Object-Oriented Programming Systems Languages and Applications, 2008.

[24] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 1979.

[25] William Pugh and David Wonnacott. Constraint-based array dependence analysis. Transactions on Programming Languages and Systems, 1998.

[26] Nir Shavit and Alex Matveev. Towards a fully pessimistic stm model. Transactional Computing, 2012.

[27] Nir Shavit and Dan Touitou. Software transactional memory. Principles of Distributed Computing, 1995.

[28] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. Object-Oriented Programming Systems Languages and Applications, 2007.

[29] Nehir Sonmez, Tim Harris, Adrian Cristal, Osman S. Unsal, and Mateo Valero. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. International Symposium on Parallel and Distributed Processing, 2009.

[30] Michael F. Spear, Virendra J. Marathe, Luke Daless, and Michael L. Scott. Privatization techniques for software transactional memory. Principles of Distributed Computing, 2007.

[31] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. Symposium on Parallelism in Algorithms and Architectures, 2008.

[32] Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A uniform transactional execution environment for java. European Conference on Object-Oriented Programming, 2008.