Static Detection of Place Locality and Elimination of Runtime Checks

Shivali Agarwal, RajKishore Barik, V.Krishna Nandivada, Rudrapatna K. Shyamasundar, and Pradeep Varma

IBM India Research Lab, New Delhi

Abstract. Harnessing parallelism particularly for high performance computing is a demanding topic of research. Limitations and complexities of automatic parallelization have led to programming language notations wherein a user programs parallelism explicitly and partitions a global address space for harnessing parallelism. X10 from IBM uses the notion of places to partition the global address space. The model of computation for such languages involves threads and data distributed over local and remote places. A computation is said to be *place local* if all the threads and data pertaining to it are at the same place. Analysis and optimizations targeting derivations of place-locality have recently gained ground with the advent of partitioned global address space (PGAS) languages like UPC and X10, wherein efficiency of place local accesses is performance critical.

In this paper, we present a novel framework for statically establishing place locality in X10. The analysis framework is based on a static abstraction of activities (threads) incorporating places and an extension to classical escape analysis to track the abstract-activities to which an object can escape. Using this framework, we describe an algorithm that eliminates runtime checks that are inserted by the X10 compiler to enforce place locality of data access. We also identify place locality checks that are guaranteed to fail. Our framework takes advantage of the high level abstraction of X10 distributions to reason about place locality of array accesses in loops as well. The underlying issues, the framework and its power are illustrated through a series of examples.

1 Introduction

As multi-core systems are gaining popularity, there is a definite need for languages and tools that can simplify programming high performance machines to exploit the hardware features to a significant level and achieve higher throughput. X10 [22] is an object-oriented explicitly parallel programming language being designed at IBM under the DARPA HPCS program that enables scalable, high-performance, and high-productivity programming for high-end computer systems.

X10 provides a notion of an activity as an independent execution sequence. An activity runs at a *place*. Multiple activities (zero or more) could be running at any one particular place at any point of time. Notion of activities and places becomes clear through the association of activities to threads of execution and places to processors in the program. Each place has a local memory and runs multiple activities. An object created at place p is considered local to p, and for any activity running at place $p'(\neq p)$ the location of the object is considered remote. X10 restricts accesses to remote memory and a runtime exception is thrown if an activity accesses remote data; note that X10 disallows migration of objects and activities.

```
L0:  async (FirstPlace) {
L1:    final Y y = new Y();
L2:    ateach(AllPlaces) {
L3:      X x = new X();
L4:      ... = x.f;
L5:      final Z z = new Z();
L6:      async (FirstPlace) {
L7:        y.f = z;
L8:        ... = y.f.g; }}}
```

Fig. 1. `X10` compiler inserts `pca`s before `L4, L7` and `L8`.

For any object o, the field *o.location* yields the place at which the object was created. The current `X10` compiler conservatively inserts a place check assertion (`pca`) to do a place check before every object dereference – leading to inefficient code. These checks, if they fail, throw a runtime exception called `BadPlaceException`. A `pca` preceding an object dereference gets translated to the following runtime code:

```
if (o.location != here) throw BadPlaceException
```

Each `pca` not only introduces additional code but also introduces additional control flow nodes. Since every object dereference needs to do a place check, the program is peppered with pc-assertions all around; this has a severe impact on the performance (execution time). The table below presents an experimental confirmation to this effect by presenting the runtime impact of `pca`s (execution time with and without the `pca`s being disabled) on two of the largest NAS [3] parallel benchmarks (note that this benchmark suite contains highly parallel applications).

Benchmark	Exec Time -`pca`s (seconds)	Exec Time +`pca`s (seconds)
CG	135	355
LU	22	60

It can be seen that the overhead of checking these assertions is significant. Besides the runtime impact of these pc-assertions, pervasive presence of *asserts* makes it hard to analyze and optimize programs. Also, certain constructs (for example, *atomic*) in the `X10` language require that no `pca` is violated in the body of the statement. Our analysis can also be used by the compiler to enforce such a language guarantee.

We use the snippet of sample X10 program in Fig. 1 to motivate the problem further; we shall use this program as the running example throughout the paper. The program has been simplified in syntax for readability. Line `L0` creates an *activity* at `FirstPlace` that executes the compound statement `L1-L8`. Line `L1` allocates an object at the current place (`FirstPlace`). Line `L2` creates an activity at each of the places in the set `AllPlaces`; each of these activities execute the compound statement `L3-L8`. At each place, we create a new local object at `L3`. This object is being dereferenced in `L4`. In `L5`, we create another local object, and assign it to a final variable. Another activity is created in `L6`. This activity runs at `FirstPlace` and dereferences objects pointed-to by `y` (in `L7`) and `y.f` (in `L8`). In this example, there are three object dereferences, and the current `X10` compiler introduces `pca`s before each of the object dereferences `L4, L7`, and `L8`. However, it can be seen that variable `y` holds an object at place `FirstPlace` and hence, it's dereference in `L7` that happens at place `FirstPlace` does not need a preceding `pca`. Similarly, variable `x` is local to each activity and holds an object local to the respective places. Again, there is no need of a preceding `pca` before `L4`. However, the one preceding `L8` is needed, in particular, for dereferencing the field `g` of `y.f`.

In this paper, we present a static analysis framework to eliminate unnecessary `pca`s during compilation and/or identify assertions that will always fail. The analysis thus either leads to faster code or identifies illegal accesses or leads to programmer productivity (or both). In other place-based languages like UPC [5] and ParAlfl [10] where remote accesses are legal, such reasoning can be used to specialize accesses (local and remote). We have manually applied our analysis on several benchmarks and show that we can eliminate several `pca`s statically.

Our framework of eliminating `pca`s statically consists of the following steps: `1.` Abstraction of activities: In order to reason about different object dereferences and their places of creation, it is important to be able to reason about all the possible activities and places statically. We define an abstraction for `X10` places and use this to design a system for activity abstraction.
`2.` Extension to the classical escapes-to analysis: For guaranteeing that object dereferences do not violate `pca`, we need to guarantee that the object access happens at the place, where the object is created. This is done by tracking all the object creations and copying. For this purpose, we extend the traditional escape analysis to incorporate the target activity that the object can escape and present our escapes-to analysis by analyzing the heap using extensions to the connection graph [9].
`3.` Algorithm to detect place locality of programs with arrays: Besides objects, array accesses constitute the other source of `pca` insertions. In `X10` a distributed array has its slots distributed across multiple places and each access of an array-slot is restricted to the place where the array slot resides. This is enforced by a compiler generated `pca` before the access of each array element. Reasoning about array accesses involves additional analysis about the values of the index expressions used for access. We use a constraint-based system to analyse the values of the index expressions. Further,we extend the escapes-to analysis to reason about array accesses within `X10` loops.

Our contributions are summarized below.
`i.` **Place Locality for Objects**: We define a notion of place-locality and present a framework to statically prove locality or non-locality of data. This allows us to identify `pca`s that are guaranteed to be either true or false; these identifications have special significance both to `X10` and other PGAS languages.
`ii.` **Place Locality for Arrays**: We present a novel constraint based scheme for reasoning about place locality of `X10` arrays distributed over multiple places as an extension to the algorithm described in `(i)`. We present an optimized scheme for solving our generated constraints with regards to two popular `X10` distributions *UNIQUE* and *CYCLIC*.

Rest of the paper is organized as follows. First, we present a brief overview of the relevant constructs of `X10` in section 2. The activity abstraction is described in section 3 followed by the escapes-to analysis in section 4. Section 5 presents a place locality analysis and its' application to verify the need of `pca`s. In section 6, we describe a few case studies. This is followed by an analysis of distributed arrays and an illustrative example in section 7. Comparison of our work with related work is presented in section 8 followed by conclusions in section 9.

2 A brief overview of X10 language

In this paper, we confine ourselves to simplified `X10` programs that have only simple expressions (similar to the expressions in three-address-codes) and every statement has an associated label with it. Details about standard X10 can be found in the X10 reference manual [22].

Some of the constructs, we use are as follows. `async (p) S` creates a concurrent activity to execute the statmenet `S` at place `p`. Every program can be considered as an async statement which recursively starts off all parallel computation. The place argument is optional, so `async (here) S` can be written more compactly as `async S`.

Any activity can reference only the `final` variables of its surrounding lexical environment. Attempt to reference other variables is a static error.

`finish S` is a structured barrier statement wherein `S` is executed with a surrounding barrier such that all activities created (recursively) inside S have to terminate before the barrier completes.

Parallel computation can be spawned with a `future` besides an `async`. The value of `expr` can be evaluated remotely and received using (`future (p) expr).force()`. A `future` is dif-

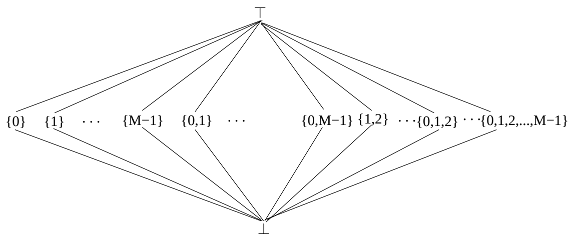

Fig. 2. Place Lattice

ferent from an `async` in terms of returning a value; `force` awaits for the value to become available before returning.

A *distribution* is a function from a set of points (called region) to a set of places. X10 has several pre-defined distributions, such as UNIQUE and CYCLIC. The former associates a set of points in one-to-one correspondence with the set of places; the latter wraps a set of points on a set of places in a cyclic fashion.

`ateach (point p: D) S` is a parallel, distributed loop wherein the statement `S` is evaluated for each point `p`, in the domain of a distribution `D`, at the place given by `D(p)`. An `ateach` can be written in terms of explicit `async` statements in a loop; however, our rules target `ateach`s explicitly for analysis. For this paper, we allow a shorthand for `ateach` statements over UNIQUE distributions by simply letting the user list the set of the distribution's places `X` as `ateach (X) S`.

A distributed array is described by its type and distribution `T[D]`. Construction of the array carries out a `new` operation over this specification e.g., `new T[UNIQUE]`.

3 Activity Representation

Efficient representation of parallel activities(threads) is critical to the complexity of static analysis of fine-grained parallel programs. At the same time, the precision of a static analysis would require enumeration of all instances of runtime activities during compile-time and track all of their interactions. In circumstances where parallel activities are created in loops, it is hard to estimate the upper bound of the loops during compile-time. To take into account precision and complexity of compile-time analysis, we describe an abstract activity representation that efficiently captures both single and multiple runtime activity instances. This work extends the previous work by Barik [4].

We use the following notation to define abstract-activities

Ŀ	: Set of all labels in the program.
$\mathcal{P}$	: Set of all abstract places in the program.
`AA`	: Set of abstract-activities in the program.

We first present an abstract representation of the places to aid in the activity representation. In X10, places are represented by integer place identifiers that range from 0 to $\mathcal{M} - 1$, where $\mathcal{M}$ is the total number of places. We represent the place information of an abstract-activity using a place lattice shown in Fig. 2. The set of abstract places $\mathcal{P}$ is given by

$$\{\perp, \{0\}, \{1\} \cdots \{0, 1\}, \cdots \{1, 2\} \cdots \{0, 1, 2\} \cdots \{0, 1 \ldots \mathcal{M} - 1\}\}$$

It consists of all the possible combinations of $\{0 \cdots \mathcal{M}{-}1\}$, besides $\perp$, and $\top$. The special place $\perp$ indicates an undefined place. This captures the place information of an activity before its' creation. Singleton sets $\{0\}, \{1\} \cdots \{\mathcal{M} - 1\}$ correspond to places $0, 1, \cdots \mathcal{M} - 1$ respectively. An abstract-activity might be created at multiple places. For example, if an activity is created in each iteration of a loop (iterating over a set of places S_p), then we say that there exists a single abstract-activity that represents all of the instances of the activity and this abstract-activity *must* run at an abstract place given by S_p. Non-singleton sets are used to represent the abstract places for such activities and they provide *must* information. Thus, the abstract place $\{0, 1, \cdots \mathcal{M} - 1\}$ is used to represent the place of an abstract-activity created at all the places. The element $\top$ indicates that the activity *may* be created at more than one place. Note that, unlike the *must*-information represented by the other elements of the lattice, $\top$ represents *may*-information; our place check analysis handles may-information conservatively.

An abstract-activity $at \in$ `AA` is represented by a tuple $\langle Label, Places\rangle$, where `Label` $\in$ Ł and $Places \in \mathcal{P}$.The label uniquely identifies an abstract-activity. *Places* denote the abstract-place where the activity runs. Since, we use program labels to identify an activity, multiple activities (at different 'places') might be mapped to the same abstract-activity. We shall extend the notion of abstract-activities to suit our array analysis in section 7.

Consider the program shown in Fig. 1. The async statement in line `L0` is represented in our abstract-activity representation by $\langle L0, \{0\}\rangle$, where 0 is the value of the place `FirstPlace`. However, the async statement in line `L2` is represented by $\langle L2, \{0, 1, ..., \mathcal{M} - 1\}\rangle$. Looking at `L6`, it may be seen that the async statement is invoked at every place, but the activity is created only at `FirstPlace`. We represent the corresponding abstract-activity by $\langle L6, \{0\}\rangle$. [1]

Issues in Computing Abstract-Activities Set (`AA`) for X10

There are two components in the representation of an abstract-activity: label and place.Our simplified program representation helps us to compute unique labels for each statement and the expressions there in.

In X10, the target place for an activity can be specified as the return value of any arbitrary place expression [22]. That is, place expressions can be in terms of arrays, object dereferences and function calls. For analyzing such non-trivial place expressions we can use techniques similar to standard global value numbering mechanisms [17] or flow analysis [12]. Even though the escapes-to connection graph presented in section 4 can be extended to compute the values of the place expressions, we avoid doing that to keep the paper focused. We use a precomputed map

$$pV : Ł \times V \to \mathcal{P},$$

where V is the set of all variables, and $pV(L, v)$ returns the abstract place value of variable v at the statement labeled L. Note that, our intermediate language only has simple expressions and hence, each expression will have an unique label associated with it.

We present an algorithm to compute the abstract-activity set `AA` in section 4 as part of the escapes-to analysis (See the rule to handle the *Async* statement).

4 Escapes To Analysis

Escape analysis in the context of Java like languages consists of determining whether an object (1) may escape a method - the object is not local to the method, and (2) may escape a thread - other threads access the object. Escape analysis results can be applied in various compiler optimizations: (1) determining if an object should be stack-allocatable, (2) determining if an object is thread-local (used for eliminating synchronization operations on an object). For an extensive study of escape analysis the reader is referred to [8, 25].

In this section, we describe *escapes-to* analysis by extending the classical escape analysis that is needed for analyzing X10-like languages. The key difference lies in computing the set of threads (activities in the context of X10) to which an object escapes. To our knowledge, this is the first generalization of the escape analysis that takes into consideration the target activity to which an object escapes.

[1] In general, it may be useful to know if an abstract-activity represents an aggregation of actual activities, or not. And this can be made part of the activity representation. But for the scope of this paper we do not seek such detailed information.

In X10, objects are created by activities at various places. Once created, the object is never migrated to another place during its entire lifetime. Objects created at a local place can be accessed by all the activities associated with that place.

Definition 1. *An object O is said to* escape-to *an activity A, if it is accessed by A but not created in A.*

We represent the escapes-to information by using a map

$$\texttt{nlEscTo} \in \texttt{Objs} \to \mathrm{P}(\texttt{AA}),$$

where `Objs` is the set of abstract-objects (we create an unique abstract-object for each static allocation site). For each object in the program, `nlEscTo` returns a set consisting of abstract-activities that the object might escape to; $\mathrm{P}(S)$ denotes the power set of S.

Prior escape analysis techniques track the 'escapement' of an object based on a lattice consisting of three values: (1) *NoEscape*: the object does not escape an activity; (2) *ArgEscape*: the object escapes a method via its argument; (3) *GlobalEscape*: the object is accessed by other activities and is globally accessible.

Escapes-to Connection Graph (ECG)

We present the escapes-to analysis by extending the connection graph of Choi et al. [8]. We define an abstract relationship between activities and objects through an *Escapes-To Connection Graph* (ECG). Apart from tracking points-to information, ECG also tracks abstract activities in which objects are created and accessed.

An ECG is a directed graph G_e=(N, E), The set of nodes $N = N_O \cup N_v \cup N_a \cup \{O_\top, A_\top\}$ where N_O denotes the set of nodes corresponding to objects created in the program, N_v denotes the set of nodes corresponding to variables in the program, N_a is the set of nodes corresponding to different abstract-activities, $O_\top$ denotes a special node to summarize all the objects that we cannot reason about and $A_\top$ denotes a node corresponding to a special activity used to summarize all the activities that cannot be reasoned about.

The set of edges E comprises of four types of edges:

• *points-to edges*: E_p is the set of points-to edges resulting out of assignments of objects to variables. For $x \in N_v$ and $y \in N_O \cup \{O_\top\}$ $x \xrightarrow{\texttt{p}} y$ denotes a points-to edge from x to y.

• *field edges*: E_f is the set of field edges resulting from the assignment to the fields of different variables. For $x, y \in N_O \cup \{O_\top\}$, $x \xrightarrow{\texttt{f,g}} y$ denotes a field edge from x to y for field g in x.

• *created-by edges*: E_c: For each object O_i, created in an abstract-activity A_i, $(O_i, A_i) \in E_c$. For $x \in N_O \cup \{O_\top\}$ and $y \in N_a$, $x \xrightarrow{\texttt{c}} y$ denotes a created-by edge from x to y.

• *escapes-to edge*: E_e is the set of edges resulting from accessing of an object at a remote activity. For $x \in N_O \cup \{O_\top\}$ and $y \in N_a$, $x \xrightarrow{\texttt{e}} y$ denotes an escapes-to edge from x to y.

For simplification, we have omitted the *deferred edges* used by Choi et al. [8], which are used to invoke the *bypass* function in a lazy-manner. However, we invoke the *bypass* function eagerly.

Intraprocedural Flow-sensitive analysis

The goal of our escapes-to algorithm is to track the abstract-activities that any object is created or accesses in. We present an intra-procedural, flow sensitive, iterative data-flow analysis (standard abstract-interpretation), that maintains and updates G_e at each program point. The algorithm terminates when we reach a fix point.

Initialization Our initial graph consists of nodes N_v, $O_\top$ and $A_\top$. Our intra-procedural analysis makes conservative assumptions about the function arguments (including *this* pointer). For each $v \in V_a$, where $V_a \subseteq N_v$ is the set of all the arguments to the current function:

– add $(v \xrightarrow{\texttt{p}} O_{a_i})$ to E. The special object O_{a_i} represents the object referenced by the i^{th} argument passed to the function. Thus, we conservatively assume that each argument points to an object that is unknown, but not $O_\top$. This helps us reason about the activities created at the native place of these objects more precisely.

– add $(O_{a_i} \xrightarrow{\texttt{c}} A_{a_i})$ to E. For each argument i create a new activity A_{a_i} and use it to represent the activity that created the object referenced by the i^{th} argument.

$$
\begin{array}{lll}
(N,E) & \overset{L:\texttt{async}(p)}{\Longrightarrow} & (N \cup \{\langle L, pV(L,p)\rangle\}, E) \\
(N,E) & \overset{L:a=\texttt{new}\ T}{\Longrightarrow} & (N \cup \{O_L\}, E - \{(a \xrightarrow{\texttt{p}} y) | y \in N \wedge (a,y) \in E\} \cup \{(a \xrightarrow{\texttt{p}} O_L), (O_L \xrightarrow{\texttt{c}} A_c)\}) \\
(N,E) & \overset{L:a=b}{\Longrightarrow} & (N, E - \{(a \xrightarrow{\texttt{p}} y) | y \in N \wedge (a,y) \in E\} \cup \{(a \xrightarrow{\texttt{p}} z) | z \in N \wedge (b \xrightarrow{\texttt{p}} z) \in E\}) \\
(N,E) & \overset{L:a=b.g}{\Longrightarrow} & (N, E - \{(a \xrightarrow{\texttt{p}} y) | y \in N \wedge (a,y) \in E\} \\
 & & \quad \cup \{(a \xrightarrow{\texttt{p}} z), (x \xrightarrow{\texttt{e}} A_c) | x, z \in N \wedge (b \xrightarrow{\texttt{p}} x) \in E \wedge (x \xrightarrow{\texttt{f,g}} z) \in E \wedge (x \xrightarrow{\texttt{c}} A_c) \notin E\}) \\
(N,E) & \overset{L:a.g=b}{\Longrightarrow} & (N, E - \{(x \xrightarrow{\texttt{f,g}} y) | x, y \in N \wedge (a \xrightarrow{\texttt{p}} x) \in E \wedge (x \xrightarrow{\texttt{f,g}} y) \in E\} \\
 & & \quad \cup \{(x \xrightarrow{\texttt{f,g}} z), (x \xrightarrow{\texttt{e}} A_c) | x, z \in N \wedge (a \xrightarrow{\texttt{p}} x) \in E \wedge (b \xrightarrow{\texttt{p}} z) \in E \wedge (x \xrightarrow{\texttt{c}} A_c) \notin E\}) \\
 & & \text{(Strong Update).} \\
(N,E) & \overset{L:a.g=b}{\Longrightarrow} & (N, E \cup \{(x \xrightarrow{\texttt{f,g}} z), (x \xrightarrow{\texttt{e}} A_c) | x, z \in N \wedge (a \xrightarrow{\texttt{p}} x) \in E \wedge (b \xrightarrow{\texttt{p}} z) \in E \wedge (x \xrightarrow{\texttt{c}} A_c) \notin E\}) \\
 & & \text{(Weak Update).} \\
(N,E) & \overset{L:a=f(b)}{\Longrightarrow} & (N, E \cup \{(a \xrightarrow{\texttt{p}} O_\top)\} \cup \{(x \xrightarrow{\texttt{e}} A_\top) | (b \xrightarrow{\texttt{p}} z) \in E \wedge (z \xrightarrow{\texttt{f,*}} x) \in E\})
\end{array}
$$

Fig. 3. Rules for different instructions.

– for any field dereferenced from $O_\top$ and O_{a_i} add $(O_\top \xrightarrow{\texttt{f,*}} O_\top)$, and $(O_{a_i} \xrightarrow{\texttt{f,*}} O_\top)$ to E. The special edge $\xrightarrow{\texttt{f,*}}$ denotes all possible field access.

We distinguish objects O_{a_i} and $O_\top$, and activities A_{a_i} and $A_\top$, as we want to distinguish a field referenced from an argument to a field referenced from some unknown object.

Statements and Operations Fig. 3 presents the effects of the relevant X10 instructions on our analysis. The transformations to the ECG with respect to the labeled construct `L:S` is denoted by

$$(N,E) \overset{L:\texttt{S}}{\Longrightarrow} (N', E')$$

where (N', E') denotes the updated graph as a result of the execution of statement `S` labelled `L`. The updates can include the addition of new nodes, addition of new edges, or updates to existing edges.

In X10, the statements that are of interest to us are: (a) `async (p) S`; (b) `a = new T`; (c) `a=b`; (d) `a.f=b`; (e) `a=b.f`; (f) `a = f(b)`. We now discuss the effect of processing each of these statements.

`Async`: An async statement creates an abstract-activity node in the ECG. $pV(L, v)$ returns the place value of the variable p at the statement L (see section 3). The X10 construct `ateach` can be represented using a basic `async` statement inside a loop. The X10 construct `future` which also creates an activity is handled similar to async.

$a =$ `new` $T()$: We create an object node O_L in the ECG. Since statements are executed within the scope of an activity, we add a created-by edge from O_L to current abstract-activity, given by A_c (O_L is 'local' to A_c). Each statement is in the syntactic scope of exactly one activity. If the current statement is in the syntactic scope of an async labeled L_1, then the current abstract-activity is of the form $\langle L_1, *\rangle$. Note that, for all those cases where we cannot reason about the current abstract-activity, A_c is set to $\perp$. We eliminate any existing points-to edges of the form $(a \xrightarrow{\texttt{p}} y) \in E$, for any $y \in N$. We introduce a new points-to edge $a \xrightarrow{\texttt{p}} O_L$.

$a = b$: We delete all the existing points-to edges starting at a, and for each points-to edge that b points to, we add a points-to edge from a.

$a = b.g$: We process this statement exactly like the copy statement ($a = b$), above. For every dereference of an object, we add a new 'escape-to' edge from the object to the current abstract-activity, if the object is not created in the current activity.

$a.g = b$: Assignments to the fields is a bit more involved because we have to take into consideration possible *weak* and *strong* updates. If there can be multiple points-to edges from the node a, or if a has a single points-to edge to a node x but multiple activities could be updating the field g of x in parallel (See may happen analysis [1]) then $\forall y \in N : (a \xrightarrow{\texttt{p}} y) \in E$, we add a new edge $(y \xrightarrow{\texttt{f,g}} b)$ to the edge set E (weak update). Otherwise, we process the

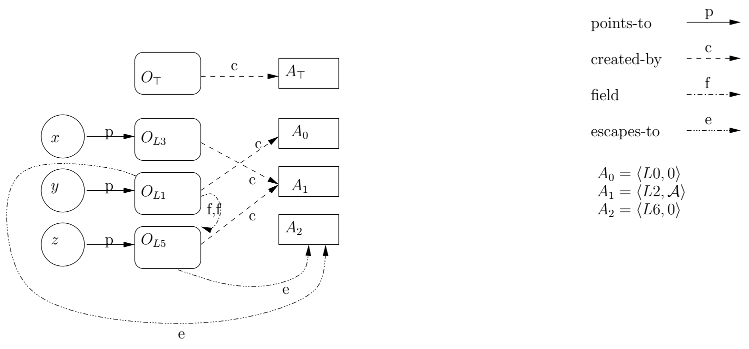

Fig. 4. ECG generated by our algorithm for the example shown in Fig. 1 (as seen after processing the last statement).

statement like the copy statement above - eliminate existing points-to edges and add new edge (strong update).

$a = f(b)$: Since we are doing intra procedural analysis, we can only make conservative assumptions about the arguments (that includes the receiver) and the return values of a function call. We add an escapes-to edge from objects pointed to by the arguments of the function to $A_\top$ and a points-to edge from a to $O_\top$.

Merge Operation: The meet/merge operation of the escapes-to analysis is the union of the two ECGs emanating from two different control flow paths: Merge$((N_1, E_1), (N_2, E_2))$=$((N_1 \cup N_2), (E_1 \cup E_2))$.

While processing all the above assignment statements and the merge operation, we ensure the following invariant: if a variable or a field of an object x has a points-to edge ($x \xrightarrow{\mathtt{p}} O_\top$) or $\exists i$ ($x \xrightarrow{\mathtt{p}} O_{a_i}$), then $\nexists y \in N - \{O_\top, \forall i\ O_{a_i}\}$ such that $(x \xrightarrow{\mathtt{p}} y) \in E$. We ensure this property by checking for the special object $O_\top$ and O_{a_i} while editing the edges of the graph (not shown in the Fig. 3).

Termination: Our iterative data flow analysis follows the standard method of execution and stops only after reaching a fixed point. It can be seen that for any node the maximum number of edges (points-to or escapes-to) is bound by the number of nodes. And in every iteration at least one new edge is added to the set of edges, compared to any previous iteration. Hence, after a finite number of iterations we cannot add any more edges and the algorithm will terminate.

Compute `nlEscTo` map: Given an ECG at program point l, for each object O_i we populate `nlEscTo` (non local escape-to) and `PtsTo` (points-to) maps:

$$\texttt{nlEscTo}(l, O_i) = \{(\langle L, p\rangle | \langle L, p\rangle \in N_a \wedge (O_i \xrightarrow{\mathtt{e}} \langle L, p\rangle) \in E \wedge \neg \exists L' \text{ such that } (O_i \xrightarrow{\mathtt{c}} \langle L', p\rangle)\}$$
$$\texttt{PtsTo}(l, v_i) = \{(x | x \in N_o \wedge (v_i \xrightarrow{\mathtt{p}} x) \in E\}$$

Analysis of the running example Fig. 4 shows the ECG for the example shown in Fig. 1 after processing the last statement.

5 Place Locality

Traditionally, the notion of thread locality is used to denote the access of objects created in the same thread. In this section, we show that in languages like X10, where each activity is associated with a place, activity locality provides insufficient information when we reason about locality of objects with respect to places. Later in the section, we extend the notion of locality to places.

Fig. 5 illustrates the distinctions between the notion of place locality and activity locality.

```
finish async (p) {                         // th0
   S0: final global1 = new G();
   async     (p) {S1: global1.x = new Baz();} // th1
   async     (p) {S2: global1.x = new Baz();} // th2
   async     (p.next()) { ... }            // th3

   async     (p) {global1.x.f ++;}         // th4
}
```

Fig. 5. Place local ⇒ Does not require PC assertion

Traditional thread-local analysis would deduce that the object created in statement S0 is not activity (thread) local. Using this information to decide on the locality of the object being dereferenced would result in insertion of pcas before every dereference of global1 in th1, th2, and th4. Similar argument would follow for the dereference of global1.x in activity th4. However, say it is verified that in async th3, global1 is not accessed (that is, the object accessible by global1 does not escape place p and remains confined to the activities at place-p). Thus, we can conclude that we do not need a pca before statement S1. This is because although global1 is accessed in multiple activities, all of the activities execute at the place of creation (p). An important point to note is that none of these activities *escape* the object under consideration outside place p. That is, global1 holds a place local reference (local to place p) and hence, dereferencing of global1 does not require a preceding pca in th1, th2 and th4 (which execute at place p). Similarly, global.x can hold multiple references but all are local to place p. Hence, we do not require a pca in th4 for accessing global1.x.f. In the following section, we propose an analysis that takes the abstraction of places into consideration while reasoning about the locality of an object.

Given an object O_a, we would like to know if dereference of O_a is done at the place of its allocation or not. Note that, multiple activities can run at the same place and hence, even if an object escapes to another activity running at the same place, we still will be accessing the object locally; thus no place check is required to dereference it. In essence, first we would like to know if object O_a escapes an activity and if so, to which activity. We get this information from the *escapes-to* analysis described in section 4. Further, for each of the dereferencing sites of object O_a, if we can deduce that the dereference happens at the same place as the place of allocation of O_a (say p) then we do not need a preceding pca. For each other activity that O_a can escape to, if we can guarantee that the activity executes only at place p, then we would not need a pca for any of the dereferences of O_a.

Traditional thread-local algorithms declare an object to be thread-non-local, if it gets assigned to a global (read static in Java/X10 context) variable or to a field of an object accessible via any global variable. In comparison, a place-local object can be assigned to a global g and/or be reachable via fields of objects accessible via a global g'. The constraint is that globals g and g' must only be accessible from activities running at only one place. That is, an object may escape to another activity running at the same place and still be place-local, as long as it does not escape to an activity running at a different place. Traditional notions of thread-locality caters to ‘activity-locality’ in X10.

In this section we present (i) the concept of place locality, (ii) an algorithm to deduce place local information, and (iii) a scheme to apply the place local information to eliminate useless pcas, and notify guaranteed pca failures.

Definition 1: An object is *native* to a place p, if it is created at place p.

Definition 2: An object is considered *local to place* p, if it is native to place p and is accessed only at place p.

Definition 3: Dereference of an object o at place p is considered *safe*, if o is local to p.

Remark : *Safe dereferences of an object p do not need a preceding* pca. Later, we present an algorithm to eliminate pcas for safe dereferences of objects.

```
foreach $l_i \in L,$ and $v_i \in V$ do
    $\wp[(l_i, v_i) \leftarrow$ unknown];
foreach $l_i \in L,$ and $v_i \in V$ do
    Say $l_i$ is part of the activity $\langle L_j, p_j \rangle$;
    if $p_j == \bot$ OR $p_j == \top$ then
        continue;
    boolean place-local = true ;
    boolean place-non-local = true;
    foreach $o_k \in$ PtsTo$(l_i, v_i)$ do
        if $\langle L_j, p_j \rangle \in$ nlEscTo$(l_i, o_k)$ then
            place-local=false;
        if $\langle L_j, p_j \rangle \notin$ nlEscTo$(l_i, o_k)$ then
            place-non-local=false;
    if place-local then
        $\wp[(l_i, v_i) \leftarrow$ local];
    if place-non-local then
        $\wp[(l_i, v_i) \leftarrow$ place-non-local];
```

Fig. 6. Algorithm to identify place local references.

```
foreach dereference of of $x \in V$ at program point $l_i$ in activity $\langle L_i, p_i \rangle \in$ AA do
    if $\wp(l_i, x) ==$ local then
        eliminate pca before $l_i$.
    else if $\wp(l_i, x) ==$ nonLocal then
        report that pca before $l_i$ will always fail.
```

Fig. 7. Algorithm to eliminate useless `pca`s and identify guaranteed `pca` failures

Algorithm

Fig. 6 presents an algorithm to identify variables that point to only place-local objects. This algorithm is run after the escape-to analysis is run (i.e., `nlEscTo` map is populated. See section 4).

The algorithm presented in Fig. 6 populates the following map.

$$\wp : (Ł \times V) \rightarrow \langle \texttt{local}, \texttt{nonLocal}, \texttt{unknown} \rangle$$

At a program point l, map $\wp(l, v)$ returns `local` if all the objects pointed-to by variable v are place local, `nonLocal` if all the objects pointed-to by v are non-local, and `unknown` otherwise. The algorithm first initializes the $\wp$ map conservatively to indicate that at all program points the locality of the set of objects pointed-to by all the variables is unknown (Line numbers 1-2). Lines 4-13 identify accesses of variables whose target objects are either guaranteed to be local or non-local. We update the $\wp$ map at Lines 15 and 17.

In Fig. 7, we present a simple algorithm to apply place locality information to eliminate useless place-local-checks and report guaranteed place check failures. For each dereference (field access or method call) in activity a_i, the algorithm eliminates the preceding `pca`, if all the objects pointed-to by the variable are place local. Similarly, it reports cases where `pca` will always fail. This can be used to alert a user of the access error (in X10 context), or to specialize the memory access to remote access (in UPC context).

Analysis of the Running Example

Fig. 8 shows a run of the algorithms presented in this section.

It can be seen that out of the four `pca`s, our algorithm eliminates three of them. The remaining one assertion (before $L8$) must be present and cannot be eliminated.

Improvements to the Algorithm

It can be noted that the above presented algorithm does not take into account the possible control flow between two statements. For example, Fig. 9 presents a case where an object is created at place `p1`, and dereferenced at the same place. After that, the object escapes to an activity executing at place `p2`. Any dereferencing of the object at place `p2` requires a preceding `pca`. However, our algorithm would declare object to be non-place-local and would eliminate neither of the `pca`s preceding `S1` and `S2`. An analysis aware of may-happen-parallelism [19] can recognize such idioms and result in more precise results.

6 Examples

Here, we describe four examples that showcases the strengths and drawbacks of our algorithm.

Consider first the example shown in Fig. 5 of section 5. Our analysis identifies that reference O_{S0} is place local, and hence, its dereference at `th1, th2` and `th4` does not require a preceding `pca`.

Fig. 10(a) shows a snippet of a program for updating a dynamic linked list, as part of master-slave work paradigm. The master goes over the list and invokes the slave server if a boolean flag *done* is not set. The master also adds nodes regularly to the end of the list. The slave, when invoked, sets the flag *done* randomly. Our algorithm attaches a unique abstract object to each of the arguments and thus, is able to eliminate all the `pca`s.

Fig. 10(b) shows a snippet of a program of a postorder traversal of a tree. In this example, Tree is a *value* class. In X10, value classes have the property that after initialization the fields of the objects cannot be modified. After recognizing that a Tree object does not change, similar to the linked list example, we can infer that all the `pca`s in the function `postOrder` are redundant and can be eliminated.

Fig. 10(c) shows an example where an object created at place p0, is assigned to a final variable `z`. A field `g` of `z` is initialized in place p0. Now, two activities are created which can run in parallel. In one activity, running at place p1, the object referenced by `z.g` escapes to place p1. In the other activity, running at place p0, a field of the object referenced by `z.g` is dereferenced. Our algorithm identifies that the object referenced by `z.g` is created at place p0 and escapes to place p1. Hence, we declare it as place-non-local, and add `pca` before the dereference of `z.g.h`. However, the key point to note is that even though the object escapes to another place (p1), none of the fields are updated there. Hence, we do not require a `pca` for dereferencing the sub-fields, at p0. Our algorithm needs to be extended to recognize such idioms.

We have also applied our analysis on the LU NAS parallel benchmark. Our analysis could not remove most of the `pca`s as the program consisted of a large number of tiny helper functions that were invoked at many places. We attributed this to our conservative handling of function calls. Hence, we applied our analysis on another version of the source file after inlining these functions. As guessed, we could eliminate all the `pca`s in this version.

`Objs`=$\{O_{L1}, O_{L3}, O_{L5}\}$ `nlEscTo`=$\{((L8, O_{L5}), \langle L_6, \{0\}\rangle)\}$

Compute Place Local Information
Iter 1: $\wp[(L4, \mathtt{x}) \leftarrow$ `place-local`] Iter 2: $\wp[(L7, \mathtt{y}) \leftarrow$ `place-local`]
Iter 3: $\wp[(L8, \mathtt{y.f}) \leftarrow$ `unknown`]

Eliminate Place Local Assertion
Eliminate the `pca` before $L4$ Eliminate the `pca` before $L7$

Fig. 8. Eliminate `pca`s in the running example, shown in Fig. 1.

```
finish {
   finish async (p1) {global1.x = new G();
                   S1:   ... = global1.x.y}// th0

   async (p2) { S2: ... = global1.x.y } // th1
}
```

Fig. 9. Limitations of of our Analysis: A smarter algorithm can eliminate the pca before S1.

```
void master() {
  // Assert (head != null)
  i = 0;
  while (true) {
    nullable Node tmp = head;
    while (tmp != null) {
      // goto the end of the worklist.
      final node = tmp;
      tmp = getNext(node);
      boolean status=getStatus(node);
      if (status) slave(node); }
    final tail = tmp;
    i = (i + 1)%NumPlaces;
    addNewNode(tail,i); } }
void slave(Node n){
  if (random()%2 == 1)
     async (n) {n.done = true;} }
void getNext(Node n){
future (n) {n.next}.force();}
void getStatus(Node n){
future (n) {!n.done}.force();
void addNewNode(Node n, int i) {
finish async (i) {
     finish async (n) {
       n.next = future (i)
     {new Node()}.force();
     } } }
                (a)
```

```
value class Tree {
   int value;
   Tree left, right;

   public void postOrder() {
      if (left != null) {
        future (left)
          {left.postOrder() }.force();}
      if (right != null) {
        future (right)
          {right.postOrder()}.force();}
      print(value); } }

            (b)
```

```
async (p0) {
    final z = new Z();
    z.g = new G();
    async (p1) {
        final z1 = future (p0) z.g;
        ... = z1.h // escapes
    }
    async (p0) {
        ... = z.g.h.k; } }

          (c)
```

Fig. 10. Three example programs (a)Master slave update program. (b) PostOrder traversal : traverse a distributed binary tree (c) Dummy copies lead to non-elimination of pca

7 PCA Handling for Distributed Arrays

Arrays in X10 can be distributed over multiple places according to some predefined distributions provided as part of the language. X10 guarantees that any array slot located at place p can only be accessed by the activity running at p. Similar to object dereferences, the current X10 compiler inserts a pca before each array element access to maintain the above guarantee. In this section, we present a scheme to eliminate the useless pcas and report pcas that are guaranteed to fail for array accesses.

A distribution can be represented as a map $\mathcal{D} : X \rightarrow \mathcal{P}$, where X is the set of points (integers for one dimensional regions) over which the distribution is defined. For each point i, $\mathcal{D}(i)$ returns the place of the point i. Similarly, we define the inverse distribution function $\mathcal{D}^{-1} : \mathcal{P} \rightarrow \mathrm{P}(X)$ that maps each place to the corresponding set of points. For example, in a scenario with k places $p_0, \ldots p_{k-1}$, the cyclic distribution over n points can be defined as

$$
\begin{array}{lll}
(N,E) & \overset{L:\texttt{ateach(point } x{:}A)\ \mathit{Stmt}}{\Longrightarrow} & (N \cup \{A_c\}, E) \\
(N,E) & \overset{L:a=new\ [A]T}{\Longrightarrow} & (N \cup \{O_L, A_c\}, E \cup \{(a \xrightarrow{\mathtt{p}} O_L), (O_L \xrightarrow{\mathtt{c}} A_c)\}) \\
(N,E) & \overset{L:a=b}{\Longrightarrow} & (N, E - \{(a \xrightarrow{\mathtt{p}} y) | y \in N \wedge (a,y) \in E\} \\
 & & \quad \cup \{(a \xrightarrow{\mathtt{p}} z) | z \in N, \wedge (b \xrightarrow{\mathtt{p}} z) \in E\}) \\
(N,E) & \overset{L:\texttt{ateach(point } x{:}A)\ \{..B[e]..\}}{\Longrightarrow} & (N, E \cup \{(x \xrightarrow{\mathtt{e}} A_c) | \mathrm{C1} \wedge (B \xrightarrow{\mathtt{p}} x) \in E\}) \\
 & & \quad \text{and } \mathrm{C1} = \forall (x, p_i) \in A, \forall (B \xrightarrow{\mathtt{p}} z) \in E,\ [e/x] \in \mathcal{D}_z^{-1}(p_i) \\
(N,E) & \overset{L:\texttt{ateach(point } x{:}A)\ \{..B[e]..\}}{\Longrightarrow} & (N, E \cup \{(x \xrightarrow{\mathtt{e}} A_f) | \mathrm{C2} \wedge (B \xrightarrow{\mathtt{p}} x) \in E\}) \\
 & & \quad \text{and } \mathrm{C2} = \forall (x, p_i) \in A, \forall (B \xrightarrow{\mathtt{p}} z) \in E,\ [e/x] \notin \mathcal{D}_z^{-1}(p_i) \\
(N,E) & \overset{L:a=f(b)}{\Longrightarrow} & N, E \cup \{(a \xrightarrow{\mathtt{p}} O_\top)\}
\end{array}
$$

Fig. 11. Generate ECG for reasoning about `pca`s before array accesses. $A_c = \langle L, A \rangle$.

follows: $\forall i \in \{0..n-1\}$, $\mathcal{D}(i) = i \bmod k$. Similarly the inverse distribution function for cyclic distribution can be defined as $\forall i \in \{0..k{-}1\} \mathcal{D}^{-1}(p_i) = \{x | x = k{\times}j{+}i,\ x \in N,\ 0 \le j \le \frac{(n-1)}{k}\}$.

Fig. 11 presents some additions to the escapes-to analysis presented in section 4, to make it suitable for reasoning about array accesses. These rules are given to process only arrays and non-nested loops; these are to be used on top of the rules for non-array operations given in 3.

`ateach(point` x : A`)` *Stmt*: To model ateach loop bodies iterating over a distribution, we use a new type of abstract-activity of the form: $\langle \mathit{Label}, \mathcal{D} \rangle$, where *Label* is the label of the ateach statement and $\mathcal{D}$ is the distribution with respect to the ateach statement. We use a special abstract-activity $A'_\top$ to denote those activities where a specific distribution cannot be statically determined.

`a = new [A] T`: An array object distributed over A is considered to be created by an activity $\langle L, A \rangle$. We use `ArrObjs` to denote the set of all array objects, and map $\mathcal{D}_x$ returns the underlying distribution of the array object $x \in$ `ArrObjs`. Similarly, map $\mathcal{D}_x^{-1}$ returns the underlying inverse distribution map.

`L: ateach(point` x : A`) {...B[e]...}`: An array access in the body of an `ateach` loop introduces an escape-to edge from object pointed to by `B` to the activity corresponding to the loop provided the distribution of the array matches that of the access pattern in the loop body (given by C1). The index expression e may be data/control dependent on the value of x. Constraint C1 ensures that for each point x located at place p_i in the distribution A, the array slot number e is located at place p_i. To reason about `pca`s that are guaranteed to fail, we use another special activity A_f, where all the `pca`s are guaranteed to fail. For such a scenario (given by C2), we add an escapes-to edge from the objects pointed to by `B` to A_f.

`a = b`: Assigning an array to another results in the removal of all the existing edges from `a` and points-to edges are created from `a` to all the nodes that `b` points-to. Note that, the rules specified for statements of the form `a = b.f` do not require any special treatment in the context of arrays.

`a = f(b)`: It may be noted that any array object that is passed as argument doesn't get its' distribution modified. Thus, we need to add only a points-to edge from a to $O_\top$.

We generate constraints C1 and C2 for each array access in the given program and invoke a constraint solver (for example, [11]) to derive the ECGs at different program points and then invoke the algorithm shown in Fig. 12. Unlike scalar variables which may point to different objects at different program points, the array distribution is an immutable state of the array and thus, it simplifies our algorithm to remove `pca`s (compared to the algorithm presented in section 5).

We now present a optimized scheme for generating and evaluating constraints C1 and C2. The generation of constraints is illustrated through a subset of a priori defined array distributions in X10 : UNIQUE and CYCLIC. Note that the above distributions cover some of the most common idioms of X10 programs including the set of X10 benchmarks presented

```
foreach access of the array variable $v$ at program point $l_i$ in activity $a_i \in \texttt{AA}$ do
    boolean local-access = true;
    foreach $o_i \in \texttt{PtsTo}(l_i, v_i)$ do
        Say $\{cp\} = \texttt{nlEscTo}(l_i, o_i)$ ; Say $a_i = \langle L, D\rangle$ ;
        if $\mathcal{D}_{cp} \neq D$ then
            local-access=false;
    boolean non-local-access = true;
    foreach $o_i \in \texttt{PtsTo}(l_i, v_i)$ do
        if $A_f \notin \texttt{nlEscTo}(l_i, o_i)$ then
            non-local-access = false;
    if local-access then
        eliminate pca.
    else if non-local-access then
        report guaranteed pca failure.
```

Fig. 12. Algorithm to eliminate useless `pca`s and identify guaranteed `pca` failures for array accesses.

at the HPC Challenge in the fall of 2007 (and won the class II challenge). The presented techniques can be extended to other distributions as well. We first show that for the above distributions, the number of elements in the range of x is bounded by the number of places.

Lemma 1. *$\forall(x, p_i) \in \mathcal{D}_A\ [e/x] \in \mathcal{D}_B^{-1}(p_i)\ iff\ \forall p_i \in \texttt{places}(A)\ \exists x \in \texttt{points}(p_i),\ [e/x] \in \mathcal{D}_B^{-1}(p_i)$, where `points`$(p_i)$ returns the set of points mapped onto p_i.*

Proof. Proof omitted for space.

The above lemma makes the constraint solving efficient by reducing the search space. We take advantage of the lemma and the nature of the distributions to present a simplification to constraints C1 and C2, for these distributions.

Let I denote the set of simplified (syntactic) index expressions (of the form $a \times i + b$) in the loop body (linear expressions over i), where i is a loop induction variable. Constraints C1 and C2 for the above two distributions can be reformulated as follows ('mod' is the modulo function) (say $\mathcal{M}$ = number of places in the ateach loop):

	UNIQUE	CYCLIC
$C1$	$\forall i \in \{0..\mathcal{M}-1\} \bigwedge_{e \in I}(\lvert e - i\rvert == 0)$	$\forall i \in \{0..\mathcal{M}-1\} \bigwedge_{e \in I}(\text{mod}(\lvert e - i\rvert, \mathcal{M}) == 0)$
$C2$	$\forall i \in \{0..\mathcal{M}-1\}. \bigwedge_{e \in I}(\lvert e - i\rvert \neq 0)$	$\forall i \in \{0..\mathcal{M}-1\} \bigwedge_{e \in I}(\text{mod}(\lvert e - i\rvert \mathcal{M}) \neq 0)$

Constraint C1 (constraint C2) for UNIQUE states that for each index expression, the absolute value of its difference from i must (must not) be zero. Constraint C1 (C2) for CYCLIC states that for each index expression, the absolute value of its' difference from i should (should not) be a multiple of $\mathcal{M}$.

Illustrative Example: Consider the scenario wherein one set of workers is directed to compute a function `f` on a test-space using the ateach loop on one half the set of machines and the second set of workers use the other half to compute the inverse of the result and check whether the function and its inverse compose to the identity function. The program snippet only shows the code to create the distributed array of jobs, and code to access the elements of the array in the two `ateach` loops.

```
final int N = NumPlaces/2 - 1;      int master() {
       // Assume NumPlaces > 1       finish ateach(i : unique([0..N]))
//Places 0..N reflect half the m/c        { job j = jobs[i]; ...f... }
final job[UNIQUE] jobs =             finish ateach(i : unique([N+1..2*N+1]))
     new (i:UNIQUE(AllPlaces))             { job j = jobs[id-N];
       {return new job(initNum(i));}         ...inv_f... } }
```

Our analysis would find that in the first ateach loop constraint C1 is satisfied, and the second loop satisfies C2. Thus, our analysis will eliminate the pcas before the array access in the first loop, and warn about the illegal array access in the second loop.

We have manually applied our analysis on two other NAS parallel benchmarks: RandomAccess, and CG. After inlining different utility functions, we could eliminate all the pcas in both the benchmarks.

8 Related Work

In this section, we place our work in the context of the existing literature.
Abstraction of runtime components: Abstraction of runtime components like objects have long been used to help in static analysis [18]. We have extended the thread abstraction techniques of Barik [4] to reason about the activities of X10 and also presented an abstraction of places, that is critical to our framework.
Locality of Context: The work that relates most closely to our addressed problem is that of Chandra et al. [7], who improve upon the works of Zhu and Hendren [26]. They present a dependent type system that can be used by the programmer to specify place locality information. They further present an inter-procedural type inference algorithm to infer the place locality information and use it to eliminate useless pcas. We have presented an intra-procedural data flow analysis based approach to infer place locality information, without depending on the programmer input. Their unification based approach would lead to conservative results compared to the results we obtain from the escapes-to-graph: our precise representation of places and activities lead to precise reasoning of activities and objects within loops. The following example clarifies the same.

```
ateach (p: A) { final X x = new X(); async (p) {... =x.f;}}
```

While our algorithm can detect that the dereference of x is place local, their unification based algorithm cannot detect so. Further, we have partially integrated may-happen-parallel analysis into our scheme to generate more precise results. Besides elimination of useless pcas our analysis reports guaranteed failures. It would be interesting to combine our algorithm with the techniques of Chandra et al. use for inter-procedural analysis.

Barton et al. [5] present memory affinity analysis, whereby local data accesses are statically identified and that is used to avoid overhead of routing the access through a machine-wide shared variable directory. Our language setting is more general as unlike in UPC multiple activities can share the same place. We have presented a scheme that tracks the heap statically to prove local, non-local properties.

Work involving inference of place information for programs without user specified places is extensive. For example, false-sharing identification tries to partition data so that a place does not have to deal with data not used by it and hence, does not pay for it through bus traffic and cache-coherency costs [13, 15, 23]. These approaches all differ from us since in our framework activities are explicitly programmed with places, which requires its own set of extensive static representations, analysis and optimizations.

There has been significant interest in proving the thread locality of data [14, 24, 25, 9]. All of these approaches limit themselves to identifying if the object can be accessed in any thread besides the thread of creation. In this paper, we have extended the context of thread locality further to the 'place' of creation. An object whose reference is stored in a global variable and is accessed in another thread might still be 'place-local', provided that all the activities in which the object might be accessed are created at the same place.

Points-to and Escape Analysis: There has been a wide spread interest and good amount of research in the area of points-to and escape analysis [2, 8, 25, 21]. They propose different applications to points-to analysis and different solutions there of. However, we are not aware of any work that tracks not only if an object escapes a context, but also the target context.

Our Escapes-to Connection Graph (ECG) is inspired from the connection graphs of Choi et al. [8]. Apart from tracking the points-to information, ECG also tracks abstract activities in which objects are created and accessed.

Exceptions and Performance: Safe programming languages like Java introduce a lot of runtime assertions, which may throw a runtime exception, in case the assertion fails. Some well known runtime exceptions are Null-pointer-exception, array-out-of-bounds exception and so on. Due to the nature of these assertions un-optimized code is littered with these exceptions. Researchers have shown that a majority of these can be eliminated statically [6, 16]. Systems like CCured [20] have a notion of a dynamic (runtime verified) pointers which are expensive to use. They present a scheme to verify and thus, translate the dynamic pointers to static (statically, type safe) pointers. In our work, we have shown the elimination of `pca`s introduced in `X10` (an explicitly parallel language) arising due to places.

9 Conclusion and Future Work

In this paper, we have presented a static analysis framework for conservatively computing the notion of place locality and have demonstrated the application of the framework for objects and arrays in the context of X10. Our representation of the abstract activities and places is general enough to allow us to extend our intra-procedural analysis to inter-procedural analysis.

Our framework supports reasoning about locality of activities, which can be useful for optimizing the invocation of the activities. We are working towards refining the analysis via may-happen-parallelism analysis and generalizations of the notion to other features of X10 and experimental validation of the concepts.

Acknowledgment: The results reported are based upon the work supported by the Defense Advanced Research Projects Agency under its agreement no. HR0011-07-9-0002. We thank Vivek Sarkar, Vijay Saraswat and the `X10` team for valuable discussions. Thanks go to Raghavan Komondoor and Mangala Gowri for comments on an earlier draft.

References

1. S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *Proceedings of the 12th ACM SIGPLAN symposium on PPoPP*, pages 183–193, 2007.
2. L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
3. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
4. R. Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In *The International Workshop on LCPC*, 2005.
5. C. Barton, C. Cascaval, G. Almasi, Y. Zheng, M. Farreras, S. Chatterjee, and J. N. Amaral. Shared memory programming for large scale machines. In *Proceedings of the Conference on PLDI*, pages 108–117, 2006.

6. R. Bodík, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 conference on PLDI*, pages 321–333, 2000.
7. S. Chandra, V. A. Saraswat, V. Sarkar, and R. Bodík. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN symposium on PPoPP*, pages 11–22. ACM, 2008.
8. J-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *Proceedings of the ACM SIGPLAN conference on OOPSLA*, pages 1–19, 1999.
9. J-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, 2003.
10. P. Hudak and L. Smith. Para-functional programming: A paradigm for programming multiprocessor systems. In *Proc. of the ACM SIGPLAN Symposium on PoPP*, pages 243–254. ACM Press, 1986.
11. T. Huynh, L. Joskowicz, C. Lassez, and J-L. Lassez. Reasoning about linear constraints using parametric queries. In *Proceedings of the 10th conference on FSTTCS*, pages 1–20. Springer-Verlag, 1990.
12. S. Jagannathan, P. Thiemann, S. Weeks, and A.K. Wright. Single and loving it: Must-alias analysis for higher-order languages. In *In Proceedings of the 25th ACM SIGPLAN Symposium on POPL*, January 1998.
13. T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Symposium on PPoPP*, pages 179–188, 1995.
14. R. Jones and A. King. A fast analysis for thread-local garbage collection with dynamic class loading. In *Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 129–138, September 2005.
15. M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. On reducing false sharing while improving locality on shared memory multiprocessors. In *Proceedings of the International Conference on PACT*, pages 203–211, 1999.
16. M. Kawahito, H. Komatsu, and T. Nakatani. Effective null pointer check elimination utilizing hardware trap. *SIGPLAN Notices*, 35(11):139–149, 2000.
17. S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
18. V. K. Nandivada and D. Detlefs. Compile-time concurrent marking write barrier removal. In *Proceedings of the international symposium on CGO*, pages 37–48, 2005.
19. G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the ACM SIGSOFT International symposium on FSE*, pages 24–34, 1998.
20. G. C. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN symposium on POPL*, pages 128–139, New York, NY, USA, 2002. ACM.
21. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.
22. V. Saraswat. Report on the experimental language X10, x10.sourceforge.net/docs/x10-101.pdf, 2006.
23. Y. Shuf, M. Gupta, H. Franke, A. Appel, and J. P. Singh. Creating and preserving locality of java applications at allocation and garbage collection times. In *Proceedings of the conference on OOPSLA*, pages 13–25, 2002.
24. B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the 2nd international symposium on Memory management*, pages 18–24, 2000.
25. J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN conference OOPSLA*, pages 187–206, 1999.
26. Y. Zhu and L. Hendren. Locality analysis for parallel C programs. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):99–114, 1999.