Optimizing Remote Data Transfers in X10

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 threads 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 that involves *serialization, data transfer* and *deserialization.* However, identifying the required data during each place-change operation is a non-trivial task. While the current X10 compiler handles the copying of distributed arrays efficiently, it leads to significant overheads (in terms of performance and memory consumption) 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 required to be sent during a place-change operation. In this paper, we present a new optimization called AT-Opt to minimize these overheads.

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 present scope and place. 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 on two different systems (a 32-core Intel system and a 64-core AMD system). Compared to the current X10 compiler, the AT-Opt optimized code achieved a geometric mean speedup of 3.06× and 3.44×, on the Intel and AMD systems, respectively.

ACM Reference Format:

. 2018. Optimizing Remote Data Transfers in X10. In *Proceedings of ACM Conference (PACT'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

(a) MST: min spanning tree

```
for(pt in bz.D) {
val sTrait=bz.traitorId;
at(bz.D(pt)){
 for(var i:long=0; i<
       sTrait.size; i++)
  if(sTrait(i)==pt(0))
  bz.ifTraitor(pt)=true;
} }
```

(b) BY: byzantine consensus

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 distribution of data and computation across shared and distributed memory systems. While such expressiveness aids in data distribution, it may lead to significant communication overheads. We explain the same using a motivating example.

Figure 1a shows a snippet (in X10) of the MST (builds a minimum spanning tree) kernel from IMSuite [13]; the setChildSignal function checks if any of the children can start processing in parallel. X10 uses the abstraction of *places*, where each place has a unique local memory associated with it, has its own local free variables and one or more associated activities. An activity running at a place may access data items local to that place (that is, created at that place). To access remote data, the activity has to perform a place-change operation (using an at-construct). X10 supports two main types of data: distributed arrays (distributed across places at the time of creation) and non-distributed objects (need to be sent to a place, if referred at that place).

In Figure 1a, a child ready to start processing will set its corresponding element in the distributed boolean array setCheck (a field of this object). The shown loop checks if any node has set its element in setCheck (using the at expression) and if so the function returns true (code not shown); here the distribution D specifies how the array elements are distributed. The variable atVal is set to the value of the expression.

As specified in the X10 manual [23], to translate an at-construct, the X10 compiler emits code to send a serialized message, containing all the referred objects, to the remote place. Consequently, for the code shown in Figure 1a, the serialized message contains 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 the `this` object, and evaluate the expression. In general, this sending of remote data may incur a significant amount of overhead and since 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 the `this` object 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, it only copies the remote-reference of `setCheck` during the copy of `this` object.

Similar to Figure 1a, Figure 1b shows a snippet of the BY (builds byzantine consensus) kernel of IMSuite. The shown snippet does the initialization of the distributed array `ifTraitor` – the value of `ifTraitor(i)` tells, if the node `i` should behave like a traitor. Besides the local variable `sTrait`, like before, the `X10` compiler transmits the complete object pointed-to by `bz`, whereas the code requires only a reference to a single field `ifTraitor` of `this`.

We have studied many distributed kernels and found that the amount of such copied data can be prohibitively large. For example, for the MST and BY kernels (snippets shown in Figure 1), at two places, for a graph with 256 nodes (MST) and with 128 nodes (BY), the original program copied 76.48 GB and 68.12 GB data, respectively. Our studies have indicated that a large portion of this data is redundant and need not be copied.

Note that, in general, it may not be 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, context-insensitive, flow-sensitive analysis that precisely tracks the communication across places in terms of the created objects. Unlike other prior works [1, 2], we do not depend on global-value numbering (can be imprecise) to reason about the places. Once `AT-Opt` identifies the required data (in terms of the variables in the present scope and place), 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 the execution time). For example, for the MST (input 256 nodes) and BY (input 128 nodes) kernels, at two places, `AT-Opt` reduces the amount of copied data from 76.48 GB to 10.33 GB (leading to a speedup of 6.50×) and from 68.12 GB to 0.26 GB (leading to a speedup of 3.01×), respectively, on a 32-core Intel system.

Barik et al. [6] present a scheme to reduce communication by doing (i) scalar replacement of *globals*, (ii) object splitting for array of objects, (iii) replicating arrays across places to avoid repeated copying, and (iv) distributing loops to specialize local-place accesses. Though their scheme is impactful, it does not try to identify the exact heap locations required at the remote place to reduce the data required for communication. We believe that these techniques can be applied on top of our optimization to realize further gains. Even though our proposed techniques are discussed in the context of `X10`, we believe that `AT-Opt` can also be applied to other PGAS languages like Chapel, Cilk, HJ, and so on.

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 benchmark kernels of IMSuite [13] on two different hardware systems (a 32-core Intel system and a 64-core AMD system). Our evaluations shows that compared to the baseline x10v2.6.0 compiler, `AT-Opt` leads to significant speedups (geometric mean speedups of 3.06× and 3.44×, on the Intel and the AMD system, respectively).

2 BACKGROUND

In this section, we briefly discuss three `X10` constructs and some of the 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 `at-construct` using two instructions: `at-entry` and `at-exit`, with the body of the `at-construct` in between.

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

At runtime, each worker is assigned one or more activities to execute and can be seen as a software thread. The initial count for places and workers can be set at runtime using the environment variables `X10_NPLACES` and `X10_NTHREADS`. To execute on distributed environment, `X10` uses `X10_HOSTFILE` and `X10_HOSTLIST` as environment variables to get the names of the host nodes.

3 AT-OPTIMIZATION

In this section, we propose a compile-time technique to optimize `X10` programs that use `at-constructs`. We assume that the programs do not use arrays; see Section 6 for a discussion on the same. 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 (we call it as `AT-Opt`) reduces these overheads. For each `at-construct`, our approach conservatively identifies the data "required" at the remote-place (in terms of the program 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 any exceptions. In Section 5, we discuss how we handle the cases with exceptions.

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

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 `at-constructs` is given by L_p (see Figure 2). An APT is defined by the pair (N_a, E_a), where $N_a = N_p = \{p_i | L_i \in L_p\}$. Thus, each $p_i \in N_a$ represents an abstract place-change operation. Given two nodes $p_i, p_j \in N_a$, we say that $(p_i, p_j) \in E_a$, if L_j is present in the body of p_i. An interesting aspect of the APT data structure is that it makes it easy to understand the flow of data 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 parent (and recursively to the grand-parents, and so on) and siblings. That is, in the APT, data flows only from the parent to the children.

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: Set of definitions.

```
1  def f():void{
2    val a:A = new A();
3    a.r1 = 36;
4    at(P){
5     a.r2 = 27.0;
6     ... = a.r2;
7     val b:B = new B();
8     b.r1 = 24;
9     for(i in D){
10     b.r7 = a;//use of b.r7
           is not shown
11     at(i){
12      ... = b.r1;
13      val c:A = a;
14      c.r1 = 91;
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 3: Example synthetic X10 code.

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, 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 label 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:

- *points-to edges* $\subseteq N_v \times N_o \cup \{O_\top\}$: These edges of the form $v \xrightarrow{\text{p}} o$ are created because of assignment of objects (for example, o) to variables (for example, v).
- *field edges* $\subseteq N_o \cup \{O_\top\} \times \text{Fields} \times N_o \cup \{O_\top\}$: These edges of the form $o_1 \xrightarrow{\text{f,g}} o_2$ are created because of assignment of objects (for example, o_2) to the fields (for example, g) of objects (for example, o_1).

We consider an edge $v \xrightarrow{\text{p}} o$ (or $o_1 \xrightarrow{\text{f,g}} o$) to be a *weak-edge*, if $\exists o' \neq o$, such that $v \xrightarrow{\text{p}} o' \in E$ (or $o_1 \xrightarrow{\text{f,g}} o' \in E$). Otherwise, the edge is considered as a *strong edge.*

[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.

$$L_j : \texttt{at-entry}(p) \quad (N_a, E_a) \quad \Rightarrow (N_a \cup \{p_j\}, E_a \cup \{(\text{pOf}(L_j) \rightarrow p_j)\})$$

(a) Impact on the APT

1.	$L_j : a = \texttt{new T()}$	(N, E)	$\Rightarrow (N \cup \{o_j\}, E - \{a \xrightarrow{p} y \mid a \xrightarrow{p} y \in E\} \cup \{a \xrightarrow{p} o_j\})$
		POC	$\Rightarrow POC \cup \{(o_i, \text{pOf}(L_j))\}$
2.	$L_j : a = b$	(N, E)	$\Rightarrow (N, E - \{a \xrightarrow{p} y \mid a \xrightarrow{p} y \in E\} \cup \{a \xrightarrow{p} z \mid b \xrightarrow{p} z \in E\})$
		AAL	$\Rightarrow AAL \cup \{b\}$ // *if b has weak-edges.*
3.	$L_j : a = b.g$	(N, E)	$\Rightarrow (N, E - \{a \xrightarrow{p} y \mid a \xrightarrow{p} y \in E\} \cup \{a \xrightarrow{p} z \mid b \xrightarrow{p} x \in E \wedge x \xrightarrow{f,g} z \in E\}$
		RS	$\Rightarrow RS \cup \{\langle o_i, g\rangle \mid b \xrightarrow{p} o_i \in E \wedge \langle o_i, g\rangle \notin MustWS\}$
		AAL	$\Rightarrow AAL \cup \{b\}$ // *if b has weak-edges.*
		AAL	$\Rightarrow AAL \cup \{\langle o_i, g\rangle \mid b \xrightarrow{p} o_i \in E\}$ // *if b.g has weak-edges.*
4*a*.	$L_j : a.g = b$	(N, E)	$\Rightarrow (N, E - \{y \xrightarrow{f,g} z \mid a \xrightarrow{p} y \in E \wedge y \xrightarrow{f,g} z \in E\} \cup \{y \xrightarrow{f,g} x \mid b \xrightarrow{p} x \in E \wedge a \xrightarrow{p} y \in E\})$ (Strong update)
		$MayWS$	$\Rightarrow MayWS \cup \{\langle o_i, g\rangle \mid a \xrightarrow{p} o_i \in E\}$
		$MustWS$	$\Rightarrow MustWS \cup \{\langle o_i, g\rangle \mid a \xrightarrow{p} o_i \in E\}$
		AAL	$\Rightarrow AAL \cup \{b\}$ // *if b has weak-edges.*
4*b*.	$L_j : a.g = b$	(N, E)	$\Rightarrow (N, E \cup \{y \xrightarrow{f,g} x \mid b \xrightarrow{p} x \in E \wedge a \xrightarrow{p} y \in E\})$ (Weak update)
		$MayWS$	$\Rightarrow MayWS \cup \{\langle o_i, g\rangle \mid a \xrightarrow{p} o_i \in E\}$
		AAL	$\Rightarrow AAL \cup \{a\}$ // *a might not have been added to AAL so far*
		AAL	$\Rightarrow AAL \cup \{b\}$ // *if b has weak-edges.*
5.	$L_j : a = x.foo(b)$	(N, E)	$\Rightarrow (N, E \cup \{a \xrightarrow{p} O_\top\} \cup \{y \xrightarrow{f,*} O_\top \mid x \rightarrow^+ y \in E\} \cup \{z \xrightarrow{f,*} O_\top \mid b \rightarrow^+ z \in E\})$
		RS	$\Rightarrow RS \cup \{\langle o_i, *\rangle \mid x \rightarrow^+ o_i \in E \vee b \rightarrow^+ o_i \in E\}$
		AAL	$\Rightarrow AAL \cup \{x\}$ // *if x has weak-edges in E.*
		AAL	$\Rightarrow AAL \cup \{b\}$ // *if b 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 explicitly shown).

Figure 4: Rules of instruction for intra-procedural analysis.

Consider the code fragment (synthetic) shown in Figure 3. For this code, $L_p = \{p_4, p_{11}, p_{18}\}$ and $L_o = \{o_2, o_7\}$.

In addition to maintaining APT (global) and PG (at each statement), we maintain few other data-structures: (a) *place-of-creation (POC)*: a global map ($N_o \rightarrow N_p$), indicating the original place where the object is created. (b) *Place-of-execution*: a global map pOf : $L \rightarrow N_p$ that returns the place node of each statement in the function. Note that for each function foo, all the statements not contained inside any `at-construct` are assumed to be executed at the special place p_{foo}. (c) *read-set (RS)*: At each label L_j, $RS(j) \subseteq N_o \times Fields$. A pair $\langle o_i, f\rangle \in RS(j)$ indicates that the field f of the object o_i is defined at one of the predecessors of pOf(j) in the APT and that definition may reach L_j. (d) *write-sets*: At each label L_j, we maintain two sets: $MayWS(j) \subseteq N_o \times Fields$ and $MustWS(j) \subseteq N_o \times Fields$. A pair $\langle o_i, f\rangle \in MayWS(j)$ (or, $MustWS(j)$) indicates that the field f of the object o_i may (or, must) be defined at a predecessor of L_j at pOf(j). Note that $MustWS(j) \subseteq MayWS(j)$. (e) *Ambiguous-access list (AAL)*: At each statement L_j, we maintain a set $AAL \subseteq N_v \cup (N_o \times Fields)$. An entry $k \in AAL(j)$ indicates that k (could be either 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. 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 foo, at the first instruction: (1) APT is initialized to a single root node ($p_{foo} \in N_p$); (2) PG is initialized to (N_i, E_i), where $N_i = N_v \cup V_a \cup \{O_\top\}$, where V_a is the set of function parameters (including the 0^{th} argument *this*). For each $a_j \in V_a$, conservatively, we include an edge $a_j \xrightarrow{p} O_\top$ in E_i. Also, we add an edge $O_\top \xrightarrow{f,*} O_\top$ 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. Each transformation is shown as a transition of the form

$$L : Stmt \quad (N, E) \quad \Rightarrow (N', E')$$

where (N, E) is the APT (or PG) before processing the statement ($Stmt$) at label L and (N', E') is the updated APT (or PG) after the processing. Unless otherwise specified, APT, PG and the auxiliary data structures are 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.foo(b);` and (vii) L `:at-exit`. We now discuss how the processing of each of these statements updates APT, PG and the other 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 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. The rest of the instructions discussed below update the PG (standard) and/or the auxiliary data-structures.

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

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 – standard way), we add (o_i, pOf(L_j)) to POC.

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

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 (Rules 4a, 4b, 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 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:$a = x.foo(b)$: Here we make conservative assumptions on the impact of the function call on the arguments, receiver and the return

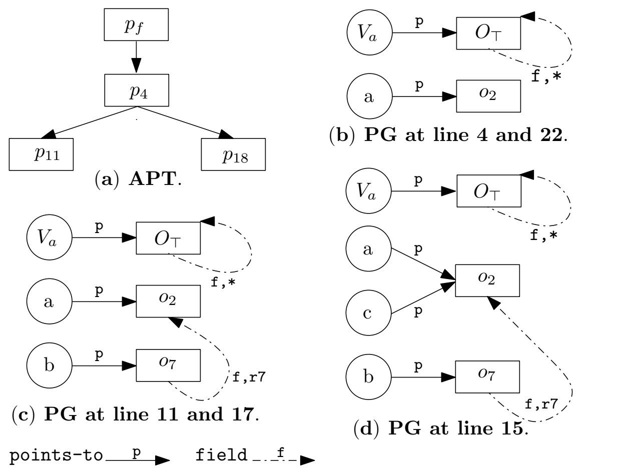


Figure 5: APT and the points-to graph generated by our analysis for the example shown in Figure 3.

RS_{16}	$\langle o_7, r1\rangle$
RS_{21}	$\langle o_2, r1\rangle$
	$\langle o_2, r2\rangle$

(a)

POC	(o_2, p_f)
	(o_7, p_4)

(b)

$CRS_{\langle 1,23\rangle}$	$\langle o_2, r1\rangle$
$CRS_{\langle 4,22\rangle}$	$\langle o_2, r1\rangle$
	$\langle o_2, r2\rangle$
	$\langle o_7, r1\rangle$

(c)

$MayWS_4$	$\langle o_2, r1\rangle$
$MayWS_{11}$	$\langle o_2, r2\rangle$
	$\langle o_7, r1\rangle$
	$\langle o_7, r7\rangle$
$MayWS_{18}$	$\langle o_2, r2\rangle$
	$\langle o_7, r1\rangle$
	$\langle o_7, r7\rangle$

(d)

$MustWS_4$	$\langle o_2, r1\rangle$
$MustWS_{11}$	$\langle o_2, r2\rangle$
	$\langle o_7, r1\rangle$
	$\langle o_7, r7\rangle$
$MustWS_{18}$	$\langle o_2, r2\rangle$
	$\langle o_7, r1\rangle$

(e)

Figure 6: Auxiliary data structures at different program points and nodes of APT.

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 points-to graph) after traversing one or more edges (points-to, or field). We add all the weak-edges reachable from x and b to AAL. We add all the fields reachable from x and b 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*: To refer to the individual auxiliary maps at different program labels, we subscript the maps with the labels. For example, RS_i refers to the *RS* map before processing the statement with label L_i. Note that each node in *APT* can be represented by a pair $\langle i, j\rangle$, where, L_i and L_j correspond to 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 $n = \langle x, y\rangle$ we 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: Figure 5 shows the APT and PGs for the example code shown in the Figure 3. 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 $\langle 4, 22\rangle$.

Figures 6a, 6d, and 6e show the contents of *RS*, *MayWS* and *MustWS* sets at different representative program points. Figures 6b and 6c show the *CRS* and *POC* maps as computed after the analysis of the function *foo*, respectively. For this example, $AAL = \phi$.

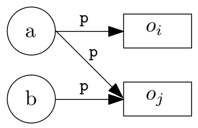
3.2 Code Generation

This section discusses our code generation scheme that takes into consideration 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 3 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 3. 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

```
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 7: An example to illustrate 'ambiguous' objects.

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.

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 7 shows a small snippet of `X10` code and its corresponding 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, both o_i and o_j, in Figure 7) 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`. Consider a statement L_j:`at-entry(p)`, and say its 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 $CAAL_{\langle j,k\rangle}$: $AOS = \{o_i | n \in CAAL_{\langle j,k\rangle} \wedge n \longrightarrow^{+w} o_i \in PG_j\}$; here we use notation $n \longrightarrow^{+w} o_i$ to indicate o_i is reachable from n (in PG_j) after traversing one or more weak edges. During a place-change operation, we will not be optimizing the copying of the objects in *AOS* and those will be copied fully. For each node $\langle j, k\rangle$ in the APT, we use the corresponding *AOS* set to generate code to copy the relevant data. This code generation process consists of two parts:

(A) Code emitted immediately before label L_j: Algorithm 8 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

```
1  Input : $PG_j, AOS_j, CRS_{\langle j,k\rangle}, MayWS_j, POC.$
2  begin
3      foreach $\langle o_i, g\rangle \in CRS_{\langle j,k\rangle} \wedge o_i \notin AOS_j$ do
4          if $POC(o_i) == pOf(j) \vee \langle o_i, g\rangle \in MayWS_j$ then
5              bool flag1 = $\exists\ (o_i \xrightarrow{f,g} o_l) \in PG_j.E$;
6              bool flag2 = flag1 $\wedge$ $(o_l \in AOS_j)$ ;
7              if $\neg flag1 \vee flag2$ 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 $PG_j.E$;
                       // Input is in
                          three-address-code form.
                          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($\langle o_i, g\rangle$,T1) // Remember the
                     mapping.
```

Figure 8: Function to emit the required code before each `at-construct` **corresponding to the APT node $\langle j, k\rangle$.**

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 Algorithm 9, 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 the `at-construct` (`at-exit` statement, that is).

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 stored 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: Algorithm 9 emits code to create an object for o_i and restore the value of $\langle o_i, g\rangle$ from temporaries (added in Step A); see Section 6 for a discussion on how it can be further optimized. For each pair $\langle o_i, g\rangle$ read in the body of `at-construct`

```
1  Input : $PG_j, AOS, RS_k, MayWS_k$
2  begin
3      Set $S = \phi$;
4      foreach $\langle o_i, g\rangle \in RS_k \wedge o_i \notin AOS$ do
5          createObject($o_i$, $H$, $S$);
6          if $\exists\ (o_i \xrightarrow{f,g} o_j) \in PG_j.E \wedge o_j \notin AOS$ then
7              createObject($o_j$, $H$, $S$);
8              Emit($H$.get($o_i$) || "." || g || "=" || $H$.get($o_j$) || ";");
9          else
10             Emit($H$.get($o_i$) || "." || g || "=" || $tMap$.get($\langle o_i, g\rangle$) ||
                 ";");
11     foreach $\langle o_i, g\rangle \in MayWS_k \wedge o_i \notin AOS$ do
12         createObject($o_i$, $H$, $S$);
13
14 Function createObject
15 Input : $o, H, S.$
16 begin
17     if $\neg S.contains(o)$ then
18         $S = S \cup o$;
19         String T1 = new TempName();
20         Emit ("val " || T1 || "=new " || typeOf($o$) || "();");
21         $H$.put($o$, T1);
```

Figure 9: Emit code to copy data from the temporaries, emitted in Figure 8. Function `createObject` **emits code to create a "substitute" object for o_i.**

and o_i is not an ambiguous object, we first call the function `createObject`(o_i, H, S) to emit code to create a substitute object for o_i (Line 5); say it is stored in a variable named tx_0. Then we 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, it is 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 10). 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 11- 12). For discussions on `at-expr` see Section 5.

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

```
1  def f():void{
2   val a:A = new A();
3   a.r1 = 36;
4   t1 = a.r1;
5   at(P){
6    val a1:A = new A();
7    a1.r2 = 27.0;
8    ... = a1.r2;
9    val b:B = new B();
10   b.r1 = 24;
11   b.r7 = a1;
12   for(i in D){
13    t3 = b.r1;
14    at(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 = 91;
21     ... = a2.r1;
22     }}
23  t2 = a1.r2;
24  at(Q){
25   val a3:A = new A();
26   a3.r1 = t1;
27   a3.r2 = t2;
28   ... = a3.r1;
29   ... = a3.r2; }}}
```

Figure 10: X10 synthetic code after code generation phase

Code generation for our running example: For the example shown in the Figure 3, our code generation pass takes the computed data-structures (shown in Figures 5 and 6) and generates code as shown in Figure 10.

As it can be seen, unlike the default `X10` compiler that copies the complete object (for example, in Figure 3 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 10 see lines 4, 13, and 23), creates the required substitute objects therein (for example, in Figure 10 see lines 6, 15, and 25), 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 25 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).

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 context-insensitive 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 *foo*, besides creating a summary for the points-to graph, we set the *CRS* (and *CAAL*) in output summary as the *CRS* (and *CAAL*) of the APT node corresponding to the function *foo*; 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 11). 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 *foo* may resolve to statically.

Impact on IS of the function foo. In points-to graph present in the IS of the function *foo*, 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 foo. Say the APT node for the function *foo* is represented by $\langle m, n\rangle$. (1) For each each object $o_{\langle i,[C]\rangle}$in the merged *PG*, if it is created in the function *foo*, 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 *foo* is pointing to in the OS. (3) We set *a* to point to whatever

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

Figure 11: Rules to translate a function-call instruction for inter-procedural analysis.

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 *foo*. (5) We update CRS by taking a union of current RS with $CRS_{\langle m,n\rangle}$, after removing the local object pairs of *foo*.

5 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 `p1` may be caught by the `catch` statement at `p1` 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, instead of the simple codes to store in a temporary and dereference the temporary, we emit code as shown in Figure 12.

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. Else, we use the *null* object as the substitute object.

6 DISCUSSION

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

No exceptions	With exceptions

t1=x.f1; t2=x.f2; …	var flag: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 12: Code generation in presence of exceptions

(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 help in enabling more optimizations (involving remote data) in future.
(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 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.
(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 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 (A). 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

	I/P	#at		s-Data (GB)		% reduction	
		Stat	Dyn	Base	AT-Opt	c-refer	c-miss
BF	256	45	6K	3.00	0.13	84.79	91.94
DST	256	101	15K	7.40	0.51	82.43	88.58
BY	128	69	549K	68.12	0.26	75.51	91.14
DR	256	39	311K	152.80	0.98	87.21	95.80
DS	256	195	286K	140.22	0.27	72.40	86.98
KC	256	121	83K	0.33	0.16	11.05	21.97
DP	256	97	65K	32.31	0.16	88.89	95.60
HS	256	130	400K	1.02	0.22	10.05	22.05
LCR	256	48	197K	0.48	0.09	10.74	22.75
MIS	256	68	18K	8.94	0.13	83.16	88.55
MST	256	254	145K	76.48	10.33	76.99	78.95
VC	256	86	5K	2.65	0.13	81.06	86.39

Figure 13: Characteristics of the IMSuite kernels. Abbreviations: s-data: Serialized data, c-refer: Cache reference, c-miss: Cache misses.

dummy constructor for each user-defined type of the input X10 program to assist creating the substitute objects.
(e) **Code generation for the statement `a = b`:** During code generation phase, if (1) the object o_i pointed by b is non-ambiguous, and (2) the code generator has not emitted code to create substitute object for o_i, then it indicates that there is no de-reference (or use) of o_i in the body of the at-construct; likely dead-code. Hence, we can either initialize a to null (will be removed by a later phase) or eliminate the statement altogether.
(f) **Code generation for at-expr**: X10 admits the at-constructs in two forms: as a statement (discussed in detail, in this paper) or expression (as shown in Figure 1a). Code generations for at-expr are 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, with 16 cores per processor and 64GB RAM; and a two node AMD system, where each node has a AMD Abu Dhabi 6376 processor containing 16 cores per processor, with 512GB RAM.

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 George et al. [11], we report the execution times by taking a geomean 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 13, columns 2 to 4, shows the chosen input sizes, 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 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.

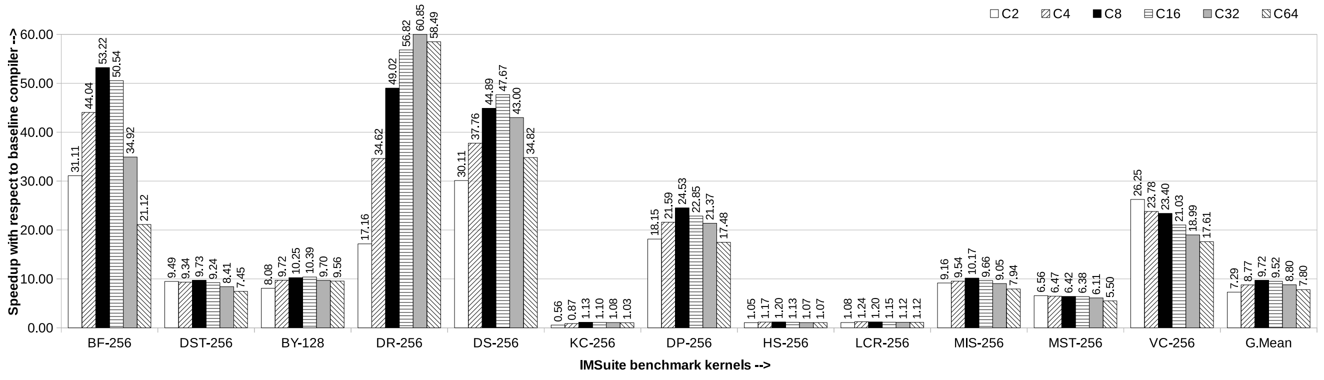
We executed the chosen kernels on the specified inputs by varying the number of places (in powers of two) and threads per place such that at any point of time the total number of threads (= #places × #num-threads-per-place) is equal to the number of cores. This is achieved by setting the runtime environment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

7.1 Evaluation of AT-Opt

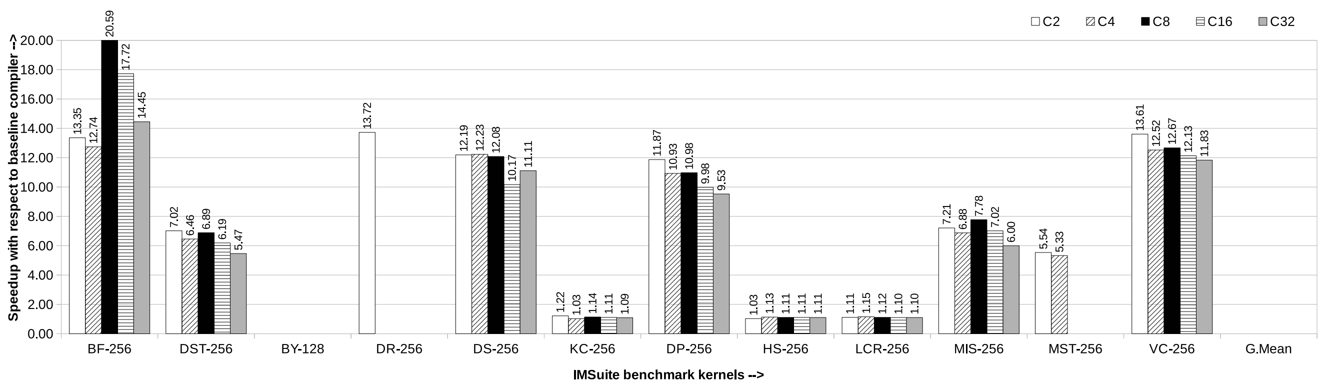
We report experimental results for two cases: (a) Base - the baseline version without any communication optimizations; (b) AT-Opt - the optimized version that uses the techniques described in this paper. In Figure 13, the columns 5 and 6 report the amount of data (excluding some common metadata and body 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-construct and the data serialized at each of them. We present the evaluation in two parts: one on a multi-node (distributed) setup, and one on a single-node (shared memory) setup.

Multi-node setup. Figure 14a shows the speedups achieved by using AT-Opt, on the two node Intel systems (32 cores each) for varying number of places and threads. We can see that the AT-Opt leads to significantly large speedups (geometric mean of 8.61×). For kernels other than VC and MST, as the number of places increase (from 2 to 8), due to the increase in the amount of inter-place communication (within and across the nodes), the speedups more or less increase.

For C_16 to C_64, the speedups decrease because of the TODO though the amount of communicated data remains the same, TODO TODO the obtained speedups for C_4 and



(a) Speedups on the two node Intel system; totalCores=64.



(b) Speedups on two node AMD system. totalCores=32.

Figure 14: Speedups for varying number of places (#P) and threads (#T). Configuration C_i denotes #P=i and #T=totalCores/i; Speedup = (execution time using `Base` / execution time using `AT-Opt`).

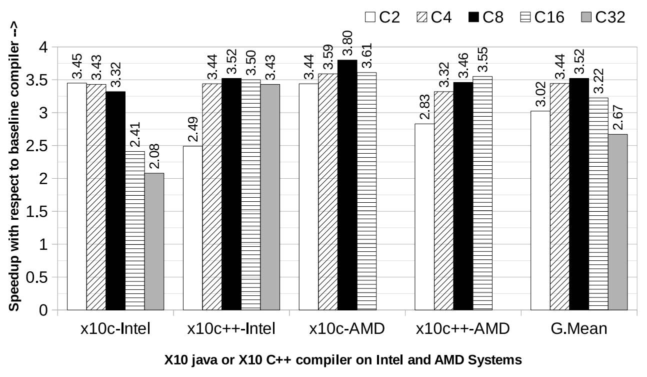


Figure 15: GeoMean speedups for varying number of places (#P) and threads (#T) for one core Intel or AMD system in x10c (Java) or x10c++ (Cpp) backend. Configuration C_i denotes #P=i and #T=xx/i;

C_8 are significantly better than C_2. This is because, for both `Base` and `AT-Opt` the programs run much slower (more inter-place communication) in configuration C_2 compared to their counterparts in the configurations C_4 and C_8. This is due to the way in which the input graph is distributed among the nodes.

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 13). 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).

Figure 14b shows the speedups achieved by the kernels on two AMD system (xx cores each) for varying number of places and threads. It can be seen that with respect to `Base`, the `AT-Opt` optimizer achieved large speedups (geometric mean of $xx\times$).

Shared environment setup. Figure 15 shows the geometric mean speedups achieved by the kernels on an Intel (32 cores) or on an AMD (xx cores) system in different `X10` compiler backend (x10c-Java and x10c++-Cpp). 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 because of less memory provided 64GB RAM, but as `AT-Opt` optimizes data across places it ran successfull within the provided memory space. We can see that the `AT-Opt` leads to significantly geometric mean speedups of 3.06× on Intel, 3.44× on AMD and overall speedups of 3.24× for one node.

In Figure 13, the columns 7 and 8 reports the % reduction in cache-references (10-88%) and cache-misses (22-95%) for x10c++ backend compiler run on an Intel system (32 cores) with respect to baseline compiler. Reduction in cache-references ⇒ less cache-polution ⇒ less cache-misses. Thus, overall gains = gains from reduced copying and data-transfer + reduced memory-access cost.

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 `at-constructs` executed. (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, 14] that aim to reduce the communication overheads across places resulting from redundant data transfer. Barik and Sarkar [5] eliminate memory loads by scalar replacement for `X10` constructs like `async`, `finish`, and `isolated`. Barik et al. [6] propose compiler optimizations to reduce communication and synchronization overheads by applying local optimizations such as scalar replacement for object fields and array accesses, and task localization, combined with supporting transformations such as loop distribution, scalar expansion, loop tiling and loop splitting for distributed memory environment. However, our technique has the following main advantages or differences with [6]:
(a) We go beyond scalars and reduce remote-data transfers involving heap objects.
(b) [6] does not handle writes to fields; `AT-Opt` can handle reads and writes of both mutable and immutable-fields.
(c) `AT-Opt` can handle 'ambiguous' objects (Section 3.2, Figure 7); [6] does not.
(d) Similarly, accesses of the form `a.f` and `b.f`, where `a` and `b` both are aliases, cannot be scalar replaced by [6]. But, `AT-Opt` can handle it. Lines 13 - 16 of Example 3.
(e) [6] cannot be applied where the object may be passed to a function as argument or receiver.

In the IMSuite benchmark [13] used for evaluation, the opportunities mainly involved points (a) and (e), listed above and as results [6] leds to no benefits. Thus, our work compliments their work by focusing on objects of user-defined types (other than distributed arrays and globals) that may be accessed across many functions.

Hayashi et al. [14] propose a common LLVM-based framework to perform communication optimization by creating LLVM IRs for PGAS programming languages. Once the programs in the PGAS languages are translated to the LLVM IR, the authors take advantage of the existing LLVM optimizations to optimize the code.

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 an framework called Fotran 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 that perform points-to and shape analysis [4, 10, 12, 20, 21, 24]. Chandra et al. [8] use a dependent type system to reason about locality of objects in X10 programs. We extend the escapes-to-connection graph (ECG) 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 32-core Intel system and a 64-core AMD system). We achieved significant gains in execution time with speedups of 3.06× and

3.44× on the Intel and AMD systems, respectively. Additionally, the experimental results show that the `AT-Opt` optimized programs scale better than the baseline versions. Even 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, Cilk, HJ, and so on.

REFERENCES

[1] S Agarwal, R Barik, V K Nandivada, R K Shyamasundar, and P Varma. 2008. Static Detection of Place Locality and Elimination of Runtime Checks. In *APLAS*. Springer Berlin Heidelberg, 53–74.

[2] S Agarwal, R Barik, V Sarkar, and R K Shyamasundar. 2007. May-happen-in-parallel Analysis of X10 Programs. In *PPoPP*. ACM, 183–193.

[3] M Alvanos, M Farreras, E Tiotto, J N Amaral, and X Martorell. 2013. Improving Communication in PGAS Environments: Static and Dynamic Coalescing in UPC. In *ICS*. ACM, 129–138.

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

[5] R Barik and V Sarkar. 2009. Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs. In *PACT*. 41–52.

[6] R Barik, J Zhao, D Grove, I Peshansky, Z Budimlic, and V Sarkar. 2011. Communication Optimizations for Distributed-Memory X10 Programs. In *IPDPS*. 1101–1113.

[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 (2007), 291–312.

[8] S Chandra, V Saraswat, V Sarkar, and R Bodik. 2008. Type Inference for Locality Analysis of Distributed Data Structures. In *PPoPP*. ACM, 11–22.

[9] Wei-Yu Chen, C Iancu, and K Yelick. 2005. Communication optimizations for fine-grained UPC applications. In *PACT*. 267–278.

[10] J Choi, M Gupta, M Serrano, V C Sreedhar, and S Midkiff. 1999. Escape Analysis for Java. In *OOPSLA*. ACM, 1–19.

[11] A Georges, D Buytaert, and L Eeckhout. 2007. Statistically rigorous Java performance evaluation. In *OOPSLA*. ACM, 57–76.

[12] R Ghiya and L J Hendren. 1996. Is It a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C. In *POPL*. ACM, 1–15.

[13] S Gupta and V K Nandivada. 2015. IMSuite: A benchmark suite for simulating distributed algorithms. *JPDC* 75 (2015), 1–19.

[14] A Hayashi, J Zhao, M Ferguson, and V Sarkar. 2015. LLVM-based Communication Optimizations for PGAS Programs. In *LLVM-HPC (LLVM)*. ACM, 1–11.

[15] S Hiranandani, K Kennedy, and Chau-Wen Tseng. 1991. Compiler Optimizations for Fortran D on MIMD Distributed-memory Machines. In *SC*. ACM, 86–100.

[16] S Hiranandani, K Kennedy, and Chau-Wen Tseng. 1992. Compiling Fortran D for MIMD Distributed-memory Machines. *Commun. ACM* 35, 8 (Aug. 1992), 66–80.

[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. *TOPLAS* 21, 6 (Nov. 1999), 1251–1297.

[18] S S Muchnick. 1997. *Advanced compiler design implementation*. M Kaufmann.

[19] J Paudel, O Tardieu, and J N Amarai. 2014. Optimizing shared data accesses in distributed-memory X10 systems. In *HiPC*. 1–10.

[20] N Rinetzky and M Sagiv. 2001. Interprocedural Shape Analysis for Recursive Programs. In *CC*. Springer Berlin Heidelberg, 133–149.

[21] A Salcianu and M Rinard. 2001. Pointer and Escape Analysis for Multithreaded Programs. In *PPoPP*. ACM, 12–23.

[22] A Sanz, R Asenjo, J López, R Larrosa, A Navarro, V Litvinov, S E Choi, and B L Chamberlain. 2012. Global Data Re-allocation via Communication Aggregation in Chapel. In *SBAC-PAD*. 235–242.

[23] V Saraswat, B Bloom, I Peshansky, O Tardieu, and D Grove. 2016. X10 Language Specification Version 2.6.0. http://x10.sourceforge.net/documentation/languagespec/x10-260.pdf. (2016).

[24] J Whaley and M Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *OOPSLA*. ACM, 187–206.