Interval Analysis for Concurrent Trace Programs using Transaction Sequence Graphs

Malay K. Ganai and Chao Wang

NEC Labs America, Princeton, NJ, USA

Abstract. Concurrent trace programs (CTPs) are slices of the concurrent programs that generate the concrete program execution traces, where inter-thread event order specific to the given traces are relaxed. For such CTPs, we introduce transaction sequence graph (TSG) as a model for efficient concurrent data flow analysis. The TSG is a digraph of thread-local control nodes and edges corresponding to transactions and possible context-switches. Such a graph captures all the representative interleavings of these nodes/transactions. We use a mutually atomic transaction (MAT) based partial order reduction to construct such a TSG. We also present a non-trivial improvement to the original MAT analysis to further reduce the TSG sizes. As an application, we have used interval analysis in our experiments to show that TSG leads to more precise intervals and more time/space efficient concurrent data flow analysis than the standard models such as concurrent control flow graph.

1 Introduction

Verification of multi-threaded programs is hard due to the complex and often un-expected interleaving between the threads. Exposing concurrency related bugs—such as atomicity violations and data races—require not only bug-triggering inputs but also bug-triggering execution interleavings. Unfortunately, testing a program for every interleaving on every test input is often practically impossible. Runtime-based program analysis [1–13] infer and predict program errors from an observed trace. Compared to static analysis [14–20], runtime analysis often result in fewer false alarms.

Runtime analysis can be broadly classified into three categories: *runtime monitoring*, *runtime prediction*, and *runtime model checking*. In the *first* category, analysis such as [1–6] monitor the observed trace events (such as shared memory accesses) and flag true or potential violations of intended atomic transactions. In the *second* category, the analysis can also predict violations in other interleavings of the events in the observed trace. Some of these approaches [7, 8] use data abstraction, and thereby report false alarms as the interleaving may not be feasible; while other approaches such as [21] use happens-before causal relation to capture only (but may not be all) the feasible interleavings, and thereby, report no bogus (but may miss some true) violations. The *third* category includes more heavy-weight approaches such as dynamic model checking [9–11] and satisfiability-based symbolic analysis [12,13]. These methods search for violations in all feasible alternate interleavings of the observed trace and thereby, report a true violation if and only if one exists.

In dynamic model checking, for a given test input, systematic exploration of a program under all possible thread interleavings is performed. Even though the test input is fixed, explicit enumeration of interleavings can still be quite expensive. Although partial order reduction techniques (POR) [9, 22] reduce the set of necessary interleavings to explore, the reduced set often remains prohibitively large. Some previous work used ad-hoc approaches such as perturbing program execution by injecting artificial delays at every synchronization points [23], or randomized dynamic analysis to increase the chance of detecting real races [24].

In trace-based symbolic analysis [12,13], explicit enumeration is avoided via the use of symbolic encoding and decision procedures to search for violations in a concurrent trace program (CTP) [25]. A CTP corresponds to data and control slice of the concurrent program (unrolled, if there is a thread local loop), and is constructed from both the observed trace and the program source code. One can view a CTP as a *generator* for both the original trace and all the other traces corresponding to feasible interleavings of the events in the original trace.

In this paper, we present a light-weight concurrent data flow analysis which can be used as an efficient preprocessor to reduce the subsequent efforts of the more heavy-weight symbolic analysis for concurrency verification such as [12, 13]. Our primary focus is on a suitable graph representation of CTP to conduct more precise and scalable concurrent data flow analysis than the standard models such as concurrent control flow graph (CCFG). In the sequel, we use interval analysis as an example.

In a nutshell, our approach proceeds as follows: from a given CCFG (corresponding to a CTP), we construct a transaction sequence graph (TSG) denoted as $G(V, E)$ which is a digraph with nodes V representing thread-local control states, and edges E representing either transactions (sequences of thread local transitions) or possible context switches. On the constructed TSG, we conduct an interval analysis for the program variables, which requires $O(|E|)$ iterations of interval updates, each costing $O(|V| \cdot |E|)$ time. Our main contributions are two fold:

– Precise and effective interval analysis using TSG
– Identification and removal of redundant context switches

For construction of TSGs, we leverage our mutually atomic transaction (MAT) analysis [26]—a partial-order based reduction technique that identifies a subset of possible context switches such that *all* and *only* representative schedules are permitted. Using MAT analysis, we first derive a set of so-called *independent transactions.* (As defined later, an independent transaction is globally atomic with respect to a set of schedules.) The beginning and ending control states of each independent transaction form the vertices of a TSG. Each edge of a TSG corresponds to either an independent transaction or a possible context switch between the inter-thread control state pairs (also identified in MAT analysis). Such a TSG is much reduced compared to the corresponding CCFG, where possible context switches occur between every pair of shared memory accesses. Most prior work such as [15–19] apply the analysis directly on CCFGs. In contrast, we conduct interval analysis on TSGs which leads to more precise intervals, and more time/space-efficient analysis than doing on CCFGs.

We improve our original MAT analysis further by reducing the set of possible context switches, and at the same time guarantee that such a reduced set captures all necessary schedules. Such improvement is important because:

– It significantly reduces the size of TSG, both in the number of vertices and in the number of edges; this in turn, results in a more precise interval analysis with improved runtime performance.
– The more precise intervals reduce the size and the search space of decision problems that arise during the more heavy-weight symbolic analysis.

The outline of the rest of the paper is as follows: We provide formal definitions and notations in Section 2. In Section 3, we give an informal overview of our approach, and in Section 4, we present our approach formally. We present our experimental results in Section 5, followed by conclusions, related, and future work in Section 6.

2 Formal Definitions

A multi-threaded concurrent program P comprises a set of threads and a set of shared variables, some of which, such as locks, are used for synchronization. Let M_i $(1 \leq i \leq n)$ denote a thread model represented by a control and data flow graph of the sequential program it executes. Let V_i be a set of local variables in M_i and $\mathcal{V}$ be a set of (global) shared variables. Let $\mathcal{S}$ be the set of global states of the system, where a state $s \in \mathcal{S}$ is valuation of all local and global variables of the system. A global transition system for P is an interleaved composition of the individual thread models, M_i.

A thread transition $t \in \rho$ is a 4-tuple (c, g, u, c') that corresponds to a thread M_i, where c, c' represent the control states of M_i, g is an enabling condition (or *guard*) defined on $V_i \cup \mathcal{V}$, and u is a set of update assignments of the form $v := exp$ where variable v and variables in expression exp belong to the set $V_i \cup \mathcal{V}$. As per interleaving semantics precisely one thread transition is scheduled to execute from a state.

A *schedule* of the concurrent program P is an interleaving sequence of thread transitions $\rho = t_1 \cdots t_k$. In the sequel, we focus only on sequentially consistent [27] schedules. An event e occurs when a unique transition t is fired, which we refer to as the *generator* for that event, and denote it as $t = gen(P, e)$. A *run* (or concrete execution trace) $\sigma = e_1 \cdots e_k$ of a concurrent program P is an ordered sequence of events, where each event e_i corresponds to firing of a unique transition $t_i = gen(P, e_i)$. We illustrate the differences between schedules and runs in Section 3.

Let $begin(t)$ and $end(t)$ denote the beginning and the ending control states of $t = \langle c, g, u, c' \rangle$, respectively. Let $tid(t)$ denote the corresponding thread of the transition t. We assume each transition t is atomic, i.e., uninterruptible, and has at most one shared memory access. Let T_i denote the set of all transitions of M_i.

A *transaction* is an uninterrupted sequence of transitions of a particular thread. For a transaction $tr = t_1 \cdots t_m$, we use $|tr|$ to denote its length, and $tr[i]$ to denote the i^{th} transition for $i \in \{1, \cdots, |tr|\}$. We define $begin(tr)$ and $end(tr)$ as $begin(tr[1])$ and $end(tr[|tr|])$, respectively. In the sequel, we use the notion of *transaction* to denote an uninterrupted sequence of transitions of a thread as *observed* in a system execution.

We say a transaction (of a thread) is *atomic* w.r.t. a schedule, if the corresponding sequence of transitions are executed uninterrupted, i.e., without an interleaving of another thread in-between. For a given set of schedules, if a transaction is atomic w.r.t. all the schedules in the set, we refer to it as an *independent transaction* w.r.t. the set. [1]

Given a run σ for a program P we say e *happens-before* e', denoted as $e \prec_\sigma e'$ if $i < j$, where $\sigma[i] = e$ and $\sigma[j] = e'$, with $\sigma[i]$ denoting the i^{th} access event in σ. Let $t = gen(P, e)$ and $t' = gen(P, e')$. We say $t \prec_\sigma t'$ iff $e \prec_\sigma e'$. We use $e \prec_{po} e'$ and $t \prec_{po} t'$ to denote that the corresponding events and the transitions are in thread

[1] We compare the notion of atomicity used here, vis-a-vis previous works [2, 6, 8]. In our work, the atomicity of transactions corresponds to the observation of the system, which may not correspond to the user intended atomicity of the transactions. Previous work assume that the atomic transactions are system specification that should always be enforced, whereas we infer atomic (or rather independent) transactions from the given system under test, and intend to use them to reduce the search space of symbolic analysis.

program order. We extend the definition of $\prec_{po}$ to thread local control states such that corresponding transitions are in the thread program order.

Reachable-before relation ($\sqsubset$): We say a control state pair (a, b) is reachable-before (a', b'), where each pair corresponds to a pair of threads, represented as $(a, b) \sqsubset (a', b')$ such that one of the following is true: 1) $a \prec_{po} a', b = b'$, 2) $a = a', b \prec_{po} b'$, 3) $a \prec_{po} a', b \prec_{po} b'$.

Dependency Relation ($\mathcal{D}$): Given a set T of transitions, we say a pair of transitions $(t, t') \in T \times T$ is dependent, i.e. $(t, t') \in \mathcal{D}$ iff one of the following holds (a) $t \prec_{po} t'$, (b) (t, t') is conflicting, i.e., accesses are on the same global variable, and at least one of them is a write access. If $(t, t') \notin \mathcal{D}$, we say the pair is *independent*.

Equivalency Relation ($\simeq$): We say two schedules $\rho_1 = t_1 \cdots t_i \cdot t_{i+1} \cdots t_n$ and $\rho_2 = t_1 \cdots t_{i+1} \cdot t_i \cdots t_n$ are equivalent if $(t_i, t_{i+1}) \notin \mathcal{D}$. An equivalent class of schedules can be obtained by iteratively swapping the consecutive independent transitions in a given schedule. A *representative* schedule refers to one of such an equivalent class.

Definition 1 (Concurrent Trace Programs (CTP), Wang 09). *A concurrent trace program with respect to an execution trace $\sigma = e_1 \cdots e_k$ and concurrent program P, denoted as CTP_σ, is a partial ordered set $(T_\sigma, \prec_{\sigma,po})$*

– $T_\sigma = \{t \mid t = gen(P, e)$ where $e \in \sigma\}$ is the set of generator transitions
– $t \prec_{\sigma,po} t'$ iff $t \prec_{po} t' \; \exists \, t, t' \in T_\sigma$

Let $\rho = t_1 \cdots t_k$ be a schedule corresponding to the run σ, where $t_i = gen(P, e_i)$. We say schedule $\rho' = t'_1, \cdots, t'_k$ is an *alternate schedule* of CTP_σ if it is obtained by interleaving transitions of ρ as per $\prec_{\sigma,po}$. We say ρ' is a *feasible schedule* iff there exists a concrete trace $\sigma' = e'_1 \cdots e'_k$ where $t'_i = gen(P, e'_i)$.

We extend the definition of CTP over multiple traces by first defining a *merge* operator [13] that can be applied on two CTPs, CTP_σ and CTP_ψ as: $(T_\tau, \prec_{\tau,po}) \overset{def}{=} merge((T_\sigma, \prec_{\sigma,po}), (T_\psi, \prec_{\psi,po}))$, where $T_\tau = T_\sigma \cup T_\psi$ and $t \prec_{\tau,po} t'$ iff at least one of the following is true: (a) $t \prec_{\sigma,po} t'$ where $t, t' \in T_\sigma$, and (b) $t \prec_{\psi,po} t'$ where $t, t' \in T_\psi$. A merged CTP can be effectively represented as a CCFG with branching structure but no loop. In the sequel, we refer to such a merged CTP as a CTP.

3 Our Approach: An Informal View

In this section, we present our approach informally, where we motivate our readers with an example. We use that example to guide the rest of our discussion. In the later sections, we give a formal exposition of our approach.

Consider a system P comprising interacting threads M_a and M_b with local variables a_i and b_i, respectively, and shared (global) variables X, Y, Z, L. This is shown in Figure 1(a) where threads are synchronized with *Lock/Unlock*. Thread M_b is created and destroyed using fork-join primitives. Figure 1(b) is the lattice representing the complete interleaving space of the program. Each node in the lattice denotes a global control state, shown as a pair of the thread local control states. An edge denotes a shared event write/read access of global variable, labeled with $W(.)/R(.)$ or *Lock(.)/Unlock(.)*. Note, some interleavings are not feasible due to Lock/Unlock, which we crossed out $(\times)$ in the figure. We also labeled all possible context switches with **cs**. The highlighted interleaving corresponds to a concrete execution (run) σ of program P

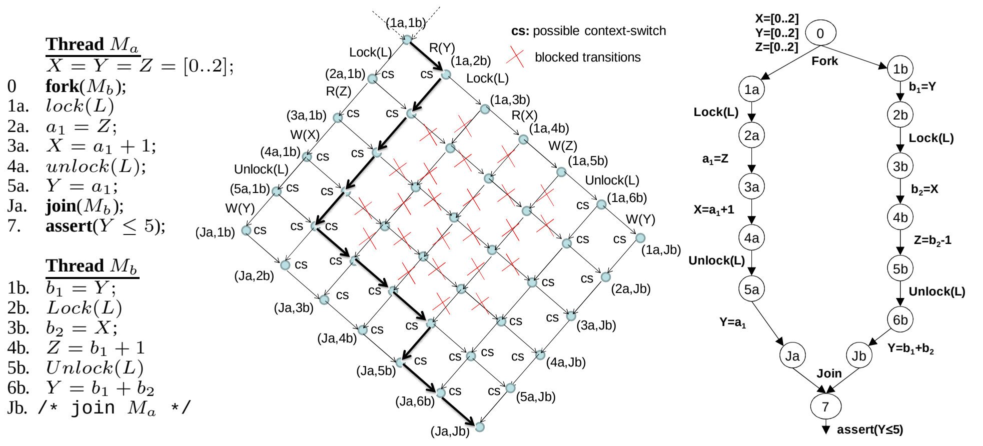
$$\sigma = R(Y)_b \cdot Lock(L)_a \cdots Unlock(L)_a \cdot Lock(L)_b \cdots W(Z)_b \cdot W(Y)_a \cdot Unlock(L)_b \cdot W(Y)_b$$

where the suffices a, b denote the corresponding thread accesses.

A thread transition $(1b, true, b_1 = Y, 2b)$ (also represented as $1b \xrightarrow{b_1=Y} 2b$) is a generator of access event $R(Y)_b$ corresponding to the read access of the shared variable Y. The corresponding schedule ρ of the run σ is

$$\rho = (1b \xrightarrow{b_1=Y} 2b)(1a \xrightarrow{Lock(L)} 2a) \cdots (4a \xrightarrow{Unlock(L)} 5a)(2b \xrightarrow{Lock(L)} 3b) \cdots (6b \xrightarrow{Y=b_1+b_2} Jb)$$

From σ (and ρ), we obtain a slice of the original program called concurrent trace program (CTP) [25]. A CTP can be viewed as a generator of concrete traces, where the inter-thread event order specific to the given trace are relaxed. Figure 1(c) show the CTP_σ of the corresponding run σ shown as a CCFG (This CCFG happens to be the same as P, although it need not be the case). Each node in CCFG denotes a thread control state (and the corresponding thread location), and each edge represents one of the following: thread transition, a context switch, a fork, and a join. To not clutter up the figure, we do not show edges that correspond to possible context switches (30 in total). Such a CCFG captures all the thread schedules of CTP_σ.



```
     Thread M_a
     X = Y = Z = [0..2];
0    fork(M_b);
1a.  lock(L)
2a.  a_1 = Z;
3a.  X = a_1 + 1;
4a.  unlock(L);
5a.  Y = a_1;
Ja.  join(M_b);
7.   assert(Y ≤ 5);

     Thread M_b
1b.  b_1 = Y;
2b.  Lock(L)
3b.  b_2 = X;
4b.  Z = b_1 + 1
5b.  Unlock(L)
6b.  Y = b_1 + b_2
Jb.  /* join M_a */
```

(a) (b) (c)

Fig. 1. (a) Concurrent system P with threads M_a, M_b and local variables a_i, b_i respectively, communicating with shared variable X, Y, Z, L. (b) lattice and a run σ (c) CTP_σ as CCFG

3.1 Transaction Sequence Graph

We now briefly describe the construction of TSG from the CCFG obtained above. Assuming we have computed—using MAT analysis (described in the next section)—independent transactions sets AT_a and AT_b and necessary context switches for threads M_a and M_b, where $AT_a = \{1a \cdots 5a, 5a \cdot Ja\}$, $AT_b = \{1b \cdot 2b, 2b \cdots 6b, 6b \cdot Jb\}$, and the context switching pairs are $\{(2b, 1a), (Ja, 1b)(6b, 1a)(5a, 2b), (Ja, 6b)(Jb, 1a)(Ja, 2b)(Jb, 5a)\}$. The independent transactions are shown in Figure 2(a) as shaded rectangles.

Given such sets of independent transactions and context switching pairs, we construct a transaction sequence graph (TSG), a digraph as shown in Figure 2(b), as follows: the beginning and ending of each independent transaction forms nodes, each independent transaction forms a transaction edge (solid bold edge), and each context-switching pairs forms a context-switch edge (dash edge). We use V, TE, and CE to denote the set of nodes, transaction edges, and context-switch edges, respectively. Such a graph captures all and only the representative interleaving, where each interleaving is a sequence of independent transactions connected by directed edges. The number of nodes ($|V|$) and the number of transaction edges ($|TE|$) in TSG are linear in the number of independent transactions, and the number of context-switch edges ($|CE|$) is quadratic in the number of independent transactions. The TSG (in Figure 2(b)) has 7 nodes and 13 edges (= 5 transaction edges + 8 context-switch edges).

If we do not use MAT analysis, a naive way of defining an independent transaction would be a sequence of transitions such that only the last transition has a global access. This is the kind of graph representation used by most of the prior work in the literature [15–19]. In the sequel, we refer to a TSG obtained without MAT analysis as a CCFG. Such a graph would have 13 nodes, and 41 edges (=11 transaction edges + 30 context-switch edges).

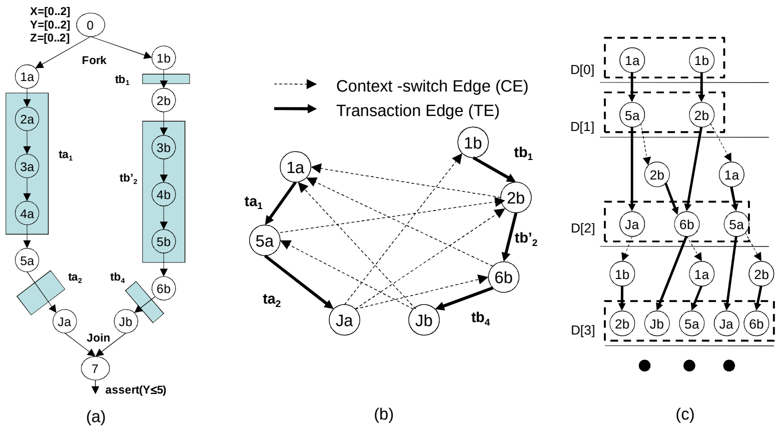


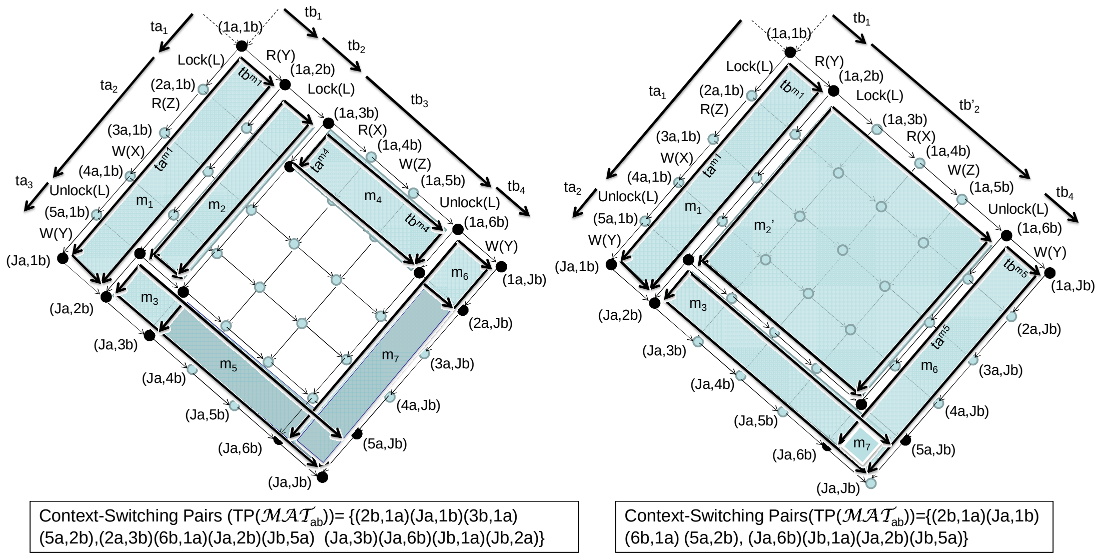
Fig. 2. (a) CCFG with independent transactions (b) TSG (c) Traversal on TSG

Range Propagation on TSG Although TSG may have cycles (as shown in Figure 2(b)), the sequential consistency requirement does not permit such cycles in any feasible path. A key observation is that any feasible path will have a sequence of transactions of length at most $|TE|$. As per the interleaving semantics, any schedule can not have two or more consecutive context switches. Thus, a feasible path will have at most $|TE|$ context switches. For example, path $Ja \cdot 2b \cdot 1a \cdot 5a$ involves two consecutive context switches, and therefore, can be ignored for range propagation. Clearly, one does not require a fixed point computation for range propagation, but rather a bounded number of iterations of size $O(|TE|)$.

Let $D[i]$ denote a set of TSG nodes reachable at BFS depth i from an initial set of nodes. Starting from each node in $D[i]$, we compute range along one transaction edge or along one context switch edge together with its subsequent transaction edge. We show such a traversal on TSG in Figure 2(c), where dashed and solid edges correspond to context switch and transaction edges, respectively. The nodes in $D[i]$ are shown in dotted rectangles. As a transaction edge is associated with at most one context switch edge, a range propagation would require $O(|V| \cdot |TE|)$ updates per iteration.

3.2 MAT Analysis

We now discuss the essence of MAT analysis used to obtain TSG. Consider a pair (ta^{m_1}, tb^{m_1}), shown as the shaded rectangle m_1 in Figure 3(a), where $ta^{m_1} \equiv Lock(L)_a \cdot R(Z)_a \cdots W(Y)_a$ and $tb^{m_1} \equiv R(Y)_b$ are transactions of threads M_a and M_b, respectively. For the ease of readability, we use an event to imply the corresponding generator transition.



Context-Switching Pairs (TP($\mathcal{MAT}_{ab}$))= {(2b,1a)(Ja,1b)(3b,1a) (5a,2b),(2a,3b)(6b,1a)(Ja,2b)(Jb,5a) (Ja,3b)(Ja,6b)(Jb,1a)(Jb,2a)}

Context-Switching Pairs(TP($\mathcal{MAT}_{ab}$))={(2b,1a)(Ja,1b) (6b,1a) (5a,2b), (Ja,6b)(Jb,1a)(Ja,2b)(Jb,5a)}

Fig. 3. MATs m_i shown as rectangles, obtained using (a) `GenMAT` (b) `GenMAT'`

From the control state pair $(1a, 1b)$, the pair $(Ja, 2b)$ can be reached by one of the two representative interleavings $ta^{m_1} \cdot tb^{m_1}$ and $tb^{m_1} \cdot ta^{m_1}$. Such a transaction pair (ta^{m_1}, tb^{m_1}) is *atomic pair-wise* as one avoids interleaving them *in-between*, and hence, referred as *Mutually Atomic Transaction*, MAT for short [26]. Note that in a MAT only the last transitions pair is dependent. Other MATs $m_2 \cdots m_7$ are similar. A MAT is formally defined as:

Definition 2 (Mutual Atomic Transactions (MAT), Ganai 09). *We say two transactions tr_i and tr_j of threads M_i and M_j, respectively, are mutually atomic iff except for the last pair, all other transitions pairs in the corresponding transactions are independent. Formally, a Mutually Atomic Transactions (MAT) is a pair of transactions, i.e., $(tr_i, tr_j), i \neq j$ iff $\forall k\ 1 \leq k \leq |tr_i|, \forall h\ 1 \leq h \leq |tr_j|$, $(tr_i[k], tr_j[h]) \notin \mathcal{D}$ $(k \neq |tr_i|$ and $h \neq |tr_j|)$, and $tr_i[|tr_i|], tr_j[|tr_j|]) \in \mathcal{D}$.*

The basic idea of MAT-based partial order reduction [26] is to restrict context switching only between the two transactions of a MAT. A context switch can only occur from the ending of a transaction to the beginning of the other transaction in the same MAT. Such a restriction reduces the set of necessary thread interleavings to explore. For a given MAT $\alpha = (f_i \cdots l_i, f_j \cdots l_j)$, we define a set $TP(\alpha)$ of possible context switches as ordered pairs, i.e., $TP(\alpha) = \{(end(l_i), begin(f_j)), (end(l_j), begin(f_i))\}$. Note that there are exactly two context switches for any given MAT.

Let TP denote a set of possible context switches. For a given CTP, we say TP is *adequate* iff for each feasible thread schedule of the CTP there is an equivalent schedule that can be obtained by choosing context switching only between the pairs in TP. Given a set $\mathcal{MAT}$ of MATs, we define $TP(\mathcal{MAT}) = \bigcup_{\alpha \in \mathcal{MAT}} TP(\alpha)$. A set $\mathcal{MAT}$ is called *adequate* iff $TP(\mathcal{MAT})$ is adequate. For a given CCFG, one can use an algorithm `GenMAT` [26] to obtain an adequate set of $\mathcal{MAT}$ that allows only representative thread schedules, as claimed in the following theorem.

Theorem 1 (Ganai, 2009). *`GenMAT` generates a set of MATs that captures all (i.e., adequate) and only (i.e., optimal) representative thread schedules. Further, its running cost is $O(n^2 \cdot k^2)$, where n is number of threads, and k is the maximum number of shared accesses in a thread.*

The `GenMAT` algorithm on the running example proceeds as follows. It starts with the pair $(1a, 1b)$, and identifies two MAT candidates: $(1a \cdots Ja, 1b \cdot 2b)$ and $(1a \cdot 2a, 1b \cdots 6b)$. By giving M_b higher priority over M_a, it selects the former MAT (i.e., m_1) uniquely. Note that the choice of M_b over M_a is arbitrary but is fixed through the MAT computation, which is required for the optimality result. After selecting MAT m_1, it inserts in a queue Q, three control state pairs $(1a, 2b), (Ja, 2b), (Ja, 1b)$ corresponding to the *begin* and the *end* pairs of the transactions in m_1. These correspond to the three corners of the rectangle m_1. In the next step, it pops out the pair $(1a, 2b) \in Q$, selects MAT m_2 using the same priority rule, and inserts three more pairs $(1a, 3b), (5a, 2b), (5a, 3b)$ in Q. Note that if there is no transition from a control state such as Ja, no MAT is generated from $(Ja, 2b)$. The algorithm terminates when all the pairs in the queue (denoted as ● in Figure 3(a)) are processed. Note that the order of pair insertion can be arbitrary, but the same pair is never inserted more than once.

For the running example, a set $\mathcal{MAT}_{ab} = \{m_1, \cdots m_7\}$ of seven MATs is generated. Each MAT is shown as a rectangle in Figure 3(a). The total number of context switches allowed by the set, i.e., $TP(\mathcal{MAT}_{ab})$ is 12. The highlighted interleaving (shown in Figure 1(b)) is equivalent to the representative interleaving $tb^{m_1} \cdot ta^{m_1} \cdot tb^{m_3}$ (Figure 3(a)). One can verify (the optimality) that this is the only representative schedule (of this equivalence class) permissible by the set $TP(\mathcal{MAT}_{ab})$.

Reduction of MAT We say a MAT is *feasible* if the corresponding transitions do not disable each other; otherwise it is *infeasible*. For example, as shown in Figure 3(a), MAT $m_2 = (ta^{m_2}, tb^{m_2})$ is infeasible, as the interleaving $tb^{m_2} \cdot ta^{m_2}$ is infeasible due to locking semantics, although the other interleaving $ta^{m_2} \cdot tb^{m_2}$ is feasible.

The `GenMAT` algorithm does not generate infeasible MATs when both the interleavings are infeasible. Such case arises when control state pairs such as $(2a, 3b)$ are simultaneously unreachable. However, it generates an infeasible MAT if such pairs are simultaneously reachable with only one interleaving of the MAT (while the other one is infeasible). For example, it generates MAT m_2 as $(5a, 3b)$ is reachable with only interleaving $Lock(L)_a \cdots Unlock(L)_a \cdot Lock(L)_b$ while the other one $Lock(L)_b \cdot Lock(L)_a \cdots Unlock(L)_a$ is infeasible. Such infeasible MAT may result in generation of other MATs, such as m_5 which may be redundant, and m_4 which may be infeasible. Although the interleaving space captured by $\mathcal{MAT}_{ab}$ is still adequate and optimal, the set apparently may not be "minimal" as some interleavings may be infeasible.

To address the minimality, we modify `GenMAT` such that only feasible MATs are chosen as MAT candidates. We refer to the modified algorithm as `GenMAT`'. We use additional static information such as lockset analysis [1] to obtain a reduced set $\mathcal{MAT}'_{ab}$ and later show (Theorem 2) that such reduction do not exclude any feasible interleaving. The basic modification is as follows: stating from the pair $(begin(f_i), begin(f_j))$, if a MAT $(f_i \cdots l_i, f_j \cdots l_j)$ is infeasible, then we select a MAT $(f_i \cdots l'_i, f_j \cdots l'_j)$ that is a feasible, where $end(l_i) \prec_{po} end(l'_i)$ or $end(l_j) \prec_{po} end(l'_j)$ or both.

With this modified step, `GenMAT'` produces a set $\mathcal{MAT}'_{ab} = \{m_1, m'_2, m_3, m_6, m_7\}$ of five MATs, as shown in Figure 3b. Note that infeasible MATs m_2 and m_4 are replaced with MAT m'_2. MAT m_5 is not generated as m_2 is no longer a MAT, and therefore, control state pair $(5a, 3b)$ is no longer in Q.

The basic intuition as to why m_5 is redundant is as follows: For m_5, we have $TP(m_5) = \{(Ja, 2b), (5a, Jb)\}$. The context switching pair $(Ja, 2b)$ is infeasible, as the interleaving allowed by m_5, i.e., $R(Y)_b \cdot Lock(L)_b \cdot Lock(L)_a \cdot W(Y)_a \cdot R(X)_a \cdots$ is an infeasible interleaving. The other context switching pair $(5a, Jb)$ is included in either $TP(m_3)$ or $TP(m_7)$, where m_3, m_7 are feasible MATs (Figure 3(b)). The proof that $TP(\mathcal{MAT}'_{ab})$ allows the same set of feasible interleavings as allowed by $TP(\mathcal{MAT}_{ab})$, is given in Section 4.

Independent Transactions Given a set of MATs, we obtain a set of independent transactions of a thread M_i, denoted as AT_i, by splitting the pair-wise atomic transactions of the thread M_i as needed into multiple transactions such that a context switching (under MAT-based reduction) can occur either to the beginning or from the end of such transactions.

For the running example, the sets of independent transactions corresponding to $\mathcal{MAT}'_{ab}$ are $AT_a = \{1a \cdots 5a, 5a \cdot Ja\}$ and $AT_b = \{1b \cdot 2b, 2b \cdots 6b, 6b \cdot Jb\}$. These are shown in Figure 2(a) as shaded rectangles, and are shown as outlines of the lattice in Figure 3(b). The size of set of independent transaction determines the size of TSGs.

If we used $\mathcal{MAT}_{ab}$, we would have obtained $AT_a = \{1a \cdot 2a, 2a \cdots 5a, 5a \cdot Ja\}$ and $AT_b = \{1b \cdot 2b, 2b \cdot 3b, 3b \cdots 6b, 6b \cdot Jb\}$, as shown outlining the lattice in Figure 3(a). A TSG constructed using $\mathcal{MAT}_{ab}$ (not shown) would have 8 nodes and 17 edges (= 7 transaction edges + 10 context-switch edges). Note, out of the 12 context-switches, one can remove $(3b, 1a)$ and $(2a, 3b)$ as they are simultaneously unreachable.

4 Our Approach: TSG-based Interval Analysis

We now present our approach formally. We first discuss MAT reduction step. Then we describe the construction of TSGs in Section 4.1, followed by interval analysis on TSG in Section 4.2. For comparison, we introduce a notion of interval metric in Section 4.3.

Given a CTP with threads $M_1 \cdots M_n$, and a dependency relation $\mathcal{D}$, we use `GenMAT` [26] to generate $\mathcal{MAT}_{ij}$ for each pair of threads M_i and M_j, $i \neq j$, and obtain $\mathcal{MAT} = \bigcup_{i \neq j} \mathcal{MAT}_{ij}$. Note that $\mathcal{D}$ may not include the conflicting pairs that are unreachable. We now define the feasibility of MAT to improve the MAT analysis.

Definition 3 (Feasible MAT). *A MAT $m = (tr_i, tr_j)$ is feasible such that both representative (non-equivalent) interleavings, i.e., $tr_i \cdot tr_j$ and $tr_j \cdot tr_i$, are feasible; otherwise it is infeasible. In other words, in a feasible MAT, the corresponding transitions do not disable each other.*

We modify `GenMAT` such that only feasible MATs are chosen as MAT candidates. We denote the modified algorithm as `GenMAT'`. The modified step is as follows: starting from the pair (f_i, f_j), if a pair $(l_i, l_j) \in \mathcal{D}$ is found that yields an infeasible MAT, then

– we select another pair $(l'_i, l'_j) \in \mathcal{D}$ such that $(l_i, l_j) \sqsubset (l'_i, l'_j)$ and $(f_i \cdots l'_i, f_j \cdots l'_j)$ is a feasible MAT, and
– there is no pair $(l''_i, l''_j) \in \mathcal{D}$ such that $(l_i, l_j) \sqsubset (l''_i, l''_j) \sqsubset (l'_i, l'_j)$ and $(f_i \cdots l''_i, f_j \cdots l''_j)$ is a feasible MAT.

where $\sqsubset$ is the reachable-before relation defined before. Interested readers may refer to the complete algorithm in Appendix A (also available at [28]).

Let $\mathcal{MAT}$ and $\mathcal{MAT}'$ be the set of MATs obtained using `GenMAT` and `GenMAT'`, respectively. We state the following MAT reduction theorem:

Theorem 2 (MAT reduction). *$\mathcal{MAT}'$ is adequate, and $TP(\mathcal{MAT}') \subseteq TP(\mathcal{MAT})$.*

The proof is provided in Appendix B (also available at [28]).

4.1 Transaction Sequence Graph

To build a TSG, we first identify independent transactions of each thread, i.e., those transactions that are atomic with respect to all schedules allowed by the set of MATs, as discussed in the following. Here we use $\mathcal{MAT}$ to denote the set of MATs obtained.

Identifying Independent Transactions Given a set $\mathcal{MAT} = \bigcup_{i \neq j\ \in \{1,\cdots,n\}} \mathcal{MAT}_{ij}$, we identify independent transactions, denoted as AT_i as follows:

– We first define a set of transactions $\mathcal{MAT}_i$ of thread M_i:

$$\mathcal{MAT}_i = \{tr_i | m = (tr_i, tr_j) \in \mathcal{MAT}_{ij}\ i \neq j \in \{1, \cdots, n\}\}$$

In other words, $\mathcal{MAT}_i$ comprises all transactions of thread M_i that are pairwise atomic with some other transactions.

– Given two transactions $tr, tr' \in \mathcal{MAT}_i$, we say $begin(tr) \prec_{po} begin(tr')$ if $tr[1] \prec_{po} tr'[1]$. Using the set $\mathcal{MAT}_i$, we obtain a partial order set of control states S_i, referred as *transaction boundary* set, that is defined over $\prec_{po}$ as follows:

$$S_i \equiv \{begin(tr_{i,1}), begin(tr_{i,2}), \cdots, begin(tr_{i,m}), end(tr_{i,m})\}$$

where $tr_{i,k} \in \mathcal{MAT}_i$, and $tr_{i,m}$ denote the last transaction of the thread M_i. Note that due to conditional branching the order may not be total.

– Using the set S_i, we obtain a set of transactions AT_i of thread M_i as follows:

$$AT_i = \{t \cdots t' \mid c \xrightarrow{t \cdots t'} c' \text{ where } c \prec_{po} c' \text{ and } c, c' \in S_i \text{ and } t, \cdots, t' \in T_i \text{ and there is no } c'' \in S_i \text{ such that } c \prec_{po} c'' \prec_{po} c'\}$$

Recall that T_i is the set of transitions in M_i.

Proposition 1. *Each transaction $tr \in AT_i$ for $i \in \{1, \cdots, n\}$ is an independent transaction and is maximal, i.e., can not be made larger without it being an independent transaction. Further, for each transition $t \in T_i$, there exists $tr \in AT_i$ such that $t \in tr$.*

Constructing TSG Given a set of context-switching pairs $TP(\mathcal{MAT})$, a set of independent transactions $\bigcup_i AT_i$, and a set of transaction boundaries $\bigcup_i S_i$, we construct a transaction sequence graph, a digraph $G(V, E)$ as follows:

– $V = \cup_i V_i$ is the set of nodes, where V_i denotes a set of thread local control states corresponding to the set S_i,
– $E = TE \bigcup CE$ is the set of edges, where
  - TE is the set of *transaction edges* corresponding to the independent transactions i.e., $TE = \{(begin(tr), end(tr)) \mid tr \in \bigcup_i AT_i\}$
  - CE is the set of *context switch edges* corresponding $TP(\mathcal{MAT})$ i.e., $CE = \{(c_i, c_j) \mid (c_i, c_j) \in TP(\mathcal{MAT})\}$

A TSG $G(V, E = (CE \cup TE))$, as constructed, has $|V| = O(\Sigma_i |AT_i|)$, $|TE| = (\Sigma_i |AT_i|)$, and $|CE| = (\Sigma_{i \neq j} |AT_i| \cdot |AT_j|)$, where $i, j \in \{1, \cdots, n\}$, and n is number of threads. In the worst case, however, $|V| = O(n \cdot k)$, $|TE| = O(n \cdot k)$, and $|CE| = O(n^2 \cdot k^2)$ where k is the maximum number of shared accesses in any thread.

Proposition 2. *TSG as constructed captures all and only the representative interleaving (of a given CTP), each corresponding to a total ordered sequence of independent transactions where the order is defined by the directed edges of TSG.*

4.2 Range Propagation on TSG

Range propagation uses data and control structure of a program to derive range information. In this work, we consider intervals for simplicity, although other abstract domains are equally applicable. For each program variable v, we define an interval $\langle l_v^c, u_v^c \rangle$, where l_v^c, l_v^c are integer-valued lower and upper bounds for v at a control location c. One can define, for example, the lower bound(L)/upper bound (U) of an expression $exp = exp_1 + exp_2$ at a control location c as $L(exp, c) = L(exp_1, c) + L(exp_2, c)$ and $U(exp, c) = U(exp_1, c) + U(exp_2, c)$, respectively (more details in [29]).

We say an interval $\langle l_v^c, u_v^c \rangle$ is *adequate* if value of v at location c, denoted as $val(v, c)$ is bounded in all program executions, i.e., $l_v^c \leq val(v, c) \leq u_v^c$. As there are potentially many feasible paths, range propagation is typically carried out iteratively along bounded paths, where the adequacy is achieved conservatively. However, such bounded path analysis can still be useful in eliminating paths that do not satisfy sequential consistency requirements. As shown in Figure 2(c), a sequence $5a \cdot 2b \cdot 6b \cdot 1a$ does not follow program order, and therefore, paths with such a sequence can be eliminated.

At an iteration step i of range propagation, let $r^{c,p}[i]$ denote the range information (i.e., a set of intervals) at node c along a feasible path p, and is defined as:

$$r^{c,p}[i] = \{\langle l_v^{c,p}[i], u_v^{c,p}[i] \rangle | \text{ interval for } v \text{ computed at node } c \text{ along path } p \text{ at step } i\}$$

One can merge $r^{c,p}[i]$ and $r^{c,p'}[i]$ conservatively as follows:

$$r^{c,p}[i] \sqcup r^{c,p'}[i] = \{\langle l_v^{c,p}[i], u_v^{c,p}[i] \rangle \sqcup \langle l_v^{c,p'}[i], u_v^{c,p'}[i] \rangle | \text{ interval for } v \text{ computed at node } c \text{ along paths } p, p' \text{ at step } i\}$$

where the interval merge operator ($\sqcup$) is defined as: $\langle l, u \rangle \sqcup \langle l', u' \rangle = \langle min(l, l'), max(u, u') \rangle$. Let $r^c[i]$ denote the range information at node c at step i, i.e.,

$$r^c[i] = \{\langle l_v^c[i], u_v^c[i] \rangle \mid \text{ interval for } v \text{ computed at node } c \text{ at iteration step } i\}.$$

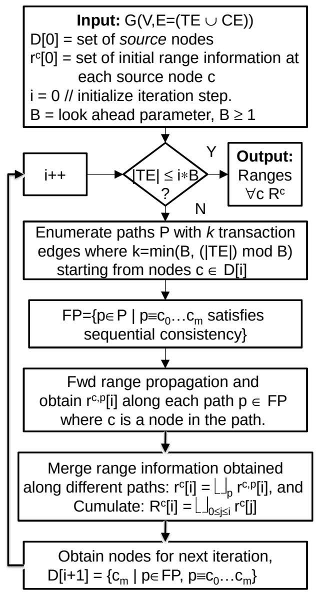
Let FP denote a set of feasible paths starting from nodes $D[i]$ of length $B \geq 1$, where B is a *lookahead* parameter that controls the trade off between precision and update cost. Given $r^{c,p}[i]$ with $p \in FP$, we obtain the range information at step i as $r^c[i] = \sqcup_{p \in FP}\ r^{c,p}[i]$ and cumulative range information at step i as $R^c[i] = \sqcup_{j=0}^{j=i} r^c[j]$.

We present a self-explanatory flow of our forward range propagation procedure, referred as RPT, for a given TSG $G = (V, E)$ in Figure 4(a). As observed in Section 3.1, in any representative feasible path, a transaction edge is associated with at most one context switch edge. Thus, the length of such a path is at most $2 \cdot |TE|$. At every iteration of range propagation, we compute the range along a sequence of $|B|$ transaction edges interleaved with at most $|B|$ context switch edges. Such a range propagation requires $\lceil |TE|/B \rceil$ iterations. The cost of range propagation at each iteration is $O(|V| \cdot |TE|^B)$. After RPT terminates, we obtain the final cumulative range information $R^c[i]$ at each node c, denoted as R^c.

Proposition 3. *Given a TSG $G = (V, E = (TE \cup CE))$ that captures all feasible paths of a CTP, the procedure RPT generates adequate range information R^c for each node $c \in V$, and the cost of propagation is $O(|V| \cdot |TE|^{B+1})$.*

We show a run of RPT in Figure 4(b) on the TSG shown in Figure 2(b). At each iteration step i, we show the range computed $r^c[i]$ (for each global variable) at the control states $1a, 5a, Ja, 1b, 2b, 6b, Jb$. Since there are 5 TE edges in the TSG, we require

5 iterations with $B = 1$. The cells with $\langle -, - \rangle$ correspond to no range propagation to those nodes. The cells in **bold** at step i correspond to nodes in $D[i]$. The final intervals at each node c, i.e., R^c, is equal to the data-union of the range values at c computed at each iteration $i = 1 \cdots 5$. We show the corresponding cumulative intervals obtained for the CCFG after 11 iterations (as it has 11 TE edges). Note that using TSG, RPT not only obtains more refined intervals, but also requires fewer iterations. Also observe that the assertion $Y \leq 5$ (line 7, Figure 1(a)) holds at Jb with the final intervals for Y obtained using TSG, while it does not hold at Jb when obtained using CCFG.


Input: G(V,E=(TE ∪ CE))
D[0] = set of source nodes
r^c[0] = set of initial range information at each source node c
i = 0 // initialize iteration step.
B = look ahead parameter, B ≥ 1
i++
|TE| ≤ i*B ?
Y
N
Output: Ranges ∀c R^c
Enumerate paths P with k transaction edges where k=min(B, (|TE|) mod B) starting from nodes c ∈ D[i]
FP={p∈ P | p≡c0…cm satisfies sequential consistency}
Fwd range propagation and obtain r^{c,p}[i] along each path p ∈ FP where c is a node in the path.
Merge range information obtained along different paths: r^c[i] = ⊔p r^{c,p}[i], and Cumulate: R^c[i] = ⊔0≤j≤i r^c[j]
Obtain nodes for next iteration, D[i+1] = {cm | p∈ FP, p≡c0…cm}

(a)

Ranges $r^c[i]$ at each step i=0..5 on TSG with B=1								
i	V	1a	5a	Ja	1b	2b	6b	Jb
0	X	$\langle\mathbf{0,2}\rangle$	$\langle -,-\rangle$	$\langle -,-\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle -,-\rangle$	$\langle -,-\rangle$	$\langle -,-\rangle$
	Y	$\langle\mathbf{0,2}\rangle$	$\langle -,-\rangle$	$\langle -,-\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle -,-\rangle$	$\langle -,-\rangle$	$\langle -,-\rangle$
	Z	$\langle\mathbf{0,2}\rangle$	$\langle -,-\rangle$	$\langle -,-\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle -,-\rangle$	$\langle -,-\rangle$	$\langle -,-\rangle$
1	X	$\langle 0,2\rangle$	$\langle\mathbf{1,3}\rangle$	$\langle -,-\rangle$	$\langle 0,2\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle -,-\rangle$	$\langle -,-\rangle$
	Y	$\langle 0,2\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle -,-\rangle$	$\langle 0,2\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle -,-\rangle$	$\langle -,-\rangle$
	Z	$\langle 0,2\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle -,-\rangle$	$\langle 0,2\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle -,-\rangle$	$\langle -,-\rangle$
2	X	$\langle 0,2\rangle$	$\langle\mathbf{1,3}\rangle$	$\langle\mathbf{1,3}\rangle$	$\langle 0,2\rangle$	$\langle 0,3\rangle$	$\langle\mathbf{0,3}\rangle$	$\langle -,-\rangle$
	Y	$\langle 0,2\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle 0,2\rangle$	$\langle 0,2\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle -,-\rangle$
	Z	$\langle 0,2\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle 0,2\rangle$	$\langle 0,2\rangle$	$\langle\mathbf{-1,2}\rangle$	$\langle -,-\rangle$
3	X	$\langle 0,3\rangle$	$\langle\mathbf{0,3}\rangle$	$\langle\mathbf{1,3}\rangle$	$\langle 0,3\rangle$	$\langle\mathbf{0,3}\rangle$	$\langle\mathbf{0,3}\rangle$	$\langle\mathbf{0,2}\rangle$
	Y	$\langle 0,2\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle 0,2\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle\mathbf{0,5}\rangle$
	Z	$\langle -1,2\rangle$	$\langle\mathbf{-1,2}\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle 0,2\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle\mathbf{-1,2}\rangle$	$\langle\mathbf{-1,2}\rangle$
4	X	$\langle 0,3\rangle$	$\langle\mathbf{0,3}\rangle$	$\langle\mathbf{0,3}\rangle$	$\langle 0,3\rangle$	$\langle\mathbf{0,3}\rangle$	$\langle\mathbf{0,3}\rangle$	$\langle\mathbf{0,3}\rangle$
	Y	$\langle 0,5\rangle$	$\langle\mathbf{0,5}\rangle$	$\langle\mathbf{-1,2}\rangle$	$\langle 0,2\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle\mathbf{0,2}\rangle$	$\langle\mathbf{0,5}\rangle$
	Z	$\langle -1,2\rangle$	$\langle\mathbf{-1,2}\rangle$	$\langle\mathbf{-1,2}\rangle$	$\langle 0,2\rangle$	$\langle\mathbf{-1,2}\rangle$	$\langle\mathbf{-1,2}\rangle$	$\langle\mathbf{-1,2}\rangle$
5	X	$\langle 0,3\rangle$	$\langle\mathbf{0,3}\rangle$	$\langle\mathbf{0,3}\rangle$	$\langle 0,3\rangle$	$\langle\mathbf{0,3}\rangle$	$\langle\mathbf{0,3}\rangle$	$\langle\mathbf{0,3}\rangle$
	Y	$\langle 0,5\rangle$	$\langle\mathbf{0,5}\rangle$	$\langle\mathbf{-1,2}\rangle$	$\langle -1,2\rangle$	$\langle\mathbf{-1,5}\rangle$	$\langle\mathbf{-1,5}\rangle$	$\langle\mathbf{0,5}\rangle$
	Z	$\langle -1,2\rangle$	$\langle\mathbf{-1,2}\rangle$	$\langle\mathbf{-1,2}\rangle$	$\langle -1,2\rangle$	$\langle\mathbf{-1,2}\rangle$	$\langle\mathbf{-1,2}\rangle$	$\langle\mathbf{-1,2}\rangle$
R^c	X	$\langle 0,3\rangle$	$\langle 0,3\rangle$	$\langle 0,3\rangle$	$\langle 0,3\rangle$	$\langle 0,3\rangle$	$\langle 0,3\rangle$	$\langle 0,3\rangle$
	Y	$\langle 0,5\rangle$	$\langle 0,5\rangle$	$\langle -1,2\rangle$	$\langle -1,2\rangle$	$\langle -1,5\rangle$	$\langle -1,5\rangle$	$\langle 0,5\rangle$
	Z	$\langle -1,2\rangle$	$\langle -1,2\rangle$	$\langle -1,2\rangle$	$\langle -1,2\rangle$	$\langle -1,2\rangle$	$\langle -1,2\rangle$	$\langle -1,2\rangle$
Final ranges on CCFG (Figure 1(c)) with B=1 at i=11								
R^c	X	$\langle 0,3\rangle$	$\langle 0,3\rangle$	$\langle 0,3\rangle$	$\langle 0,3\rangle$	$\langle 0,3\rangle$	$\langle 0,3\rangle$	$\langle 0,3\rangle$
	Y	$\langle -1,7\rangle$	$\langle -1,7\rangle$	$\langle -1,2\rangle$	$\langle -1,7\rangle$	$\langle -1,7\rangle$	$\langle -1,7\rangle$	$\langle 0,7\rangle$
	Z	$\langle -1,2\rangle$	$\langle -1,2\rangle$	$\langle -1,2\rangle$	$\langle -1,2\rangle$	$\langle -1,2\rangle$	$\langle -1,2\rangle$	$\langle -1,2\rangle$
Notes: Cells with ranges in **bold** correspond to nodes in $D[i]$. $\langle -,-\rangle$ corresponds to unreachable node at depth i								

(b)

Fig. 4. (a) RPT: Range Propagation on TSG (b) A run of RPT on TSG (Figure 2) and CCFG

4.3 Interval Metric

Given the final intervals $\langle l_v^c, u_v^c \rangle \in R^c$, we use the total number of bits needed (the fewer the better) to encode each interval, as a metric to compare effectiveness of interval analysis on CCFG and TSGs. We refer to that as *interval metric*. It has two components: local (denoted as RB_l) and global (denoted as RB_g) corresponding to the total range bits of local and global variables, respectively.

The local component RB_l is computed as follows:

$$RB_l = \Sigma_{t \in \bigcup_i T_i} \Sigma_{v \in assgn_l(t)} \ \log_2(u_v^{end(t)} - l_v^{end(t)})$$

where $assgn_l(t)$ denotes a set of local variables assigned (or updated) in transition t.

For computing the global component RB_g, we need to account for context switching that can occur between global updates. Hence, we add a synchronization component, denoted as RB_g^{sync}, in the following:

$$RB_g = \Sigma_{t \in \bigcup_i T_i} \Sigma_{v \in assgn_g(t)} \ \log_2(u_v^{end(t)} - l_v^{end(t)}) \ + \ RB_g^{sync}$$

where $assgn_g(t)$ denotes a set of global variables assigned in transition t, and RB_g^{sync} is the synchronization component corresponding to a global state before an independent transaction begins, and is computed as follows:

$$RB_g^{sync} = \Sigma_{tr \in \bigcup_i AT_i} \Sigma_{v \in \mathcal{V}} \log_2(u_v^{begin(tr)} - l_v^{begin(tr)})$$

where $v \in \mathcal{V}$ is a global variable, and tr is an independent transaction.

For the running example, the interval metrics obtained are as follows: CCFG: $RB_l = 8, RB_g = 95$; TSG using $\mathcal{MAT}_{ab}$: $RB_l = 6, RB_g = 57$; TSG using $\mathcal{MAT}'_{ab}$: $RB_l = 6, RB_g = 43$.

5 Experiments

In our experiments, we use several multi-threaded benchmarks of varied complexity with respect to the number of shared variable accesses. There are 4 sets of benchmarks that are grouped as follows: simple to complex concurrent programs [26] (`cp`), our Linux/Pthreads/C implementation [12] of atomicity violations reported in apache server [30] (`atom`), bank benchmarks [31] (`bank`), and indexer benchmarks [9] (`index`). Each set has concurrent trace programs (CTP) generated [25] from the runs of the corresponding concurrent programs. These benchmarks are publicly available at [32]. We used constant propagation algorithm [16] to preprocess these benchmarks in order to expose the benefits of our approach.

Our experiments were conducted on a linux workstation with a 3.4GHz CPU and 2GB of RAM, and a time limit of 20 minutes. From these benchmarks, we first obtained `CCFG`. Then we obtained `TSG` and `TSG′` after conducting MAT analysis on the CCFGs, using `GenMAT` and `GenMAT′`, respectively, as described in Section 4.1. For all three graphs, we removed context switch edges between node pairs that are found unreachable using lockset analysis [1].

Comparison of `RPT` on `CCFG`, `TSG`, and `TSG′` are shown in Table 1 using lookahead parameter $B = 1$. The characteristics of the corresponding CTPs are shown in Columns 2-6, the results of `RPT` on `CCFG`, `TSG` and `TSG′` are shown in Columns 7-11, and Columns 12-17, and Columns 18-23, respectively. Columns 2-6 describe the following: the number of threads (n), the number of local variables (#L), the number of global variables (#G), the number of global accesses (#A), and the number of total transitions (#T), respectively. Columns 7-11 describe the following: the number of context switch edges (#CE), the number of transaction edges (#TE) (same as the number of iterations of `RPT`), the time taken (t, in sec), the number of local bits RB_l, and number of global bits RB_g, respectively. Columns 12-17 and 18-23 describe similarly for `TSG` and `TSG′` including the number of MATs obtained (#M). In case of `CCFG`, we obtained a transaction by combining sequence of transitions such that only the last transition has exactly one global access. The time reported includes MAT analysis (if performed) and run time of `RPT`.

As we notice, `RPT` on `TSG` and `TSG′` (except `index4`) completes in less than a minute, and is an order of magnitude faster compared to that on `CCFG`. Also, the interval metric (RB_l, RB_g) for `TSG` and `TSG′` are significantly lower compared to `CCFG`. Further, between `TSG′` and `TSG`, the former generates tighter intervals.

Using the adequate intervals obtained, we also checked for unreachability of control states. We found that using both `TSG` and `TSG′`, we were able to show a few control states (shown in brackets) unreachable in the following examples: `bank2` (1), `bank3` (1), `index1` (64), and `index2` (32). However, we could not show a single unreachable control state using `CCFG` in these benchmarks.

Next we evaluate the reduction in the efforts of a heavy-weight trace-based symbolic analysis tool `CONTESSA` [13] using `RPT` results. For each benchmark, we selected a reachability property corresponding to a reachability of a thread control state. Using

Table 1. Comparison of RPT on CCFG, TSG and TSG'.

Ex	Characteristics					CCFG					TSG						TSG′					
	n	#L	#G	#A	#T	#C	#TE	t (s)	RB_l	RB_g	#M	#C	#TE	t (s)	RB_l	RB_g	#M	#C	#TE	t (s)	RB_l	RB_g
cp1	3	4	3	41	28	90	24	< 1	6	131	18	22	13	< 1	6	82	9	14	9	< 1	6	50
cp2	3	4	3	185	108	1562	88	< 1	22	531	330	342	45	< 1	22	354	121	222	25	< 1	22	178
cp3	3	4	3	905	508	35802	408	13	102	2531	7650	7702	205	2	102	1714	2601	5102	105	1	102	818
atom1	3	1	2	27	25	44	16	< 1	29	493	11	14	11	< 1	29	300	6	9	8	< 1	29	200
atom2	3	2	3	37	31	68	20	< 1	30	647	14	19	13	< 1	30	389	8	12	9	< 1	30	245
atom3	3	2	11	412	243	4748	153	< 1	61	10.5K	1321	1350	91	< 1	61	6945	478	865	48	< 1	61	3605
atom4	3	3	13	435	251	5336	160	1	32	12.4K	1344	1342	95	< 1	32	7933	508	899	50	< 1	32	4115
bank1	9	59	16	383	286	12.6K	180	10	855	22.8K	178	278	75	< 1	808	10.6K	178	278	75	< 1	808	10.6K
bank2	9	67	25	540	369	25.1K	231	63	818	35.9K	440	559	155	1	771	25.1K	277	409	115	< 1	771	19K
bank3	9	67	26	599	386	24.8K	240	38	834	38.3K	384	454	147	< 1	786	24.7K	212	320	99	1	786	16.6K
index1	9	11	24	229	168	7224	98	2	52	7653	6	12	12	< 1	32	452	6	12	12	< 1	32	452
index2	19	21	54	514	363	40.1K	213	51	154	43.5K	351	513	106	1	132	11.6K	225	366	64	1	132	6613
index3	31	33	184	2125	1490	627K	821	TO	NA	NA	2573	3386	496	40	883	399K	1399	2024	265	22	883	121K
index4	33	35	246	3914	2793	1.98M	1490	TO	NA	NA	29.6K	31.4K	922	822	1814	1M	10.8K	11.9K	479	275	1814	307K

Notes: n: num. of threads, #L: num. of local vars, #G: num. of global vars, #A: num. of global accesses, #T: num. of transitions, #CE: num. of context switch edges, #TE: num. of transaction edges (=num. of iterations), t(s): time in sec (TO: t $>$ 1200s), #M: num. of MATs, RB_l: num. of local bits, RB_g: num. of global bits, $B = 1$ for the experiments

the tool, we then generated Satisfiability Modulo Theory (SMT) formula such that the formula is satisfiable if and only if the control state is reachable. We then compared the solving time of two such SMT formula, one encoded using the bit-widths of variables as obtained using RPT (denoted as ϕ_R), and other encoded using integer bit-width of 32 (denoted as ϕ_{32}). We used SMT solver Yices-1.0.28 [33]. For each benchmark, we present the time taken by the SMT solver to solve the property in Table 2 for a time limit of 1500s.

Column 1 lists the benchmarks. Column 2 show whether the property is satisfiable (S) or unsatiable (U). Columns 3–4, 5–6, 7–8 present the time taken by the solver on ϕ_{32} and ϕ_R on CCFG, TSG, and TSG′, respectively. We observe that the intervals obtained using RPT reduces the efforts of the symbolic analysis by 1-2 orders of magnitude.

Table 2. Time Taken (in sec) by Symbolic Analysis [13] on CCFG, TSG and TSG'.

Ex	S/	CCFG		TSG		TSG′	
	U?	ϕ_{32}	ϕ_R	ϕ_{32}	ϕ_R	ϕ_{32}	ϕ_R
cp1	S	< 1	< 1	< 1	< 1	< 1	< 1
cp2	S	4	< 1	3	< 1	1	< 1
cp3	S	TO	432	366	10	34	5
atom1	S	< 1	< 1	< 1	< 1	< 1	< 1
atom2	S	< 1	< 1	< 1	< 1	< 1	< 1
atom3	S	6	3	15	6	3	1
atom4	S	35	44	15	4	4	2
bank1	U	6	6	1	< 1	1	< 1
bank2	U	16	15	2	1	2	1
bank3	U	16	15	2	1	2	1
index1	U	4	4	1	< 1	1	< 1
index2	U	40	41	2	1	2	1
index3	U	TO	TO	30	23	24	15
index4	U	TO	TO	TO	TO	TO	252

Notes: S/U: Satisfiable/Unsatisfiable instance
ϕ_{32}: using 32-bit for each variable
ϕ_R: using bit-width obtained using RPT
TO: Time out (time taken $>$ 1500s)

6 Conclusion, Related and Future Work

We presented an interval analysis for CTPs using the new notion of TSGs, which is often more precise and space/time efficient than using the standard CCFGs. We use a MAT analysis to obtain independent transactions and to minimize the size of the TSGs. We also propose a non-trivial improvement to the MAT analysis to further simplify the TSGs. Our work is related to the prior work on static analysis for concurrent programs such as [15-19], although such analysis were directly applied to the CCFG of a whole program. Our notion of TSG is also different from the transaction graph (TG) [20] and the task interaction concurrency graph (TICG) [14] that have been used in concurrent data flow analysis. Such graphs, i.e, TG and TICG, represent a product graph where nodes correspond to the global control states and edges correspond to thread transitions—such graphs are often significantly bigger in size than TSGs.

Although we have applied our TSG approach only to CTPs, we plan to generalize it for concurrent programs with loops. Such generalization would involve extending the MAT analysis to handle loops (e.g. by considering the loop back-edges during MAT generation) and introducing abstract domains to handle the interleaving of interacting loops (e.g. by considering independent transactions in a loop). We leave that as a future work.

References

1. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. In *ACM Trans. Comput. Syst.*, 1997.
2. C. Flanagan and S. N. Freund. Atomizer: A dynamic atomcity checker for multithreaded programs. In *Proc. of IPDPS*, 2004.
3. M. Xu, R. Bodik, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *Programming Language Design and Implementation*, 2005.
4. L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. In *IEEE Transactions on Software Engineering*, 2006.
5. A. Farzan and P. Madhusudan. Monitoring Atomicity in Concurrent Programs. In *Proc. of CAV*, 2008.
6. C. Sadowski, S. N. Freund, and C. Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *European Symposium on Programming*, 2009.
7. A. Farzan and P. Madhusudan. Meta-analysis for atomicity violations under nested locking. In *Proc. of CAV*, 2009.
8. L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Symposium on Principles and Practice of Parallel Programming*, 2006.
9. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of POPL*, 2005.
10. M. Musuvathi and S. Quadeer. CHESS: Systematic stress testing of concurrent software. In *Logic-based Program Synthesis and Transformation*, 2006.
11. Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A Runtime Model Checker for Multi-threaded C Programs. Technical Report UUCS-08-004, University of Utah, 2008.
12. C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *Proc. of TACAS*, 2010.
13. S. Kundu, M. Ganai, and C. Wang. CONTESSA: CONcurrency TESting Augmented with Symbolic Analysis. In *Proc. of CAV*, 2010.
14. D. L. Long and L. A. Clarke. Task interaction graphs for concurrent analysis. In *International Conference on Software Engineering*, 1989.

15. M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *International Symposium on the Foundations of Software Engineering*, 1994.
16. J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler algorithms for parallel programs. *Symposium on Principles and Practice of Parallel Programming*, 1999.
17. A. Farzan and P. Madhusudan. Causal dataflow analysis for concurrent programs. In *Proc. of TACAS*, 2007.
18. R. Chugh, J. W. Voung, R. Jhala, and S. Lerer. Dataflow analysis for concurrent programs using datarace detection. In *Programming Language Design and Implementation*, 2008.
19. A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. In *Proc. of TACAS*, 2008.
20. V. Kahlon, S. Sankaranarayanan, and A. Gupta. Semantic reduction of thread interleavings in concurrent programs. In *Proc. of TACAS*, 2009.
21. F. Chen and G. Rosu. Parametric and sliced causality. In *Proc. of CAV*, 2007.
22. G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian partial-order reduction. In *Proc. of SPIN Workshop*, 2007.
23. O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for Testing Multi-threaded Java Programs. In *Concurrency and Computation: Practice and Experience*, 2003.
24. K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.
25. C. Wang, S. Chaudhuri, A. Gupta, and Y. Yang. Symbolic pruning of concurrent program executions. In *ESEC-FSE*, 2009.
26. M. K. Ganai and S. Kundu. Reduction of Verification Conditions for Concurrent System using Mutually Atomic Transactions. In *Proc. of SPIN Workshop*, 2009.
27. L. Lamport. How to make multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 1979.
28. M. K. Ganai. Conference Notes. *http://www.nec-labs.com/∼malay/notes.htm*.
29. R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Programming Language Design and Implementation*, 2000.
30. S. Lu, J. Tucekt, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Architectural Support for Programming Languages and Operating Systems*, 2006.
31. E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Parallel and Distributed Processing Symposium*, 2003.
32. System Analysis and Verification Team. NECLA SAV Benchmarks. *http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php*.
33. SRI. Yices: An SMT solver. *http://fm.csl.sri.com/yices*.

`***The appendix should not be considered as a part of the submission.***`

A Appendix: MAT Generation Algorithm

We present the algorithm `GenMAT'` (Algorithm 1), where we use **OLD/NEW** predicate to show the difference between previous [26] and our proposed improvements, respectively.

Given a CTP with threads $M_1 \cdots M_n$, and a dependency relation D, we use `GenMAT`' to generate $\mathcal{MAT}_{ij}$ for each pair of threads M_i and M_j, $i \neq j$, and obtain $\mathcal{MAT} = \bigcup_{i \neq j} \mathcal{MAT}_{ij}$. Note that D may not include the conflicting pairs that are unreachable.

For the ease of explanation, we assume there is no conditional branching in each thread. (For threads with conditional branching, please refer [28].) We also assume that each shared variable has at least one conflicting accesses in each pair of threads. (Such an assumption can be easily met by adding a dummy shared write access at the end of each thread without affecting the cost of MAT analysis. Note that such an assumption is needed for the adequacy and optimality for validity of Theorem 1 for a multi-threaded system).

With abuse of notation, we use transition t to also indicate $begin(t)$, the control state of the thread where the transition t begins. Further, we use $+t$ to denote the transition immediately after t in program order, i.e., $begin(+t) = end(t)$.

We discuss the inner loop (lines 15–18) to generate $\mathcal{MAT}_{ij}$ for a thread pair M_i and M_j, $i \neq j$. Let $(\vdash_i, \vdash_j)$ and $(\dashv_i, \dashv_j)$ denote the start and end control pair locations, respectively, of the threads M_i and M_j. We first initialize a queue Q with control state pair $(\vdash_i, \vdash_j)$ representing the beginning of the threads, respectively. For a previously unchosen pair (f_i, f_j) in the Q, we can obtain a MAT $m = (tr_i = f_i \cdots l_i, tr_j = f_j \cdots l_j)$. There can be other MAT-candidates $m' = (tr'_i = f_i \cdots l'_i, tr'_j = f_j \cdots l'_j)$ such that $l'_i \prec_{po} l_i$ or $l'_j \prec_{po} l_j$ but not both, as that would invalidate m as a candidate. Let $\mathcal{M}_c$ denote a set of such choices as obtained using the method **OLD** [26]. Using our proposed method **NEW**, we will restrict our choices to feasible MAT candidates only.

The algorithm selects $m \in \mathcal{M}_c$ uniquely by assigning thread priorities and using the following selection rule. If a thread M_j is given higher priority over M_i, the algorithm prefers $m = (tr_i = f_i \cdots l_i, tr_j = f_j \cdots l_j)$ over $m' = (tr'_i = f_i \cdots l'_i, tr'_j = f_i \cdots l'_j)$ if $l_j \prec_{po} l'_j$, i.e., $|tr_j| < |tr'_j|$. The choice of M_j over M_i is arbitrary but fixed through the MAT computation, which is required for the optimality result. We presented MAT selection (lines 10–11) in a declarative style for better understanding. However, algorithm finds the unique MAT using the selection rule, without constructing the set $\mathcal{M}_c$.

We add m to the set $\mathcal{MAT}_{ij}$. If $(+l_i \neq \dashv_i)$ and $(+l_j \neq \dashv_j)$, we update Q with three pairs, i.e., $(+l_i, +l_j), (+l_i, f_j), (f_i, +l_i)$; otherwise, we insert selectively as shown in the algorithm (lines 14—16). The algorithm terminates when all the pairs in the queue are processed. Note that the order of pair insertion can be arbitrary, but the same pair is never inserted more than once.

A run of `GenMAT`: We present a run of `GenMAT` (**OLD**) in Figure 5(a) for the running example. We gave M_2 higher priority over M_1. The table columns provide each iteration step (#I), the pair $p \in Q \backslash Q'$ selected, the chosen $\mathcal{MAT}_{ab}$, and the new pairs added in $Q \backslash Q'$ (shown in bold). It starts with the pair $(1a, 1b)$, and identifies two MAT candidates: $(1a \cdots Ja, 1b \cdot 2b)$ and $(1a \cdot 2a, 1b \cdots 6b)$. Note that the pair $(1a \cdot 2a, 1b \cdots 3b)$

is not a MAT candidate as the pair $(2a, 3b)$ is an unreachable pair. By giving M_b higher priority over M_a, it selects a MAT uniquely from the MAT candidates. The choice of M_b over M_a is arbitrary but fixed through the MAT computation, which is required for the optimality result. After selecting MAT m_1, it inserts in a queue Q, three control state pairs $(1a, 2b), (Ja, 2b), (Ja, 1b)$ corresponding to the *begin* and the *end* pairs of the transactions in m_1. These correspond to the three corners of the rectangle m_1. In the next step, it pops out the pair $(1a, 2b) \in Q$, selects MAT m_2 using the same priority rule, and inserts three more pairs $(1a, 3b), (5a, 2b), (5a, 3b)$ in Q. Note that if there is no transition from a control state such as Ja, no MAT is generated from $(Ja, 2b)$. Also, if a pair such as $(2a, 2b)$ is unreachable, no MAT is generated from it. One may not insert such pair in the first place. The algorithm terminates when all the pairs in the queue (denoted as ● in Figure 3(a)) are processed.

Note that the order of pair insertion can be arbitrary, but the same pair is never inserted more than once. For the running example, a set $\mathcal{MAT}_{ab} = \{m_1, \cdots m_7\}$ of seven MATs is generated. Each MAT is shown as a rectangle in Figure 3(a). The total number of context switches allowed by the set, i.e., $TP(\mathcal{MAT}_{ab})$ is 12.

A run of `GenMAT'`: We present a run of `GenMAT'` (**NEW**) in Figure 5(b) for the same running example. The table columns have similar description. In the second iteration, starting from the pair $(1a, 2b)$, the infeasible MAT $(1a \cdots 5a, 2b \cdots 3b)$ is ignored as the interleaving $2a \cdots 3b \cdot 1a \cdots 5a$ is infeasible. As $(1a, 3b)$ is no longer in Q, m_4 is not generated (which is infeasible). Similarly, as $(5a, 3b)$ is no longer in Q, m_5 is not generated (which is feasible). There are 5 MATs m_1, m_2', m_3, m_6, m_7 generated, shown as rectangles in Figure3(b). The total number of context switching allowed by the set is 8.

Algorithm 1 `GenMAT`': Obtain a set of MATs

```
1:  input: Thread Models: $M_1 \cdots M_n$; Dependency Relation $D$
2:  output: $\mathcal{MAT}$
3:  for all pairs of thread ($M_i$,$M_j$) do
4:      $\mathcal{MAT}_{ij} := \emptyset$; $Q := \{(\vdash_i, \vdash_j)\}$; $Q' := \emptyset$ {Initialize Queue};
5:      while $Q \backslash Q' \neq \emptyset$ do
6:          Select $(f_i, f_j) \in Q \backslash Q'$
7:          $Q := Q \backslash \{(f_i, f_j)\}$; $Q' := Q' \cup \{(f_i, f_j)\}$
8:          if OLD MAT-candidates set, $\mathcal{M}_c$ = { $m$ |$m$ is MAT from $(f_i, f_j)$} [26]
9:          if NEW MAT-candidates set, $\mathcal{M}_c$ = {$m$ |$m$ is feasible MAT from $(f_i, f_j)$}
10:         Select a MAT $m = (tr_i = f_i \cdots l_i, tr_i = f_j \cdots l_j) \in \mathcal{M}_c$ such that
11:                 $\forall m' = (tr_i', tr_j') \in \mathcal{M}_c, m' \neq m \; |tr_j| < |tr_j'|$, (i.e., $M_j$ has higher priority).
12:         $\mathcal{MAT}_{ij} := \mathcal{MAT}_{ij} \cup \{m\}$
13:         if $(+l_i = \dashv_i \wedge +l_j = \dashv_j)$ then continue;
14:         elseif $(+l_i = \dashv_i)$ then $q := \{(f_i, +l_j)\}$;
15:         elseif $(+l_j = \dashv_j)$ then $q := \{(+l_i, f_j)\}$;
16:         else $q := \{(+l_i, +l_j), (+l_i, f_j), (f_i, +l_j)\}$;
17:         $Q := Q \cup q$;
18:     end while
19:     $\mathcal{MAT} := \mathcal{MAT} \cup \mathcal{MAT}_{ij}$
20: end for
21: return $\mathcal{MAT}$
```

#i	p∈Q\Q'	$\mathcal{MAT}_{ab}$	Q\Q'
			(1a,1b)
1	(1a,1b)	m1:(1a⇒Ja,1b⇒2b)	**(1a,2b)**
2	(1a,2b)	m2:(1a⇒5a,2b⇒3b)	**(5a,2b)(1a,3b)(5a,3b)**
4	(5a,2b)	m3:(5a⇒Ja,2b⇒Jb)	
3	(1a,3b)	m4:(1a⇒2a,3b⇒6b)	**(1a,6b)(2a,6b)**
5	(5a,3b)	m5:(5a⇒Ja,3b⇒Jb)	
6	(1a,6b)	m6:(1a⇒Ja,6b⇒Jb)	
7	(2a,6b)	m7:(2a⇒Ja,6b⇒Jb)	

#I	p∈Q\Q'	$\mathcal{MAT}_{ab}$	Q\Q'
			(1a,1b)
1	(1a,1b)	m1:(1a⇒Ja,1b⇒2b)	**(1a,2b)**
2	(1a,2b)	m2':(1a⇒5a,2b⇒6b)	**(5a,2b)(1a,6b) (5a,6b)**
3	(5a,2b)	m3:(5a⇒Ja,2b⇒Jb)	
4	(1a,6b)	m6:(1a⇒Ja,6b⇒Jb)	
5	(5a,6b)	m7:(5a⇒Ja,6b⇒Jb)	

Fig. 5. (a) Run of (a) `GenMAT` and (b) `GenMAT'` on example in Figure 1

B Appendix: MAT Reduction Theorem

Let $\mathcal{MAT}$ and $\mathcal{MAT}'$ be the set of MATs obtained using `GenMAT` and `GenMAT'`, respectively.

Theorem 1 (MAT reduction) *$\mathcal{MAT}'$ is adequate, and $TP(\mathcal{MAT}') \subseteq TP(\mathcal{MAT})$.*

Proof. Consider a pair of threads M_a and M_b such that the chosen priority of M_a is higher than M_b. Let (a_1, b_1) be a pair picked at line 6, and the corresponding MAT selected by `GenMAT` be $m_1 = (ta_1, tb_1)$. `GenMAT` algorithm then inserts pairs (a_2, b_1), (a_1, b_2), and (a_2, b_2) in the worklist Q, shown as • in Figure 6(a). Assume that tb_1 disables ta_1, i.e., $tb_1 \cdot ta_1$ is an infeasible interleaving, and rest are feasible interleaving. Thus, m_1 is an infeasible MAT. Continuing the run of `GenMAT`, we have the following MAT

- $m_2 = (ta_1, tb_2 \cdot tb_3)$ from the pair (a_1, b_2),
- $m_3 = (ta_2, tb_1 \cdot tb_2)$ from the pair (a_2, b_1),
- $m_4 = (ta_2, tb_2)$ from the pair (a_2, b_2).

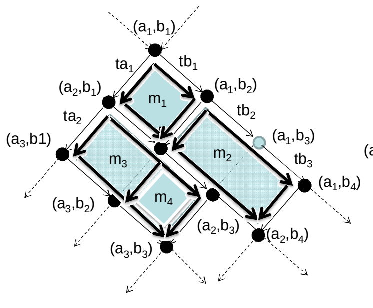
Note, since tb_1 disables ta_1, there exists some $tb_2 \cdot tb_3$ that enables ta_1, such that its last transition have a conflicting access with that of ta_1. (If not, one observe that any interleaving of the form $tb_1 \cdots tb_j \cdot ta_1$ is infeasible. In that case we will not have m_2). Also, since M_a is prioritized higher, we have the MAT m_3 with $|tb_2| \geq 0$. The context switching allowed by MATs $m_1 \cdots m_4$ are

$$TP(\{m_1, m_2, m_3, m_4\}) = \\ \{(b_2, a_1), (a_2, b_1), (a_2, b_2)(b_4, a_1)(a_3, b1)(b_3, a_2)(a_3, b_2)\}.$$

Now we consider the corresponding run of `GenMAT'` from (a_1, b_1) where only feasible MATs are generated. Such a run would produce MATs

- $m'_1 = (ta_1, tb_1 \cdot tb_2 \cdot tb_3)$ from the pair (a_1, b_2),
- $m_3 = (ta_2, tb_1 \cdot tb_2)$ from the pair (a_2, b_1).

The context switching allowed by MATs m'_1, m_3 are



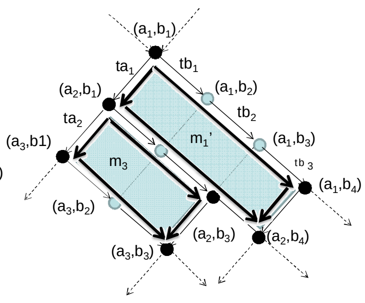


Fig. 6. MATs generated using (a) GenMAT and (b) GenMAT'

$TP(\{m'_1, m_3\}) = \{(a_2, b_1)(b_4, a_1), (a_3, b1)(b_3, a_2)\}$.

In the rest of the proof discussion, we consider the interesting case where $|tb_2| > 0$. (A similar proof discussion can be easily made for the other case $|tb_2| = 0$.) All the interleaving I_1-I_{11} (including the infeasible ones), as allowed by MAT m_1, m_2, m_3, m_4, are shown as follows:

$I_1:$	$\cdots ta_1 \cdot ta_2 \cdots$		
$I_2:$	$\cdots tb_1 \cdot tb_2 \cdot tb_3 \cdots$		
$I_3:$	$\cdots ta_1 \cdot ta_2 \cdot tb_1 \cdot tb_2 \cdots$		allowed by $\{m_3\}$
$I_4:$	$\cdots ta_1 \cdot tb_1 \cdot tb_2 \cdot ta_2 \cdots$		allowed by $\{m_1, m_3\}$
$I_5:$	$\cdots ta_1 \cdot tb_1 \cdot tb_2 \cdot tb_3 \cdots$		allowed by $\{m_1\}$
$I_6:$	$\cdots tb_1 \cdot tb_2 \cdot tb_3 \cdot ta_1 \cdots$		allowed by $\{m_2\}$
$I_7:$	$\cdots tb_1 \cdot ta_1 \cdot ta_2 \cdots$	(infeasible)	allowed by $\{m_1\}$
$I_8:$	$\cdots tb_1 \cdot ta_1 \cdot ta_2 \cdot tb_2 \cdots$	(infeasible)	allowed by $\{m_1, m_4\}$
$I_9:$	$\cdots tb_1 \cdot ta_1 \cdot ta_2 \cdots$	(infeasible)	allowed by $\{m_1\}$
$I_{10}:$	$\cdots tb_1 \cdot ta_1 \cdot tb_2 \cdots$	(infeasible)	allowed by $\{m_1, m_2\}$
$I_{11}:$	$\cdots tb_1 \cdot ta_1 \cdot tb_2 \cdot ta_2 \cdots$	(infeasible)	allowed by $\{m_1, m_2, m_4\}$

One can verify that all but infeasible interleavings, i.e., I_1-I_6, are also captured by m'_2 and m_3.

All the pairs that are inserted in Q are shown using • in the Figures 6(a)-(b). After the MATs $\{m_1, m_2, m_3, m_4\}$ are selected (by `GenMAT`), the following pairs in Q that are yet to be processed are

$$Q \backslash Q' = \{(a_3, b_1), (a_3, b_2), (a_3, b_3), (a_2, b_3), (a_2, b_4)(a_1, b_4)\}$$

Similarly, after the MATs $\{m'_1, m_3\}$ are selected (by `GenMAT'`), the following pairs in Q that are yet to be processed are

$$Q \backslash Q' = \{(a_3, b_1), (a_3, b_3), (a_2, b_3), (a_2, b_4)(a_1, b_4)\}.$$

Note that MAT from (a_3, b_2), as selected in `GenMAT`, allows exclusively an interleaving $\cdots tb_1 \cdot ta_1 \cdot ta_2 \cdots$; however such an interleaving is infeasible. For the remaining pairs we apply our argument inductively to show that from a control state pair, one can obtain a set of MATs from both `GenMAT` and `GenMAT′` respectively, that allow the same set of feasible interleaving. These arguments show the adequacy of our claim.

Further, `GenMAT′` inserts in the worklist a set of pairs that is a subset of pairs inserted by `GenMAT`. The claim $TP(\mathcal{MAT}') \subseteq TP(\mathcal{MAT})$ trivially holds as the worklist set is smaller with `GenMAT′` as compared to `GenMAT`. Thus, the interleaving space captured by $\mathcal{MAT}'$ is not increased. As $\mathcal{MAT}$ captures only representative schedules as per Theorem 1, clearly, $\mathcal{MAT}'$ captures only representative schedules. $\square$.