

Sound Transaction-based Reduction without Cycle Detection

Vladimir Levin¹, Robert Palmer², Shaz Qadeer³, and Sriram K. Rajamani³

¹ Microsoft vladlev@microsoft.com

² University of Utah rpalmer@cs.utah.edu

³ Microsoft Research {qadeer,sriram}@microsoft.com

Abstract. Partial order reduction is widely used to alleviate state space explosion in model checkers for concurrent programs. Traditional approaches to partial order reduction are based on *ample* sets. Natural ample sets can be computed for threads that communicate with each other predominantly through message queues. For threads that communicate with shared memory using locks for synchronization, Lipton's theory of reduction provides a promising way to aggregate several fine-grained transitions into larger transactions. In traditional partial order reduction, actions that are not in the ample set are delayed, thus avoiding the redundant exploration of equivalent interleaving orders. Delaying the execution of actions indefinitely can lead to loss of soundness. This is called the *ignoring problem*. The usual solution to the ignoring problem is by Cycle Detection. Explicit state model checkers usually use Depth First Search, and when a cycle is detected, disallow using a reduced ample set that closes the cycle.

The ignoring problem exists in transaction-based reduction as well. We present a novel solution to the ignoring problem in the context of transaction-based reduction. We designate certain states as *commit points* and track the exploration to discover whether the reduced exploration guarantees a path from each commit point to a state where the transaction is completed. If such a path does not exist, we detect this at the time a commit point is popped from the stack, and schedule all threads at the commit point. This paper presents our algorithm, called Commit Point Completion (CPC). We have implemented both CPC and Cycle Detection in the Zing model checker, and find that the CPC algorithm performs better.

1 Introduction

Partial order methods have been widely used as an optimization in building model checkers for concurrent software [1–6]. Traditional partial order reduction methods are based on the notion of *independence* between actions. Two actions α and β are independent if (1) they do not disable one another and (2) if both actions are enabled in a state s , then executing them in the order α followed by β from s , or in the order β followed by α from s , leads to the same resulting state. Partial order reduction algorithms explore a subset of enabled actions

in each state called the *ample* set. The set of all actions enabled in a state s is denoted $Enabled(s)$ and the ample set of actions in a state s is denoted $Ample(s)$. Obviously, $Ample(s) \subseteq Enabled(s)$. For partial order reduction to be sound, ample sets need to be chosen in such a way that a transition that is dependent on a transition in $Ample(s)$ cannot execute without a transition in $Ample(s)$ occurring first (see condition **C1** in [1] page 148). Choosing a minimal ample set satisfying **C1** is a very hard problem. In practice, ample sets are formed from local actions, and from restricted versions of send and receive actions, such as: sending to a queue, with the sender having exclusive rights of sending to the queue, and receiving from a queue, with the receiver having exclusive rights of receiving from the queue [3]. If the system consists of threads interacting via shared memory, Lipton's theory of reduction [7] provides an alternate way to do partial order reduction. Reduction views a transaction as a sequence of actions $a_1, \dots, a_m, x, b_1, \dots, b_n$ such that each a_i is a right mover and each b_i is a left mover. A right mover is an action that commutes to the right of every action by another thread; a left mover is an action that commutes to the left of every action by another thread. Thus, to detect transactions we need to detect right and left movers. Most programs consistently use mutexes to protect accesses to shared variables, we can exploit this programming discipline to infer left and right movers:

- The action `acquire(m)`, where m is a mutex, is a right mover.
- The action `release(m)` is a left mover.
- An action that accesses only a local variable or shared variable that is consistently protected by a mutex is both a left mover and a right mover.

A transaction is a sequence of right movers, followed by a *committing* action that is not a right mover, followed by a sequence of left movers. A transaction can be in two states: pre-commit or post-commit. A transaction starts in the pre-commit state and stays in the pre-commit state as long as right movers are being executed. When the committing action is executed, the transaction moves to the post-commit state. The transaction stays in the post-commit state as long as left movers are being executed until the transaction completes. In addition to being able to exploit programmer-imposed discipline such as protecting each shared variable consistently with the same lock, transaction-based reduction allows extra optimizations such as summarization [8].

Ignoring Problem. All partial order reduction algorithms work by delaying the execution of certain actions, thus avoiding the redundant exploration of equivalent executions. For instance, if thread t_1 executes an action from state s_1 that reads and writes only local variables, then thread t_2 does not need to be scheduled to execute in s_1 , and t_2 's scheduling can be delayed without losing soundness. For any interleaving that starts from s_1 and ends in a state where some thread t goes wrong, there exists an equivalent interleaving where the execution of t_2 is delayed at s_1 . However, unless we are careful, the scheduling of thread t_2 can be delayed indefinitely resulting in loss of soundness. This situation is called the *ignoring problem* in partial order reduction.

```

int g = 0;

void T1() {
L0:  g = 1;
L1:  skip;
L2:  while(true){
L3:    skip;
    }
L4:  return;
}

void T2() {
M0:  assert(g == 0);
M1:  return;
}

P = { T1() } || { T2() }

```

Fig. 1. Ignoring problem

Consider the example in Figure 1. The initial state of this program has two threads t_1 and t_2 starting to execute functions T1 and T2 respectively. The program has one global variable g , which has an initial value 0. A typical model checking algorithm first schedules t_1 to execute the statement at line L0, which updates the value of g to 1. Let us call this state s_1 . Since the next statement executed by thread t_1 from s_1 reads and writes only local variables of t_1 (namely its program counter) and does not read or write the global variables, partial order reduction algorithms delay execution of thread t_2 at state s_1 . Continuing, the while loop in lines L2 and L3 also reads and writes only the local variables of t_1 and thus execution of t_2 can be delayed during the execution of these statements as well. However, since reached states are stored, and a newly generated state is not re-explored if it is already present in the set of reached states, a fix-point is reached after executing the loop in T_1 once. Thus, the execution of t_2 is delayed indefinitely, and the reduction algorithm can be unsound, and say that the assertion in line M0 is never violated.

Most partial order reduction algorithms “fix” the ignoring problem by detecting cycles, and disallowing the actions of a thread to be ample when a cycle is “closed” (see condition **C3**, pages 150 and 158 in [1]). Since explicit-state model checkers usually use depth first search (DFS), cycle detection can be performed by detecting whether a newly generated state is already present in the DFS stack. In the SPIN model checker this is implemented using a bit in the hash table entry for reached states. This bit indicates whether the newly generated successor state is currently also on the depth first search stack.

Cycle detection is neither necessary nor sufficient for transaction-based reduction. Consider the variant of our current example in Figure 2. Here, we have introduced a nondeterministic choice in line L2 of procedure T1. In one branch of the nondeterministic choice, we have a while-loop with statements reading and writing only local variables of thread t_1 (lines L3-L4). The other branch of the nondeterministic choice just terminates the procedure. In this case, even without doing any cycle detection, since one branch of the nondeterministic choice terminates, a partial order reduction algorithm can schedule thread t_2 after procedure T1 terminates, and thus the assertion violation in line M0 can be detected. If we consider a variant of this example, where the entire “if” statement (from

```

int g = 0;

void T1() {
L0:   g = 1;
L1:   skip;
L2:   if (*) {
L3:       while(true){
L4:           skip;
L5:       }
L6:   return;
}

void T2() {
M0:   assert(g == 0);
M1:   return;
}

P = { T1() } || { T2() }

```

Fig. 2. Cycle detection is not necessary for transaction-based reduction

line L2 to L6 is replaced by `assume(false)` at line L2, some other mechanism in addition to cycle detection is needed to schedule the thread t_2 after t_1 executes the statement L1.

In the current literature on transaction-based reduction, the ignoring problem is addressed indirectly by disallowing certain types of infinite executions, such as those consisting of only internal hidden actions, within each thread (see Condition **C** from Section 4.2 in [9] which forbids the transaction from having infinite executions after committing, but without completing, and well-formedness assumption **Wf-ifinite-invis** from Section 4 in [10]). These assumptions do not hold in practice. In particular, when we analyze models that arise from abstractions (such as predicate abstraction) of programs, it is common to have loops with non-deterministic termination conditions, which violate the above assumptions. Thus, a more direct and computationally effective solution to the ignoring problem is required for wide applicability of transaction-based reduction. This paper presents a novel solution to this problem.

CPC Algorithm. We propose a new technique called *Commit Point Completion* (CPC) to solve the ignoring problem without cycle detection. We keep track of the state immediately after the committing action is executed, called the *commit point*. When a committed transaction completes, we simply mark the commit point as completed. When an unmarked commit point is about to be popped from the DFS stack, we schedule all threads from that state. Our insight is that we can delay the decision to forcibly end a transaction up to the time when commit point is about to be popped from the stack, avoiding taking such a decision pre-maturely when cycles are closed.

In the example from Figure 1 the state immediately after t_1 executes the statement at line L0 is a commit point. Due to the non-terminating while loop, the transaction that is committed here never completes. Thus, when this commit point is about to be popped from the DFS stack, it is unmarked, and the CPC algorithm schedules thread t_2 from this state, and the assertion violation in line M0 is detected. The example from Figure 2, has an identical commit point. However, since one nondeterministic branch completes the transaction, the commit point gets marked. Thus, when the commit point gets popped from the

```

Mutex m;
int x = 0; /* all accesses to x will be guarded by m*/
int y = 0; /* accesses to y are not guarded */

void T1() {
L0: acq(m);
L1: y := 42;
L2: x := 1;
L3: rel(m);
L4: while (true)
    { skip; }
}
P = { T1() } || { T2() } || { T3() }

void T2() {
M0: acq(m);
M1: assert(x == 0);
M2: rel(m);
}

void T3() {
N0: y = 10;
}

```

Fig. 3. CPC algorithm in the presence of left movers

DFS stack, the other thread t_2 is not scheduled. Note that the assertion failure at M0 is detected even without scheduling thread t_2 from the commit point, because t_2 will be scheduled by the reduction algorithm after the transaction in t_1 completes on one of the nondeterministic branches.

The above description of the CPC algorithm is simplistic. In the presence of left movers there may be more than one commit point for a transaction, and all of these commit points need to reach a state where the transaction completes to ensure sound reduction. For example, consider the example shown in Figure 3. In this example, there are two global variables x and y and one mutex m . All accesses to x are protected by mutex m , and are thus both movers. Accesses to y are unprotected, and are hence non-movers. Acquires of mutex m are right movers and releases are left movers as mentioned earlier. Thus, when thread T1 executes the assignment to y at label L1, its transaction commits, since the access to y is a non-mover. The resulting state, where y has just been assigned 42 and the program counter of the thread T1 is at L2 is a commit point. Due to the infinite while-loop at L4 this committed transaction never completes, and the CPC algorithm can schedule threads at the above commit point when it is about to be popped from the stack. However, for us to detect the assertion violation at line M1 of thread T2, another commit point needs to be established in T1 after the assignment to x at line L2. We handle this case by designating every state in a committed-transaction obtained by executing a “pure” left mover (i.e, a transaction that is a left mover but not a both-mover) as a commit point. Thus, in T1, the state after executing the release at line L3 is also designated as a commit point, and the algorithm schedules T2 when this state is about to be popped, leading to the assertion violation.

We have implemented the CPC algorithm in the Zing model checker at MSR. Section 5 presents experimental results that compare the CPC algorithm with a Cycle Detection algorithm for various Zing programs. The results clearly demonstrate that the CPC algorithm generally explores far fewer states than the Cycle Detection algorithm.

Outline. The rest of the paper is organized as follows. Section 2 introduces notations for describing multithreaded programs precisely. Section 3 gives an abstract framework for sound transaction-based reduction. Section 4 presents the CPC algorithm and a statement of its correctness. This section contains the core new technical results of the paper. Section 5 presents experimental results from the implementation of the CPC algorithm in the Zing model checker. Section 6 compares the CPC algorithm with related work, and Section 7 concludes the paper.

2 Multithreaded programs

The store of a multithreaded program is partitioned into the global store *Global* and the local store *Local* of each thread. We assume that the domains of *Local* and *Global* are finite sets. The set *Local* of local stores has a special store called *wrong*. The local store of a thread moves to *wrong* on failing an assertion and thereafter the failed thread does not make any other transitions.

$$\begin{aligned}
 t, u &\in \text{ \textit{Tid}} = \{1, \dots, n\} \\
 i, j &\in \text{ \textit{Choice}} = \{1, 2, \dots, m\} \\
 g &\in \text{ \textit{Global}} \\
 l &\in \text{ \textit{Local}} \\
 ls &\in \text{ \textit{Locals}} = \text{ \textit{Tid}} \rightarrow \text{ \textit{Local}} \\
 \text{ \textit{State}} &= \text{ \textit{Global}} \times \text{ \textit{Locals}}
 \end{aligned}$$

A multithreaded program (g_0, ls_0, T) consists of three components. g_0 is the initial value of the global store. ls_0 maps each thread id $t \in \text{ \textit{Tid}}$ to the initial local store $ls_0(t)$ of thread t . We model the behavior of the individual threads using two transition relations:

$$\begin{aligned}
 T_G &\subseteq \text{ \textit{Tid}} \times (\text{ \textit{Global}} \times \text{ \textit{Local}}) \times (\text{ \textit{Global}} \times \text{ \textit{Local}}) \\
 T_L &\subseteq \text{ \textit{Tid}} \times \text{ \textit{Local}} \times \text{ \textit{Choice}} \times \text{ \textit{Local}}
 \end{aligned}$$

The relation T_G models system visible thread steps. The relation $T_G(t, g, l, g', l')$ holds if thread t can take a step from a state with global store g and local store l , yielding (possibly modified) stores g' and l' . The relation T_G has the property that for any t, g, l , there is at most one g' and l' such that $T_G(t, g, l, g', l')$. We use functional notation and say that $(g', l') = T_G(t, g, l)$ if $T_G(t, g, l, g', l')$. Note that in the functional notation, T_G is a partial function from $\text{ \textit{Tid}} \times (\text{ \textit{Global}} \times \text{ \textit{Local}})$ to $(\text{ \textit{Global}} \times \text{ \textit{Local}})$. The relation T_L models thread local thread steps. The relation $T_L(t, l, i, l')$ holds if thread t can move its local store from l to l' on choice i . The nondeterminism in the behavior of a thread is captured by T_L . This relation has the property that for any t, l, i , there is a unique l' such that $T_L(t, l, i, l')$.

The program starts execution from the state (g_0, ls_0) . At each step, any thread may make a transition. The transition relation $\rightarrow_t \subseteq \text{ \textit{State}} \times \text{ \textit{State}}$ of thread t is the disjunct of the system visible and thread local transition relations defined below. For any function h from A to B , $a \in A$ and $b \in B$, we write $h[a := b]$ to denote a new function such that $h[a := b](x)$ evaluates to $h(x)$ if $x \neq a$, and to b if $x = a$.

$$\frac{T_G(t, g, ls(t), g', l')}{(g, ls) \rightarrow_t (g', ls[t := l'])} \quad \frac{T_L(t, ls(t), i, l')}{(g, ls) \rightarrow_t (g, ls[t := l'])}$$

The transition relation $\rightarrow \subseteq State \times State$ of the program is the disjunction of the transition relations of the various threads:

$$\rightarrow = \exists t. \rightarrow_t$$

3 Transactions

Transactions occur in multithreaded programs because of the presence of right and left movers. Inferring which actions of a program are right and left movers is a problem that is important but orthogonal to the contribution of this paper. In this section, we assume that right and left movers are available to us as the result of a previous analysis (see, e.g. [11]).

Let $RM, LM \subseteq T_G$ be subsets of the transition relation T_G with the following properties for all $t \neq u$:

1. If $RM(t, g_1, l_1, g_2, l_2)$ and $T_G(u, g_2, l_3, g_3, l_4)$, there is g_4 such that $T_G(u, g_1, l_3, g_4, l_4)$ and $RM(t, g_4, l_1, g_3, l_2)$.
2. If $T_G(u, g_1, l_1, g_2, l_2)$ and $RM(t, g_2, l_3, g_3, l_4)$, then for all g', l' ($T_G(t, g_1, l_3, g', l') \Rightarrow RM(t, g_1, l_3, g', l')$).
3. If $T_G(u, g_1, l_1, g_2, l_2)$ and $LM(t, g_2, l_3, g_3, l_4)$, there is g_4 such that $LM(t, g_1, l_3, g_4, l_4)$ and $T_G(u, g_4, l_1, g_3, l_2)$.
4. If $T_G(u, g_1, l_1, g_2, l_2)$ and $LM(t, g_1, l_3, g_3, l_4)$, there is g_4 such that $LM(t, g_2, l_3, g_4, l_4)$.

The first property states that a right mover of thread t commutes to the right of a transition of a different thread u . The second property states that if a right mover of thread t is enabled in the post-state of a transition of another thread u , and thread t is enabled in the pre-state, then the transition of thread t is a right mover in the pre-state. The third property states that a left mover of thread t commutes to the left of a transition of a different thread u . The fourth property states that a left mover that is enabled in the pre-state of a transition by another thread is also enabled in the post-state.

Our analysis is parameterized by the values of RM and LM and only requires that they satisfy these four properties. The larger the relations RM and LM , the longer the transactions our analysis infers. Therefore, these relations should be as large as possible in practice.

In order to minimize the number of explored interleaving orders and to maximize reuse, we would like to **infer** transactions that are as long as possible (i.e., they are maximal with respect to a given thread). To implement this inference, we introduce in each thread a boolean local variable to keep track of the phase of that thread's transaction. Note that this instrumentation is done automatically by our analysis, and not by the programmer. The phase variable of thread t is

true if thread t is in the right mover (or pre-commit) part of the transaction; otherwise the phase variable is false. We say that the transaction *commits* when the phase variable moves from true to false. The initial value of the phase variable for each thread is *false*.

$$\begin{aligned} p, p' \in \text{Boolean} &= \{\text{false}, \text{true}\} \\ \ell, \ell' \in \text{Local}^\# &= \text{Local} \times \text{Boolean} \\ \ell s, \ell s' \in \text{Locals}^\# &= \text{Tid} \rightarrow \text{Local}^\# \\ \text{State}^\# &= \text{Global} \times \text{Locals}^\# \end{aligned}$$

Let $\text{Phase}(t, (g, \ell s))$, the phase of thread t in state $(g, \ell s)$ be the second component of $\ell s(t)$.

The initial value of the global store of the instrumented program remains g_0 . The initial value of the local stores changes to ℓs_0 , where $\ell s_0(t) = \langle \ell s_0(t), \text{false} \rangle$ for all $t \in \text{Tid}$. We instrument the transition relations T_G and T_L to generate a new transition relation $T^\#$.

$$T^\# \subseteq \text{Tid} \times (\text{Global} \times \text{Local}^\#) \times \text{Choice} \times (\text{Global} \times \text{Local}^\#)$$

$$T^\#(t, g, \langle l, p \rangle, i, g', \langle l', p' \rangle) \stackrel{\text{def}}{=} \begin{cases} \vee T_G(t, g, l, g', l') \wedge \\ p' = (\text{RM}(t, g, l, g', l') \wedge (p \vee \neg \text{LM}(t, g, l, g', l'))) \\ \vee T_L(t, l, i, l') \wedge g = g' \wedge p' = p \end{cases}$$

In the definition of $T^\#$, the relation between p' and p reflects the intuition that if p is true, then p' continues to be true as long as it executes right mover transitions. The phase changes to false as soon as the thread executes a transition that is not a right mover. Thereafter, it remains false as long as the thread executes left movers. Then, it becomes true again as soon as the thread executes a transition that is a right mover and not a left mover. A transition from T_L does not change the phase. We overload the transition relation \rightarrow_t defined in Section 2 to represent transitions in the instrumented transition relation. Similar to the functional notation defined for T_G in Section 2, we sometimes use functional notation for $T^\#$.

Given an instrumented transition relation $T^\#$, we define three sets for each thread t : $\mathcal{R}(t), \mathcal{L}(t), \mathcal{N}(t) \subseteq \text{State}^\#$. These sets respectively define when a thread is executing in the right mover part of a transaction, the left mover part of a transaction, and outside any transaction. These three sets are a *partition* of $\text{State}^\#$ defined as follows:

$$\begin{aligned} - \mathcal{R}(t) &= \{ (g, \ell s) \mid \exists l. \ell s(t) = \langle l, \text{true} \rangle \wedge l \notin \{\ell s_0(t), \text{wrong}\} \}. \\ - \mathcal{L}(t) &= \left\{ (g, \ell s) \mid \begin{array}{l} \exists l. \ell s(t) = \langle l, \text{false} \rangle \wedge l \notin \{\ell s_0(t), \text{wrong}\} \wedge \\ (\exists i, g', l'. \text{LM}(t, g, l, g', l') \vee T_L(t, l, i, l')) \end{array} \right\}. \\ - \mathcal{N}(t) &= \text{State}^\# \setminus (\mathcal{R}(t) \cup \mathcal{L}(t)). \end{aligned}$$

The definition of $\mathcal{R}(t)$ says that thread t is in the right mover part of a transaction if and only if the local store of t is neither its initial value nor *wrong*

and the phase variable is true. The definition of $\mathcal{L}(t)$ says that thread t is in the left mover part of a transaction if and only if the local store of t is neither its initial value nor *wrong*, the phase variable is false, and there is an enabled transition that is either a left mover or thread-local. Note that since the global transition relation is deterministic, the enabled left mover is the only enabled transition that may access a global variable. Since $(\mathcal{R}(t), \mathcal{L}(t), \mathcal{N}(t))$ is a partition of $State^\#$, once $\mathcal{R}(t)$ and $\mathcal{L}(t)$ have been picked, the set $\mathcal{N}(t)$ is implicitly defined.

$$p = p_1 \xrightarrow{+}_{t(1)} p_2 \xrightarrow{+}_{t(2)} p_3 \cdots p_k \xrightarrow{+}_{t(k)} p_{k+1} = q_1 \xrightarrow{+}_{u(1)} q_2 \xrightarrow{+}_{u(2)} q_3 \cdots q_l \xrightarrow{+}_{u(l)} q_{l+1} = q$$

$$\underbrace{(p_2 = p_{2,1} \xrightarrow{+}_{t(2)} \cdots \xrightarrow{+}_{t(2)} p_{2,x} = p_3)}_{\text{}} \quad \underbrace{(q_2 = q_{2,1} \xrightarrow{+}_{u(2)} \cdots \xrightarrow{+}_{u(2)} q_{2,x} = q_3)}_{\text{}}$$

Fig. 4. A sequence of transactions.

A sequence of states from Figure 4 is called a *sequence of transactions* if

- for all $1 \leq m \leq k$, if $p_m = p_{m,1} \xrightarrow{+}_{t(m)} \cdots \xrightarrow{+}_{t(m)} p_{m,x} = p_{m+1}$, then (1) $p_{m,1} \in \mathcal{N}(t(m))$, (2) $p_{m,2}, \dots, p_{m,x-1} \in \mathcal{R}(t(m)) \vee \mathcal{L}(t(m))$, and (3) $p_{m,x} \in \mathcal{L}(t(m)) \vee \mathcal{N}(t(m))$.
- for all $1 \leq m \leq l$, if $q_m = q_{m,1} \xrightarrow{+}_{u(m)} \cdots \xrightarrow{+}_{u(m)} q_{m,x} = q_{m+1}$, then (1) $q_{m,1} \in \mathcal{N}(u(m))$, and (2) $q_{m,2}, \dots, q_{m,x} \in \mathcal{R}(u(m))$.

Intuitively, for every i , $p_i \xrightarrow{+}_{t(i)} p_{i+1}$ is a committed transaction and for every j , $q_j \xrightarrow{+}_{u(j)} q_{j+1}$ is an uncommitted transaction.

The following theorem says that for any sequence in the state space that reaches a state where some thread t goes wrong, there exists a corresponding sequence of transactions that reaches a corresponding state at which thread t goes wrong.

Theorem 1. *Let $P = (g_0, ls_0, T^\#)$ be the instrumented multithreaded program. For all $t \in Tid$, let $\mathcal{W}(t) = \{(g, ls) \mid \exists p. ls(t) = \langle wrong, p \rangle\}$. For any state $(g', ls') \in \mathcal{W}(t)$ that is reachable from (g_0, ls_0) , there is another state $(g'', ls'') \in \mathcal{W}(t)$ that is reachable from (g_0, ls_0) by a sequence of transactions.*

A detailed proof of this theorem can be found in our technical report [12]. As a consequence of this theorem, it suffices to explore only transactions to find errors. This is the basis for the our reduction algorithm. Using the values of $\mathcal{N}(t)$ for all $t \in Tid$, we model check the multithreaded program by computing the least fixpoint of the set of rules in Figure 5. This model checking algorithm schedules a thread only when no other thread is executing inside a transaction.

This algorithm is potentially unsound for the following reason. If a transaction in thread t commits but never finishes, the shared variables modified by this transaction become visible to other threads. However, the algorithm does not explore transitions of other threads from any state after the transaction commits. Section 4 presents a more sophisticated algorithm which ensures that all threads are explored from some state in the post-commit phase of every transaction.

$$\begin{array}{c}
\text{(INIT)} \\
\hline
\Sigma(g_0, \ell s_0) \\
\\
\text{(STEP)} \\
\frac{\forall u \neq t. (g, \ell s) \in \mathcal{N}(u) \quad \Sigma(g, \ell s) \quad T^\#(t, g, \ell s(t), i, g', \ell')}{\Sigma(g', \ell s[t := \ell'])}
\end{array}$$

Fig. 5. Model checking with unsound reduction.

4 Commit Point Completion

This section presents the CPC algorithm and its soundness theorem, which are the core new technical contributions of this paper. The algorithm uses Depth First Search (DFS). Each state in the DFS stack is encapsulated using a `TraversalInfo` record. In addition to the state, the `TraversalInfo` records the following 7 fields:

1. `tid`, the id of the thread used to reach the state,
2. `numTids`, the number of threads active in the state,
3. `choice`, the current index among the nondeterministic choices executable by thread `tid` in this state,
4. `LM`, a boolean which is set to true iff the action used to reach this state is a left mover,
5. `RM`, a boolean which is set to true iff the action used to reach this state is a right mover,
6. `Xend`, a boolean which is set to true iff the algorithm decides to schedule other threads at this state, and
7. `CPC`, a boolean which is relevant for only states with `phase` equal to false, and is set to true by the algorithm if there exists a path of transitions of the thread generating the state to a state where all threads are scheduled.

Figure 6 gives two variants of the CPC algorithm (with and without line L19). The statement at L4 peeks at the `TraversalInfo` `q` on top of the stack and explores all successors of the state using actions from thread `q.tid`. If the phase of `q` is false, then for each such successor `q'`, if the action used to generate `q'` is not a left-mover, then we update `q.Xend` to true at label L7. The invariant associated with the CPC flag is the following: If `q` is about to be popped from the stack and `q.CPC` is true and `Phase(q.tid, q.state)` is false then there exists a path to a state where `Xend` is true. Thus, at label L8 we set `q.CPC` to true if `q.Xend` is true. The `Xend` and `CPC` fields are also updated when a `TraversalInfo` is popped from the stack. In particular, at label L18, when `q` is about to be popped from the stack, if its phase is false and `q.CPC` is false, then we set `q.Xend` to true and force scheduling of all threads at `q`. If `q.Xend` is true, then at label L24 we ensure that all threads are scheduled from `q`. Figure 7 contains helper procedures for the CPC algorithm.

```

Hashtable table;
Stack stack;
TraversalInfo q, q', q'', pred;

stack = new Stack
table = new Hashtable

L0: q' = { state = (g0, ls0),
        tid = 1,
        numTids = 1,
        choice = 1,
        CPC = true,
        Xend = true,
        LM = false,
        RM = false }

L1: table.Add(q'.state, q')
L2: stack.Push(q')

L3: while (stack.Count > 0)
L4:   q = stack.Peek()
L5:   if (Enabled(q))
L6:     q' = Execute(q)
L7:     q.Xend = q.Xend || (¬Phase(q.tid, q.state) && ¬q'.LM)
L8:     q.CPC = q.CPC || q.Xend
L9:     if (IsMember(table, q'.state))
L10:      q'' = Lookup(table, q'.state)
L11:      q.CPC = q.CPC || q''.CPC
L12:     else /* undiscovered state */
L13:       table.Add(q'.state, q')
L14:       stack.Push(q')
L15:     end if
L16:     q.choice = q.choice + 1
L17:   else
L18:     q.Xend = q.Xend || (¬Phase(q.tid, q.state) && ¬q.CPC
L19:                        (* && ¬q.RM *))
L20:     q.CPC = q.CPC || q.Xend

L21:   stack.Pop()
L22:   pred = stack.Peek()
L23:   pred.CPC = pred.CPC || q.CPC

L24:   if (q.Xend && q.numTids < |Tid|)
L25:     q' = Update(q)
L26:     stack.Push(q')
L27:   end if
L28: end if
L29: end while

```

Fig. 6. CPC algorithm for sound reduction.

A key invariant preserved by the algorithm is the following: Suppose a `TraversalInfo` record q is about to be popped from the search stack and $q.CPC$ is true. Then there is a sequence of left mover transitions of thread $q.tid$ to a state represented in some `TraversalInfo` record q' such that $q'.Xend$ is true. We can show this by induction on the order in which `TraversalInfo` records are popped from the stack (See our technical report [12]).

Without the optimization in line L19, the CPC algorithm ensures that for every `TraversalInfo` record q explored by the algorithm such that $q.state$ is in the post-commit part of the transaction, there exists a sequence of transitions to some other state where all threads are scheduled. With the optimization in line L19, the CPC algorithm guarantees this property only for a subset of states in the post-commit part of the transaction that are reached by pure left movers as stated below.

Theorem 2. *Let q be a `TraversalInfo` constructed during the execution of the CPC algorithm such that $q.RM = false$. Then at line L21 there exists a sequence of left-mover transitions of thread $q.tid$ from $q.state$ to (g', ls') and all threads are explored from (g', ls') .*

Finally, Theorem 3 concludes that if there is a state in the multithreaded program where a thread goes wrong that is reachable from the initial state the CPC algorithm will find a state that is reachable from the initial state where that thread goes wrong.

Theorem 3. *If there is an execution of the multithreaded program from (g_0, ls_0) to (g, ls) and a thread t such that $ls(t) = wrong$, then there is another state (g', ls') where the CPC algorithm visits (g', ls') and $ls'(t) = wrong$.*

The proof involves using Theorem 1 to first produce a sequence of transactions that also reach a state where thread t goes wrong, and then using Theorem 2 to transform this latter sequence into another sequence that will be explored by the CPC algorithm. Details can be found in [12].

5 Experimental results

We implemented the CPC algorithm in Zing, which is a software model checker being developed in Microsoft Research. Table 1 gives the number of states explored by Zing on various example programs using three variants of the reduction algorithm. The column labeled “Loc” gives the number of lines of code in the Zing program. The column labeled “Unsound Reduction” gives the number of states explored by a reduction algorithm which does not solve the ignoring problem. This gives a lower bound on the number of states that need to be explored by any sound algorithm. The column labeled “CPC” gives the number of states explored by the CPC algorithm. The column labeled “Cycle Detection” gives the number of states explored by a sound algorithm which forcibly ends a transaction whenever a cycle is encountered in the post-commit part of the transaction.

Example	Loc	Unsound Reduction	CPC	Cycle Detection
AuctionHouse	798	108	108	108
FlowTest	485	4656	4656	4656
Shipping	1844	206	206	222
Conc	392	512	512	2063
Peterson	793	1080	1213	3427
Bluetooth	2768	48109	52092	116559
TransactionManager	6927	1220517	1264894	1268571
AlternatingBit	130	1180	1180	1349
Philosophers	76	87399	87399	428896
Bakery	104	10221	14935	14254

Table 1. Number of states visited by Unsound Reduction, CPC and Cycle Detection algorithms.

The number of states explored is a measure of the running time of the algorithm. The smaller the number of states explored by a sound algorithm, the faster the tool is.

The programs are classified into four groups. The first 3 programs, `AuctionHouse`, `FlowTest` and `Shipping` programs were produced by translating to Zing from a process co-ordination language called BPEL. They represent workflows for business processes, and have mostly acyclic state spaces. In these examples, the number of states explored by the all three algorithm are almost identical.

The next 3 programs `Conc`, `Peterson` and `Bluetooth` were produced by automatic abstraction refinement from concurrent C programs. We have adapted the SLAM toolkit [13] to concurrent programs by using Zing as a back-end model checker instead of Bebop. These examples all have loops that terminate non-deterministically in the abstraction. Thus, the cycle detection algorithm forces interleaving of all threads in these loops whereas the CPC algorithm avoids interleaving all threads in the loops without losing soundness. The CPC algorithm really shines in comparison with the Cycle Detection algorithm on these examples.

The `TransactionManager` program was obtained from a product group in Microsoft. It was automatically translated to Zing from `C#`, after a few manual abstractions and manually closing the environment. It is one of the larger Zing examples we currently have. Since the manual abstraction did not result in non-deterministically terminating loops, the CPC algorithm performs only marginally better than the Cycle Detection algorithm.

The final 3 programs, `AlternatingBit`, `Philosophers` and `Bakery` are standard toy examples used by the formal verification community. In the first two examples, CPC performs better than Cycle Detection. In the `Bakery` example we find that the Cycle Detection algorithm performs slightly better than the CPC algorithm. This is possible, since the total number of states is counted over all transactions, and the CPC algorithm gives optimality only within a single

transaction. Heuristically, this should translate to smaller number of states explored over all the transactions, but this example shows that this is not always the case.

Overall, the results clearly demonstrate that CPC is a good algorithm for making reduction sound, without forcing the interleaving of other threads in all loops. It generally explores fewer states than Cycle Detection, and out-performs Cycle Detection in examples with nondeterministic loops. Such examples arise commonly from automatic abstraction refinement.

6 Related work

Partial order reduction has numerous variants. The most commonly used ones are stubborn sets of Valmari [2], ample sets [4, 1], and sleep sets [5]. Most of these approaches handle the ignoring problem by using some variant of cycle detection. In another paper, Valmari proposes detecting Strongly Connected Components (SCCs) to solve the ignoring problem [14]. This algorithm from [14] involves detecting terminal strongly connected components, and forces scheduling of other threads from at least one state in each of the terminal strongly connected components (see Algorithm 1.28, Section 5 in [14]). In contrast, the CPC algorithm does not directly compute any strongly connected components. Also the CPC algorithm terminates transactions at fewer points than Valmari’s algorithm. See Appendix B for an example.

Transaction based reduction was originally developed by Lipton [7]. Work by Stoller and Cohen [10] uses a locking discipline to aggregate transitions into a sequence of transitions that may be viewed atomically. Flanagan and Qadeer augment this approach with right movers to get further reduction [9]. This idea is combined with procedure summarization by Qadeer, Rajamani, and Rehof in [8]. As mentioned earlier, all of these papers address the ignoring problem only indirectly by disallowing certain types of infinite executions, such as those consisting of only internal hidden actions, within each thread (see Condition **C** from Section 4.2 in [9] which forbids the transaction from having infinite executions after committing, but without completing, and well-formedness assumption **Wf-ifinite-invis** from Section 4 in [10]). It is not clear how these assumptions are enforced. Two of the above papers [9, 8] do not have any accompanying implementation, and it is unclear how the ignoring problem is solved in the implementation associated with [10]. Our guess is that they use some form of cycle detection.

The Verisoft [6] implementation does not use the detection of cycles or strongly connected components, rather a timeout is used to detect an infinite execution that is local to a particular process. Other cycles are broken by limiting the search depth or using a driver that generates a finite number of external events. Dwyer et al [15] use the notion of a locking discipline is used to increase the number of transitions that can form an ample set for a process. The algorithms presented use the standard cycle detection technique to insure soundness.

7 Conclusion

Partial order reduction methods with ample sets usually use Cycle Detection to solve the ignoring problem. In the context of transaction based reduction, we propose a new technique called Commit Point Completion (CPC) to solve the ignoring problem. We have proved that this algorithm is correct, and have implemented it in the Zing model checker. Our experimental results demonstrate that with transaction based reduction, the CPC algorithm performs better than Cycle Detection. Though the CPC algorithm was presented using the terminology of Lipton's transactions, we believe that the idea is applicable to other variants of partial order reduction as well. Exploration of this idea is left to future work. The ignoring problem also arises when we attempt to build summaries for multithreaded programs[8]. Though not mentioned here, our implementation of summaries in Zing also uses the core idea of the CPC algorithm to ensure soundness.

References

1. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
2. Valmari, A.: A stubborn attack on state explosion. In: CAV 91: Computer Aided Verification, Springer-Verlag (1991) 156–165
3. Holzmann, G., Peled, D.: An improvement in formal verification. In: FORTE 94: Formal Description Techniques, Chapman & Hall (1994) 197–211
4. Peled, D.: Partial order reduction: Model-checking using representatives. In: MFCS 96: Mathematical Foundations of Computer Science, Springer-Verlag (1996) 93–112
5. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. LNCS 1032. Springer-Verlag (1996)
6. Godefroid, P.: Model checking for programming languages using Verisoft. In: POPL 97: Principles of Programming Languages. (1997) 174–186
7. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. In: Communications of the ACM. Volume 18:12. (1975) 717–721
8. Qadeer, S., Rajamani, S.K., Rehof, J.: Summarizing procedures in concurrent programs. In: Principles of Programming Languages, ACM (2004) 245–255
9. Flanagan, C., Qadeer, S.: Transactions for software model checking. In: SoftMC 03: Software Model Checking Workshop. (2003)
10. Stoller, S.D., Cohen, E.: Optimistic synchronization-based state-space reduction. In: TACAS 03. LNCS 2619, Springer-Verlag (2003) 489–504
11. Flanagan, C., Qadeer, S.: Types for atomicity. In: TLDI 03: Types in Language Design and Implementation, ACM (2003) 1–12
12. Levin, V., Palmer, R., Qadeer, S., Rajamani, S.K.: Sound transaction-based reduction without cycle detection. Technical Report MSR-TR-2005-40, Microsoft Research (2005) <ftp://ftp.research.microsoft.com/pub/tr/TR-2005-40.pdf>.
13. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: POPL 02: Principles of Programming Languages, ACM (2002) 1–3
14. Valmari, A.: Stubborn sets for reduced state space generation. In: Advances in Petri nets. LNCS 483, Springer-Verlag (1990)
15. Dwyer, M.B., Hatcliff, J., Robby, Ranganath, V.P.: Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. Formal Methods in System Design **25** (2004) 199–240

A Helper functions for the CPC algorithm

```
Boolean Enabled(TraversalInfo q) {
  let (g, ℓs) = q.state in
  return (∃ g', ℓ'. T#(q.tid, g, ℓs(q.tid), q.choice, g', ℓ'))
}

TraversalInfo Execute(TraversalInfo q) {
  let (g, ℓs) = q.state in
  let (g', ℓ') = T#(q.tid, g, ℓs(q.tid), q.choice) in
  State succ = (g', ℓs[q.tid := ℓ'])
  return { state = succ,
          tid = q.tid,
          numTids = 1,
          choice = 1,
          CPC = false,
          Xend= false,
          LM = LM(q.tid, q.state, succ)
          RM = RM(q.tid, q.state, succ) }
}

TraversalInfo Update(TraversalInfo q) {
  Tid nextTid = ite((q.tid == |Tid|), 1, q.tid + 1)
  return { state = q.state,
          tid = nextTid,
          numTids = q.numTids + 1,
          choice = 1,
          CPC = q.CPC,
          Xend= q.Xend,
          LM = q.LM,
          RM = q.RM }
}
```

Fig. 7. Helper procedures for the CPC algorithm.

The helper functions for the CPC algorithm perform the following actions. `Enabled` determines whether the current thread has a transition enabled at a given state. `Execute` applies the transition relation $T^\#$ to the current state. `Update` schedules the next thread to run.

B Comparison with Valmari's SCC algorithm

Consider the example from Figure 8. In this example, a transaction commits at the state after executing line L0, followed by a non-deterministic branch at line

```

int g = 0;

void T1() {
L0:   g = 1;
L1:   skip;
L2:   if (*) {
L2:       while(true){
L3:           skip;
L4:       }
      }
      else {
L5:       while(true){
L6:           skip;
L7:       }
      }

L8:   return;
}

P = { T1() } || { T2() }

void T2() {
M0:   assert(g == 0);
M1:   return;
}

```

Fig. 8. Distinction between CPC algorithm and SCC-based algorithms

L2. Each of the branches produce terminal SCCs in the state space. Valmari's algorithm appears to force scheduling T2 at each of these terminal SCCs, whereas the CPC algorithm forces scheduling T2 only once, at the commit-point (label L1).