Implied Set Closure and Its Application to Memory Consistency Verification

Surender Baswana, Shashank K. Mehta, and Vishal Powar

Indian Institute of Technology, Kanpur - 208016, India,
{sbaswana,skmehta,vishalp}@cse.iitk.ac.in

Abstract. Hangal et. al. [3] have developed a procedure to check if an instance of the execution of a shared memory multiprocessor program, is consistent with the Total Store Order (TSO) memory consistency model. They also devised an algorithm based on this procedure with time complexity $O(n^5)$, where n is the total number of instructions in the program. Roy et. al. [6] have improved the implementation of the procedure and achieved $O(n^4)$ time complexity.
We have identified the bottleneck in these algorithms as a graph problem of independent interest, called *implied-set closure* (ISC) problem. In this paper we propose an algorithm for ISC problem and show that using this algorithm, Hangal's consistency checking procedure can be implemented with $O(n^3)$ time complexity. We also experimentally show that the new algorithm is significantly faster than Roy's algorithm.

Keywords Memory consistency model verification, Incremental transitive closure, Total store order, Shared memory multi-processor

1 Introduction

Modern processors aggressively employ chip multiprocessing and simultaneous multi-threading to achieve high processor performance. Traditionally memory being the slower subsystem various techniques, such as hierarchical implementation of the memory, have been employed to reduce the bottleneck. With the fast multiprocessors, modern architectures exploit new techniques to improve the memory performance. Foremost of these techniques is executing the memory instructions out of program-order according to a predetermined consistency model.

A memory consistency model specifies the restriction on the order in which memory instructions may be executed. Commercial architectures support a variety of memory consistency models. The strictest of these is sequential-consistency (SC), which requires that all the instructions of one processors must be executed in their program-order. An execution is valid in this model if and only if the global order of the instructions result from interleaving the processor-programs. Relaxing the restrictions progressively lead to Total-Store-Order (TSO) and Release Consistency (RC), these along with further relaxed models are given in [3].

The problem of *memory consistency verification* is to ensure that a given architecture always executes every program in accordance to the memory consistency model adopted in its design. This problem is shown to be NP complete by Catlin et. al. [2]. With increasingly relaxed consistency model the verification of compliance becomes extremely difficult. Adve and Gharachorloo [1] discuss many issues related to these models and their implementation.

A more practical approach, adopted in industry, is to execute a test program and then verify from its execution trace that indeed it was executed in accordance with the consistency model. This approach can never prove that the design is error free but a large number of tests can give significant confidence. The test programs are multi-threaded programs of memory instructions: Store, Load, Memorybar etc. We call each thread a *processor-program*.

In this paper we address the problem of analyzing the execution trace. Hangal et. al. [3] have developed TSOtool for Sun Microsystem's processors to analyze the outcome of test programs run under TSO model. Their algorithm has $O(n^5)$ time complexity and $O(n^2)$ space complexity, where n is the number of instructions in the test program. The basic procedure is general enough to be applicable to other memory consistency models.

Roy et. al. [6] have developed Intel MPRIT Tool for the same task by improving the general algorithm by Hangal, reducing the time complexity to $O(n^4)$. Once again, this algorithm can be applied to any consistency model without change in the time complexity.

Manovit and Hangal [5] have also implemented the procedure of [3] with time complexity $O(k.n^3)$ where k is the number of processors. This implementation crucially depends on the total order on Store instructions in TSO. Therefore their approach cannot be applied to consistency model other than TSO.

Our contribution in this paper is to identify a graph problem called *implied-set-closure*(ISC) which is the abstraction of the bottleneck of the above mentioned general high-level algorithm. We present an efficient incremental algorithm for ISC problem based on the incremental transitive closure algorithm by Italiano [4]. The application of this algorithm reduces the time complexity of the general memory consistency verification algorithm to $O(n^3)$, which is a significant improvement over the $O(n^4)$ bound achieved by the previous best algorithm of Roy et al. [6]. The space requirement of the algorithm remains $\Theta(n^2)$. Other salient feature of our algorithm is its compact and simple description and no hidden constant, which makes it an ideal candidate to be a practical algorithm (a quality not possessed by many theoretically efficient algorithms). We also compared our algorithm with the algorithm of Roy et al. [6] experimentally in an identical computing environment. In MPRIT, the algorithm of Roy et. al. is implemented using vector instructions. For our experiment we implemented both algorithms without vector instructions. The results show that the new algorithm outperforms their algorithm in real time.

The remainder of the paper is organized as follows. In the following section, we reproduce the description of the memory consistency verification problem and its algorithm given in [3, 6]. In Section 3 a graph problem, which we refer by

the name *implied set closure* (ISC) problem, is formally defined and its relation to the memory consistency problem is established. In Section 4, we develop an algorithm for ISC, and also argue about the optimality of its theoretical time complexity. In Section 5, experimental results comparing the performance of the new algorithm to the previous best algorithm are presented.

2 Formal description of TSO model and consistency verification algorithm

We reproduce here the formal description of TSO model and the high level consistency verification algorithm presented in [3]. The axiomatic description of the model is originally borrowed from Sindhu et. al. [7]. The model used by [6] is slightly generalized which will also be covered in the following discussion.

A Load instruction is considered executed when the issuing processor receives the data, while a Store instruction is considered executed when it is visible to all processors in the system. It is assumed that each instruction eventually gets executed, i.e., no instruction takes infinite time to complete. The notations used here are as follows.

L_a^i	a *Load* from location a by processor i
S_a^i	a *Store* to location a by processor i
$\lfloor L_a^i; S_a^i \rfloor$	an *Atomic operation* to location a by processor i
$Val[L_a^i]$	the value read by L_a^i
$Val[S_a^i]$	the value written by S_a^i
O_a^i	either a L_a^i or a S_a^i
$\mathcal{S}(L_a^i)$	the store instruction S_a^j s.t. $Val(S_a^j) = Val(L_a^i)$ (it is assumed that each Store instruction in the test program stores a unique value so $\mathcal{S}(L_a^i)$ is well defined.)

There are two partial-ordering relations defined over the set of all memory instructions: local ';' and global '$\leq$'. $x; y$ iff x and y are in the same processor-program with x before y; and $x \leq y$ iff it is required by TSO axioms or by the data dependency, that instruction x must be executed before y. In the latter ordering x and y may belong to different processor-programs. Now we present the axioms of total-store-order (TSO) memory consistency model.

Total Store Order $\forall S_a^i, S_b^j : (S_b^j \leq S_a^i \vee S_a^i \leq S_b^j)$.

Atomic Operation $\lfloor L_a^i; S_a^i \rfloor \Rightarrow (L_a^i \leq S_a^i) \wedge (\forall S_b^j : S_b^j \leq L_a^i \vee S_a^i \leq S_b^j)$.

Value Coherence $Val[L_a^i] = Val[\max_{\leq}[\{S_a^j : S_a^j \leq L_a^i\} \cup \{S_a^i : S_a^i; L_a^i\}]]$.

Local Ordering This tells when the local ordering must be preserved in the global ordering. We first present Roy et.al.'s [6] version of the axiom. If $O_a^i; O_b^i$ then subject to various criteria depending on the the type of the operations (load or store) and the type of the locations a, b (write-back, write-through, write protected, uncacheable, uncacheable speculative write combine), we require $O_a^i \leq O_b^i$. The criteria can be expressed by a function $f : (\{load, store\} \times \{location\text{-}types\})^2 \rightarrow \{0, 1\}$ and state the axiom as
$(O_a^i; O_b^i) \wedge (f(type(O_a^i), type(a), type(O_b^i), type(b)) = 1) \Rightarrow (O_a^i \leq O_b^i)$.

This axiom is stated in a restricted way in [3] where a specific f is assumed, namely, $f(i_1, l_1, i_2, l_2) = 1$ iff $i_1 \neq$ store or $i_2 \neq$ load.

Axiomatic description of other memory consistency models are similarly described in [7].

2.1 Consistency verification problem

TSOtool developed by Hangal et. al. [3] generates a test program, for a multiprocessor system (consisting only of memory instructions), executes it on the system or a system simulator, and then analyzes the outcome of the run to check if it is consistent with the axioms of the TSO model. In order to carry out the analysis each Store instruction stores a unique value. It allows to determine $\mathcal{S}(L_a^i)$ for each L_a^i unambiguously. Their main contribution is the consistency checking algorithm. We will present the high level description of the algorithm in this section. The implementation of this algorithm has $O(n^5)$ time complexity, where n is the total number of (memory) instructions included in all processor-programs. Roy at. al. [6] improved the implementation leading to the time complexity $O(n^4)$. In the following sections we will show that it can be further improved to $O(n^3)$.

Algorithm 1 analyzes the program output by computing a directed graph (V, E) based on the outcome of the program. The nodes, V, of this graph are the instructions of the program. An edge is placed from node O_1 to O_2 if $O_1 \leq O_2$. If the graph contains a cycle, i.e., $\leq$ is not found to be a partial ordering, then we can conclude that the requirement of the consistency model must have been violated in the execution.

We reproduce here the rules of including the edges in the graph from [3, 6] which incorporate the TSO axioms.

Static Edges: This rule is due to the local ordering axiom.

R_1: If $f(type(O_a^i), type(a), type(O_b^i), type(b)) = 1$, then include edge (O_a^i, O_b^i).

The edges due to this rule can be determined from the program itself and they are independent of the outcome of the run. Assuming that the function can be evaluated in $O(1)$ time, the static edges can be computed in $O(n^2)$ time.

Observed Edges: These rules are direct implication of the Value-Coherence and the Total-Store axioms.

R_2: If $(\mathcal{S}(L_a^i) = S_a^j) \wedge (i \neq j)$, then add the edge (S_a^j, L_a^i).

R_3: If $(\mathcal{S}(L_a^i) = S_a^j) \wedge (S_a'^i; L_a^i)$, then add the edge $(S_a'^i, S_a^j)$

The first part of these conditions is decided by finding S such that $Val(L) = Val(S)$ for a given L. This is because each Store instruction stores a unique value. Thus these edges can be computed only after the outcome of the run is known. The second part of the conditions only depend on the program. Hence it is easy to see that the observed edges can be computed in $O(n^2)$ time.

Inferred Edges: These rules are the indirect implications of Value Coherence axiom.

R_4: If $(\mathcal{S}(L_a^i) = S_a^j) \wedge (S_a'^k \leq L_a^i) \wedge (S_a'^k \neq S_a^j)$, then add the edge $(S_a'^k, S_a^j)$.

R_5: If $(\mathcal{S}(L_a^i) = S_a^j) \wedge (S_a^j \leq S_a'^k)$, then add the edge $(L_a^i, S_a'^k)$.

The conditions in this case, unlike in the earlier cases, depend on the structure (edges) of the graph. As the new inferred edges are added to E new pairs of vertices $L_a^i, S_a'^k$ or $S_a'^k, S_a^j$ may satisfy the respective precondition and be eligible for an edge between them. Thus inferred edges need to be computed iteratively. The computation of the inferred edges is the bottleneck in the performance of the algorithm, so the complexity of Algorithm 1 is determined by steps 9 and 12. In Sections 3 and 4 we describe an efficient solution for these steps.

2.2 Example

We present an example borrowed from [3] to show how above described rules detect inconsistency.

Let $S[X]\#n$ denote a store instruction which stores n in location X; and $L[X] = n$ denote a load instruction which loads value n from location X. In this example we assume that $f(i_1, l_1, i_2, l_2) = 1$ iff $i_1 \neq$ store or $i_2 \neq$ load. Consider the following 4-thread program along with the relevant information from an execution.

P_1	P_2	P_3	P_4
$S[B]\#91$	$S[A]\#2$	$S[B]\#92$	$L[B] = 92$
$S[A]\#1$		$L[A] = 2$	$L[B] = 91$
$L[A] = 2$		$L[B] = 92$	

In Figure 1 we show the directed graph generated using the rules. All edges due to rules R_1, R_2 and R_3 are easy to deduce from the rules. Here is the explanation for the two R_4 edges. Since $S[B]\#91 \leq S[A]\#1 \leq S[A]\#2 \leq L[A] = 2 \leq L[B] = 92$. So from R_4 there must be an edge from $S[B]\#92$ to $S[B]\#91$. We also have $S[B]\#92 \leq L[B] = 92 \leq L[B] = 91$. Again from R_4, there should be an edge from $S[B]\#91$ to $S[B]\#92$. These edges from a cycle so we conclude that the TSO model is violated in this execution.

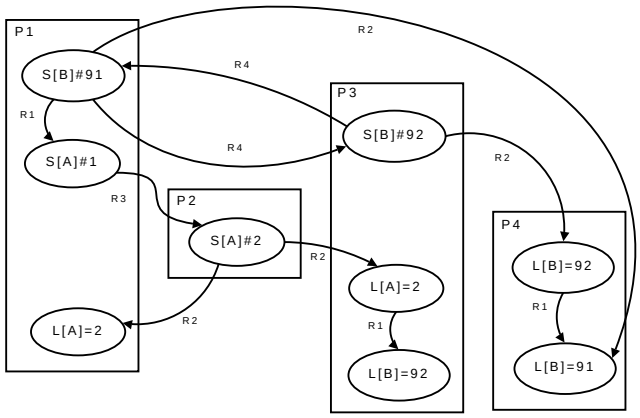

Fig. 1. Edges due to application of rules

2.3 Limitation of Algorithm 1

We have seen that a cycle in the graph implies that TSO model is violated but the converse is not true.

The TSO requires that all Stores must be totally ordered. There may be some cases where insufficient data dependence information is present to totally order all the Stores. To remain true to TSO, all total orders must be considered which are consistent with the partial order on stores determined by the data dependence. This would make the worst case complexity of the algorithm exponential [3]. The procedure chooses to ignore this lacunae in an attempt to trade off accuracy for reasonable analysis time.

All the three algorithms that implement this procedure are designed with this limitation. As explained above, the Store instructions do not get totally ordered in the global ordering (as required by TSO) only when there is insufficient data dependence. This situation arises when different threads do not have sufficient interaction. In practice this situation can easily be resolved by having Load instruction for each location in several threads. Therefore these algorithms are extremely useful in spite of the incompleteness.

3 Implied-Set Closure (ISC) problem

Definition 1. *Given a directed graph* $G = (V, E_0)$ *and and a mapping* $T : V \times V \rightarrow powerset(V \times V)$. *Then the* implied-set closure *of* G *is* $G^{ISC} = (V, E_0^{ISC})$ *where* E_0^{ISC} *is the smallest edge set such that*
(i) $E_0 \subset E_0^{ISC}$,
(ii) if E_0^{ISC} *has a (directed) path from* a *to* b, *then* $T(a, b) \subseteq E_0^{ISC}$ $\forall a, b \in V$.

If any two edge sets satisfy the above conditions, then their intersection also satisfies the same. Therefore there is a unique smallest set satisfying the conditions. Observe that if $T(a, b) = \{(a, b)\}$, then ISC problem reduces to transitive closure problem.

3.1 Relation to Memory Consistency Verification Problem

The computation of the edges generated by the rules R_4 and R_5 is an instance of the computation of implied-set closure problem as explained below.

Let E_0 be the edge set E in Algorithm 1 after step 8. Define (i) $T(S_a^k, L_a^i) = \{(S_a^k, \mathcal{S}(L_a^i)\}$ for every S_a^k and L_a^i such that $S_a^k \neq \mathcal{S}(L_a^i)$, (ii) $T(S_a^j, S_a^k) = \{(L_a^i, S_a^k) : \mathcal{S}(L_a^i) = S_a^j\}$, (iii) $T(x, y) = \emptyset$ for all the remaining ordered pairs (x, y). Then the final set E in Algorithm 1, computed after step 14, is E_0^{ISC} with respect to the T mapping described above.

4 Algorithm for ISC problem

We present an algorithm for implied-set closure problem which is based on incremental algorithm for transitive closure proposed by Italiano [4].

Data: Sequence of memory-instructions for each processor (Swap is considered both, a Load and a Store instruction), value associated with each Load/Store instruction. V denotes the set of all instruction

Result: It outputs *true* if the execution of the program obeys all TSO axioms, else outputs *false*

```
1  for each L ∈ V do
2  |  if Val[L] = Val[S] then
3  |  |  S(L) = S
4  |  end
5  end
6  initialize graph (V, E) to (V, ∅);
   /* R1 :                                                    */
7  Add edges to E according to R1;
   /* R2, R3 :                                                */
8  Add edges to E according to R2 and R3;
   /* R4 :                                                    */
9  while ∃L, S' such that S'_a ≠ S(L_a) = S_a and (S'_a, L_a) ∈ E do
10 |  Add edge (S'_a, S_a) to E;
11 end
   /* R5 :                                                    */
12 while ∃L_a, S'_a such that S'_a ≠ S(L_a) = S_a and (S_a, S'_a) ∈ E do
13 |  Add edge (L_a, S'_a) to E;
14 end
15 if E contains a cycle then
16 |  declare that TSO violation found;
17 end
18 else
19 |  declare no violation found;
20 end
```

Algorithm 1: Algorithm to analyze a program output for TSO violation

4.1 An incremental algorithm for ISC problem

Algorithm 2 computes the implied-set closure of a set of directed-edges $E_0 \subseteq V \times V$. For convenience we shall denote the existence of a directed path from a to b in graph H, in the algorithm, by $a \rightsquigarrow b$ and the transitive closure of H by H^c.

The correctness of the algorithm can be established by observing that at the end of each iteration of the *while*-loop following four assertion are always true: (i) $E_0 \subseteq H \cup X$, (ii) $HC = H^c$, (iii) For each pair of vertices u, v, if $u \rightsquigarrow v$ in H, then $T(u, v) \subseteq H \cup X$, (iv) $H \cup X \subseteq E_0^{ISC}$, where E_0^{ISC} is the implied-set-closure of E_0. Condition (iii) is equivalent to: for each pair of vertices u, v, if $(u, v) \in HC$, then $T(u, v) \subseteq H \cup X$.

The algorithm terminates since H grows monotonically. On termination, X is empty so the loop invariant conditions imply that finally $H = E_0^{ISC}$.

In the next step we will show how to efficiently compute the incremental transitive closure of step 8 of Algorithm 2.

Data: vertex set V; edge set E_0; implied-edge-sets $T(a,b)$ for all pairs $(a,b) \in V \times V$
Result: $H = E_0^{ISC}$ and $HC = H^c$ (transitive closure of H)

$H \leftarrow \emptyset$;
$HC \leftarrow \emptyset$;
$X \leftarrow E_0$;
while $X \neq \emptyset$ **do**
  $(x,y) \leftarrow Select(X)$;
  `/* pick an arbitrary edge` (a,b) `from` X `and delete it from the set */`
  $H' \leftarrow H \cup \{(x,y)\}$;
  if $(x,y) \notin HC$ **then**
    $HC' \leftarrow (HC \cup \{(x,y)\})^c$;
  end
  else
    $HC' = HC$;
  end
  `/* here superscript 'c' denotes transitive-closure */`
  for *each* $(u,v) \in (HC' - HC)$ **do**
    $X \leftarrow X \cup (T(u,v) - H')$;
  end
  $H \leftarrow H'$;
  $HC \leftarrow HC'$;
end
return H;

Algorithm 2: An incremental algorithm to compute implied-set closure of E_0

4.2 Improved Algorithm for ISC problem

In order to update the transitive closure of the graph upon insertion of an edge, the algorithm uses following observation to minimize the computation required. Let (x,y) be an edge added to H. If y was already reachable from x, then the transitive closure of the graph will remain unchanged. Otherwise, transitive closure needs to be updated. In particular, we need to add the edges implied by the transitivity in context of all those vertices w such that $w \rightsquigarrow x \wedge w \not\rightsquigarrow y$ prior to insertion of edge (x,y) because vertex reachable from y has subsequently become reachable from w too. In order to update the transitive closure for each such vertex w, a simple way is to scan all vertices reachable from y. This will require $O(n)$ work per vertex and leads to $O(n^4)$ time algorithm as designed by Roy et al. [6]. However, note that it would suffice if we can efficiently compute only those vertices which are reachable from y but not reachable from w (prior to the current edge insertion). Let us denote this set by $\mathcal{D}(w,y)$. The following Lemma would pave the way for its efficient computation, and hence updating the transitive closure.

Lemma 1. *For each vertex $v \in \mathcal{D}(w,y)$, and any path $\mathcal{P}$ from y to v, each vertex lying on $\mathcal{P}$ is also present in $\mathcal{D}(w,y)$.*

The set $\mathcal{D}(w, y)$ is a subset of the set of vertices reachable from y, and we know that the latter can be computed by performing DFS (or BFS) traversal in the graph starting from y. In order to compute $\mathcal{D}(w, y)$ efficiently, it would suffice to perform a *bounded* DFS traversal in the graph starting from y wherein we extend DFS recursively only along those vertices which were not reachable from w prior to insertion of the edge (x, y). This is because, as follows from Lemma 1, the DFS traversal pursued from a vertex already reachable from w won't lead to any vertex of set $\mathcal{D}(w, y)$, and so there is no point extending DFS traversal beyond such vertices.

Algorithm 3 is the bounded depth-first search based procedure to update transitive closure for a vertex w upon insertion of an edge. This algorithm implicitly computes $\mathcal{D}(w, y)$.

```
1 HC ← HC ∪ (w, y);
2 for each (y, z) ∈ H do
3 |   if (w, z) ∉ HC then
4 |   |   bDFS(w, z);
5 |   end
6 end
```

Algorithm 3: $bDFS(w, y)$

Based on the above discussion, it follows that we can replace Step 8 in Algorithm 2 by the following step.

Step 8 for Algorithm 2:
for each $w \in V$ **do**
if $((w, x) \in HC)\&((w, y) \notin HC)$ **then** $bDFS(w, y)$; **end**
end

Along with this modification we also absorb the for-loop at step 13 of Algorithm 2 in the bounded DFS routine. The final algorithm is given in Algorithm 4. Here H is stored in two data-structures, adjacency-list HL as well as adjacency-matrix HM. $HL[a]$ points to the list of vertices to which there are edges in H from a, and $HM[a, b] = 1$ iff $(a, b) \in H$. The transitive closure of H is stored as adjacency matrix, where $HC[a, b] = 1$ iff (a, b) belongs to H^c.

4.3 Time and Space Complexity

Analysis of running time: Let the number of vertices in V be n, number of edges in E_0 be m, and $\overline{m}$ denote the number of edges in the implied-set closure of E_0.

The **for**-loop in Algorithm 4 runs n times in each call and **while**-loops iterates $\overline{m}$ times so total time complexity of the algorithm, excluding the cost of $bDFS'$-calls is $O(\overline{m}n)$.

We bound the cost due to $bDFS'$-calls in two parts, one for each for-loop. Observe that the $bDFS'$-routine is never called again with the same arguments. Therefore the cumulative cost of all calls due to the second for-loop

Data: vertex set V; edge set E_0; induced-edge-sets $T(a, b)$ for all pairs $(a, b) \in V \times V$

Result: HL is the adjacency list of the implied-set-closure, and HC stores its transitive closure

```
1  HL ← ∅;
2  HM ← ∅;
3  HC ← ∅;
4  X ← E_0;
5  while X ≠ ∅ do
6  |   (x, y) ← Select(S);
7  |   Insert y in HL[x];
8  |   HM[x, y] ← 1;
9  |   if HC[x, y] = 0 then
10 |   |   for each w ∈ V do
11 |   |   |   if HC[w, x] = 1 and HC[w, y] = 0 then
12 |   |   |   |   bDFS'(w, y)
13 |   |   |   end
14 |   |   end
15 |   end
16 end
17 return HL, HC;
```

Algorithm 4: Algorithm to compute implied-set closure: Final version

```
1  HC[w, y] ← 1;
2  for each (u, v) ∈ T(w, y) do
3  |   if HM[u, v] ≠ 1 then
4  |   |   X ← X ∪ {(u, v)}
5  |   end
6  end
7  for each z ∈ HL[y] do
8  |   if HC[w, z] = 0 then
9  |   |   bDFS'(w, z);
10 |   end
11 end
```

Algorithm 5: subroutine $bDFS'(w, y)$

is $\sum_w \sum_y deg_H(y)$, where $deg_H(y)$ denotes the out-degree of y in H, which is $O(\overline{m}n)$. The contribution of the first for-loop can be estimated by observing that each $T(w,y)$ is scanned at most once and the membership test takes $O(1)$ time. So its cost is $O(\sum_{(w,y)\in HC} |T(w,y)|)$. The total time complexity of the algorithm is $O(\overline{m}n + \sum_{(w,y)\in HC} |T(w,y)|)$.

Note that the second component in the expression can't be got rid of by any algorithm of ISC problem because this is also the input size for the problem and each algorithm has to scan it at least once. The additional cost is $O(\overline{m}n)$. It seems quite difficult, if not infeasible, to beat this bound since the ordinary transitive closure problem is a special case of the ISC problem and there does not exist any combinatorial algorithm for transitive closure problem with running time $o(mn)$ where $m = |E_0|$.

Space requirement: The algorithm uses adjacency list as well as adjacency matrix representation for the graph H which amounts to $\Theta(\overline{m} + n^2) = \Theta(n^2)$ space. In addition we use $n \times n$ matrix HC. Hence, in addition to the input which consists of original edge set $E_)$ and the lists $T(a,b)$ for each $(a,b) \in V \times V$, the algorithm uses only $\Theta(n^2)$ additional space.

Theorem 1. *For a given graph on n vertices, a base set of edges E_0, and sets $T(a,b)$ $\forall (a,b) \in V \times V$ of implied edges, there exists an algorithm which solves the ISC problem in $O(\overline{m}n + \sum_{a,b} |T(a,b)|)$ time and $\Theta(n^2)$ space, in addition to the space used by input, where $\overline{m}$ is the number of edges in the implied-set closure of E.*

In Section 3 we have seen that computation of edges due to rules R_4 and R_5 in Algorithm 1, which dominates the time complexity, is the implied-set closure of the edge set resulting after applications of the first three rules. Therefore the time complexity for memory consistency verification problem is the same as that for computing the corresponding implied-set closure.

In this ISC problem the non-empty $T(a,b)$ sets are either of the form $T(S,S')$ or $T(S,L)$, so $\sum_{a,b} |T(a,b)| = \sum_{S,S'} |T(S,S')| + \sum_{S,L} |T(S,L)|$. From the definition it is clear that $T(S,S') \cap T(x,y) \neq \emptyset$ iff $S = x$ and $S' = y$. Hence $\sum_{S,S'} |T(S,S')| \leq |V \times V|$. Again from the definition $|T(S,L)| \leq 1$ so $\sum_{S,L} |T(S,L)| \leq \sum_{S,L} 1 \leq |V \times V|$. This gives $\sum_{a,b} |T(a,b)| < 2n^2$. We have the following corollary.

Corollary 1. *Memory consistency verification problem can be solved in $O(n^3)$ time and $O(n^2)$ space, where n is the total number of instructions in the test program.*

5 Experimental Results

Theoretically, the new algorithm achieves a speed-up by a factor of n over the worst case time complexity of the previous best algorithm of Roy et al. [6]. To show that there are no hidden large constants, we compared it with the algorithm of Roy et. al. [6] experimentally (see Table 1).

Our experiments are based on test programs with 2, 4, 6, and 8 program-threads. In each case the number of instructions per thread were kept the same and these varied from 100 instruction per thread to 500 instruction per thread. Each result reported here is the average of 100 programs in each category. The test programs were generated by Intel's MPRIT tool [6]. Here Algorithm A refers to the algorithm reported here and Algorithm B is that of Roy et. al. [6] .

To make a fair comparison, the algorithms were executed in an identical environment consisting of a single processor. The experiment used implementation of Roy et. al. [6] algorithm without vector instructions.

Time (ms)	Instructions per thread									
	100		200		300		400		500	
#threads	algo A	algo B	algo A	algo B	algo A	algo B	algo A	algo B	algo A	algo B
2	119	82	230	240	591	626	1216	1350	2216	2476
4	149	163	614	953	1834	3289	4198	7913	8181	16071
6	225	332	1222	2997	3926	11023	9508	27914	18593	57671
8	328	688	2082	7216	7204	28183	16846	69851	32904	146350

Table 1. Experimental results

In each experiment the number of instructions is fixed. So n is equal to the number of instructions-per-thread times the number of threads. As number of threads increase, n increases and consequently performance gap increases. Besides, we see that algorithm A performs significantly better compared to algorithm B in case of 100 instructions per thread with 8 thread, while the gap is insignificant when instruction per thread is 400 and threads are 2. In both the cases $n = 800$, but the reason for this difference is the number of threads. The number of inferred edges increases with more threads, and expensive computation involves the computation of these edges.

Intel's MPRIT Tool incorporates an implementation of the algorithm of Roy et al. [6] using vector instructions on SIMD processor which performs 128 bit operations per instruction. This results in a significant speedup (about a factor of 100). Yet asymptotically, for large values of n, the new algorithm will outperform even the vector-instruction aided implementation of the algorithm in [6].

5.1 Parallelization of the new algorithm

Step 10 in Algorithm 4 involves processing for each $w \in V$. In each iteration of the *for*-loop all computations exclusively depend on the HC edges of the form $(w, *)$ and the sets $T(w, *)$. This implies that computations in different iterations are mutually independent. Therefore in a multi-processor environment different processor could be assigned different passes of this loop to be performed in parallel.

6 Conclusion

In this work we studied the memory consistency compliance algorithm of Hangal et.al. [3] which was improved by Roy et. al. [6]. The former had $O(n^5)$ complexity, while the latter was improved to $O(n^4)$. We identified the bottleneck in these algorithms and proposed it as a graph problem called induced-set closure problem. We proposed an efficient algorithm for this problem using Italiano's [4] incremental algorithm for transitive closure and showed that using this approach the memory consistency compliance algorithm can be implemented in $O(n^3)$ time. An efficient parallel implementation of our algorithm remains a task for the future research.

7 Acknowledgment

We thank Amitabha Roy for introducing the memory consistency compliance problem to us. We thank Mainak Chaudhuri for explaining various concepts in the area of memory architecture. We also thank Mayur Shardul for pointing out an error in the bound for $|\sum T(S, S')|$ in an earlier draft.

References

1. S. V. Adve and K. Gharachorloo. Shared memory consistency models : A tutorial. In *Digital Western Research Laboratory Technical Report*, 1995.
2. J. Cantin, M. Lipasti, and J. Smith. The complexity of verifying memory coherence. In *Proceedings of 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 254–255, 2003.
3. S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu. TSOtool : A program for verifying systems using the memory consistency model. In *In Proceedings of the 31st annual international symposium on computer architecture (ISCA)*, pages 114–123, 2004.
4. G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273 –281, 1986.
5. C. Manovit and S. Hangal. Efficient algorithm for verifying memory consistency. In *In Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures (SPAA'05)*, pages 245–252, 2005.
6. A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang. Fast and generalized polynomial time memory consistency verification. In *In Proceedings of the 18th International Conference of Computer Aided Verification (CAV)*, pages 503–516, 2006.
7. P. S. Sindhu, J. M. Frailong, and M. Cekleov. Formal specification of memory models. In *Xerox PARC Technical Report*, 1991.