Optimizing Remote Data Transfers in X10

Arun Thangamani
IIT Madras
arunt@cse.iitm.ac.in

V Krishna Nandivada
IIT Madras
nvk@iitm.ac.in

ABSTRACT

X10 is a partitioned global address space (PGAS) programming language that supports the notion of *places*; a place consists of some data and some lightweight tasks called activities. Each activity runs at a place and may invoke a place-change operation (using the `at-construct`) to synchronously perform some computation at another place. These place-change operations need to copy all the required data from the current place to the remote place. However, identifying the required data during each place-change operation is a non-trivial task, especially in the context of irregular applications (like graph applications) that contain large amounts of cross-referencing objects – not all of those objects may be actually required, at the remote place. In this paper, we present a new optimization AT-Opt that minimizes the amount of data serialized and communicated during place-change operations.

AT-Opt uses a novel abstraction called *abstract-place-tree* to capture place-change operations in the program. For each place-change operation, AT-Opt uses a novel inter-procedural analysis to precisely identify the data required at the remote place, in terms of the variables in the current scope. AT-Opt then emits the appropriate code to copy the identified data-items to the remote place. We have implemented AT-Opt in the x10v2.6.0 compiler and tested it over the IMSuite benchmark kernels. Compared to the current X10 compiler, the AT-Opt optimized code achieved a geometric mean speedup of 8.61× and 5.57×, on a two-node (32 cores each) Intel and two-node (16 cores each) AMD system, respectively.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; **Distributed programming languages**; • **Software and its engineering** → **Compilers**;

KEYWORDS

Remote communication; Data serialization; PGAS languages

PACT '18, November 1–4, 2018, Limassol, Cyprus

ACM ISBN 978-1-4503-5986-3/18/11…$15.00
https://doi.org/10.1145/3243176.3243209

```
def setChildSignal() :
    boolean
{//tell children to start
for(i in D){
 var atVal:boolean
  =at(D(i))setCheck(i);
  ...  } ...  }
```

(a) MST: min spanning tree

```
def Start() { ...
at(D(p)){
 for(j=0;j<nSet(p).neig.
     size;j++) {
  var k:Point = ...
  setDist(k,nSet(p).d+1);
}}}
```

(b) BF: breadth first search

Figure 1: Snippets from two IMSuite kernels.

1 INTRODUCTION

With the rapid advancement of many-core systems, it is becoming important to efficiently perform computations in models where the memory may be distributed. X10 [23] is a parallel programming language that uses the PGAS (partitioned global address space) model and provides support for task parallelism. Importantly, X10 supports the distribution of data and computation across shared and distributed memory systems. X10 uses the abstraction of *places*, where each place has some local data (created at that place) and one or more associated activities performing computation over the local data. To access remote data, the activity has to perform a place-change operation (using an `at-construct`). While such expressiveness aids in programmability and data distribution, it may lead to significant communication overheads. We explain the same using a motivating example.

Figure 1a shows a code snippet in X10, of the MST (builds a minimum spanning tree) kernel from IMSuite [13]; the `setChildSignal` function checks if any child can start processing in parallel. A child ready to start processing would have already set its corresponding element in the distributed boolean array `this.setCheck`. The `at` expression checks if a node `i` has set `setCheck(i)`; this value is stored in `atVal`. If any node has set it to true, then the function returns `true` (code not shown); `D` is the distribution of the array.

X10 supports two main types of non-primitive data: distributed arrays (distributed across one or more places at the time of creation) and non-distributed objects (need to be sent to a remote place, if referred at that place). Consequently, for the code shown in Figure 1a, the X10 compiler emits code to serialize and send a message containing a deep copy of the complete `this` object and a pointer to the code containing the expression `setCheck(i)`. At the remote location, the X10 emitted code will deserialize the message, build a copy of the `this` object, and evaluate the expression. In general, this

sending of remote data may incur a significant amount of overhead and since in Figure 1a this remote communication happens inside a loop, the overall cost increases with the number of iterations of the loop.

However, it may be noted that in the code shown, only the `setCheck` field of `this` is getting de-referenced to evaluate the expression; whereas the `X10` compiler copies and transmits the complete `this` object, which has many other fields. Note that the current `X10` compiler handles distributed arrays efficiently, and does not require further optimizations. For example, while copying `this.setCheck` (as part of copying `this`) it only copies the remote-reference of `setCheck`.

Similar to Figure 1a, Figure 1b shows a snippet of the BF (breadth first search) kernel of IMSuite. The shown snippet starts the BFS creation with the root setting the distance for its neighbors. The function `setDist` uses the distribution `this.D` and the distributed array `this.nSet`; code not shown. Like before, at the `at-construct`, the `X10` compiler emits code to transmit the complete `this` object, though the code only requires `this.D` and `this.nSet`.

We have studied many distributed kernels and found that the amount of such copied data can be prohibitively large. For example, for graphs with 256 nodes, the `X10` compiled MST and BF kernels (snippets in Figure 1), led to copying of 76.5 GB and 3.0 GB data, respectively. We have found that a large portion of this data is unused and need not be copied.

In general, it is not trivial to identify the precise data to be copied, especially in the presence of nested `at-constructs` and arbitrary operations involving heap.

In this paper, we present a new optimization technique called `AT-Opt` that analyzes the whole `X10` input program and then identifies the data required at the remote places. The crux of the proposed `AT-Opt` optimization is a novel inter-procedural, summary-based, flow-sensitive analysis that precisely tracks the communication across places in terms of the created objects. Unlike other prior works [1, 2], that reason about the places, we do not depend on global-value numbering, as it can be imprecise. Once `AT-Opt` identifies the required data in terms of the variables in the present scope, it modifies the `X10` input program such that only the required data is copied to the remote places. This leads to a significant reduction in remote communication and (consequently) execution time. For example, for the MST and BF kernels, `AT-Opt` reduces the amount of copied data from 76.5 GB to 10.3 GB, and 3.0 GB to 0.13 GB, respectively. This in turn, at two places, leads to a speedup of 6.6× and 31.1×, respectively, on a two-node (32 cores/node) Intel system.

Barik et al. [6] present a related scheme to reduce the data communicated during place-change operations by doing scalar replacement. Though their scheme is interesting, its impact is limited in irregular benchmark kernels (like the IMSuite kernels) that have many cross-referencing objects. For example, in Figure 1, the scalar-replacement scheme of Barik et al. cannot be applied. This is because (i) They only focus on scalar fields; `setCheck` and `nSet` are distributed arrays, not scalars. (ii) If an `at-construct` calls a method *m* by passing an object as an argument or receiver, then the field accesses of that object in *m*, or in the `at-construct` after the call to *m*, cannot be scalar replaced (Figure 1b). Note: though `setCheck(i)` and `nSet(p).neig.size` are scalars, they cannot be scalar replaced, as the associated arrays are distributed across multiple places and cannot be dereferenced without performing a place-change operation.

Though we discuss `AT-Opt` in the context of `X10`, it can also be applied to other PGAS languages like HJ [14] and Chapel [7]. Similar to `X10`, these languages also support the abstractions/constructs like places/`at-constructs` and while executing a place-change operation, the reachable non-distributed data is required to be copied to the target place.

Contributions:

- We propose a novel analysis to track the flow of objects across places. We are not aware of any other prior work that does so, for minimizing communication overheads.
- We propose a new optimization technique (called `AT-Opt`) that improves the performance of `X10` programs by avoiding the copying of redundant data across places.
- We have implemented `AT-Opt` in the x10v2.6.0 compiler.
- We have evaluated `AT-Opt` over all the IMSuite kernels on two different hardware systems: a two node (32-cores/node) Intel system and a two node (16-cores/node) AMD system. Our evaluations show that compared to the baseline x10v2.6.0 compiler, `AT-Opt` leads to geometric mean speedups of 8.61× and 5.57×, on the Intel system and AMD system, respectively.

2 BACKGROUND

In this section, we briefly discuss three `X10` constructs and some pertinent `X10` concepts. Interested readers may refer to the `X10` specification [23] for details.

`async S` : spawns a new asynchronous task to execute `S`.

`finish S` : acts as join point and waits for the all spawned tasks in `S` to terminate.

`at(P) S` : the place-change operator is a synchronous construct and executes the statement `S` at place `p`. For the ease of presentation, we represent each such `at-construct` using the sequence of three instructions: `at-entry`; `S`; `at-exit`.

In x10v2.6.0, the implementation of an `at-construct` of the form '`at(p) S`' involves sending the serialized data needed to execute `S`, to the remote place `p`, and deserializing the data at `p`. To determine the required data the compiler analyzes `S` and identifies the referenced variables and sends across all the non-distributed data reachable from them.

At runtime, the initial count for places and workers can be set by using the environment variables `X10_NPLACES` and `X10_NTHREADS`, respectively.

3 AT-OPTIMIZATION

In this section, we propose a compile-time technique to optimize X10 programs that use at-constructs. During the compilation of each at-construct, the existing X10 compiler emits code to conservatively serialize all the objects (and variables of primitive type) that may be referred at the remote-place. As discussed in Section 1, this leads to significant overheads. Our proposed approach (called AT-Opt) reduces these overheads. For each at-construct, our approach conservatively identifies the data "required" at the remote-place (in terms of the local variables, and the reachable fields thereof, in the current scope) and emits code to send/receive only that data. For simplicity, in this section, we assume that the input programs do not throw exceptions. In Section 5, we discuss how we handle exceptions.

AT-Opt has two main phases: (1) Analysis phase: to identify the required data, (2) Code generation phase: to emit the optimized code. We now discuss both of these phases.

3.1 AT-Opt Analyzer

For each function, in the input program, the analysis phase of AT-Opt creates two graphs: (1) an *Abstract-place-tree* that captures the place-change operations (from a "source" place to "target" place), and (2) a flow-sensitive *points-to graph* that captures the points-to information of X10 objects (as an extension to the escape-to connection-graph described by Agarwal et al. [1]). We first elaborate on these two graphs.

3.1.1 Abstract-place-tree (APT). For each function in the input X10 program, an APT defines the relationship among the instances of different at-constructs in the function. Each at-construct corresponds to one or more place-change operations at runtime. Say, the set of labels[1] of these constructs is given by L_p (see Figure 2). An APT is defined by the pair (N_p, E_p), where $N_p = \{p_i | L_i \in L_p\}$. Thus, each $p_i \in N_p$ represents an abstract place-change operation. Given two nodes $p_i, p_j \in N_p$, we say that $(p_i, p_j) \in E_p$, if L_j is present in the body of p_i. An interesting aspect of the APT data structure is that it exposes the data-flow between places, as per the X10 semantics: changes done to any data structure (not a global reference or a distributed array) at a place node are not visible to its ancestors and siblings. That is, in the APT, data flows only from the parent to the children.

3.1.2 Points-to Graph (PG). We use the definitions in Figure 2 and a special object node $O_\top$ (that represents all the non-analyzable abstract-objects in the program) to define a points-to graph. A points-to graph is a directed graph (N, E), where $N = N_o \cup N_v \cup \{O_\top\}$. Similar to the discussion of APT,

[1]Without any loss of generality, we assume that the input is a simplified X10 program in three-address-code form [18], each statement has a unique associated label, and variables have unique names.

L	= Set of all the labels in the program.
$L_p \in L$	= Set of labels of the at-constructs.
$L_o \in L$	= Set of labels of the new-statements.
N_o	= Set of all the abstract-objects created in the program.
N_v	= Set of all the variables in the program.
N_p	= Set of all the abstract places in the program.

Figure 2: Definitions of different sets.

POC	$: N_o \rightarrow N_p$	place of creation
pOf	$: L \rightarrow N_p$	place of statement
RS_j	$\subseteq N_o \times Fields$	read before L_j; but defined at ancestor place
$MayWS_j$, $MustWS_j$	$\subseteq N_o \times Fields$	written before L_j, at the current place
AAL_j	$\subseteq N_v \cup (N_o \times Fields)$	Ambiguous access list

Figure 3: Auxiliary data-structures

each abstract object $\in N_o (= \{o_i | L_i \in L_o\})$, may represent one or more instances of objects created at the corresponding labels at runtime. We call an abstract object that represents multiple runtime object instances, as a *summary object.*

The set E comprises of two types of edges:
1. *points-to edges* $\subseteq N_v \times N_o \cup \{O_\top\}$: These edges of the form $v \rightarrow^{\mathrm{p}} o$ are created because of assignment of objects (for example, o) to variables (for example, v).
2. *field edges* $\subseteq N_o \cup \{O_\top\} \times \text{Fields} \times N_o \cup \{O_\top\}$: These edges of the form $o_1 \rightarrow^{\mathrm{f,g}} o_2$ are created because of assignment of objects (for example, o_2) to the fields (for example, g) of objects (for example, o_1).

We call an edge $v \rightarrow^{\mathrm{p}} o$ (or $o_1 \rightarrow^{\mathrm{f,g}} o$) to be a *weak-edge*, if $\exists o' \neq o$, such that $v \rightarrow^{\mathrm{p}} o' \in E$ (or $o_1 \rightarrow^{\mathrm{f,g}} o' \in E$). Otherwise, we call it a *strong-edge*. We use this classification later in this section to mark objects that can be tracked precisely.

In addition to maintaining APT (global) and PG (at each statement), we maintain a few other data-structures; listed in Figure 3. *POC* returns the place of object-creation and pOf returns the place node of each statement. For each function g, we assume that all the statements not contained inside any at-construct are executed at the special place p_g. A pair $\langle o_i, f \rangle \in RS_j$ indicates that the field f of o_i is used (read) at a predecessor of L_j at the current place pOf(j), and the definition reaching this use is present in one of the APT-ancestors of pOf(j). At each label L_j, we maintain two 'write-sets': $MayWS_j$ and $MustWS_j$ to hold the may and must information indicating that the object-field may-/must-be defined at a predecessor of L_j, at pOf(j). Note: $MustWS_j \subseteq MayWS_j$. An entry $k \in AAL_j$ indicates that k (a variable or an obj-field pair) has some weak-edges in the PG_j and k is used at pOf(j).

3.1.3 Intra-procedural flow-sensitive analysis. We now discuss our algorithm to perform a flow-sensitive iterative data-flow analysis to build the APTs (global), the points-to graphs (at each label) and the auxiliary data-structures, in a combined manner. The points-to-graph construction is standard and is shown here for completeness. For the ease of presentation, in this section, we focus just on the intra-procedural component of the analysis. We discuss the inter-procedural extension in Section 4.

Initialization. For each function *bar*, at the first instruction: (1) APT is initialized to a single root node ($p_{bar} \in N_p$); (2) PG is initialized to (N_i, E_i), where $N_i = N_v \cup \{O_\top\}$. In *bar*, for each function parameter $a_j \in N_v$ (including the 0^{th} argument *this*), conservatively, we include an edge $a_j \rightarrow^{\mathrm{p}} O_\top$ in E_i. Also, we add an edge $O_\top \rightarrow^{\mathrm{f},*} O_\top$ to indicate that all field edges from $O_\top$ will points to $O_\top$. The rest of the auxiliary data structures are initialized to empty.

Statements and related operations. Figure 4 shows how we update the APT, PG and auxiliary data structures on processing the different `X10` statements. For each statement $L : Stmt$, each transformation is shown as a transition of the form $X \Rightarrow X'$, where X is a data-structure before processing the statement *Stmt* at label L and X' is the updated data-structure after the processing. Unless otherwise specified, each data structure is copied (cloned) to the next statement.

The statements which are of interest to our analysis are: (i) L: `at-entry(p)` (ii) L: `a=new T()`; (iii) L: `a=b`; (iv) L: `a=b.f`; (v) L: `a.f=b`; (vi) L: `a=x.bar(b)`; and (vii) L: `at-exit`. We now discuss how the processing of each of these statements updates APT, PG and the other auxiliary data-structures.

Entering `at`. L_j:`at-entry(p)`: We create a new place node p_j and add an edge from the current place (given by $\mathrm{pOf}(L_j)$) to the target place (p_j) in APT (Figure 4a). Further, we reset $AAL = MayWS = MustWS = RS = \Phi$ (not shown). Note: `at-entry` is the only instruction that updates the APT.

Exiting `at`. L_j:`at-exit(`L_i`)`: We restore *PG*, *RS*, *MayWS*, *MustWS*, and *AAL* to their values at L_i, which has the corresponding `at-entry` instruction.

Allocation statement. $L_j : a =$ `new T()`: Besides updating the PG (Rule 1, Figure 4b – creates a new object node o_i and updates the points-to edges), we add $(o_i, \mathrm{pOf}(L_j))$ to *POC*.

Copy statement. L_j:$a = b$: Besides updating the PG (Rule 2. Figure 4b), if b has weak-edges, we update *AAL* to include b, since b is used here.

Load statement. L_j:$a = b.g$: Besides updating the PG (Rule 3. Figure 4b), we update *RS*, keeping in mind that no definitions from the current place are added to *RS*. If b or $b.g$ has weak-edges then we add them to *AAL* set appropriately.

Store statement. L_j: $a.g = b$: If a has weak-edges or a points-to a summary node, we perform a weak update; else we perform a strong update (Rule 4, Figure 4b). Besides updating the PG, we add all of the object-field pairs that may be referred to by $a.g$ to *MayWS*. These pairs are also added to *MustWS*, if we are performing a strong update. we update *AAL* to include the variable b, if b has weak-edges. In case of weak-update, we also add a to *AAL*, as a might not have been added so far to the current *AAL*.

Function call (intra-procedural analysis). L_j:$a = x.bar(b)$: Here we make conservative assumptions on the impact of the function call on the arguments, receiver and the return value. We first update the PG (Rule 5. Figure 4b). We use the notation $p \rightarrow^{+} q$ to indicate that q is reachable from p (in the current PG) after traversing one or more edges (points-to, or field). We add all the weak-edges reachable from x and b to

$\mathbf{L}_j$: `at-entry`(p) $\;(N_p, E_p) \Rightarrow (N_p \cup \{p_j\}, E_p \cup \{(\mathrm{pOf}(L_j) \rightarrow p_j)\})$

(a) Impact on the APT

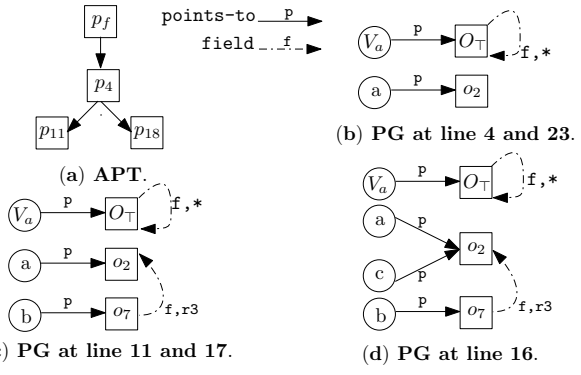
1. $L_j : a =$ `new T()`	
(N, E)	$\Rightarrow (N \cup \{o_j\}, (E - \{a \rightarrow^{\mathrm{p}} y \mid a \rightarrow^{\mathrm{p}} y \in E\}) \cup \{a \rightarrow^{\mathrm{p}} o_j\})$
POC	$\Rightarrow POC \cup \{(o_i, \mathrm{pOf}(L_j))\}$
2. $L_j : a = b$	
(N, E)	$\Rightarrow (N, (E - \{a \rightarrow^{\mathrm{p}} y \mid a \rightarrow^{\mathrm{p}} y \in E\}) \cup \{a \rightarrow^{\mathrm{p}} z \mid b \rightarrow^{\mathrm{p}} z \in E\})$
AAL	$\Rightarrow AAL \cup \{b\}$ // *if b has weak-edges.*
3. $L_j : a = b.g$	
(N, E)	$\Rightarrow (N, (E - \{a \rightarrow^{\mathrm{p}} y \mid a \rightarrow^{\mathrm{p}} y \in E\}) \cup$ $\{a \rightarrow^{\mathrm{p}} z \mid b \rightarrow^{\mathrm{p}} x \in E \wedge x \rightarrow^{\mathrm{f,g}} z \in E\})$
RS	$\Rightarrow RS \cup \{\langle o_i, g\rangle \mid b \rightarrow^{\mathrm{p}} o_i \in E \wedge \langle o_i, g\rangle \notin MustWS\}$
AAL	$\Rightarrow AAL \cup \{b\}$ // *if b has weak-edges*
AAL	$\Rightarrow AAL \cup \{\langle o_i, g\rangle \mid b \rightarrow^{\mathrm{p}} o_i \in E \wedge b.g$ has weak-edges$\}$
4(i). $L_j : a.g = b$ (Strong update)	
(N, E)	$\Rightarrow (N, (E - \{y \rightarrow^{\mathrm{f,g}} z \mid a \rightarrow^{\mathrm{p}} y \in E \wedge y \rightarrow^{\mathrm{f,g}} z \in E\}) \cup$ $\{y \rightarrow^{\mathrm{f,g}} x \mid b \rightarrow^{\mathrm{p}} x \in E \wedge a \rightarrow^{\mathrm{p}} y \in E\})$
MayWS	$\Rightarrow MayWS \cup \{\langle o_i, g\rangle \mid a \rightarrow^{\mathrm{p}} o_i \in E\}$
MustWS	$\Rightarrow MustWS \cup \{\langle o_i, g\rangle \mid a \rightarrow^{\mathrm{p}} o_i \in E\}$
AAL	$\Rightarrow AAL \cup \{b\}$ // *if b has weak-edges.*
4(ii). $L_j : a.g = b$ (Weak update)	
(N, E)	$\Rightarrow (N, E \cup \{y \rightarrow^{\mathrm{f,g}} x \mid b \rightarrow^{\mathrm{p}} x \in E \wedge a \rightarrow^{\mathrm{p}} y \in E\})$
MayWS	$\Rightarrow MayWS \cup \{\langle o_i, g\rangle \mid a \rightarrow^{\mathrm{p}} o_i \in E\}$
AAL	$\Rightarrow AAL \cup \{a\}$ // *a might not be* $\in AAL$*, yet*
AAL	$\Rightarrow AAL \cup \{b\}$ // *if b has weak-edges.*
5. $L_j : a = x.bar(b)$	
(N, E)	$\Rightarrow (N, E \cup \{a \rightarrow^{\mathrm{p}} O_\top\} \cup \{y \rightarrow^{\mathrm{f},*} O_\top \mid x \rightarrow^{+} y \in E\} \cup$ $\{z \rightarrow^{\mathrm{f},*} O_\top \mid b \rightarrow^{+} z \in E\})$
RS	$\Rightarrow RS \cup \{\langle o_i, * \rangle \mid x \rightarrow^{+} o_i \in E \vee b \rightarrow^{+} o_i \in E\}$
AAL	$\Rightarrow AAL \cup \{z \mid z \in \{x, b\} \wedge\ z$ has weak-edges in $E\}$
AAL	$\Rightarrow AAL \cup \{\langle o_i, g\rangle \mid (x \rightarrow^{+} o_i \in E \vee b \rightarrow^{+} o_i \in E) \wedge$ $\langle o_i, g\rangle$ is a weak-edge$\}$

(b) Impact on the points-to-graph PG and the auxiliary data structures (only the updated data structures are shown).

Figure 4: Rules for intra-procedural analysis.

```
1 def f():void{
2   val a:A = new A();
3   a.r1 = ...
4   at(P){
5    a.r2 = ...
6    ... = a.r2;
7    val b:B = new B();
8    b.r1 = ...
9    for(i in D){
10    b.r3 = a;//use of b.r3
          is not shown
11   at(D(i)){
12      ... = b.r1;
13      val c:A = a;
14      c.r1 = ...
15      ... = a.r1;
16    } // end of at(i)
17   } // end of for
18   at(Q){
19    ... = a.r1;
20    ... = a.r2;
21   } // end of at (Q)
22  } // end of at (P)
23 } // end of f
```

Figure 5: Example synthetic X10 code.



(a) **APT**.
(b) **PG at line 4 and 23**.
(c) **PG at line 11 and 17**.
(d) **PG at line 16**.

Figure 6: APT and the points-to graph generated by our analysis for the example shown in Figure 5.

AAL. We conservatively assume that all the fields reachable from x and b are read in the method *bar* and add them to *RS*.

Merge Operation. At each control-flow join point, we merge the *APT*, *PG* and the auxiliary sets. The merging of graphs is done by taking a union of the nodes and edges. For *RS*, *MayWS*, and *AAL*, we merge the sets by performing the set-union operation. We merge the *MustWS* sets by performing the set-intersection operation.

Termination. We follow the standard iterative data-flow analysis approach [18] and stop after reaching a fixed point (from the point of view of *APT* and *PG*).

Post analysis. After computing the *APT*, *PG* and auxiliary maps, we populate two cumulative sets for each node in the *APT*: (i) cumulative read-set *CRS*; and (ii) cumulative ambiguous-access-list *CAAL*. A pair $\langle o_i, f\rangle \in CRS_n$ indicates that the field f of the object o_i is defined at one of the predecessors of n in the APT and that definition may reach a statement in n or one of its successors in the APT. An entry $k \in CAAL_n$ indicates that k is in the ambiguous-access-list of either n or one of its APT successors.

Computation of *CAAL* and *CRS*: We use X_i to refer to the data-structure X before processing the statement labeled L_i. Note that each node in *APT* can be represented by a pair $\langle i, j\rangle$,

RS_{16}	$\{\langle o_7, r1\rangle\}$	RS_{21}	$\{\langle o_2, r1\rangle, \langle o_2, r2\rangle\}$	POC	$\{(o_2, p_f), (o_7, p_4)\}$
$CRS_{\langle 1,23\rangle}$	$\{\langle o_2, r1\rangle\}$	$CRS_{\langle 4,22\rangle}$	$\{\langle o_2, r1\rangle, \langle o_2, r2\rangle, \langle o_7, r1\rangle\}$		
$MayWS_4$	$\{\langle o_2, r1\rangle\}$	$MayWS_{11}$	$\{\langle o_2, r2\rangle, \langle o_7, r1\rangle, \langle o_7, r3\rangle\}$		
		$MayWS_{18}$	$\{\langle o_2, r2\rangle, \langle o_7, r1\rangle, \langle o_7, r3\rangle\}$		
$MustWS_4$	$\{\langle o_2, r1\rangle\}$	$MustWS_{11}$	$\{\langle o_2, r2\rangle, \langle o_7, r1\rangle, \langle o_7, r3\rangle\}$		
		$MustWS_{18}$	$\{\langle o_2, r2\rangle, \langle o_7, r1\rangle\}$		

Figure 7: Auxiliary data structures at different program points and nodes of APT.

where, L_i and L_j are the labels of the first and the last instruction, respectively, of the node. We traverse the APT in post-order and for any APT node $n = \langle x, y\rangle$ set: (1) $CRS_n = RS_y \cup_{\langle p,q\rangle \in aptChild(n)} (RS_q \cup (CRS_{\langle p,q\rangle} - MustWS_q))$, and (2) $CAAL_n = AAL_y \cup_{\langle p,q\rangle \in aptChild(n)} CAAL_{\langle p,q\rangle}$.

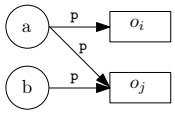
Example: Consider the example code shown in the Figure 5. Here, $L_p = \{p_4, p_{11}, p_{18}\}$ and $L_o = \{o_2, o_7\}$. Figure 6 shows the generated APT and PGs. For brevity, we only show the contents of the *PG* just before the `at-constructs`. Note that the nodes in the APT can also be represented as a pair of indices. For example p_f can be represented as $\langle 1, 23\rangle$, and p_4 as $\langle 4, 22\rangle$. Figure 7 shows the contents of *RS*, *MayWS* and *MustWS* sets at different representative program points. *CRS* and *POC* maps are shown as seen after the analysis of the function f. For this example, $AAL = \phi$.

3.2 Optimized Code Generation

We now discuss our code generation scheme that uses the information (*APT*, *PG*, *RS*, *CRS*, *MayWS*, *POC*, and *CAAL*) computed in Section 3.1.3 to generate code which ensures that only required data is copied to the target places during a place-change operation. For the ease of explanation, we show the rules to emit `X10` optimized code, which can be fed to the current `X10` compiler to generate efficient target code.

While compiling an `at-construct`, the current `X10` compiler captures all the free variables (including `this`) that are referenced in the body of the `at-construct`, and emits code to copy all the data reachable from these free variables. For example, in Figure 5 for the `at-construct` at line 11, the whole of object `b` will be copied to the target place. We take advantage of this approach of the `X10` compiler and use a simple scheme to emit optimized code. We first illustrate our scheme using the code in Figure 5. We emit code to copy `b.r1` to a temporary (say `t3`) just before line 11. In the body of the `at-construct`, we create an empty object (say `b1`, of type `B`), set `b1.r1=t3`, and replace every occurrence of `b.r1` in the body of the `at-construct` with `b1.r1`. Note that in this generated code, `b` is not one of the captured free variables and hence the object pointed-to by `b` will not be copied (at the `at-construct`) by the `X10` compiler; `t3` will be copied instead. We now first present a brief discussion on the impact of objects pointed-to by weak-edges and then detail our code generation scheme.

```
1  at(p1){..//o_i and o_j created here.
2   at(p2){
3     a.r1 = 9;
4     ... = b.r1; }}
```



(a) x10 code snippet **(b) PG at Line 2.**

Figure 8: An example to illustrate 'ambiguous' objects.

Ambiguous objects: In a points-to graph, we call an object o_i to be ambiguous, if o_i is reachable from some node say $x \in PG.N$, and the path from x to o_i in PG contains a weak-edge. Unlike a non-ambiguous object, we cannot precisely determine which of the fields of the ambiguous objects are to be copied. For example, Figure 8 shows a small snippet of `X10` code and its points-to graph (before line 2). At run-time line 3 may write to field $\langle o_i, r1\rangle$ or $\langle o_j, r1\rangle$ and hence, the field dereference at 4 may read the value of the field $\langle o_j, r1\rangle$ written at line 3 or in place p_1. Consequently, at compile time we would not know if `b.r1` should be copied from p_1. Because of such inexactness during compilation time (and to be sound), we deal with the ambiguous objects (for example, o_i and o_j, in Figure 8) conservatively and copy the full objects (provided, the objects are being dereferenced).

We now describe how our code-generation pass handles the statements discussed in Section 3.1.3. Of these statements, handling the `at-construct` (described first) is more involved than the rest (described at the end).

Handling `at-construct` L_j:`at-entry(p)`. Say the corresponding APT node is $\langle j, k\rangle$. We first compute the set of all the ambiguous objects that are reachable from the elements of $CAAL_{\langle j,k\rangle}$: $AOS = \{o_i | n \in CAAL_{\langle j,k\rangle} \wedge n \rightarrow^{+w} o_i \in PG_j\}$; here $n \rightarrow^{+w} o_i \Rightarrow o_i$ is reachable from n (in PG_j) ,after traversing one or more weak edges. During a place-change operation, the objects in AOS will be copied fully. This code generation phase for the remaining objects has two parts:

(A) Code emitted immediately before label L_j: Figure 9 emits code to save the field $\langle o_i, g\rangle$ that is used in the successor(s) of pOf(j) (in the APT) if (1) $CRS_{\langle j,k\rangle}$ contains $\langle o_i, g\rangle$ and o_i is non-ambiguous; (2) o_i created at pOf(j) or $MayWS_j$ contains $\langle o_i, g\rangle$ (that is, $\langle o_i, g\rangle$ may be written to at the parent place pOf(j)); and (3) either $\langle o_i, g\rangle$ is a scalar field, or it points to an ambiguous object (Lines 3-7).

As we will see in Figure 10, we may create new substitute objects inside an `at-construct`. And in the body of the `at-construct`, whenever there is a reference to the original object, those references have to be replaced by the substitute objects. To aid in this process, we maintain a map H which takes as input an object o_i and returns the name of the variable pointing to the substitute object. We save the old value of H before processing an `at-construct` (`at-entry` statement, that is) and restore H to the saved value after completing the processing of `at-exit`.

```
1  Input : PG_j, AOS_j, CRS_⟨j,k⟩, MayWS_j, POC.
2  begin
3      foreach ⟨o_i, g⟩ ∈ CRS_⟨j,k⟩ ∧ o_i ∉ AOS_j do
4          if POC(o_i) == pOf(j) ∨ ⟨o_i, g⟩ ∈ MayWS_j then
5              bool flag1 = ∃ (o_i →^{f,g} o_l) ∈ PG_j.E;
6              bool flag2 = flag1 ∧ (o_l ∈ AOS_j) ;
7              if ¬flag1 ∨ flag2 then // Emit code to copy
8                  if H.contains(o_i) then // o_i is being
                     referred to by a new temp name
9                      x=H.get(o_i);
10                 else
11                     Set x to the name of one of the variables
                         pointing-to o_i in PG_j.E;
                       // Three-address-code input.
                          Hence, each object is pointed
                          to by at-least one variable.
12                 String T1 = new TempName();
13                 Emit ("val " || T1 || "=" || x || ".g;");
14                 tMap.put(⟨o_i, g⟩,T1)// Store the mapping
```

Figure 9: Function to emit the required code before the `at-construct` of the APT node $\langle j, k\rangle$.

To emit the correct code, we need to identify a variable `x` that points to the substitute object of o_i (if present), or o_i in PG_j (Lines 8-11). Then we create a temporary (name in `T1`) and emit code to copy `x.g` to the temporary. We remember this mapping of $\langle o_i, g\rangle$ to `T1` in a global map $tMap$.

(B) Code emitted in the body of the `at-construct` at L_j: Figure 10 emits code to create an object for o_i and re-store the value of $\langle o_i, g\rangle$ from temporaries (added in Step A); we further improve it in Section 6. For each pair $\langle o_i, g\rangle$ read in the body of `at-construct` and o_i is not an ambiguous object, we call the function `createObject`(o_i, H, S) to emit code to create a substitute object for o_i (Line 5); say stored in a variable named tx_0. We then check if $\langle o_i, g\rangle$ is a non-scalar field pointing to a non-ambiguous object say o_j. If so we emit code (Line 7) to create a substitute object for o_j (say, stored in a variable named tx_1) and initialize $tx_0.g$ to tx_1 (Line 8). Otherwise (either $\langle o_i, g\rangle$ is a scalar, or o_j is ambiguous), we lookup (in $tMap$) the temporary (say, named tx_3) in which the value of $\langle o_i, g\rangle$ was stored before the `at-construct` and initialize $tx_0.g$ to tx_3 (Line 9). For each pair $\langle o_i, g\rangle$ written in the body of `at-construct` and o_i is not ambiguous, call the function `createObject`(o_i, H, S) to check and emit code to create a substitute object for o_i (Lines 10- 11).

Handling statements other than the `at-construct`. For statements `a=b`, `a=b.f`, `a.f=b` and `a=x.f(b)`, we check if the objects pointed to by the variables and fields have substitute objects created in the current scope (and are present in the H map), and if so we replace the variable names with

```
1  Input : PG_j, AOS, RS_k, MayWS_k
2  begin
3      Set S = φ;
4      foreach ⟨o_i, g⟩ ∈ RS_k ∧ o_i ∉ AOS do
5          createObject(o_i, H, S);
6          if ∃ (o_i →^{f,g} o_j) ∈ PG_j.E ∧ o_j ∉ AOS then
7              createObject(o_j, H, S);
8              Emit(H.get(o_i) || "." || g || "=" || H.get(o_j) || ";");
9          else Emit(H.get(o_i) || "." || g || "=" || tMap.get(⟨o_i, g⟩)
               || ";") ;
10     foreach ⟨o_i, g⟩ ∈ MayWS_k ∧ o_i ∉ AOS do
11         createObject(o_i, H, S);
12 Function createObject (o, H, S)
13 begin
14     if ¬S.contains(o) then
15         S = S ∪ o;  String T1 = new TempName();
16         Emit ("val " || T1 || "=new " || typeOf(o) || "();");
17         H.put(o, T1);
```

Figure 10: Emit code to copy data from the temporaries, emitted in Fig. 9. Function `createObject` emits code to create a "substitute" object for o_i.

names of the temporaries created. See Section 6 for a discussion on further optimization for the statement `a=b`.

Code generation for our running example: For the example shown in the Figure 5, our code generation pass takes the computed data-structures (shown in Figures 6 and 7) and generates code as shown in Figure 11.

As it can be seen, unlike the default `X10` compiler that copies the complete object (for example, in Figure 5 the objects pointed-to by `a`, `a` and `b`, and `a` for the `at-constructs` at Lines 4, 11, and 18, respectively) to the destination place, our proposed `AT-Opt` copies only the required data (for example, in Figure 11 see lines 4, 13, and 22), creates the required substitute objects therein (for example, in Figure 11 see lines 6, 15, and 24), and initializes substitute objects with the copied data. Note that at Line 6, we only create a substitute object, but do not (need to) emit additional code to initialize any of its fields. In contrast, the substitute objects created at Lines 15 and 24 have some of their fields initialized explicitly, because those fields are explicitly live from remote place and referenced later in the code (see Section 6 for a further optimization in the generated code).

In contrast to `AT-Opt`, the scalar-replacement technique of Barik et al. [6] will be serializing the complete object pointed-to by `a`, for two reasons: (i) it is copied at Lines 10 and 13. (ii) one of its fields is written to (`a.r2` at Line 5). However, their scheme can scalar-replace `b.r1`, if it is a scalar field. Otherwise, their scheme cannot scalar-replace even `b.r1` and has to serialize the complete object pointed-to by `b`.

```
1  def f():void{
2   val a:A = new A();
3   a.r1 = ...
4   t1 = a.r1;
5   at(P){
6    val a1:A = new A();
7    a1.r2 = ...
8    ... = a1.r2;
9    val b:B = new B();
10   b.r1 = ...
11   for(i in D){
12    b.r3 = a1;
13    t3 = b.r1;
14    at(D(i)){
15     val b1:B = new B();
16     b1.r1 = t3;
17     val a2:A = new A();
18     ... = b1.r1;
19     val c:A = a2;
20     c.r1 = ...
21     ... = a2.r1; }}
22  t2 = a1.r2;
23  at(Q){
24   val a3:A = new A();
25   a3.r1 = t1;
26   a3.r2 = t2;
27   ... = a3.r1;
28   ... = a3.r2; }}}
```

Figure 11: Optimized code for Figure 5.

4 INTER-PROCEDURAL AT-OPT

In this section, we present the inter-procedural extension to the intra-procedural analysis discussed in Section 3. This is based on an extension to the standard summary-based flow-sensitive analysis [18]. In addition to maintaining the standard summaries for points-to information and iterating till a fixed point, we maintain summaries for *CRS* and *CAAL*. For each function node in the call-graph, we maintain (1) Input summary: gives the summary of the points-to information of the function parameter(s) including the *this* pointer. (2) Output Summary: (A) points-to details of function parameters (as seen at the end of the function) and return value (B) cumulative-read-set: *CRS* as computed for the abstract place corresponding to the function call. (C) Cumulative ambiguous accesses list: *CAAL* as computed for the abstract place corresponding to the function call. As expected, the input and output summaries are conservative and sound.

Our inter-procedural analysis follows a standard top-down approach with additional steps to compute or maintain the specialized summaries under consideration. We now discuss some of the salient points therein.

Representation of Objects: In addition to the label in which the object is allocated, we maintain a (finite) list of labels giving a conservative estimation about the context (call-chain) in which the object is created; this list is referred as the context-list of the object.

Initialization. Unlike the intra-procedural analysis, where the analysis of each function starts with a conservative assumption of its arguments, here the analysis begins with an initial points-to graph representing the summary points-to graph of the arguments.

End of analysis of a function. Once we terminate the analysis of a function *bar*, besides creating a summary for the points-to graph, we set the *CRS* (and *CAAL*) in output

$$
\begin{array}{lll}
L_j : a = x.bar(b) & (N, E) & \Rightarrow (N \cup \{o_{\langle i,[j,C]\rangle} | o_{\langle i,[C]\rangle} \in OS.PG.N \wedge [C] = [ll, C'] \wedge ll \text{ is a label in } bar\}, \\
 & & E \cup \{a \rightarrow^{\mathrm{p}} o_{\langle i,[C]\rangle} | f_{ret} \rightarrow^{\mathrm{p}} o_{\langle i,[C]\rangle} \in OS.PG.E\} \\
 & & \quad \cup \{y \rightarrow^{\mathrm{f,g}} o_{\langle i,[C]\rangle} | this \rightarrow^{+} y \in IS.PG.E \wedge y \rightarrow^{\mathrm{f,g}} o_{\langle i,[C]\rangle} \in OS.PG.E\} \\
 & & \quad \cup \{z \rightarrow^{\mathrm{f,g}} o_{\langle i,[C]\rangle} | farg_b \rightarrow^{+} z \in IS.PG.E \wedge z \rightarrow^{\mathrm{f,g}} o_{\langle i,[C]\rangle} \in OS.PG.E\} \\
 & RS & \Rightarrow RS \cup (CRS_{\langle m,n\rangle} - \text{LocalObjects}_{bar}) \text{ // if the } \langle m, n\rangle \text{ is the APT node for bar.} \\
 & AAL & \Rightarrow AAL \cup \{x\} \text{ // if } x \text{ has weak-edges in } E. \\
 & AAL & \Rightarrow AAL \cup \{b\} \text{ // if } b \text{ has weak-edges in } E. \\
 & AAL & \Rightarrow AAL \cup \{\langle o_i, g\rangle | (x \rightarrow^{+} o_i \in E \vee b \rightarrow^{+} o_i \in E) \wedge \langle o_i, g\rangle \text{ is a weak-edge}\} \\
 & AAL & \Rightarrow AAL \cup (CAAL_{\langle m,n\rangle} - \text{LocalVars}_{bar}) \text{ // if the } \langle m, n\rangle \text{ is the APT node for bar.}
\end{array}
$$

Figure 12: Rules to translate a function-call instruction for inter-procedural analysis.

summary as the *CRS* (and *CAAL*) of the APT node corresponding to the function *bar*; recall that corresponding to each function, we create a special place node and that is the root of the APT for that function.

Processing the statements. All the statements except that of the function call are handled mostly similar to the way they were handled during the proposed intra-procedural analysis (Figure 4). The only difference is that we use the extended representation for the objects. The newly created object is represented as $o_{\langle i,[0]\rangle}$, where [0] represents the context. If this allocation site is present in a function f_1, another function f_2 calls f_1, and this object is made accessible in f_2 (for example, via a return statement in f_1) then the context-list of the object is updated to reflect the call of f_1 in f_2.

We now discuss how we process the function-call statement (shown in Figure 12). The main difference in processing is related to the handling of input-summary (IS) and taking into consideration the impact of the output-summary (OS) on the arguments and return value. This process is followed for each of the functions that *bar* may resolve to statically.

Impact on IS of the function bar. In points-to graph present in the IS of the function *bar*, the formal parameter corresponding to the actual argument *b* is updated to additionally point to the nodes pointed to by *b*.

Impact of OS of the function bar. Say the APT node for the function *bar* is represented by $\langle m, n\rangle$. (1) For each object $o_{\langle i,[C]\rangle}$in the merged *PG*, if it is created in the function *bar*, then we append the label L_j to the context-list *C*. (2) We set *b* to point to whatever the corresponding formal parameter of *bar* is pointing to in the OS. (3) We set *a* to point to whatever the return value is pointing to in the OS. (4) We merge the entries of $CAAL\langle m, n\rangle$ with the current *AAL*, after removing the local variables of *bar*. (5) We update *CRS* by taking a union of current *RS* with $CRS_{\langle m,n\rangle}$, after removing the local object pairs of *bar*.

No exceptions	With exceptions

t1=x.f1; t2=x.f2; …	var F:boolean=false;
	if (x==null)
	{*F*=true; t1=*def*(..); t2=*def*(..); …}
	else { t1=x.f1; t2=x.f2; …}

// create substitute obj
// and initialize fields
x1 = new X()
x1.f1=t1; x1.f2=t2; …
	if (*F*) x1=null;
	else {
	x1 = new X()
	x1.f1=t1; x1.f2=t1; …}

Figure 13: Code generation in presence of exceptions

5 AT-OPT WITH EXCEPTIONS

We now discuss how we handle `X10` code that may throw exceptions. In *(a) Analysis phase:* The object thrown by the `throw` statement at a place p_1 may be caught by the `catch` statement at p_1 or one of its parents in the Abstract-place-tree. Considering the complexities in identifying the precise `catch` statement and its place of execution, we treat the thrown object conservatively and assume that all the fields reachable from that object are read in the `throw` statement. Similarly, the argument of each `catch` block is also treated conservatively. Considering that the exceptions are rarely thrown, our chosen conservative design doesn't reduce our gains much. *(b) Code Generation phase:* Before an `at-construct`, we emit code (of the form, $t = x.f$) that eagerly dereference the object fields to copy their values into temporaries. If the variable x points-to *null*, such a dereference will throw a `NullPointerException`, earlier than the original dereference point (inside the `at-construct`) in the input program. Note: No other exception may be thrown because of our generated code. To preserve the semantics of the generated code, as shown in Figure 13, instead of the simple codes (shown in the left) we emit code shown in the right.

Here, we first check if the variable points-to *null*, and if so, we set a flag *F* to true and initialize the temporaries t1, t2, and so on, to their default initial values. Later inside the `at-construct`, we create a substitute object only if *F* is false.

6 DISCUSSION

We now discuss some interesting underlying points about our proposed optimization scheme.

(a) **Ambiguous object:** Our idea of ambiguous objects helps classify objects that *need* to be copied fully (non-ambiguous) and those which have to be *conservatively* copied fully (ambiguous). We believe that such a classification is novel and can enable future optimizations (involving remote data).

(b) **Array data types**: (i) Each write to an array element is also considered as a read to the array object. (ii) Our `AT-Opt` treats non-distributed arrays and rails as a single object and not as a collection of objects. This can be improved by doing precise array index analysis and only copying the relevant fields. We leave it as a future work. However, in our experience, we have found that programs rarely access non-distributed array elements.

(c) **Transient and GlobalRef fields:** `X10` allows variables and fields to be declared as `transient` – that are not copied to the remote place and are hence not relevant to our study. Similarly, for variables and fields declared as `GlobalRef`, only their memory references get serialized – not much scope to optimize and are hence not optimized by us.

(d) **Code generation for substitute object:** During the code generation phase (Section 3.2), `AT-Opt` emits code to create a new substitute object for different objects and initializes its fields from the temporaries created in the previous phase (Figure 9). We can further optimize this part by avoiding the creation of new substitute objects (altogether) and replacing the corresponding field de-references with the temporaries. Such an optimization can be done only if the object under consideration is not passed to any function call (including as the `this` argument). Note that we emit code to add a dummy constructor for each user-defined type of the input `X10` program to assist in creating the substitute objects.

(e) **Code generation for the statement a=b:** During the code generation phase, if the object o_i pointed-to by b is non-ambiguous, and the code generator has not emitted code to create substitute object for o_i, then it indicates that o_i is not de-referenced/used in the body of the `at-construct`; likely dead-code. Hence, we eliminate the statement altogether.

(f) `AT-Opt` **in the compiler:** `AT-Opt` is a high-level optimization that does not interfere negatively with other high-level optimizations but it requires a pre-pass of expression simplification (to three-address code). `AT-Opt` should be invoked before any other high-level-optimization that may change the structure of `at-constructs`.

(g) **Remote reads/writes instead of** `AT-Opt`. The alternative of using remote references for local reads/writes of non-distributed objects is unsuitable as the writes to 'local' memory of a place will be visible to other tasks running on other places – violates the underlying PGAS semantics.

Name	I/P	#at		serialized-data (GB)		% reduction in	
		Stat	Dyn	`Base`	`AT-Opt`	c-miss(I)	c-miss(A)
BF	256	45	6K	3.00	0.12	91.94	33.48
DST	256	101	15K	7.39	0.50	88.58	31.23
BY	128	69	550K	68.08	0.25	91.14	25.92
DR	256	39	489K	120.07	0.39	95.80	47.11
DS	256	195	542K	140.09	0.26	86.98	35.06
KC	256	121	84K	0.32	0.16	21.97	7.75
DP	256	97	68K	32.18	0.15	95.60	38.08
HS	256	130	400K	1.01	0.22	22.05	5.56
LCR	256	48	197K	0.47	0.09	22.75	8.61
MIS	256	68	18K	8.94	0.13	88.55	42.78
MST	256	254	193K	76.48	10.32	78.95	25.90
VC	256	86	5K	2.27	0.12	86.39	30.68

Figure 14: Characteristics of the IMSuite kernels. Abbreviations: c-miss: cache misses; (I) - Intel; (A) - AMD.

(h) **Code generation for at-expr** (such as the ones shown in Figure 1a) is handled in the same way as the at-statement.

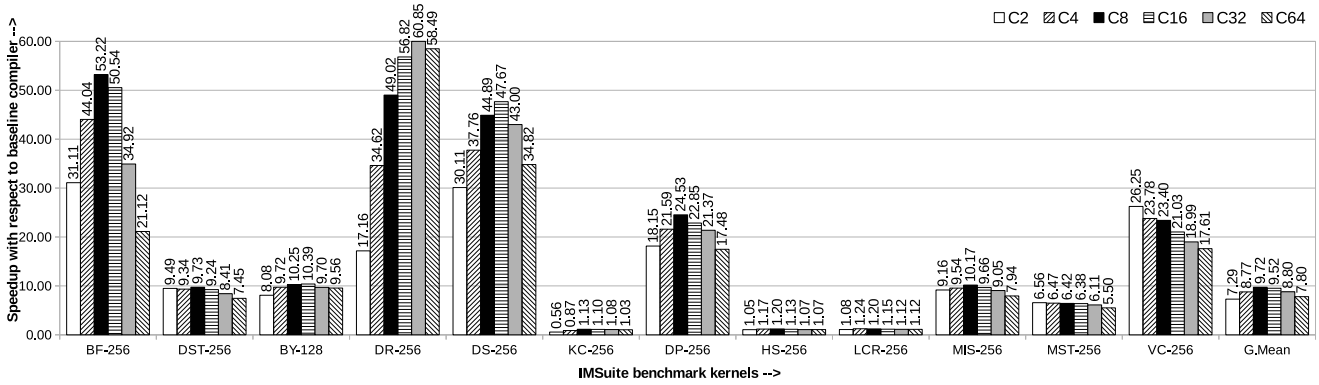
7 IMPLEMENTATION AND EVALUATION

In this section, we evaluate our proposed optimization `AT-Opt` on two different systems - a two node Intel system, where each node has two Intel E5-2670 2.6GHz processors, 16 cores per processor, 64GB RAM per node, and 20MB cache-per core; and a two node AMD system, where each node has an AMD Abu Dhabi 6376 processor, 16 cores per processor, 512GB RAM per node, and 2MB cache per core.

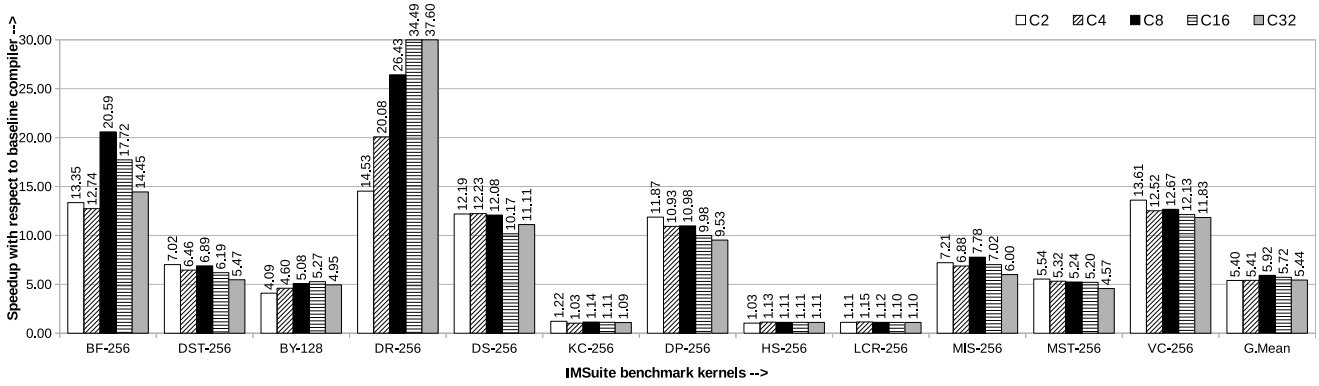
We implemented `AT-Opt` in the x10v2.6.0 compiler x10c (Java backend) and x10c++ (C++ backend). Based on the ideas from the insightful paper of Georges et al. [11], we report the execution times by taking a geometric mean over thirty runs.

We evaluated `AT-Opt` using 12 benchmark kernels from IMSuite [13]: breadth first search (BF - computes the distance of every node from the root and DST - computes the BFS tree), byzantine consensus (BY), routing table creation (DR), dominating set (DS), maximal independent set (MIS), committee creation (KC), leader election (DP - for general network, HS - for bidirectional ring network, and LCR - for unidirectional ring network), spanning tree (MST) and vertex coloring (VC). We also studied many other benchmarks made available in the `X10` distribution, but none of them met our selection requirements: (a) presence of `at-construct` in the program, and (b) de-reference of object (other than distributed arrays) fields at a remote place.

In Figure 14, columns 2 to 4, show the chosen input sizes, and the number of remote-communications (number of `at` statements) during both compile-time and run-time, for the chosen input. For all the benchmarks kernels, the chosen input size was the largest input such that on our 32-core Intel system, when the input program, compiled using default



(a) Speedups on the two node Intel system; totalCores=64. Speedup=(execution time using `Base` / execution time using `AT-Opt`).



(b) Speedups on the two node AMD system; totalCores=32. Speedup=(execution time using `Base` / execution time using `AT-Opt`).

Figure 15: Speedups for varying number of places (#P) and threads (#T). Config $C_i \equiv$ #P=i and #T=totalCores/i;

x10c, is run by setting `X10_NPLACES=2`, it does not take more than an hour to execute and does not run out-of-memory.

We executed the chosen kernels on the specified inputs by varying the number of places (in powers of two) and threads per place such that at any point of time the total number of threads (= #places × #num-threads-per-place) is equal to the number of cores. This is achieved by setting the runtime environment variable `X10_NPLACES` and `X10_NTHREADS` (threads per place) appropriately. The default `X10` runtime divides the places equally among all the provided hardware nodes.

7.1 Evaluation of `AT-Opt`

We report experimental results for two cases: (a) `Base` - the baseline version without any communication optimizations; (b) `AT-Opt` - the optimized version that uses the techniques described in this paper. In Figure 14, the columns 5 and 6 report the amount of data (excluding some common metadata and body of the `at-construct`) serialized during the execution of kernel programs, in the context of `Base` and `AT-Opt`, respectively. As it can be seen, compared to `Base` the `AT-Opt` optimized code leads to a large reduction in the amount of serialized data (2× to 527×). Note that the amount of data serialized is independent of the number of places and is only dependent on the number of `at-constructs` and the data serialized at each of them. We present the evaluation in two parts: (i) on a multi-node (distributed) setup, and (ii) on a single-node (shared memory) setup.

Multi-node setup. Figure 15a and Figure 15b show the speedups achieved by using `AT-Opt`, on the two node Intel system and AMD system, respectively, for varying number of places and threads. It can be seen that with respect to `Base`, the `AT-Opt` optimizer achieved large speedups: geomean of 8.61× on the Intel system and 5.57× on the AMD system.

It can be seen that except for KC, HS and LCR, the speedups across the Intel and AMD systems are consistent. The exact amount of speedup may vary, depending on the input program, the input, and the hardware (including the available cores, memory, cache size and so on).

For kernels KC, HS and LCR, the speedups are not substantial. This is due to the amount of data getting communicated

across places (in `Base`, itself) is very less; consequently the reduction in the communicated data is also less (order of few hundred MBs; see Figure 14). For the rest of the benchmarks, `AT-Opt` leads to significant amount of gains in the execution time (in line with the reduction in the communicated data). In case of BY, DS and MST, the relative amount of time they spend in computation (compared to the communication of serialized data) is much higher (in contrast to the kernels like BF, DR, and VC that show large gains). As a result the speedups in BY, DS and MST look comparatively less.

In general, we find that for very low values of threads per place, the speedups seem to taper off. This because, the IMSuite kernels tend to take more time, in such scenarios.

Single-node (shared environment) setup. Figure 16 shows the geometric mean speedups achieved by the kernels on an Intel (32 cores) or on an AMD (16 cores) system in different `X10` compiler backend (x10c-Java and x10c++-C++). For `Base` x10c (Java) compiler on Intel system: DR, DS and MST ran out of memory with 16 and 32 places and DP ran out of memory with 32 places, but as `AT-Opt` optimizes data across places it ran successfully, within the available memory. We can see that `AT-Opt` leads to significantly high speedups, even on a single node system: geometric mean of 3.06× and 3.44× on the Intel system and AMD system, respectively. Naturally, the speedups on the distributed-memory system are more because of the reduction in inter-node communication. We skip the detailed intra-node results for brevity.

To understand the sources of the obtained speedups, we studied the cache behavior of the generated codes (using the C++ backend) on a single node Intel and AMD systems. In Figure 14, the columns 7 - 8 report, the geomean % reduction (compared to `Base`) in cache-misses (22-95% for the Intel system and 5-47% for the AMD system), across C_2, C_4, C_8, C_{16} and C_{32}. Thus the overall gains = gains from reduced copying and data-transfer + reduced memory-access cost due to reduced cache-misses. It can be seen that reduction in cache-misses is less on the AMD system. We conjecture the reason to be related to the cache size: because `AT-Opt` reduces the amount of memory usage, which in turn leads to less cache pollution and the impact of it is more visible on a system with larger cache (Intel system).

Impact of the inter-procedural analysis. For the benchmarks under consideration, we found that the inter-procedural component was essential for getting the large speedups. This is because, the intra-procedural analysis alone leads to gains only in two benchmarks (BY and DP). Even there, the gains were minimal, as those optimized `at-constructs` were not part of the main computation. Details skipped for brevity.

Impact due to the conservative handling of ambiguous-objects. In the chosen kernels that met our selection requirements (Section 7) the impact due to ambiguous objects was non-existent. This is not unexpected, because in HPC codes,

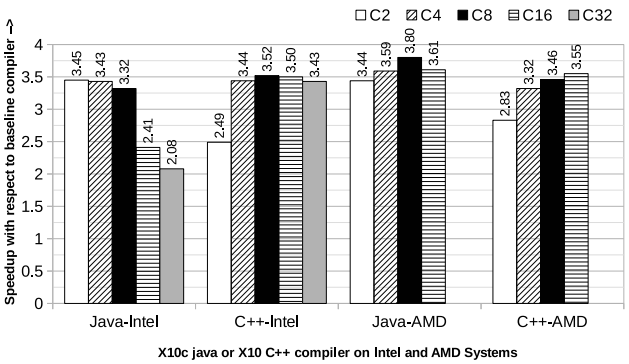


Figure 16: GeoMean speedups for varying number of places (#P) and threads (#T) for single node Intel (#cores = 32) and AMD (#cores = 16) systems using Java and C++ backends. Config: $C_i \equiv$ #P=i and #T=#cores/i.

it is not common to conditionally create objects and dereference them at remote places. However, we still show the procedure to deal with them to make sure that our proposed technique is sound (see Figure 8 and the discussion thereof).

Comparison against the prior work. Barik et al. [6] use a simple scheme of scalar-replacement to reduce the amount of data communicated across places. While that scheme identifies a subset of opportunities identified by our proposed technique and can be effective in some cases, for the IMSuite benchmarks, the scalar-replacement scheme had no impact, whatsoever. This is because in these benchmarks the objects, whose fields accesses are optimized by our technique, are passed as arguments (receiver or parameters) to functions, and consequently scalar-replacement is not performed, as it may require non-trivial modifications to the method signatures. Further, many of those fields were non-scalar in nature (violation of the requirement to perform scalar replacement [6]). This renders a comparison of `AT-Opt` against the scalar-replacement scheme of Barik et al. redundant.

Summary: We have studied the benchmarks and their behavior carefully and found that the actual amount of speedup varies depending on multiple factors: (1) Number of executed `at-constructs`. (2) Amount of data getting serialized during each communication. (3) Amount of other components of remote communication (meta-data such as runtime-type information, data related to the body of the `at-construct`, and so on) (4) Time taken to perform inter-place communication. (5) The nature of the input, runtime/OS related factors and the hardware characteristics. While the factor (2) is the only one that is different between `Base` and `AT-Opt` optimized codes, the impact of factor (2) can be felt on (4) as well. Since `AT-Opt` helps reduce the factors (2) (and consequently factor (4)) it leads to significant performance gains.

8 RELATED WORK

There have been many prior works [3, 5, 6, 9] that aim to reduce the communication overheads across places resulting from redundant data transfers. Barik and Sarkar [5] eliminate memory loads by scalar replacement in single place X10 programs. The scalar-replacement scheme of Barik et al. [6] targets multi-place X10 programs and has similarities with AT-Opt, but with the following differences: (i) They handle only scalar fields; AT-Opt goes beyond that and reduces remote-data transfers involving heap objects. (ii) They do not handle writes to fields; AT-Opt can handle reads and writes of both mutable and immutable-fields. (iii) They cannot handle 'ambiguous' objects (Section 3.2, Figure 8); AT-Opt can. (iv) Importantly, unlike AT-Opt their scheme cannot be applied where the object (whose fields may have to scalar-replaced) is passed to a function as an argument/receiver. Because of these points, the impact of their scalar-replacement scheme was negligible on the IMSuite kernels (see Section 7).

Besides the scalar-replacement technique, Barik et al. [6] also present other compiler optimizations to reduce communication and synchronization overheads: task localization, object splitting for array of objects, replicating arrays across places, distributing loops to specialize local-place accesses and so on. We believe that these techniques can be effectively used along with our techniques in an orthogonal manner, by invoking AT-Opt first and then these optimizations (that may change the structure of loops/at-construct).

There have been prior works [3, 9] that aim to optimize communication of fine-grain data by eliminating redundant communication, use of split-phase communication and coalescing. Similarly, Hiranandani et al. [15, 16] have developed a framework called Fortran D, which reduces communication overheads by applying optimizations like message vectorization, message coalescing, message aggregation and pipelining. These techniques are further extended by Kandemir et al.[17] to optimize the global communication. Our proposed work targets general communication (not just fine-grain communication) and can be invoked before their schemes to take advantage of both the schemes together.

Sanz et al. [22] optimize the communication routines and block and cyclic distribution modules in Chapel [7] by performing aggregation for array assignments. Paudel et al. [19] propose a new coherence protocol in the X10 runtime, to manage mostly-read shared variables. Our proposed technique can be used on top of such runtime optimizations to further improve the performance.

There have been many works on points-to and shape analysis [4, 10, 12, 20, 21, 24]. Chandra et al. [8] use a dependent type system to reason about the locality of X10 objects. We extend the escapes-to-connection graph of Agarwal et al. [1] to reason about the places and objects accessed thereof.

9 CONCLUSION

In this paper, we present a new optimization AT-Opt to reduce the communication overheads by paying close attention to objects getting copied across *places* in X10 programs. We implemented AT-Opt in the x10v2.6.0 compiler and evaluated the performance on two different systems (a two node × 32-core Intel system and a two node × 16-core AMD system). We achieved significant gains in execution time with speedups of 8.61× and 5.57× on the Intel and AMD systems, respectively. Additionally, the experimental results show that the AT-Opt optimized programs scale better than the baseline versions. Though our proposed techniques are discussed in the context of X10, we believe that AT-Opt can be applied to other PGAS languages like Chapel, HJ, and so on.

ACKNOWLEDGMENTS

The authors would like to thank Aman Nougrahiya and Manas Thakur for their valuable comments on an initial draft of this manuscript.

REFERENCES

[1] Shivali Agarwal, RajKishore Barik, V. Krishna Nandivada, Rudrapatna K. Shyamasundar, and Pradeep Varma. 2008. Static Detection of Place Locality and Elimination of Runtime Checks. In *Programming Languages and Systems*, G. Ramalingam (Ed.). Springer Berlin Heidelberg, San Francisco, CA, USA, 53–74.

[2] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. 2007. May-happen-in-parallel Analysis of X10 Programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*. ACM, New York, NY, USA, 183–193. https://doi.org/10.1145/1229428.1229471

[3] Michail Alvanos, Montse Farreras, Ettore Tiotto, José Nelson Amaral, and Xavier Martorell. 2013. Improving Communication in PGAS Environments: Static and Dynamic Coalescing in UPC. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 129–138. https://doi.org/10.1145/2464996.2465006

[4] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph.D. Dissertation. University of Cophenhagen.

[5] Rajkishore Barik and Vivek Sarkar. 2009. Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*. IEEE Computer Society, Washington, DC, USA, 41–52. https://doi.org/10.1109/PACT.2009.32

[6] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlic, and V. Sarkar. 2011. Communication Optimizations for Distributed-Memory X10 Programs. In *2011 IEEE International Parallel Distributed Processing Symposium*. 1101–1113. https://doi.org/10.1109/IPDPS.2011.105

[7] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 3 (Aug. 2007), 291–312. https://doi.org/10.1177/1094342007078442

[8] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. 2008. Type Inference for Locality Analysis of Distributed Data Structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. ACM, New York, NY,

USA, 11–22. https://doi.org/10.1145/1345206.1345211

[9] Wei-Yu Chen, Costin Iancu, and Katherine Yelick. 2005. Communication Optimizations for Fine-Grained UPC Applications. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*. IEEE Computer Society, Washington, DC, USA, 267–278. https://doi.org/10.1109/PACT.2005.13

[10] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*. ACM, New York, NY, USA, 1–19. https://doi.org/10.1145/320384.320386

[11] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

[12] Rakesh Ghiya and Laurie J. Hendren. 1996. Is It a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/237721.237724

[13] Suyash Gupta and V. Krishna Nandivada. 2015. IMSuite: A benchmark suite for simulating distributed algorithms. *J. Parallel and Distrib. Comput.* 75 (2015), 1 – 19. https://doi.org/10.1016/j.jpdc.2014.10.010

[14] Habanero. 2009. Habanero Java. http://habanero.rice.edu/hj. (Dec 2009).

[15] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. 1991. Compiler Optimizations for Fortran D on MIMD Distributed-memory Machines. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 86–100. https://doi.org/10.1145/125826.125886

[16] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. 1992. Compiling Fortran D for MIMD Distributed-memory Machines. *Commun. ACM* 35, 8 (Aug. 1992), 66–80. https://doi.org/10.1145/135226.135230

[17] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. 1999. A Global Communication Optimization Technique Based on Data-flow Analysis and Linear Algebra. *ACM Trans. Program. Lang. Syst.* 21, 6 (Nov. 1999), 1251–1297. https://doi.org/10.1145/330643.330647

[18] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[19] J. Paudel, O. Tardieu, and J. N. Amarai. 2014. Optimizing shared data accesses in distributed-memory X10 systems. In *2014 21st International Conference on High Performance Computing (HiPC)*. 1–10. https://doi.org/10.1109/HiPC.2014.7116889

[20] Noam Rinetzky and Shmuel Sagiv. 2001. Interprocedural Shape Analysis for Recursive Programs. In *Proceedings of the 10th International Conference on Compiler Construction (CC '01)*. Springer-Verlag, London, UK, UK, 133–149. http://dl.acm.org/citation.cfm?id=647477.727768

[21] Alexandru Salcianu and Martin Rinard. 2001. Pointer and Escape Analysis for Multithreaded Programs. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP '01)*. ACM, New York, NY, USA, 12–23. https://doi.org/10.1145/379539.379553

[22] Alberto Sanz, Rafael Asenjo, Juan Lopez, Rafael Larrosa, Angeles Navarro, Vassily Litvinov, Sung-Eun Choi, and Bradford L. Chamberlain. 2012. Global Data Re-allocation via Communication Aggregation in Chapel. In *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '12)*. IEEE Computer Society, Washington, DC, USA, 235–242. https://doi.org/10.1109/SBAC-PAD.2012.18

[23] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. 2016. X10 Language Specification Version 2.6.0. http://x10.sourceforge.net/documentation/languagespec/x10-260.pdf. (2016).

[24] John Whaley and Martin Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*. ACM, New York, NY, USA, 187–206. https://doi.org/10.1145/320384.320400

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact can be evaluated in two ways: using the provided virtual machine or manual download/build/compile/run approach. The artifact contains (i) a virtual machine image (with `Base` and `AT-Opt` compilers build in it, data sets and scripts to evaluate kernels on an x86 virtual machine), and (ii) source files, data sets, and scripts for 'manual' evaluation of `AT-Opt` techniques against `Base` compiler. `AT-Opt` techniques are implemented on the top of x10v2.6.0. The virtual machine based evaluation should be sufficient to establishing the overall claims of the paper. The manual evaluation is required to match the exact evaluation numbers such as the ones shown in Figure 14, 15a and 15b.

For each of the benchmarks under consideration, the artifact evaluation can be used to compute (i) the execution time resulting from using `AT-Opt` and `Base`, (ii) the speedup compared to `Base`, and (iii) the amount of serialized data resulting from using `AT-Opt` and `Base`. Note that the exact speedups may differ from the ones reported in the paper, because of the differences in the actual hardware.

A.2 Artifact check-list (meta-information)

• **Algorithm:** An optimized code generator for place-change operations.
• **Virtual Machine:** OVA file that can be opened using VirtualBox.
• **Programs:** `AT-Opt` compiler, and `Base` compiler.
• **Compilation:** ant, g++ and java.
• **Binary:** Binary for the x86 systems included in the virtual machine.
• **Data set:** All 12 kernels from IMSuite Benchmarks [13].
• **Run-time environment:** Our artifact has been developed and tested on Linux environment. The main software requirements are g++, Apache ant, and JAVA.
• **Hardware:** We recommend (a) a two node Intel system, where each node has two Intel E5-2670 2.6GHz processors, 16 cores per processor, 64GB RAM per node, and 20MB cache-per core; and (b) a two node AMD system, where each node has an AMD Abu Dhabi 6376 processor, 16 cores per processor, 512GB RAM per node, and 2MB cache per core for establishing the exact results presented in our paper.
• **Output:** We provide scripts to generate output files containing (1) the execution times for varying number of places (C2 to C64), (2) the speedups for varying number of places (C2 to C64), and (3) total amount of data serialized.
• **Workflow frameworks used?:** No.
• **Publicly available?:** Yes.

A.3 Description

A.3.1 How delivered. Our virtual machine image, source code, benchmarks, and scripts can be obtained from https://doi.org/10.5281/zenodo.1317758.

A.3.2 Hardware dependencies. none.

A.3.3 Software dependencies.
1) **Using VM**: For evaluation using Virtual Machine required software is already installed in the provided image. A VirtualBox (from Oracle) is required to install the image and do the evaluation.
2) **Using manual evaluation**: The below-required software needed to be installed in the system.

- g++ (preferred version 5.4.0).
- Apache ant software (preferred version 1.9.6).
- Java software.
  - Preferred jvm jdk1.8.0_151 (or) java-8-oracle.
  - After installation, set JAVA_HOME path in ~/.bashrc to point to the installed JAVA bin

A.4 Installation and Evaluation (VM)

A.4.1 Installation. Download the image (ubuntu-16.ova) from https://doi.org/10.5281/zenodo.1317758 and import it in the VirtualBox. Login information:

- User Name: ubuntu-16
- Password: admin123

A.4.2 Evaluation. Steps for evaluation:
• Start the ubuntu-16 virtual machine.
• Open a new terminal and run the below commands.

```
$ cd $ATHOME/script
```

• For establishing the impact of `AT-Opt`, when the programs are run on a 64 core system (similar to the setup used for Figure 15a):

```
$ sudo ./eval_64.sh all localhost localhost speedup
```

The first option tells the script to evaluate all the kernels. One may replace the keyword "all" with individual kernel names, such as bfsBellmanFord, bfsDijkstra, kcommitte, leader_elect_dp, mis, mst, leader_elect_hs, leader_elect_lcr, dijkstraRouting, vertexColoring, byzantine, dominatingSet, to obtain the results for that particular benchmark – much faster. The second and third options tell the script to use 'localhost' as both the nodes. Note: In the virtual machine to simulate our evaluation platform, we evaluate the kernels, by specifying both the nodes as "localhost". On our personal laptop (intel-i7 processor, 4 cores and 16GB RAM), the script took around 90 minutes to complete.
• For establishing the impact of `AT-Opt`, when the programs are run on a 32 core system (similar to the setup used for Figure 15b):

```
$ sudo ./eval_32.sh all localhost localhost speedup
```

On our laptop, the script took around 40 minutes to complete.
• To establish the impact of `AT-Opt` when the programs are

run on a single node 32 core machine:

```
$ sudo ./eval_32_one.sh all localhost speedup
```

On our laptop, the script took around 40 minutes to complete.

• To establish the results shown in Figure 14.

```
$ sudo ./eval_ser_data.sh all
```

On our laptop, the script took around five hours to complete.

For the sake of convenience, we have kept the generated results in `$ATHOME/sample-results/`

A.4.3 Expected results. The scripts execute each kernel for varying number of places (C2 to C64, when the total number of cores = 64, and C2 to C32, when the total number of cores = 32). For each kernel, we note two execution times: execution time when executing code compiled with `Base` and that when compiled with `AT-Opt`.

The output files (execution times + speedup + serialized data) are in the folder `$ATHOME/Results/`.

(1) The following command lists the speedups obtained for each kernel when evaluated on the VM assuming 64 cores:

```
$ cd $ATHOME/Results
$ grep "Speedup" 64_cores/all_speedup.txt | less
```

While the speedups listed above may not exactly match that shown in Section 7, the numbers are still indicative. For example, for BF, DR, DST, DP, MIS, MST and VC the gains are very high. And for KC, HS and LCR the gains are between 1x to 2x.

Similar speedups can be observed for the other files in the `Results` directory.

(2) Reduction in serialized data: The total "AT" calls made during run-time, and the total amount of data serialized by the code compiled using `Base` and `AT-Opt` compilers (columns 4, 5 and 6 of Figure 14) are shown in `all_data_serialized.txt` file. For example, the following commands will output the data of those columns.

```
$ cd $ATHOME/Results
$ egrep "KERNEL|Dynamic|compiler" \
      data_serialized/all_data_serialized.txt | less
```

A.5 Installation and Evaluation (Manual)

A.5.1 Installation. Steps to build the compilers:

• Install the prior discussed software pre-requisites.

• Open a new terminal and run the following commands.

```
$ git clone \
   https://github.com/arunt1204/ae-atOpt-pact2018.git
$ export ATHOME=$(pwd)/ae-atOpt-pact2018/AE/Manual
$ cd $ATHOME/script
```

• To build the x10 `Base` compilers that emit code to calculate (i) speedup and (ii) amount of data serialized by the baseline compiler: (on our laptop, each build took nearly 10 minutes)

```
$ ./build_x10_base.sh # speedup
$ ./build_x10_base_SB.sh # serialized data
```

• To build the new x10 `AT-Opt` compilers that emit code to calculate (i) speedup and (ii) amount of data serialized by `AT-Opt` compiler:

```
$ ./build_x10_AT_Opt.sh # speedup
$ ./build_x10_AT_Opt_SB.sh # serialized data
```

A.5.2 Evaluation. Steps for Execution:

• For establishing the impact of `AT-Opt`, when the programs are run on a two node (say, hostnames given by HOST1 and HOST2) 64 core system (similar to the setup used for Figure 15a):

```
$ sudo ./eval_64.sh all HOST1 HOST2 speedup
```

• For establishing the impact of `AT-Opt`, when the programs are run on a two node (say, hostnames given by HOST1 and HOST2) 32 core machine (similar to the setup used for Figure 15b):

```
$ sudo ./eval_32.sh all HOST1 HOST2 speedup
```

• To establish the impact of `AT-Opt` when the programs are run on a single node 32 core machine:

```
$ sudo ./eval_32_one.sh all HOST1 speedup
```

• To establish the results shown in Figure 14.

```
$ sudo ./eval_ser_data.sh all
```

A.5.3 Expected results. Use the commands discussed in Section A.4.3.

A.6 Experiment customization

We have written the scripts in a simple way for easy understanding and customization. Apart from evaluating the kernels through the scripts with pre-decided parameters, the kernels can also run with different parameters (like varying input size, number of places and other; see the IMSuite website for details). Further, one can compile their own code and see the performance as well.

In the VM: the `Base` compilers can be found in the directory `$ATHOME/x10-base/x10.dist/bin` and and the `AT-Opt` ones in `$ATHOME/x10-atOpt/x10.dist/bin`. The user may use the corresponding x10c and x10c++ compilers.

A.7 Notes

(1) We recommend not to run scripts in parallel.

(2) Sometimes, during executions, the kernels may throw "host not found" or "place exceptions" in both the `Base` and `AT-Opt` compiler. It is a known bug with X10 runtime. Solution: rerun the individual kernel.

(3) Detailed notes are available on the Github link (https://github.com/arunt1204/ae-atOpt-pact2018).