Constraint-Based Locality Analysis for X10 Programs

Qiang Sun\textsuperscript{1,3,4}  Yuting Chen\textsuperscript{2,4}  Jianjun Zhao\textsuperscript{2,3}

\textsuperscript{1}Department of Computer Science \& Engineering and \textsuperscript{2}School of Software, Shanghai Jiao Tong University, Shanghai, China
\textsuperscript{3}State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
\textsuperscript{4}Shanghai Key Laboratory of Computer Software Testing \& Evaluating, Shanghai, China

\texttt{sun-qiang@sjtu.edu.cn, chenyt@cs.sjtu.edu.cn, zhao-jj@cs.sjtu.edu.cn}

Abstract

X10 is a HPC (High Performance Computing) programming language proposed by IBM for supporting a PGAS (Partitioned Global Address Space) programming model offering a shared address space. The address space can be further partitioned into several logical locations where objects and activities (or threads) will be dynamically created. An analysis of locations can help to check the safety of object accesses through exploring which objects and activities may reside in which locations, while in practice the objects and activities are usually designated at runtime and their locations may also vary under different environments. In this paper, we propose a constraint-based locality analysis method called Leopard for X10. Leopard calculates the points-to relations for analyzing the objects and activities in a program and uses a place constraint graph to analyze their locations. We have developed a tool to support Leopard, and conducted an experiment to evaluate its effectiveness and efficiency. The experimental results show that Leopard can calculate the locations of objects and activities precisely.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Concurrent, distributed, and parallel languages; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program analysis

General Terms Languages, Analysis

Keywords Locality Analysis, Points-to Analysis, Concurrency

1. Introduction

Partitioned Global Address Space (PGAS) is a parallel programming model offering a shared address space [23]. The shared address space can be further partitioned into several logical places, where activities (or threads) can be dynamically created or enabled, and objects can be dynamically allocated or copied. Some PGAS programming languages such as X10 proposed by IBM also require the activities not directly access the remote objects, thus their compilers usually make some runtime safety checks for object access operations [4].

Locality analysis is an analysis technique that can help check the safety of object accesses in PGAS programs. It explores which objects and activities may reside in which locations and identifies whether an activity accesses the field of a remote object. However, locality analysis usually faces some challenges in practice. One main challenge is that objects or activities usually need to be dynamically created, and their places may be designated at runtime (e.g., a place expression is used to assign a place an activity resides in), while the space partitions can vary under different environments. Thus it is not easy, if not impossible, to statically infer the place in which objects and activities reside. In addition, some PGAS programming languages, such as X10, support cyclic places, which denote that each place has a predecessor and a successor and all places are logically organized in a circle. It also sets barriers to locality analysis in that places may be designated and evaluated through using expressions.

In this paper we propose a constraint-based locality analysis called Leopard (Locality Analyses of Partitioned Global Address Space Programs) for X10 programs. We first define a subset-based constraint system to describe the constraints on the objects, activities, and their places, and then calculate the possible values of each place expression and model the queries on all places. Leopard also analyzes the points-to relations in the program and uses a place constraint graph (PCG) to achieve the places of the objects and activities. We also take an activity- and heap-sensitive analysis of locality information in order to improve the precision of analysis.

The paper makes the following contributions:

- **Abstraction.** We define the places taking into account different topology structures of places, query operations (i.e., place expressions), and a locality model which can be specialized for a specific topology structure. We also define the satisfiability relation between the locality model and a set of constraints on the program.
- **Algorithm.** We present a context-insensitive algorithm and a context-sensitive one to achieve the constraints on objects and activities. Especially, our context-sensitive locality analysis trades off between the cost and precision through choosing different context lengths.
- **Tool support and experiments.** We have developed a tool to support Leopard, and conducted two experiments to measure its performance and precision. The experimental results show that Leopard can be used to analyze the locality information precisely.
- **Application.** The locality information computed by Leopard can be used to infer place equivalences with respect to different topology structures and check safe accesses to objects in X10 programs. We have implemented an application to utilize the locality information to verify safe accesses to instance fields, and also conduct an experiment to compare it with a place type system [4]. The result shows that Leopard provides with more support in checking safe accesses.
The remainder of the paper is organized as follows. Section 2 presents an example to illustrate how the analysis algorithm works. Section 3 presents the foundation of context-insensitive locality analysis. Section 4 presents the details of the algorithm for the context-insensitive locality analysis of X10 programs. Section 5 presents the activity-sensitive and heap-sensitive locality analysis. Section 6 describes an experiment to evaluate the performance, precision and usability of Leopard. Section 7 discusses the related work. Section 8 concludes this paper.

2. Background and Example

2.1 Place and Activity

An X10 program consists of activities executing in places plus the ability to access the remote data by creating and accessing references to the data. We next describe two language constructs satisfying the X10 specification (v2.0) [22] which needs to be focused on during our analysis.

Places. In X10, a place is a collection of resident activities and objects. Programmers can use the configuration options to set the number of places. Thus all places can be initialized before the program is executed. X10 has a type Place which denotes a certain place of execution. Values of a place-typed variable can be obtained in the following ways:

- `here` denotes a place where an activity is executing.
- Given a reference variable `v`, `v.home()` is a method invocation that returns the value of the place where the object that `v` points to resides.
- X10 organizes the places of a program through using a cycle. Given a place `p`, `p.next()` returns its successor, and `p.prev()` returns its predecessor.

Asynchronous Activities. X10 supports asynchronous through spawning and executing activities: an activity can dynamically spawn other activities in local or remote places, e.g., `async(p)` `S` creates an activity to execute the statement `S` in place `p`; once an activity (resp. object) is created in a place, it cannot migrate to another place; an activity cannot access a remote object directly, but needs to spawn remote activities to access the remote objects; a root activity is spawned in `Place(0)` by default and it corresponds to invoke the main method.

Figure 1 shows an example of X10 program containing two classes, `T` and `BoxT`, where class `T` has a field `x` of the type `int`, and class `BoxT` has a field `data` of the type `T`. An execution of the program creates a root activity which invokes the `main` method in class `BoxT`, and spawns a child activity (see lines 13-17) to update `b.data`. Note that the child activity is created and enabled in the place assigned by a place expression `q.next()`. In this example, `b.data = r;` (line 15) may cause an access violation if `b` points to a remote object.

2.2 An Illustrative Example

We use the program in Figure 1 to illustrate how Leopard works. Especially, we achieve different locality graphs for different topology structures of places in the analysis.

The first step of our locality analysis is to build an activity nesting graph (ANG) for the program, where the nodes of the ANG represent the methods and activities in the program and the edges represent the method invocations and activity creations. As Figure 2(b) shows, the ANG contains two nodes representing the root and the child activities, and one edge from the root node to the child node representing the creation of the child activity. Since the root activity invokes only one method `main()`, we omit the node for the `main()` method of Figure 2(a).

The second step is to build a place constraint graph (PCG) for each activity or method in the ANG. A PCG describes the creations of objects, points-to assignments, and place expressions. Figures 3(a) and 3(b) represent the PCGs for the root and the child, respectively. The node `cp_root` (resp. `cp_child`) represents the place in which the root activity (resp. the child one) resides. In addition, the root activity is spawned in `Place(0)` by default.
3. Foundation of Locality Analysis

3.1 X10 Program Representation

We next define some program notations following by three abstract domains representing three important program properties of X10 language (v2.0) [22]. These properties are used to capture the queries of places and object-oriented features.

A program in X10 can be represented as a collection of class types Class, which refer to the classes in the program.

A set \( F \) is defined to include all non-static fields, each of which can be accessed using the form \( v.f \), where \( v \) is a variable with a class type. \( F \) can be divided into a set \( F_0 \) containing the fields of class type and a set \( F_\rho \) containing the fields of place type.

A set \( M \) is defined to include all class methods.

A set \( R \) is defined to include all local variables, formal parameters of methods in \( M \) and static fields in the classes. \( R \) can be divided into a set \( P \) containing the variables of place type (and the auxiliary variables introduced by the analysis) and a set \( V \) containing the variables of class type.

A set \( A \) is defined to include all activity names. Since an activity is created and its place may be evaluated at runtime, we also define a mapping function to track all potential places with respect to an activity or method

\[
\mu: A \cup M \rightarrow P
\]

For example, given the program in Figure 1, we have \( \mu([a_{\text{root}}]) = \text{ep}_\text{root} \) and \( \mu([a_{\text{child}}]) = \text{ep}_\text{child} \).

A set \( O \) is defined to include the names of all objects created by the object allocation statements (e.g., \( v = \text{new } C() \)). In a context-insensitive analysis, each object name corresponds to one allocation statement.

Let \( MP \) be the maximum number of places designated at configuration time. A set \( PC \) is defined to model the place constants

\[
PC = \{0, 1, ..., MP - 1\}.
\]

In order to take an \( MP \)-independent analysis, we also define a set \( PN \) to include all place names. Each place name is specified by using an integer, and instantiated to a place constant at runtime by

\[
\text{Eval}_{MP}: PN \rightarrow PC \cong \chi z. z \mod MP,
\]

where \( \mod \) denotes the modulo operation.

For a given name \( z \), the query of its next (resp. previous) \( n^{th} \) place can be achieved by \( z + n \) (resp. \( z - n \)). For a name set \( S_{pn} \subseteq PN \), we define the right and the left shift queries by

\[
S_{pn} \triangleright n = \{ z + n | z \in S_{pn} \}, \quad S_{pn} \triangleleft n = \{ z - n | z \in S_{pn} \}.
\]

Especially, we use \( \top \) to denote the upper bound of the place name sets. We have

\[
\top \triangleleft n = \top, \quad \top \triangleright n = \top.
\]

We use \( \wp(S) \) to denote the powerset of a set \( S \). Three abstract domains are defined for modeling the program properties (e.g., locality information and points-to relations):

- The points-to analysis computes points-to relations between variables of pointer types and their allocation sites. There are two kinds of pointer variables:
  - variable \( v \in V \), and
  - object field \( o.f \), where \( o \in O \) and \( f \in F_o \).

Then a points-to relation function is

\[
\rho \in \text{AbstractReference} = (V \cup (O \times F_o) \rightarrow \wp(O)).
\]

- One object allocation statement may create an object in different places at runtime. In order to track the objects created by one statement in different places, we define

\[
\sigma \in \text{AbstractObject} = (O \rightarrow \wp(PN)).
\]

- In order to model the relationships between the place-typed variables and their values, we define

\[
\theta \in \text{AbstractPlace} = (P \cup (O \times F_p) \rightarrow \wp(PN)).
\]
3.2 Locality Analysis Specification

The principle of context-insensitive locality analysis is to define an abstract specification describing a system of constraints and a domain

\[
\text{AbstractReference} \times \text{AbstractObject} \times \text{AbstractPlace}
\]

and to calculate the least element of the domain that satisfies the constraints. The result of the locality analysis can be defined by a tuple \((\rho, \sigma, \theta)\). The formulation of the abstract specification is in the form of

\[
(\rho, \sigma, \theta) \models s
\]

where \(s\) is a statement in an X10 program and \(\models\) represents the largest acceptable relation that satisfies the abstract specification. \(\models\) is defined in the clauses shown in Figure 5.

- **[seq]**, **[if]** and **[async]** are clauses related to control flows. Each statement within a control structure must be analyzed in a consistent way using \((\rho, \sigma, \theta)\). **[seq]** and **[if]** define the acceptable analysis result for the structures of sequence and branch, respectively. These two structures have the same semantics as those in Java programs. **[async]** defines the acceptable analysis result for activity creation. In **[async]**, \(a\) is the activity created by \(\text{async}(p)\) and \(\mu(a)\) is the associated place-type variable.

- **[copy]**, **[obj]**, **[load]** and **[store]** are clauses for points-to analysis. These four clauses are inspired by the algorithm of Andersen [3]. Compared with the equality-based analysis [17], Andersen’s analysis approach is more precise due to its one-way propagation of points-to sets [7]. **[copy]** says that a copy statement \(v_1 = v_2\) makes the points-to set of \(v_2\) a subset of that of \(v_1\), **[obj]** says that an object creation statement \(v = \text{new } C()\) allocates a new object \(o\) of type \(C\), \(o\) makes \(o\) an element of the points-to set of \(v\), and records the place names of the object \(o\). Here \(\text{new } C()\) represents the current activity or method. **[load]** says that a load statement \(v_1 = v_2.f\) makes the points-to set of \(o.f\) be a subset of that of \(v_1\), where \(o\) is the object in the points-to set of \(v_2\). Similarly, **[store]** says that a store statement \(v_1.I \leftarrow v_2\) makes the points-to set of \(v_2\) a subset of that of \(o.f\), where \(o\) is the object in the points-to set of \(v_1\).

This clause is necessary in the interprocedural analysis.

4. Context-Insensitive Locality Analysis Algorithm

Our context-insensitive locality analysis includes constructing an activity nesting graph for the objective program as well as its place constraint graph, and then solving the place constraint graph.

4.1 Building Activity Nesting Graph

An activity nesting graph (ANG) describes the spawns of activities and invocations of methods in an X10 program. An ANG can be formalized as \((M \cup A, E_{ang})\), where \(E_{ang}\) contains all the edges in the form \(n_1 \rightarrow n_2\), where \(n_1, n_2 \in M \cup A\). Thus an edge 

\[
e \in E_{ang}
\]

can be either an activity creation edge whose target is an activity node or a method invocation edge whose target is a method node. An ANG is constructed by traversing the statements of the form \(\text{async}(p)\) or \(r.m()\). When an activity \(a'\) (a statement \(\text{async}(p)\)) is defined in a method \(m\) (resp. in an activity \(a\)), an activity creation edge from \(m\) (resp. \(a\)) to \(a'\) is added into the ANG. When a method call \(r.m()\) is included in a method \(m\) (resp. in an activity \(a\)), a method invocation edge from \(m\) (resp. \(a\)) to \(m'\) is added into the ANG. If a method invocation is polymorphic, the edges from the caller to all potential target methods are added into the ANG.

4.2 Building Place Constraint Graph

A place constraint graph (PCG) captures the constraints on the statements in a method or activity. A node in a PCG can be

- a variable node representing a variable in \(V\),
- an object node representing an object in \(O\),
- a place-typed variable node representing a place-typed variable in \(P\), or
- a place node representing a place name in \(PN\).

An edge represents a constraint on nodes and can be

- an allocation edge \(o \in O \rightarrow v \in V\) in the set \(E_A\),
- a copy edge \(v_1 \in V \rightarrow v_2 \in V\) in \(E_C\),
- a load edge \(v_1 \in V \overset{f_{E_L}}{\rightarrow} v_2 \in E_L\).
Add a store edge to \( E_S \), an allocation edge to \( E_A \), and a locality edge to \( E_{loc} \).

\[
v_1 = v_3.f \quad \text{(store edge to } E_S) \quad \text{Add a store edge to } E_S.
\]

\[
v_1 \rightarrow v_3 \quad \text{(store edge to } E_S) \quad \text{Add an allocation edge to } E_A.
\]

\[
v_1, f = v_3 \quad \text{(store edge to } E_S) \quad \text{Add a locality edge to } E_{loc}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add an edge } (i \rightarrow p) \text{ to } E_{pi}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a place edge to } E_{pi}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a next-place edge to } E_{ni}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a prev-place edge to } E_{pi}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a place-copy edge to } E_{pc}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a place-copy edge to } E_{pc}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a place-copy edge to } E_{pc}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]

\[
p = \text{place}(i) \quad \text{(store edge to } E_S) \quad \text{Add a load edge to } E_{Li}.
\]
Algorithm 1 is $O(N^4E)$ because (1) The complexities of solving the constraints of different types are shown in Table 2, where the complexities of propagation operation of copying one points-to set (resp. place set) and adding one place name to a place set are linear and $O(1)$, respectively. (2) The repeat-until loop at line 5 needs to execute for $O(N)$ times. (3) The complexity of detecting the loops is $O(N^3)$ because it can be expressed as a CFL-reachability problem [16].

5. Activity-Sensitive Locality Analysis

The precision of Leonard is fair because the interprocedural analysis does not distinguish the activities spawned in different contexts, and thus some place sets are propagated along with some infeasible paths. Figure 6 shows an example of X10 program, where the activities $a_1$ and $a_2$ create the objects $o_1$ and $o_2$, respectively, and then invoke the method $b.m()$. In a context-insensitive analysis, this points to the objects $o_1$ or $o_2$ and $p$ is either $Place(1)$ or $Place(2)$. Thus the activity $a_3$ can be spawned at $Place(0)$ or $Place(1)$ after $p.yaml()$ is evaluated. Therefore, $this.data=...$ is unsafe because this may refer to an object whose place is different from that of $a_3$. However, in a context-sensitive analysis, we distinguish the case of $a_1$ spawning $a_3 ([a_1,a_2])$ and $a_2$ spawning $a_3 ([a_2,a_3])$. In the former case, it can be deduced that $this$ only points to $a_1$ and the place of $o_1$ is same as that of $a_3$. Thus we can infer that $this.data=...$ is safe. In the latter case, this only points to $a_2$ and $this.data=...$ is also a safe access.

We propose an AS (Activity-Sensitive) locality analysis which adopts an activity sequence to model a context. A context defines a sequence of activities spawned in the last $k$ steps. A context $\delta \in \Delta = A^*k$ is generated by traversing an ANG, where $k$ is a user-defined constant value.

A context generation algorithm (see Algorithm 2) is adopted in the AS locality analysis for generating all the contexts and binding each context to the activities and methods. At line 7, $(a_0 \vdash a_{1 \in A})$ denotes that the activity $a_0$ directly spawns the child activity $a_1$ or through a chain of method invocations. At line 9, the new context $\delta'$ is generated by $[\delta, a_1]L$ which represents the $k$-rightmost truncation of the sequence of $[\delta, a_1]$. At line 19, $(a \vdash m_{E, M})$ denotes that the method $m$ is reachable with respect to the activity $a$.

Our AS locality analysis then combines the context information and the activities, methods, objects and variables, to form the new representation

\[
\delta \in \Delta, a \in A, m \in M, o \in O, v \in V, p \in P, \\
(\delta, a) \in A', (\delta, m) \in M', \mu : \Delta \to P', \\
(\delta, o) \in O', (\delta, v) \in V', (\delta, p) \in P',
\]

where the function $\mu'$ maps each context $\delta$ to a place-typed variable $(\delta, cp)$ denoting the places where the activity (resp. methods) of $\delta$ is executed. $\mu'$ is different from $\mu$ in Section 3.2 in that $\mu'$ requires all methods of the same context share one place-typed variable.

### Table 2. Complexity of Handling Different Constraints

<table>
<thead>
<tr>
<th>Complexity</th>
<th>$E_{PC}$</th>
<th>$E_{PC}$</th>
<th>$E_{PC}$</th>
<th>$E_{PC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(NE)$</td>
<td>$O(NE)$</td>
<td>$O(NE)$</td>
<td>$O(NE)$</td>
<td>$O(NE)$</td>
</tr>
<tr>
<td>$O(V)$</td>
<td>$O(V)$</td>
<td>$O(V)$</td>
<td>$O(V)$</td>
<td>$O(V)$</td>
</tr>
<tr>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>$O(N^2E)$</td>
<td>$O(N^2E)$</td>
<td>$O(N^2E)$</td>
<td>$O(N^2E)$</td>
<td>$O(N^2E)$</td>
</tr>
</tbody>
</table>

### Figure 6. An Example of X10 Program Analyzed in Different Context-Sensitivities

The X10 program shown in Figure 6 is analyzed in different context-sensitivities. The analysis captures the different constraints on object references and localities, and propagates them through the program to identify infeasible paths.
but μ maps each method (resp. activity) to a different place-type variable ep.

The abstract domains are
\[ \rho' \in (V' \cup (O' \times F_o)) \rightarrow \wp(O'), \]
\[ \sigma' \in (O' \rightarrow \wp(PN)), \]
\[ \theta' \in (P' \cup (O' \times F_p) \rightarrow \wp(PN)). \]

Since each activity or method is bound to some context(s), its statements need to be analyzed under current context. We extend the notation of \( \models \) and define a relation \( \models S \).

\[ \langle \rho', \sigma', \theta' \rangle \models S s. \]

Figure 7 shows the clauses corresponding to different statement types. Here δ is the current context of the analyzed statements; \( \rho', \sigma', \theta' \) denote \((\delta, o), (\delta, v), \text{ and } (\delta, p), \) respectively.

Let a be the activity spawned by the async statement in the clause [async]. The statements in a are analyzed under a context \((\delta, a), \mu'(\delta, a)\) looks up the place-type variable under the context \((\delta, a)\). Since the child activity can access the variable declared in the parent activity, CRefEq requires all the shared variables hold the same place name set (resp. points-to set), which is

\( \forall r \in Acc(a) : \varphi'((\delta, r), (\delta, a)_{k,r}) \land \varphi'((\delta, a)_{k,r}, (\delta, r)) \),

where Acc(a) is a set containing all variables defined in the parent of a and accessed by a, and \( \varphi' \) is defined by

\[ \varphi' (r_a, r_b) = \begin{cases} 
\rho'(r_a') \subseteq \rho'(r_b') & r_a', r_b' \in V', \\
\theta'(r_a') \subseteq \theta'(r_b') & r_a', r_b' \in P', \\
false & otherwise.
\end{cases} \]

\( \varphi' \) handles the parameter passing in [inv*].

[obj*] denotes that our analysis is heap-sensitive in that a static object is created under the current context.

6. Experiments

We have developed a tool to support Leopard. Figure 8 shows the framework of the tool: the X10 compiler (ver. 2.0.3) extracts the AST of an X10 program; the T.J. Watson Libraries for Analysis (WALA) [18] generates its intermediate representation; Leopard consumes the intermediate representation and produces the activity nest graph and the intraprocedural PCGs of all methods and activities; after that, Leopard combines the graphs into an intraprocedural PCG and computes the locality graph of the program.

We have conducted two experiments to evaluate the performance, precision and usability of Leopard. We have used 10 benchmark programs from the High-Performance Computing challenge (HPCC) benchmarks, the Java Grande (JG) benchmarks in X10 and the Network-Attached Storage (NAS) benchmarks. Table 3 shows the details of these benchmarks, including the lines of code (LOC), numbers of activities (#Activity), methods (#Method) and method invocations (#Call). Note that X10 language specification varies from version to version, and thus the large scale benchmark programs are not available in our experiments. Leopard also do not completely cover the standard X10 libraries (including 186 classes) in its analysis for cost-effective reason, but just includes all reachable classes in them. In the experiment we ran the tool on a PC (Intel CoreTM 2 Duo CPU 2.40GHz, 2GB memory).

6.1 Performance and Precision

The first experiment was used to evaluate the performance and precision of three analyses: context-insensitive (CI) analysis, one-level AS (activity-sensitive) analysis (i.e., \( k = 1 \)), and two-level AS analysis (i.e., \( k = 2 \)).
Table 3. Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>#Activity</th>
<th>#Method</th>
<th>#Call</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>122</td>
<td>7</td>
<td>22</td>
<td>1</td>
<td>JG</td>
</tr>
<tr>
<td>stream</td>
<td>132</td>
<td>2</td>
<td>10</td>
<td>8</td>
<td>HPC</td>
</tr>
<tr>
<td>sparsemm</td>
<td>345</td>
<td>4</td>
<td>32</td>
<td>25</td>
<td>JG</td>
</tr>
<tr>
<td>series</td>
<td>389</td>
<td>3</td>
<td>14</td>
<td>17</td>
<td>JG</td>
</tr>
<tr>
<td>crypt</td>
<td>565</td>
<td>2</td>
<td>23</td>
<td>25</td>
<td>JG</td>
</tr>
<tr>
<td>mldyn</td>
<td>700</td>
<td>14</td>
<td>16</td>
<td>25</td>
<td>JG</td>
</tr>
<tr>
<td>luact</td>
<td>853</td>
<td>7</td>
<td>24</td>
<td>40</td>
<td>JG</td>
</tr>
<tr>
<td>raytrace</td>
<td>1225</td>
<td>13</td>
<td>60</td>
<td>132</td>
<td>JG</td>
</tr>
<tr>
<td>mg</td>
<td>1868</td>
<td>57</td>
<td>122</td>
<td>248</td>
<td>NAS</td>
</tr>
<tr>
<td>mc</td>
<td>3150</td>
<td>3</td>
<td>83</td>
<td>80</td>
<td>JG</td>
</tr>
</tbody>
</table>

Table 4 shows the sizes (i.e., numbers of nodes/edges) of PCGs and the runtime for each benchmark program. Compared with the CI analysis, the AS analyses have generated some larger scale PCGs and spent more time in solving the PCGs. In the AS analyses, the sizes of PCGs and the corresponding solving time also increased along with the growth of \( k \). One reason is that \( k \) determines the variables and objects to be analyzed: the bigger \( k \) is, the more clones of variables and objects need to be produced in the analyses.

After investigating the experiment results, we believe that the performance of analyses depends on two factors: the size of the PCG and \#Activity. For a program of a small PCG or that of a few activities (e.g., the programs `stream`, `sparsemm`, `series`, and `crypt`), the AS analysis was precise when \( k \geq 2 \), but the cost was not of significant difference from that of the CI analysis. For a program of many activities or that of a large PCG, the cost increases dramatically along with the increase of \( k \). For example, it took the two-level AS analysis more time in analyzing `mg` than in analyzing `mc`.

In order to evaluate the precision of the analysis, we adopt

- Pct.1O which represents the percentage of the points-to set with only one object in order to measure the precision of the points-to analysis. The closer to 100% the Pct.1O is, the more precise the points-to analysis is.
- Pct.1P which provides the percentage of the place-type variable with one place name in order to show the possibility of an object or activity residing in one and only one place. The closer to 100% the Pct.1P is, the more precise the locality analysis is.

Figure 9 shows the Pct.1O and Pct.1P of the three analyses. The x-axes of Figures 9(a) and 9(b) represent Pct.1O and Pct.1P, respectively. It can be seen that both Pct.1O and Pct.1P of the AS analyses are higher than those of the CI analysis. The programs `raytrace` and `mg`, analyzed by the CI analysis, are of low Pct.1Ps (85.57% and 83.67%). The CI analysis of X10 program is instinctively of low precision in that the points-to sets of the formal parameters of a method called by several callers can grow rapidly, which will be further propagated to other variables and therefore reduce the precision of analysis of place-typed variables. Compared with the CI analysis, the AS analysis improves the precision through distinguishing more activities, objects and method invocations and computing different points-to and locality information for the variables and objects in the activities under different contexts.

We have also found that the precision of the points-to analysis can help improve the evaluation of the place expressions. One reason is that the evaluation of the widely used place expressions in form of \( r\cdot\text{home()} \) strongly depends on the points-to set of the variable \( r \). A place-typed variable can be propagated through the accesses of place-typed fields in form of \( v\cdot p \), the evaluation of which also depends on the points-to set of \( v \).

6.2 Safety Checking of Instance Field Access

In X10, the instance field \( v.f \) can be accessed by activities at place \( p \) which is the same place of the object \( v \) points to. A static checking of safety can help reduce the overhead of the runtime locality checking. The report of the unsafe accesses can help to find potential errors in X10 programs.

In the second experiment, we study the safety checking of the instance field accesses, which can be classified into two categories:

- Safe Access (SA) which means that the access is safe with whatever the place topology structures are,
- Conditional Safe Access (CSA) which means the access is safe with respect to certain place topology structures.

Table 5 shows the results of checking of the SAs and CSAs for each benchmark program. In the table, we have compared the SAs computed by the CI and AS analyses with the results inferred by the place type system (PTS) \[4\], and also reported the CSAs computed by the CI and AS analyses.

Table 5 shows that the CI and AS analyses have detected more SAs than PTS has. One reason is that PTS misses some accesses due to the limited inference step. That is, the UNIFICATION algorithm of PTS needs to unify the limited depth of field access path for different variables, and any function recursion needs to be unfolded up to the limited depth. Another reason is that PTS omits the place-shift queries (e.g., \( p\cdot\text{next()} \) and \( p\cdot\text{prev()} \)) and place-typed field accesses (e.g., \( v\cdot p \)). The AS analyses also have obtained more SAs than the CI analysis has. The possible reason is that the AS analyses can achieve a context-specific name set of one place name for both the activities and the objects accessed. In a CI analysis, the name set is usually rough because it may be the union of these context-specific name sets.

Leopard also has the capability of finding the CSAs in that it models the topology structure of places and place-shift queries and adopts the place name mechanism. Leopard first computes a place name set for each activity and object and then maps the place names to the concrete place based on the specific place topology structures. After that, Leopard decides whether an instance field access is safe with respect to some mappings. For example, in the program `mg`, the statement \( mg\cdot\text{results} = \text{new BMResults(...)} \) is executed at a place named \( pm_1 \), while \( mg \) only points to the object at a place named \( pn_2 \). If \( pm_1 == pn_2 \), the access is safe for any place topology structure. Otherwise, we construct a set

\[
S = \{ mp | \text{pm}_1 \mod mp == \text{pm}_2 \mod mp \}
\]

and check whether the access is safe with respect to the \( mp\)-place (\( mp \in S \)). An empty set denotes that the access is unsafe.

In the study the AS analyses have found more CSAs than the CI analysis has because the AS analysis distinguishes more static objects and activities and computes smaller place name sets for them. In addition, the AS analyses reduce the points-to set size
of \( v \) in the field access expression \( v.f \) and improve the precision of points-to analysis, which will further reduce the complexity of inferring place equivalence conditions.

7. Related Work

In this section, we discuss some work related to points-to analysis and locality analysis of X10 programs.

Points-to Analysis. In the past several years, points-to analysis has been an active research field. A survey of algorithms and metrics for points-to analysis has been given by Hind [9]. Context-sensitivity and flow-sensitivity are two major dimensions of pointer analysis precision. Context-sensitive and flow-sensitive algorithms (CSFS) [6, 12, 20, 21] are usually precise, but are difficult to scale to large programs. Kahlon [11] has proposed a bootstrapping analysis framework that improves the scalability of the context-sensitive and flow-sensitive algorithm. Yu et al. [24] propose a level-by-level algorithm that improves the scalability of context-sensitive and flow-sensitive analyses [5, 8, 10].

Different from a pointer analysis, a locality analysis is to analyze not only the points-to relations, but also the places the objects (and activities) reside in. Apart from tracking the points-to information, our PCG also tracks the places of activities and objects. In addition, the traditional points-to analyses occasionally take into account the place which is a core concept in the X10 programming model. Different from the traditional context-sensitive analyses, our activity- and heap-sensitive analysis captures the essential constructs of X10 language (e.g., activities and objects). The experiment results show that the activity-sensitive analysis improves the precision of both points-to and locality analyses.

Locality Analysis for X10. Only a few analysis approaches have been proposed for analyzing X10 programs. Agarwal et al. [2] present the algorithm for May-Happen-in-Parallel (MHP) analysis of X10 programs, in which a global place-value numbering is used to analyze X10 place expressions. However, some place expressions still cannot be precisely calculated in a static manner, for example, \( v.homed() \) or \( here \) cannot be globally assigned. Chandra et al. [4] propose a dependent type system to capture fine-grained locality information. The type system only contains two forms of place expressions: \( \text{Place}(i) \) and \( v.home() \), but omits handling the cyclic places in X10 programs. In addition, the type system can calculate the SAs but the CSAs because it can deduce that an object may reside in \( \text{here} \), a fixed place or an unknown place. Another limitation of their type system is that the system handles recursion approximately by unfolding the recursion up to some predefined

<table>
<thead>
<tr>
<th></th>
<th>CI</th>
<th>AS</th>
<th>CI</th>
<th>AS</th>
<th>CI</th>
<th>AS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>k=1</td>
<td>k=2</td>
<td>k=1</td>
<td>k=2</td>
<td>k=1</td>
<td>k=2</td>
</tr>
<tr>
<td>sor</td>
<td>538</td>
<td>1474</td>
<td>3594</td>
<td>223</td>
<td>704</td>
<td>1325</td>
</tr>
<tr>
<td>stream</td>
<td>224</td>
<td>238</td>
<td>317</td>
<td>164</td>
<td>183</td>
<td>274</td>
</tr>
<tr>
<td>spartan</td>
<td>818</td>
<td>2380</td>
<td>3725</td>
<td>576</td>
<td>1460</td>
<td>1768</td>
</tr>
<tr>
<td>series</td>
<td>2146</td>
<td>6142</td>
<td>8573</td>
<td>1264</td>
<td>2873</td>
<td>4426</td>
</tr>
<tr>
<td>crypt</td>
<td>4434</td>
<td>8232</td>
<td>14218</td>
<td>1874</td>
<td>3028</td>
<td>5072</td>
</tr>
<tr>
<td>moldyn</td>
<td>5167</td>
<td>29546</td>
<td>55127</td>
<td>2014</td>
<td>10083</td>
<td>19716</td>
</tr>
<tr>
<td>hifunc</td>
<td>6392</td>
<td>49924</td>
<td>103495</td>
<td>4373</td>
<td>15506</td>
<td>30220</td>
</tr>
<tr>
<td>raytrace</td>
<td>2187</td>
<td>2024</td>
<td>221310</td>
<td>6280</td>
<td>21906</td>
<td>27343</td>
</tr>
<tr>
<td>mc</td>
<td>13767</td>
<td>443470</td>
<td>5403810</td>
<td>3887</td>
<td>111367</td>
<td>1318121</td>
</tr>
<tr>
<td></td>
<td>15651</td>
<td>31454</td>
<td>61067</td>
<td>4273</td>
<td>13132</td>
<td>20315</td>
</tr>
</tbody>
</table>

Figure 9. Precision of Solving PCGs
depth. Leopard overcomes that limitation since it computes a fixed point for an interprocedural PCG. Agarwal et al. [1] present an intraprocedural analysis framework for statically establishing place localities in X10. In the framework, the places and activities are analyzed and the classical thread escape analysis is extended to trace which activities an object can escape. However, how to calculate a place expression containing p.next() or p.prev() is omitted. Compared with the related work, Leopard deduces a place set for each object in a program aware of the topology structures of places, and thus the analysis can be precise and efficient. We have also conducted some experiments to demonstrate the precision and efficiency of Leopard.

8. Conclusion
In this paper we presented a context-insensitive algorithm and a context-sensitive algorithm for computing the locality information of objects and activities in X10 programs. We also developed a tool to support the Leopard method and conducted some experiments to measure the performance and precision of Leopard. The experimental results show that Leopard produces precise locality information for X10 programs. In addition, we use the results of Leopard to check the safe accesses to instance fields.

In the future, we would like to combine the symbolic execution techniques and Leopard analysis to improve the precision of locality analysis. We would also like to study the code optimizations for X10 programs using our locality analysis results.

Acknowledgments
We would like to thank the anonymous reviewers for their valuable and thorough comments. We would like to thank Feng Xie and Cheng Zhang for their discussions on this work. This work is supported by the National Natural Science Foundation of China (Grant No. 91118004, 61100051, 61272102, 60970009) and Shanghai Key Laboratory of Computer Software Testing & Evaluating (Grant No. SSTL2011_02).

References