Generalized whole-program identification of efficient atomic regions

Peng Liu, Charles Zhang

The Prism Research Group

Hong Kong University of Science and Technology

{lpxz, charlesz}@cse.ust.hk

***Abstract*—Atomic regions, which are implemented with the lock or the software transactional memory, are commonly used to achieve the atomicity. As the manual specification of atomic regions is labor-intensive and error-prone, we propose an approach to identify them automatically. Our automatic approach preserves the maintainability by producing the structured atomic regions, preserves high performance by finding the minimal atomic regions, and guarantees the correctness by supporting the multi-variable atomicity throughout the program. Besides, the approach allows programmers to bound the region identification with their domain knowledge. Additionally, our identification can be used to optimize existing atomic regions or to fix atomicity violations. The evaluation shows, compared to the original version with the manual atomic region specification, the version with the identified atomic regions is 5% slower on average and 13% slower maximally, which suggests our atomic region identification is a reasonable candidate for replacing the manual specification.**

I. Introduction

The atomicity property is an important correctness criterion [5] for the shared-memory concurrent programs. The atomicity requires a group of shared memory accesses to be executed without being interleaved, and is often specified by programmers using the lock region, of which the entry marks the start of the accesses and the exit marks the end of the accesses. When specifying the atomic lock regions, programmers need to reason about the accesses to the shared memory, which are distributed in different code structures in different methods. The manual reasoning is tedious and often results in [19] atomicity violations. First, programmers may ignore some accesses to shared memory as too many accesses spread over the whole program. Second, given the accesses, programmers may find the code region that does not suffice to protect them from being interleaved, due to the complexity of the code structure layout and the method calls. On the other hand, the compiler is good at reasoning about the accesses rigorously and comprehensively. Therefore, we propose to let the compiler reason about the accesses automatically and leave to programmers the work that the compiler cannot do.

The compiler cannot do the work related to programmers' intention. First, the compiler does not know the multi-variable (or multi-field) atomicity [27], [19]. Consider the fields position and salary of an employee object, they are *correlated* and should be updated atomically, otherwise, the heap state may be left incorrect with the inconsistent field values. Although the compiler could apply the artificial intelligence technique to learn [19] the correlation intention, it cannot capture the intention faithfully. Second, suppose the accesses of the correlated variables are available, programmers often break them into groups and find a unique atomic region to protect each group of accesses. It is impossible for the compiler to determine the grouping. Given the strength and the limitations of the compiler, we propose the following workflow: (1) Programmers provide the multi-variable atomicity. Programmers can specify the multi-variable correlation with the notion of atomic set [27], which is proven simple because programmers do not need to address the complexity of the accesses. (2) The compiler finds the accesses of the multiple correlated variables and identifies an atomic region to protect them. (3) Programmers may directly adopt the compiler-generated atomic region as the alternative to the manual specification, or use it as the working atomic region and further improve it by breaking it into several finer regions. In either case, the compiler-based region identification, which is the focus of this work, provides the correctness guarantee in reasoning about the accesses and relieves programmers from the complexity of the accesses

Researchers have made the attempt in automating the region identification. For example, the approach [25] supports the single-variable atomicity, by first finding the accesses of a shared variable and then identifying the atomic region to protect them. The approaches [27], [4] support the multi-variable atomicity locally. They maintain precisely the correlation among multiple instance fields which are encoded using the ownership. However, the ownership-based encoding limits the applicability to a local scope, i.e., the declaring class of the object owning the instance fields. For example, suppose the fields owned by a List object (the root object) are correlated according to the atomic set specification, the approaches [27], [4] produce only the atomic regions that are inside the class of the root object, i.e., the List class. They cannot produce the atomic regions out of the class to protect the client-side invocations that transitively access the fields of the root object. In summary, existing approaches do not support the multi-variable atomicity or support the multi-variable atomicity within the local scope, because they cannot maintain the multi-field correlations throughout the whole program.

We propose an approach that guarantees the multi-variable atomicity *throughout the whole program*. The first phase is to recognize the accesses of the correlated fields. We need the encoding of the correlated fields, which is valid and consistent throughout the whole program. The encoding cannot be based on the ownership because the ownership is valid only in a local scope, and it cannot be based on the names of the variables in the code because the same variable name may refer to different objects inconsistently at different sites while different variable names may refer to the same object. In this work, we propose an encoding based on the static modeling of the heap. We derive the heap-based encoding from the atomic set specification, based on which we apply the side effect analysis to determine the accesses of the correlated fields in the program.

The second phase of our approach is to identify an atomic region that protects the accesses found by the first phase. This phase is also applicable to fixing the atomicity violations [18], [12], where the accesses are provided by the atomicity violation detection tools. In this phase, we have two important designs: (1) We find the *block-structured atomic region*, which is like the Java synchronization block. The block-structured atomic regions (atomic blocks) are predominately used in existing Java programs according to the study [24] of large-sized applications in Sourceforge, and therefore are the most familiar form to the programmers, which allow programmers to understand them easily and preserve the maintainability. Comparatively, existing approaches [18], [12] for fixing the atomicity violations apply the sophisticated analyses to produce the non-block-structured atomic region with multiple entries and multiple exits, which is hard to understand and prevents further optimization tuning. (2) We identify the *minimal atomic block.* By keeping the atomic block small, we leave as much irrelevant code, such as the accesses to thread-local heap locations, out of the block as possible, which minimizes the redundancy in the atomic block, minimizes the execution of the atomic block and minimizes the blocking of other threads. Comparatively, the previous approaches [27], [25] produce the large atomic regions, e.g., the whole method bodies, which may block the execution for the unnecessarily long period.

Producing the atomic blocks in the methods with the complex code structure layout is challenging because we need to guarantee the correct locking behaviors, i.e., the entry of the atomic block always matches the exit at runtime and the double unlocking/locking is avoided. Producing the minimal atomic block is also challenging as neither the atomic block nor the minimality is formally defined. In this paper, we formally define the atomic block based on the single entry single exit region [13] and define the minimality based on the partial orders among the blocks. We propose an algorithm that leverages the program structure tree [13] to address the complexity of the code structure layout and leverages the call graph analysis to address the complexity of the method calls. We also prove the minimal atomic block guarantee of the algorithm.

Finally, the third phase is optional. As aforementioned, programmers may further optimize the compiler-generated atomic region by breaking it into several children regions and minimizing each of them. The minimization in the optimization inevitably involves the manual reasoning of the accesses, which is tedious and error-prone. We propose the bounded region identification to automate the optimization. Programmers just need to tell the compiler their knowledge of the semantically independent scopes, e.g., the components developed independently by different developers. The compiler then apply the bounded identification automatically to identify the *minimal* atomic region that contains the accesses inside each scope.

We implement our approach and evaluate it on Stamp, a benchmark suite for atomicity. Our approach infers two versions of atomic blocks, one with the bounded identification and one without. The atomic blocks inferred with the bounded analysis slow down the performance by 5% on average and 13% maximally, tested with different threading configurations. The minor slowdown suggests our approach is competent for replacing manual region inference. The atomic blocks inferred without the bounded analysis slow down the performance by 43% maximally, which suggests that the bounding scope can effectively leverage the atomic block identification.

The contributions of this work are:

- We formally define the structured atomic region and design an algorithm that identifies the minimal structured atomic region which contains the accesses that require the atomicity. The minimal structured atomic regions preserve the maintainability and the high performance.
- We propose an approach to maintain the multi-variable correlation and support the multi-variable atomicity throughout the program.
- We allow programmers who have the domain knowledge to bound the identification. The bounded identification produces fine-grained atomic blocks within each scope.

II. Overview

In this section, we outline our approach with the code snippet from the Bayes benchmark, a Bayesian network application.

```
1 isTaskValid= true;
  if(op==INSERT)
3 {
   if(learnerPtr.hasEdge(fromId, toId))
5       isTaskValid=false;
  }
7 else{...}
  if(isTaskValid)
9  learnerPtr.applyOp(op, fromId, toId);
```

The code checks (line 4) if an edge exists between two nodes in the Bayesian network $learnerPtr$, and inserts (line 9) an edge if no edge is present. The operations should be protected in one atomic block to guarantee that, the insert operation is carried out only if the check operation returns false, i.e., no edge is present.

Our analysis produces the minimal atomic block to enclose the two operations as follows.

Multi-variable atomicity. Programmers first declare the program states that require the atomicity based on the notion of atomic set [27]. In our example, the state of the list transitively referenced by learnerPtr (Figure 4) needs to be accessed atomically, the corresponding atomic set declaration is, $\{List.size, List.head, List.head.next+\}$[1]. The free type variable $List$ is then bounded to a list object o_1 in Figure 4 so that the declaration becomes a concrete atomic set, $\{o_1.size, o_1.head, o_1.head.next, o_1.head.next.next\}$ or $\{o_1.size, o_1.head, o_2.next, o3.next\}$. In our static analysis, to get the concrete atomic set, we approximate the heap with the abstract heap and bind the type variable to a representation from the abstract heap. With the concrete atomic set and the side effect analysis, we identify the accesses of the program state that requires the atomicity, e.g., the accesses at line 4 and 9 in our example. More details are explained in Section V.

[1]The symbol "+"is a shortcut for one or more.

Identifying the minimal atomic block. In our running example, the minimal atomic block (i.e., a single entry single exit code structure) that contains the accesses is from line 2 to line 8. However, when the method body has a complex layout, identifying the minimal atomic block becomes more complex and error-prone. We rely on the program structure tree (Figure 1(b)), derived from the control flow graph (Figure 1(a)), to reason about the complex layout. Algorithm 1 identifies the minimal atomic block by finding the lowest common ancestor in the program structure tree.

When the accesses are distributed in different methods, we need to reason about the complex calling relations too. Suppose in the call graph (Figure 2), both the accesses are in different methods D and E, they should be protected by the atomic block in whatever calling contexts. Placing the atomic block in the method B is incorrect because the accesses will not be protected when invoked by the method C. Algorithm 2 identifies the atomic block in the immediate dominator method A so that the atomic block is always present when the accesses are executed. Section III presents our identification technique.

Bounding the atomic block identification. Programmers are allowed to bound the identification into independent scopes, based on their domain knowledge. In order to preserve the independence between different bounding scopes, the atomic blocks should be identified in the method that is "unique" to the bounding scope. Besides, the atomic blocks can be safely placed in the bound-aware dominator, which is less strict than the dominator. Besides, the bounded identification can be used as the optimization that refines existing atomic blocks specified as the bounding scopes. More details can be found in Section IV.

The rest of the paper is organized as follows. Section III, Section IV and Section V present the core of our approach. Section VI discusses the threats to validity and Section VIII presents the evaluation.

III. Identifying the minimal atomic block

Given the accesses ACC that require the atomicity, our goal is to identify the minimal atomic block, a synchronized block like code structure, to contain them. The atomic block, after being equipped with the lock or the transactional memory, excludes the buggy interleavings to the accesses. The accesses ACC may be determined (Section V) as the accesses of an atomic set, or provided by the bug detection tools [5] as the accesses in an atomicity violation.

To identify the minimal atomic block, we need to solve the following problems: (1) Although the atomic block is commonly referred to, the formal definition of it is unclear, which disables the inference. (2) As for the *minimal* atomic block that *contains* the accesses, what does the minimality and the containment mean? (3) If the accesses are nested in different code structures in a method with the complex layout, the identification algorithm needs to reason about the layout precisely. (4) If the accesses are in different methods, we need to extend the above definitions in an inter-procedural fashion and the identification algorithm needs to reason about the methods calls correctly.

In the following, we first formalize (Section III-A) the atomic block, the containment and the minimality, based on the single entry single exit region [13]. Then, we present (Section III-A) the intra-procedural analysis that identifies the minimal atomic block, on the assumption that the accesses are in the same method, and prove the guarantee of the minimality. Finally, if the accesses are not in the same method, we extend (Section III-B) the definitions and the identification algorithm.

A. Intra-procedural analysis

In this section, we assume the accesses are in the same method. Section III-B handles the scenarios where the accesses are not in the same method. Our definition of the atomic block (Definition 3) is based on the basic concepts of control flow graph (Definition 1) and domination (Definition 2).

Definition 1 (Control Flow Graph): The control flow graph G is a graph with nodes connected by edges, where each node represents the statement S and each edge represents the control flow. Two distinguished nodes, n_{start} and n_{end}, represent the unique start and the unique end. Every node occurs on some path from n_{start} to n_{end}.

Definition 2 (Domination and PostDomination): In a directed graph, a node n_x dominates node n_y if every path from n_{start} to n_y includes n_x. A node n_y postdominates node n_x if every path from n_x to n_{end} includes n_y.

Definition 3 (Atomic Block): Atomic block R is a single entry single exit code region (SESE region) in the control flow graph. It is denoted as a pair of control flow nodes, (n_a, n_b), where (1) n_a dominates n_b, (2) n_b postdominates n_a, and (3) every cycle containing n_a also contains n_b and vice verse.

The SESE region is previously defined by Johnson et al. [13] and used to study the parallel or incremental program analysis. The three conditions in Definition 3 characterize the behavioral constraints of the atomic block: (1) Take the lock implementation for example, if the lock release at the exit n_b is executed, the lock acquisition at the entry n_a must also be executed. (2) If the lock acquisition at n_a is executed, the lock release at n_b must be executed. (3) When the execution goes from the inside to the outside of the atomic block (from the outside to the inside), it must execute the lock release at the exit n_b (the lock acquisition at the entry n_a). The third constraint is especially important as it guarantees the pairing between the acquisition and release, and precludes the incorrect behaviors such as double unlocking or double locking.

Definition 4 (Containment): A control flow node n is inside or contained by the atomic block R, (n_a, n_b), if n_a dominates n and n_b postdominates n_a. We denote the containment relation as $n \in R$.

The minimal atomic block (Definition 6) that contains the accesses is defined in terms of the containment between atomic blocks (Definition 5).

Definition 5 (Containment between Atomic Blocks): Given two atomic blocks R_1, $(n_a,\ n_b)$, and R_2, $(n_s,\ n_t)$, R_2 is contained by R_1, denoted as $R_2 \subseteq R_1$, if n_a dominates n_s and n_b postdominates n_t.

Definition 6 (Minimal Atomic Block): An atomic block R_{mini} is the minimal if there is no atomic block R such that $R \subseteq R_{mini}$.

According to the results [13], the containment among atomic blocks defines a partial order, which can be graphically depicted by the graph called program structure tree or PSTree, where each node represents a block (Definition 5) and each edge represents the immediate containment. Besides, according to the results [13], in structured programs such as the C/C++/Java/Fortran programs, the blocks are either disjoint or nested, they cannot overlap partially. Therefore, each block has at most one parent as it cannot be contained within two blocks immediately, and hence the PSTree graph is a tree. The PSTree can be constructed from the control flow graph with E edges in $O(E)$ time [13].

Figure 1 shows an example control flow graph and its program structure tree. The blocks b and c are disjoint, the blocks a and b are nested, i.e., a contains b. Specially, the blocks may be sequentially composed, e.g., the blocks f and g are sequentially composed.

Our identification algorithm is based on the PSTree. In Algorithm 1, the input includes the accesses ACC that require the atomicity and the PSTree $tree$ of the current method, the output is the minimal atomic block that contains the accesses.

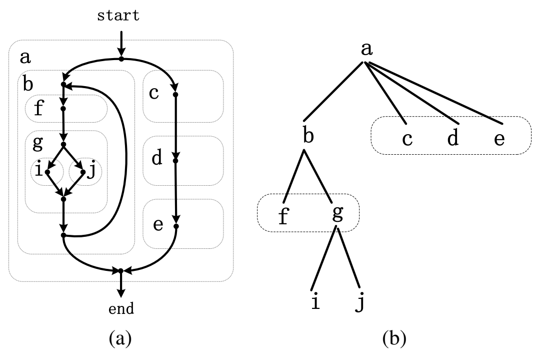


Fig. 1: (a) control flow graph labelled with atomic blocks (b) program structure tree.

At line 3, we find the atomic blocks that contain immediately the accesses. Then we find (line 5) their lowest common ancestor R_{lca} in the PSTree, which is returned as the minimal atomic block (line 16) except when the children blocks of R_{lca} are sequentially composed. Consider the example in Figure 1, the minimal atomic block that contains c and d is not the lowest common ancestor a but $sequence(c, d)$, where the helper function $sequence$ decides the minimal sequence that contains the sequentially placed items. Therefore, Algorithm 1 finds (Lines 6-12) the child R'_i of R_{lca} which is meanwhile the ancestor of R_i. Algorithm 1 returns (Line 13-15) the minimal sequence of these children blocks if they are sequentially composed.

```
1  $n = ACC.size()$;
2  for $acc_i$:$ACC$ do
3      $R_i$ = $region(acc_i)$;
4  end
5  $R_{lca}$= $tree.lowestCommonAncestor(R_1,\ R_2,\ \ldots,\ R_n)$;
6  for $R_i$: from $R_1$ to $R_n$ do
7      for child: $R_{lca}$.children() do
8          if child.isAncestor($R_i$) then
9              $R'_i$ = child;
10         end
11     end
12 end
13 if $R'_1,\ R'_2 \ldots,\ R'_n$ are sequentially composed then
14     return sequence($R'_1,\ R'_2,\ \ldots,\ R'_n$);
15 end
16 return $R_{lca}$;
```
Algorithm 1: The function $miniAtomBlock$

Theorem 1: The atomic block R_{mini} computed by Algorithm 1 is minimal, i.e., $\forall R$ that contains the accesses,

$R_{mini} \subseteq R$.

Proof 1: We divide the proof into two cases.

Case 1: If $R \nsubseteq R_{lca}$, then $R_{lca} \subseteq R$ and therefore $R_{mini} \subseteq R_{lca} \subseteq R$. More specifically, as the $\subseteq$ relation is equivalent to the "ancestor" relation in the PSTree, $\forall i \in 1 \ldots n, R_i \subseteq R$ means R is the common ancestor of the blocks R_i. Therefore, R must be[2] an ancestor of the lowest common ancestor R_{lca}, i.e., $R_{lca} \subseteq R$, therefore $R_{mini} \subseteq R_{lca} \subseteq R$.

Case 2: If $R \subseteq R_{lca}$, R must contain the child block R'_i of R_{lca} (line 9 of Algorithm 1). If the children blocks R'_i ($i \in 1 \ldots n$) are sequentially placed, R must contain R_{mini}, i.e., $sequence(R'_1, R'_i, \ldots, R'_n)$. Otherwise, R must be R_{lca}. In either case, $R_{mini} \subseteq R$.

B. Inter-procedural analysis

Our analysis in Section III-A assumes the accesses are in the same method body. In this section, we design the inter-procedural analysis which finds the minimal atomic block if the accesses are contained in different methods. Note that the resultant atomic block is still placed in a method, i.e., its entry and exit are in the same method. In the following, we first extend the definitions of the containment relation (Definition 8) and the minimality (Definition 9) in an inter-procedural way, based on the inter-procedural closure (Definition 7).

Definition 7 (Inter-procedural Closure): Given the block R, the inter-procedural closure $closure(R)$ stands for the statements in R and the statements in the transitive callee methods $callees(R)$.

Definition 8 (Containment): A control flow node n is contained inter-procedurally within the atomic block R if both the conditions hold: (1) $method(R)$[3] is present in any call chain to $method(n)$. (2) $n \in closure(R)$.

[2] R may also be a sequential composition of the ancestor of R_{lca} and other blocks, which does not affect the correctness of our proof.

[3] The helper function $method(x)$ returns the directly containing method.

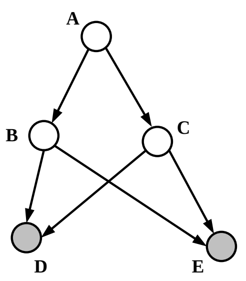


Fig. 2: Call graph. Nodes stand for methods and Edges stand for method calls.

Condition 1 requires the atomic block R to protect the access (represented by node n) in any calling context. Therefore, $method(R)$ should be the dominator in the call graph of $method(n)$, otherwise, the access n may be executed in a calling context where $method(R)$ is not present. For example, in the call graph (Figure 2), suppose the accesses are in the methods D and E. If we place the atomic block in the dominator method of D and E, i.e., the method A, we guarantee the method A and the contained atomic block are always present when D or E is executed. However, if we place the atomic block in the lowest common ancestor method such as B, the method B and the contained atomic block may not be present when D or E is executed along the call chain $A \rightarrow C$. Condition 2 specifies that the atomic block R is large enough so that its closure includes the access n.

Definition 9 (Minimal Atomic Block): An atomic block R_{mini} is the minimal if there is no atomic block R such that $closure(R) \subseteq closure(R_{mini})$.

```
$ms$=$\emptyset$;
foreach $acc_i$:$ACC$ do
    $ms.add(method(acc_i))$;
end
$m_{idom}$= $iDom(ms)$;
foreach $S'_j$: $m_{idom}.stmts()$ do
    if $S'_j.callsAny(ms)$ then
        $callsites.add(S'_j)$;
    end
end
return $miniAtomBlock(PSTree(m_{idom}), callsites)$;
```
Algorithm 2: The function $interMiniAtomBlock$

As shown in Algorithm 2, we compute the minimal atomic block by reduction to the intra-procedural scenario (Algorithm 1). Given the methods ms that contain the accesses directly, we first compute (lines 1-5) their immediate dominator m_{idom} in the call graph and then find (lines 6-10) the statements $callsites$ in m_{idom} that invoke the methods ms transitively. Finally, we identify (at line 11) intra-procedurally the minimal atomic block that contains the statements $callsites$.

Theorem 2: According to Definition 9, the atomic block R_{mini} computed by Algorithm 2 is minimal.

We sketch the proof briefly. For any atomic block R that is not in the *immediate* dominator m_{idom}, to contain all the accesses inter-procedurally, it must invoke m_{idom} and hence its closure is larger than $closure(m_{idom})$ and $closure(R_{mini})$. For any atomic block in the m_{idom}, it must contain all the statements $callsites$ intra-procedurally, otherwise some accesses may be invoked out of the atomic block. According to Algorithm 1, R_{mini} is the minimal atomic block of such.

IV. Bounding the identification

Programmers, who have domain knowledge, may specify the scope bounding the identification of the atomic block. For example, a program may consist of multiple components, each written by a developer independently, the inference can be carried out inside the scope of each component and the resultant atomic blocks are bounded by the scope. By bounding the identification, we increase the granularity of the atomic blocks and avoid the monolithic atomic block that spans the whole program.

In our work, programmers only need to specify the bounding method, e.g., the entry method of the component. The bounding method $Bound$ and the methods transitively invoked inside it constitute the bounding scope, denoted as $scope(Bound)$. Two important properties are associated with the bounding method: (1) Correctness. The identified atomic block needs to be present in any call chain from the bounding method to the methods that contain the accesses. (2) Independence. The atomic block synchronizes the execution only in the scope of the bounding method, it does not take effect in the scope of other bounding methods. In the following, we explain how to ensure the properties.

Correctness. As stated above, the atomic block is required to be present in any call chain *from the bounding method* to the methods that contain the accesses. Instead of placing the atomic block in the dominator of the containing methods, which is unnecessarily strict, we place the atomic block in the bound-aware dominator.

Definition 10 (Bound-aware Dominator): Given the bounding method $Bound$, the bound-aware dominator of the method m is the method that is present in any call chain from $Bound$ to m. It is a dominator method in the trimmed call graph, where all methods except the transitive callee methods of $Bound$ are removed[4].

Take Figure 2 for example, although the method B is not a dominator of D and E, it is a bound-aware dominator given that the bounding method is B. By placing the atomic block in the bound-aware dominator B, we can guarantee that the accesses are always protected when invoked inside B.

Independence. To guarantee the independence among the bounding scopes, we place the atomic block in the unique method of each bounding scope. The unique method, $unique(Bound_i)$, is a method in the scope $scope(Bound_i)$, but not a method in any other scope $scope(Bound_j)$. More strictly, the unique methods are computed as $scope(Bound_i) - \bigcup_{Bound_x != Bound_i} scope(Bound_x)$. It is possible that the unique methods do not exist, e.g., when the bounding method $Bound_i$ may be invoked in another bounding method $Bound_j$.

[4]The edge, of which the source or target is removed, is also removed.

That means, the atomic block placed in the scope of $Bound_i$ inevitably affects the execution in the scope of $Bound_j$. In that case, we simply set the unique method as $Bound_i$.

In Figure 2, suppose B and C are both the specified bounding methods. $unique(B) = \{B, D, E\} - \{C, D, E\} = \{B\}$. By placing the atomic block in the unique method B, the synchronization for the atomic block does not take effect in the other scope C.

To conclude, in order to ensure the correctness and the independence, we need to place the atomic block in a bound-aware dominator which is meanwhile the unique method. We may have to discard the independence property occasionally. Besides, the identification analysis in Section III and Section IV can be viewed as the optimization technique that refines the initial coarse atomic blocks specified as the bounding methods.

V. Identifying the Accesses that Require Atomicity

As mentioned in Section III, the input of the atomic block identification is the accesses that require the atomicity. We compute such accesses in this section. The accesses are those accessing the inherently correlated fields in the heap. As explained in Section I, it is crucially important to support the multi-field (or multi-variable) atomicity because otherwise the inherent consistency among the fields would be violated. Existing static analyses [25], [27] cannot support the multi-field atomicity correctly because they cannot maintain the multi-field correlation throughout the program. For example, the static analysis [27] maintains the correlation within a local scope based on the ownership, which does not apply to the whole program.

We support the multi-field atomicity based on the notion of atomic set, proposed by Vaziri et al. [27]. We support the

f	$\in$	F	fields
$type$	$\in$	TYPE	type names
sel	$\in$	SEL	selectors
$decl$	$::=$	$\emptyset \mid decl \cup \{type.sel.f\}$	
sel	$::=$	$f \mid f.sel$	

(a)

o	$\in$	O	objects
δ	$\in$	$O \times F \rightarrow O$	*heap*
δ_2	$\in$	$O \times SEL \rightarrow O$	
$bind$	$\in$	$TYPE \times SEL \times F \times O \rightarrow O \times SEL \times F$	
$alloc$	$\in$	Alloc	allocation sites
pts	$\in$	PTS	point-to sets
pts	$::=$	$\emptyset \mid pts \cup \{alloc\}$	
δ'	$\in$	$PTS \times F \rightarrow PTS$	*aheap*
δ_2'	$\in$	$PTS \times SEL \rightarrow PTS$	
$bind'$	$\in$	$TYPE \times SEL \times F \times PTS \rightarrow PTS \times SEL \times F$	

(b)

Fig. 3: (a) atomic set declaration (b) heap-related functions.

multi-variable atomicity throughout the program by maintaining the multi-variable correlation based on the whole-program reasoning of the heap.

Atomicity declaration. The atomic set consists of the correlated fields, which are referenced transitively by the same root object, e.g., the highlighted fields (Figure 4) referenced transitively by the root List object O_1. Abstracting the declaration syntax details [27] away, we represent (Figure 3(a)) each declared atomic set $decl$ succinctly as a set of syntactical items $type.sel.f$, where $type$ tells the type of the root object, sel is a field reference chain from the root object to the owner object of the field f. Here, $type$ is a free variable, once bound to a root object o_{root}, the syntactical item $type.sel.f$ refers to a concrete field $o_{root}.sel.f$ and the declaration $decl$ refers to a concrete atomic set.

We explain the binding in terms of runtime heap. As shown in Figure 3(b), the heap records the field reference relationships among objects. The semantic δ of the heap is standard, e.g., $\delta(o_1, f_1) = o_2$. The derived semantic δ_2 of the heap records the transitive field reference, e.g., $\delta_2(o_1, f_1.f_2...f_n) = \delta_2(\delta(o_1, f_1), f_2...f_n) = \delta_2(o_2, f_2...f_n) = ... = o_{n+1}$. The function $bind$, which accepts the argument of a root object o_{root}, maps each syntactical item $type.sel.f$ in the declaration $decl$ to the instance field $o_{root}.sel.f$, or equivalently, $o.f$, where $\delta_2(o_{root}, sel) = o$.

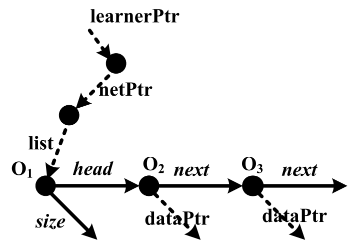


Fig. 4: Object graph rooted at `leanerPtr`. Circles represent instances, edges represent fields.

Consider the declaration $\{List.size$, $List.head$, $List.head.next+\}$, by binding $List$ to a shared list object o_1 in Figure 4, we get $\{o_1.size$, $o_1.head$, $o_1.head.next$, $o_1.head.next.next\}$. After looking up the heap, we get the atomic set of concrete fields $\{o_1.size$, $o_1.head$, $o_2.next$, $o3.next\}$.

However, the runtime heap is not available in the static analysis. Instead, we approximate the heap with the abstract heap (Aheap) and bind the free type variable to a representation in Aheap. As shown in Figure 3(b), the point-to set pts_1, computed by the pointer analysis [17], consists of the allocation sites $Alloc$, each $alloc \in Alloc$ approximating the object or objects created at the site. The Aheap, also computed by the pointer analysis, records the one-to-one field reference between point-to sets, e.g., $\delta'(pts_1, f_1) = pts_2$. Comparatively, because the Aheap summarizes all possible runtime heaps conservatively, the field reference between the $Alloc$ abstractions is one-to-many, i.e., one $Alloc$ abstraction may reference via a field many $Alloc$ abstractions. In our analysis, we choose to reason using the point-to set because of the simplicity of the one-to-one field reference between point-to sets.

We compute the atomic set as follows. Given each syntactical item $type.sel.f$ in the declaration, we first bind (Function $bind'$ in Figure 3(b)) the free $type$ variable to a root point-to set pts_{root} and produce $pts_{root}.sel.f$, or equivalently, $pts.f$, where pts is the point-to set reachable from pts_{root} along the reference chain sel. For simplicity, we refer to the item $pts.f$ as the abstract instance field, therefore, the computed atomic set $atomS$ is a set of abstract instance fields.

In the above computation, we need to know the root point-to set pts_{root}, i.e., what point-to set we bind the $type$ variable to. The point-to set pts_{root} must be (1) in the shared heap because the atomicity is a concurrency-related property and (2) type-compatible with the $type$ variable. Therefore, we find it by traversing the shared heap and checking the types involved in each traversed point-to set. According to the studies [22], [26], the shared heap consists of the point-to sets referenced transitively by the arguments at a thread-spawning call site or the static fields because only the objects referenced by such variables can be shared among threads.

Now, we proceed to identify the accesses of the fields in an atomic set $atomS = \{pts_1.f_1, ..., pts_n.f_n\}$. We traverse every field access $x.f$ in the program and compute its effect $pts(x).f$. If the effect intersects with the atomic set $atomS$, i.e., $\bigvee_i (f == f_i \&\& pts(x) \cap pts_i \neq \emptyset) = true$, we put the field access into the access set ACC^{atomS} of the atomic set $atomS$.

VI. Threats to Validity

Due to the conservativeness of the static pointer analysis, our access identification (Section V) may treat the irrelevant objects as the same abstract object representation, and therefore, determine the accesses of the irrelevant objects as the accesses of the same atomic set conservatively.

We put the identified accesses in an atomic block without distinguishing the write accesses and the read accesses. Such treatment may introduce unnecessary atomic blocks, e.g., the atomic block that protects the accesses to a shared variable which is read only by all threads.

Our atomic block identification (Section III) always finds the minimal atomic block to contain the accesses. In case that the accesses are inside the body of a loop, we find the minimal atomic block inside the loop body to contain them, which guarantees the atomicity only within each iteration. However, the atomicity may be required throughout the loop. This is an open problem faced by all region identification approaches [12], [18], [25]. We mitigate the problem by producing programmers the warnings.

In order to preserve the high performance of the program, we identify the minimal atomic blocks. Minimizing the atomic regions is a common practice that can usually improves the performance. Of course, there is no theoretical guarantee of performance improvement. Besides, the minimal atomic region is widely adopted as the oracle in the region identification researches [12], [18], [25].

VII. Implementation

Most program analyses used in our technique are standard, e.g., the pointer analysis in Section V, the program structure tree in Section III-A, the immediate dominator and closure in Section III-B. We reuse the implementation of these standard analysis provided by the Soot compiler framework.

We implement each atomic block, (n_a,n_b) as the Java synchronization block at the source code level, by injecting the block start, $synchronized(lock)\{$, before $line(a)$ and injecting the block end, $\}$, after $line(b)$. The line number information is extracted from the bytecode, upon which our program analysis is carried out. During the injection at the source code level, we leverage the abstract syntax tree (AST) to account for the syntactical details, e.g., two statements may be co-located in the same line. The lock is allocated for each concrete atomic set (Section V) and initialized as a static field.

The synchronization blocks that are identified to protect the accesses of different atomic sets may overlap or nest, which raises problems. Two synchronization blocks malfunction when they overlap. According to Algorithm 1, they must be two sequential compositions of the blocks with the same parent block. Therefore, we adopt a simple strategy which finds the union of the two sequences and places the two synchronization blocks around it. As a consequence, both synchronization blocks are lengthened but no longer overlap or malfunction.

Two synchronization blocks may introduce the circular-mutex-wait deadlocks when they nest, e.g., the locks l_a and l_b are acquired by the nested synchronization blocks in an order $l_a \rightarrow l_b$, and also acquired in another method by the nested synchronization blocks in an order $l_b \rightarrow l_a$. Given such a deadlock, we adopt a commonly used strategy [7], which substitutes all the occurrences of l_a and l_b to a new lock l_{ab}.

VIII. Evaluation

In our evaluation, we are interested in the following questions.

1) Static metrics. In terms of lexical scope, how conservative are the atomic blocks produced by the whole-program analysis?
2) Correctness. Do the atomic blocks support the general atomicity correctly?
3) Runtime performance. How efficient are the programs with the atomic blocks that are identified automatically?

We study them by comparing three program versions: orig, inf and inf-dis, which correspond to the original version, the version carrying the atomic blocks inferred automatically with the bounded identification (Section IV), the version without the bounded identification. We conduct the evaluation on nine subjects (Table I) from Java Stamp benchmark suite [3], which were written[5] in a dialect of Java and then ported to Java with the minor effort. These benchmarks are widely used [30], [1], [25] in the atomicity research, and run stably. Besides, they are not too big for manual inspection (Section VIII-A). The evaluation is conducted on a x86_64 Dell workstation with 3.0GHz quad-core Intel Xeon X5450 processors based on Core 2 micro-architecture (8 cores total). The server has 16GB RAM and 6M L2 caches, runs Ubuntu 8.04 with a Linux 2.6.22 kernel, and uses Sun's 64-Bit 1.6.0 JVM.

Declaration of atomic sets and bounding scopes The atomic sets are needed to compute the inf version and the inf-dis version, and the bounding scopes are needed in the computation of the inf version. We extract the atomic set declarations through observing the atomicity originally specified as the atomic blocks in the orig version. To help understanding the atomicity, we compute the shared fields accessed in each atomic block and map them to the object graph. We find the atomicity is often required by the popular data structures such as `List` and `Vector`. Our atomic set declarations for these data structures are almost identical to these written by Dolby et al. [4]. Plus, the fields in a declared atomic set (DAS) are often weakly connected in the object graph (Figure 4). As for the bounding scope, some of the original programs contain the annotations that specify independent phases explicitly. If such annotations exist, we treat the entry method of each phase as the bounding method (Section IV).

A. The static metrics of inferred atomic regions

In this section, we measure the lexical scopes of the identified atomic blocks by comparing them with the original atomic blocks in the orig version. The whole-program identification takes around 30 seconds for each benchmark, which is dominated by the pointer analysis. The comparison results are shown in Table I. The first columns report the total number of classes, fields, methods and source lines in the whole program. In the rest columns, we show the details about the atomic blocks. As seen, each benchmark corresponds to multiple rows, each corresponding to a declared atomic set (DAS). For each DAS, we show its root class (RootClass), the number (F)[6] of field declarations involved, the number (IAS) of identified atomic sets (IAS) and the number (AIF) of abstract instance fields (Section V). Also, we show the number ($Bound$) of bounding scopes, the number (AB) of atomic blocks (AB), the *total* lines of code (LOC) in the atomic blocks. Specially, for Columns AB and LOC, we use the "x/y/z" form to refer to the values of the versions (`inf`/`inf-dis`/`orig`) respectively. The symbol "-" in Column LOC denotes that the inferred atomic regions are not in the same method as the original atomic regions, and therefore, the absolute value is not meaningful.

From Column F, we see that 1-5 fields are typically involved in a DAS. In entries with "-", we specify the atomic set with the wild card (*)[7], meaning that all fields transitively referenced by the root class form a DAS. Specifying the atomicity using the atomic set notion incurs low cost because each atomic set declaration involves very few fields and requires only the modular reasoning, e.g., programmers only need to inspect the field declarations and do not need to inspect the field accesses. In Column IAS, one IAS is typically constructed for each DAS. Multiple IASes (e.g., the entry with "2") may be constructed for a DAS, which come from different bindings of the DAS (Section V).

Compared to the original atomic blocks in the orig version, the identified atomic blocks may be finer, identical, or coarser. Take the inf version for example, it contains 2 finer ABs, 18

[5]URL: http://demsky.eecs.uci.edu/software.php

[6]When counting the fields, we treat the array element as one special field.

[7]It is a grammar sugar. We translate the wild card into fields internally by looking up the type hierarchy.

identical ABs, 17 coarser ABs. As for the 2 finer ABs, they are finer because the ABs in the orig version include some local accesses unnecessarily. As for the 17 coarser ABs, they are coarser for two reasons: (1) the static analysis judges some safe accesses as unsafe conservatively and protect them in the ABs unnecessarily. 10 out of 17 coarser ABs fall into this category. For example, a field is initialized in the constructor by the main method and afterwards read only by threads. The field accesses are safe but the static analysis conservatively judges the them as unsafe. (2) All accesses (in the same bounding scope) of the same atomic set are protected by one AB. 7 of 17 coarser ABs fall into this category. It is due to the natural limit of the data-centric atomicity (DCA, i.e., atomic set) [27] that DCA treats the accesses at different sites homogeneously. As for the identified ABs in the inf-dis version, there are 1 finer AB, 10 identical ABs, 16 coarser ABs.

We present the details for the inf version of each benchmark.

Bayes. It has 4 atomic set declarations, which describe the atomicity in classes `Vector`, `List`, `Query`, and `Learner` respectively. Each DAS involves 1-4 fields. We have 2 bounding scopes for all the DASs as the thread code is separated into two independent phases. Interestingly, the inf version contains an AB finer than the corresponding AB in the orig version, which contains some local computations unnecessarily.

On the other hand, the inf version contains coarser ABs as compared with the orig version: some protect the safe accesses which are conservatively judged as unsafe by the static analysis, some protect the accesses distributed in different ABs in the orig version due to the natural limit of DCA. For the former case, a more precise static analysis can mitigate the problem. For the latter case, interestingly, the coarser ABs in the Bayes benchmark do not introduce high performance penalty (around 1%, as shown in Section VIII-C). Inspecting the code, we find the AB in the inf version encloses the whole branch structure, while the ABs in the orig version enclose the branch bodies for each. The inf version, although contains the lexically coarser AB, executes only one branch body each time and performs similarly to the orig version.

Genome. It is a gene sequencing application. It contains three DASes. The inf version contains coarser ABs which protect the safe accesses unnecessarily. The rest ABs in the inf version are identical to the ones in the orig version. **Intruder** is an application for the network intrusion detection. The ABs in the inf version are identical to the ones in the orig version.

KMeans, Laby, Matrix. For the three benchmarks, no bounding scopes are present, therefore, the inf version is identical to the inf-dis version. In Laby, one inf AB spans over two orig ABs. Other inf ABs are identical to the corresponding orig ABs. In the entry with "-", we declare a DAS with the wild card representing all the fields transitively referenced by the class `Solve_Arg`.

SSCA2 The first three rows declare the DASes in a class and the second three rows declare the DASes in another class. In the first three rows, the ABs in the inf version are coarser than those in the orig version because they include the safe accesses unnecessarily. In the rest three rows, the ABs in the inf version are identical to those in the orig version.

Vacation, Yada. As no bounding scopes are present, the inf version and the inf-dis version are identical. The identified ABs are coarser than the original ABs mostly due to the the natural limit of the DCA. Consider the example from Yada , the `push` and `pop` operations access the same atomic set, therefore, our identification analysis places them in the an AB, however, in the orig version, the operations are placed in two independent ABs based on the specific domain knowledge.

Programs	$Class$	$Field$	$Method$	$Line$	RootClass	F	IAS	AIF	$Bound$	AB	LOC
Bayes	33	156	181	2844	Leaner	1	1	1	2	2/1/2	4/-/7
					List	3	1	4	2	2/1/4	97/-/20
					Vector	4	1	5	2	2/1/2	132/-/2
					Query	1	1	6	2	1/1/2	66/66/6
Genome	21	82	87	1332	Sequencer	1	1	1	3	3/1/1	20/100/5
					List	5	1	6	3	2/1/1	43/59/12
					constructEntry	7	1	54	3	1/1/1	11/11/11
Intruder	18	83	74	1352	Queue	3	1	4	0	1/1/2	15/15/10
					RBTree	10	2	14	1	1/1/1	44/44/75
KMeans	7	36	25	599	GlobalArgs	1	1	6	0	1/1/1	4/4/4
					GlobalArgs	1	1	1	0	1/1/1	2/2/2
					GlobalArgs	1	1	1	0	1/1/1	1/1/1
Laby	15	74	82	1250	Queue	3	1	4	0	1/1/1	5/5/5
					List	3	1	4	0	1/1/1	1/1/1
					Solve_Arg	-	1	14	0	1/1/2	22/22/11
Matrix	2	12	4	121	Matrix	5	1	9	0	1/1/1	12/12/12
SSCA2	23	129	73	2440	ComputeGraph	1	1	1	10	1/1/1	6/6/5
					ComputeGraph	1	1	1	10	1/1/1	4/4/3
					ComputeGraph	2	1	2	10	3/1/1	29/53/15
					GenScalData	1	1	1	10	2/1/2	4/160/4
					GenScalData	1	1	1	10	2/1/2	6/378/6
					GenScalData	1	1	1	10	1/1/1	2/2/2
Vacation	19	87	133	1812	Manager	-	1	16	0	1/1/3	99/99/67
Yada	21	87	134	2144	Global_arg	2	1	2	0	1/1/1	3/3/3
					Heap	-	1	63	0	1/1/2	26/26/4
					Element	-	1	48	0	1/1/1	14/14/8
					Mesh	-	1	57	0	1/1/1	3/3/3

TABLE I: Metrics of the atomic block identification.

B. Correctness

First, the correctness of the inf and inf-dis versions cannot be proved through experiments. The experiments in this section are designed to show that the versions are likely correct. We run the inf, inf-dis and orig versions for 50 times with the same inputs. We find each version runs deterministically, producing the same output each time. Determinism of the outputs is often taken as the symptom of the correct runs [15], [2]. Besides, the inf version and the inf-dis version have the outputs that are identical to the output of the orig version, which indicates their functional correctness.

Finally, from the conceptual view, the inf and inf-dis versions contain the coarser ABs as compared to the orig version (The only exception is when the orig version includes some local computations unnecessarily.), leading to fewer run-time interleavings and accordingly fewer buggy interleavings. Therefore, they are safer, if not completely safe.

C. Performance of the program with the inferred regions

In this section, we study the impact of the identified ABs on the performance. We run the three versions with the inputs shipped with the benchmarks. For each version, we run it with different thread configurations (threads=1,2,4,8,12,16) and collect the running time. Specially, the number of threads is required to be a power of 2 for the benchmark SSCA2. To collect the running time, we run the program with each thread configuration for 10 times and compute the average value. 10 runs are sufficient in our experiment as the 10 runs exhibit very similar running time. For each run, we start the timer before the running threads are started and stop the timer after the running threads are joined.

The performance comparison is shown in Figure 5. We also compute the slowdown of the inf version as compared to the orig version, e.g., $(time(inf) - time(orig))/time(orig)$, as shown in Column inf of Table II. We show the average

TABLE II: Performance slowdown. Minus value means that the orig version is slower.

Program	inf		inf-dis	
	avg (%)	max (%)	avg (%)	max (%)
Bayes	1.0	10.3	4.8	12.7
Genome	7.4	11.3	7.9	13.6
Intruder	6.8	13.0	3.1	9.0
KMeans	0.1	0.6	0.1	0.5
Laby	-2.5	0.1	1.0	1.5
Matrix	-0.1	0.0	0.0	0.0
SSCA2	1.6	5.1	29.1	42.8
Vacation	5.8	12.7	5.9	13.1
Yada	0.1	5.0	0.1	4.6

slowdown (avg: %) and the maximal slowdown (max:%) among the thread configurations. We show the slowdown of the inf-dis version in Column inf-dis of Table II.

As seen, the inf version slows down the runtime by 0%-7% on average, and slows down the runtime by 13% maximally, which suggests our approach is a reasonable candidate for replacing the manual atomic block specification. The inf-dis version behaves similarly except for the benchmark SSCA2 where the inf-dis version incurs 29% slowdown on average and 43% slowdown maximally. Inspecting the code, we find inf-dis version contains the overly coarse ABs as compared to the inf version, which highlights the power of the bounded identification (Section IV).

IX. Related Work

Atomic region inference Besides the inference work built upon the static analysis, several work infers the atomic regions relying on the dynamic analysis or the trace analysis. Lu et al. [20] detect the atomic regions dynamically based on an assumption of correct runs: the code from the same atomic region is frequently accessed without being non-serializably interleaved. Sharing the same spirit, Muzahid et al. [21] and Weeratunge et al. [28] analyze the traces of the correct runs to infer the atomic regions.

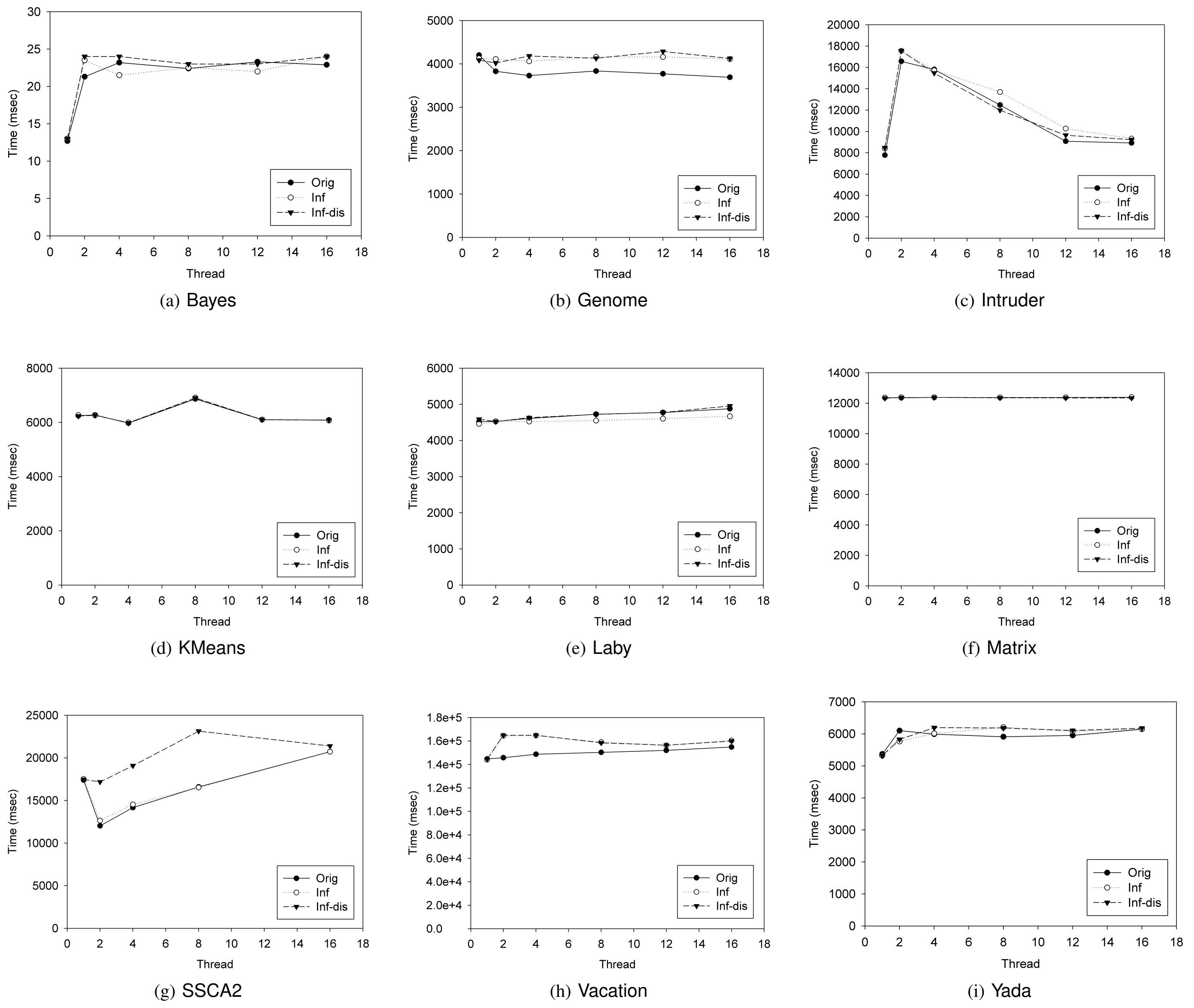
Automatic fixing of atomicity violations Automatic fixing of the atomicity violations is recently proposed [12], [28]. The input is often the dynamically detected atomicity violations, and the output is the atomic regions that remove these atomicity violations. The atomic block identification proposed in Section III can be used to fix the atomicity violations too. Different from the existing approaches, our identification produces the minimal *structured* atomic regions, which preserve the maintainability or code readability well.

Atomic set serializability It is a correctness criterion proposed by Vaziri et al. [27]. Different from the traditional serializability which describes the correct behaviors with respect to the whole heap, atomic set serializability is finer-granular as it describes the correct behaviors with respect to a subset of memory locations. This criterion is adopted commonly for detecting concurrency bugs [14], [8], [16].

Atomicity specified on data Besides Vaziri et al. [27], other researchers also observe and utilize the benefit of specifying the atomicity on data. In the area of distributed system, consistency of data is a commonly desired property. To preserve it, Weihl et al. [29] and Herlihy et al. [11] suggest to use the the atomic object, a type of objects created with their own synchronization and recovery functionality. With such objects, they achieve the serializable and recoverable executions. In the area of database, Härder et al. [9] [6] design a prototype PRIMA to support the engineering applications. In PRIMA, the molecular object, consisting of attributes, is treated as an atomic unit of data and manipulated atomically. In the software transactional memory research, DSTM2 [10] and XSTM [23] allow specifying the atomic objects by annotating the types with the keyword `@atomic`.

X. Conclusions

We present a whole-program analysis that identifies the accesses that require the atomicity and identifies the mini-



(a) Bayes (b) Genome (c) Intruder

(d) KMeans (e) Laby (f) Matrix

(g) SSCA2 (h) Vacation (i) Yada

Fig. 5: Performance comparison.

mal atomic block (structured atomic region) to protect them from buggy interleavings. Our analysis is automatic, and it preserves the maintainability, the code readability, the high performance and supports the general atomicity throughout the program. We also allow programmers to bound the atomic block identification, which improves the quality of the produced atomic blocks. The evaluation shows, the inf version slows down the program by around 5%, which represents a reasonable alternative to manual atomic block specification. The inf version outperforms the inf-dis version, especially in the SSCA2 benchmark where the inf version is 25% faster, which highlights the advantage of the bounded identification.

REFERENCES

[1] Zachary Anderson and David Gay. Composable, nestable, pessimistic atomic statements. In *OOPSLA*, 2011.

[2] Jacob Burnim and Koushik Sen. Asserting and checking determinism for multithreaded programs. In *FSE '09*.

[3] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing.

In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*.

[4] Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. A data-centric approach to synchronization. *TOPLAS '12*.

[5] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04*.

[6] Michael Gesmann, Andreas Grasnickel, Theo Härder, Christoph Hübel, Wolfgang Käfer, Bernhard Mitschang, and Harald Schöning. Prima - a database system supporting dynamically defined composite objects. *SIGMOD Rec. '1992*.

[7] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *PACT '07*.

[8] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE '08*.

[9] Theo Härder, Klaus Meyer-Wegener, Bernhard Mitschang, and Andrea Sikeler. Prima - a dbms prototype supporting engineering applications. In *VLDB '87*.

[10] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06*.

[11] M.P. Herlihy and J.M. Wing. Avalon: language support for reliable distributed systems. In *17th Symposium on Fault-Tolerant Computer Systems*.

[12] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *PLDI '11*.

[13] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: computing control regions in linear time. In *PLDI*, 1994.

[14] Nicholas Kidd, Thomas Reps, Julian Dolby, and Mandana Vaziri. Finding concurrency-related bugs using random isolation. In *VMCAI '09*.

[15] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI '07*.

[16] Zhifeng Lai, S.C. Cheung, and W.K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE '10*.

[17] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, 2002.

[18] Peng Liu and Charles Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *ICSE*, 2012.

[19] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP '07*.

[20] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS '06*.

[21] Abdullah Muzahid, Norimasa Otsuki, and Josep Torrellas. Atomtracker: A comprehensive approach to atomic region inference and violation detection. In *MICRO '10*.

[22] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI '06*.

[23] Cyprien Noël. Extensible software transactional memory. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering '10*.

[24] Weslley Torres, Gustavo Pinto, Benito Fernandes, João Paulo Oliveira, Filipe Alencar Ximenes, and Fernando Castor. Are java programmers transitioning to multicore?: a large scale study of java floss. In *SPLASH '11 Workshops*, 2011.

[25] Gautam Upadhyaya, Samuel P. Midkiff, and Vijay S. Pai. Automatic atomic region identification in shared memory spmd programs. In *OOPSLA '10*.

[26] Gautam Upadhyaya, Samuel P. Midkiff, and Vijay S. Pai. Using data structure knowledge for efficient lock generation and strong atomicity. In *PPOPP '10*.

[27] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06*.

[28] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jaganathan. Accentuating the positive: atomicity inference and enforcement using correct executions. In *OOPSLA '11*.

[29] William Weihl and Barbara Liskov. Implementation of resilient, atomic data types. *TOPLAS '1985*.

[30] Ferad Zyulkyarov, Tim Harris, Osman S. Unsal, Adrían Cristal, and Mateo Valero. Debugging programs that use atomic blocks and transactional memory. In *PPOPP*, 2010.