Optimizing Remote Communication in X10

ARUN THANGAMANI, IIT Madras
V. KRISHNA NANDIVADA, IIT Madras

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 can be very expensive as they need to copy all the required data from the current place to the remote place. However, identifying the necessary number of place-change operations and the required data during each place-change operation are non-trivial tasks, especially in the context of irregular applications (like graph applications) that contain complex code with large amounts of cross-referencing objects – not all of those objects may be actually required, at the remote place. In this paper, we present AT-Com, a scheme to optimize X10 code with place-change operations.

AT-Com consists of two inter-related new optimizations (i) AT-Opt that minimizes the amount of data serialized and communicated during place-change operations, and (ii) AT-Pruning that identifies/elides redundant place-change operations and does parallel execution of 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. AT-Pruning introduces a set of program transformation techniques to emit optimized code such that it avoids the redundant place-change operations. We have implemented AT-Com in the x10v2.6.0 compiler and tested it over the IMSuite benchmark kernels. Compared to the current X10 compiler, the AT-Com optimized code achieved a geometric mean speedup of 18.72× and 17.83×, on a four-node (32 cores per node) Intel and two-node (16 cores per node) AMD system, respectively.

ACM Reference Format:
Arun Thangamani and V. Krishna Nandivada. 2019. Optimizing Remote Communication in X10. 1, 1, Article 1 (January 2019), 25 pages. https://doi.org/10.1145/3345558

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 [26] 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.

This journal submission is an extended and reorganized version of a previous article from conference proceedings [28]. Specifically, Sections 1, 5, 6, 7 and 8 include research results that are not published elsewhere (new material: more than 30%).
Authors' addresses: Arun Thangamani, IIT Madras, arunt@cse.iitm.ac.in; V. Krishna Nandivada, IIT Madras, nvk@iitm.ac.in.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
© 2019 Association for Computing Machinery.
XXXX-XXXX/2019/1-ART1
https://doi.org/10.1145/3345558

```
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

```
def elect(var ph:Long){
finish {
for (i in D) {
 async at(D(i)) {
  if(nSet(i).status){...}
  if(!lValue.equals(0))...
}}} ... }
```
(c) HS: leader_elect_hs

Fig. 1. Snippets from three IMSuite kernels. D is the distribution of the arrays.

Fig. 1a shows a code snippet in X10, of the MST (builds a minimum spanning tree) kernel from IMSuite [14]; 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, then the function returns true (code not shown).

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 Fig. 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 (setCheck(i)). At the remote location, the compiled 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 Fig. 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 Fig. 1a, Fig. 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 (code not shown) uses the the distributed array this.nSet. Like before, at the at-construct, the X10 compiler emits code to transmit the complete this object, though only the references to nSet is required; note: the sub-partition of nSet owned by the target place is already present at the respective target places. We have studied many distributed kernels and found that the amount of such redundantly copied data can be prohibitively large. For example, the X10 compiled MST and BF kernels (snippets in Fig. 1, input size=256) copied 76.5 GB and 3.0 GB data, respectively; we 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, complex deep procedure calls and arbitrary operations involving heap. As another example depicting the issue of communication overheads incurred during the translation of code with at-constructs, Fig. 1c shows a code snippet of the HS kernel (to elect a leader in a bidirectional ring network) from IMSuite [14]. The elect function starts its computation on the input nodes to select a leader by (i) iterating through all the elements (*points*) in the domain of the Distribution D, and (ii) performing as many place-change operations. For example, the X10 compiled MST and HS kernels (snippets in Figures 1a and 1c, input size=256), led to 193,103 and 400,477 remote-communications, respectively. However, the computation can be performed with fewer place-change operations – by performing one place-change operation per place, and invoking

the code body for each point assigned to that place. We have found such similar opportunities in many distributed kernels.

In this paper, we present a new scheme (called AT-Com) to efficiently translate X10 code with at-constructs. AT-Com includes two inter-related optimizations: AT-Opt that identifies/elides the unnecessary data that is otherwise communicated to the remote places, and AT-Pruning that identifies/elides the redundant place-change operations.

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 need to depend on global-value numbering, as every place-change operation has to be handled as an independent place-change operation. 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 significantly reduces the remote communication and (consequently) execution time. For example, for MST and BF, 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 four places, leads to a speedup of 6.6× and 57.3×, respectively, on a four-node (32 cores/node) Intel system.

Our proposed optimization AT-Pruning reduces redundant place-change operations across places. This leads to a significant reduction in place-change operations and consequently execution time. For example, for the MST and HS kernel, AT-Pruning reduces the number of place-change operations from 193,103 to 118,643, and 400,777 to 7,777 respectively. AT-Pruning also emits code to execute place-change operations in parallel, wherever possible. Thus, AT-Pruning reduces the amount of data serialized and the cost of place-change operations. For example, for four places, for the MST and HS kernels, AT-Pruning leads to a speedup of 2.5× and 8.5×, respectively, on a four-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 Fig. 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 (Fig. 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-Com in the context of X10, it can also be applied to other PGAS languages like HJ [15] and Chapel [8]. 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.

Our 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 scheme AT-Com to optimize X10 programs that perform remote communication. It includes two optimizations: (i) AT-Opt that avoids the copying of redundant data across places (Sections 3 and 4), and (ii) AT-Pruning that removes redundant place-change operations and does parallel execution of place-change operations (Section 5).
- We extend AT-Com to handle programs that may throw exceptions (Section 6).
- We have implemented AT-Com in the x10v2.6.0 compiler C++ backend.

• We have evaluated AT-Com ver all the IMSuite kernels on two different hardware systems: a four-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-Com leads to geometric mean speedups of 18.72× and 17.83×, on the Intel system and AMD system, respectively (Section 8).

2 BACKGROUND

In this section, we briefly discuss a few X10 constructs and some pertinent X10 concepts. Interested readers may refer to the X10 specification [26] for details.

• `async S`: spawns a new asynchronous activity to execute `S`.
• `finish S`: acts as join point and waits for all spawned activities in `S` to terminate.
• `at(P) S`: the place-change operator is a synchronous construct and executes the statement `S` at place `p`. The syntax `at(p) async S` spawns an activity at place `p` to execute statement `S`. For the ease of presentation, we represent each such `at-construct` using the sequence of three instructions: `at-entry`; `S`; `at-exit`.
• `Distribution`: The languages provides the abstraction of a point in n-dimensional space. A set of points can be distributed across the places. Assuming `D` is a distribution, the loop '`for (i in D) S`', iterates over the points in the domain of `D`, ordered by the places.

In X10, 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. Each place is assigned with a unique id. The program execution starts at the place `0`.

At runtime, the hosts, initial count for places and workers can be set by using the environment variables `X10_HOSTFILE` (or `X10_HOSTLIST`), `X10_NPLACES` and `X10_NTHREADS`, respectively.

3 AT-OPT: OPTIMIZING REMOTE DATA TRANSFERS

In this section, we propose a compile-time technique AT-Opt 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. AT-Opt reduces these overheads. For each `at-construct`, AT-Opt 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 6, 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 AT-Opt analyzer creates two graphs: (1) an *abstract-place-tree* that captures the place-change operations (from a "source" to "target"), and (2) a flow-sensitive *points-to graph* that captures the points-to information of X10 objects (by extending 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

[1]Without any loss of generality, we assume that the input is a simplified X10 program in three-address-code form [19], each statement has a unique associated label, and variables have unique names.

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

Fig. 2. Definitions of different sets.

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

Fig. 3. Auxiliary data-structures

constructs is given by L_p (see Fig. 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 Fig. 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 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 $v \rightarrow^{\text{p}} o$ are created because of assignment of objects (say, o) to variables (say, v).
2. *field edges* $\subseteq N_o \cup \{O_\top\} \times \text{Fields} \times N_o \cup \{O_\top\}$: These edges of the form $o_1 \rightarrow^{\text{f,g}} o_2$ are created because of assignment of objects (say, o_2) to the fields (say, g) of objects (say, o_1).

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

Besides maintaining APT (global) and PG (at each statement), we maintain a few other data-structures, listed in Fig. 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 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 accessed at pOf(j); this set is used for identifying ambiguous objects during code-generation phase (Section 3.2). Note that we separately compute the AAL set instead of deriving it from the points-to graph after the first pass, as otherwise, it will lead to imprecision. For example, consider a points-to graph like the one shown in Fig. 7(b) and say the object o_j is accessed at the target place. Now, if the points-to graph is used to derive the AAL, then it would include both `a`, and `b`, even though the access might be via only the variable `b` (a strong edge).

$$\mathbf{L}_j : \texttt{at-entry}(p) \quad (N_p, E_p) \Rightarrow (N_p \cup \{p_j\}, E_p \cup \{(\mathrm{pOf}(L_j) \to p_j)\})$$

(a) Impact on the APT

1. $L_j : a = \texttt{new T()}$

$$\begin{array}{ll} (N,E) & \Rightarrow (N \cup \{o_j\}, (E - \{a \to^{\mathrm{p}} y \mid a \to^{\mathrm{p}} y \in E\}) \cup \{a \to^{\mathrm{p}} o_j\}) \\ POC & \Rightarrow POC \cup \{(o_j, \mathrm{pOf}(L_j))\} \end{array}$$

2. $L_j : a = b$

$$\begin{array}{ll} (N,E) & \Rightarrow (N, (E - \{a \to^{\mathrm{p}} y \mid a \to^{\mathrm{p}} y \in E\}) \cup \{a \to^{\mathrm{p}} z \mid b \to^{\mathrm{p}} z \in E\}) \\ AAL & \Rightarrow AAL \cup \{b\} \text{ // } \textit{if b has weak-edges.} \end{array}$$

3. $L_j : a = b.g$

$$\begin{array}{ll} (N,E) & \Rightarrow (N, (E - \{a \to^{\mathrm{p}} y \mid a \to^{\mathrm{p}} y \in E\}) \cup \{a \to^{\mathrm{p}} z \mid b \to^{\mathrm{p}} x \in E \wedge x \to^{\mathrm{f,g}} z \in E\}) \\ RS & \Rightarrow RS \cup \{\langle o_i, g\rangle \mid b \to^{\mathrm{p}} o_i \in E \wedge \langle o_i, g\rangle \notin MustWS\} \\ AAL & \Rightarrow AAL \cup \{b\} \text{ // } \textit{if b has weak-edges} \\ AAL & \Rightarrow AAL \cup \{\langle o_i, g\rangle \mid b \to^{\mathrm{p}} o_i \in E \wedge b.g \text{ has weak-edges}\} \end{array}$$

5. $L_j : a = x.bar(b)$

$$\begin{array}{ll} (N,E) & \Rightarrow (N, E \cup \{a \to^{\mathrm{p}} O_\top\} \cup \{y \to^{\mathrm{f},*} O_\top \mid x \to^{+} y \in E\} \cup \{z \to^{\mathrm{f},*} O_\top \mid b \to^{+} z \in E\}) \\ RS & \Rightarrow RS \cup \{\langle o_i, *\rangle \mid x \to^{+} o_i \in E \vee b \to^{+} o_i \in E\} \\ AAL & \Rightarrow AAL \cup \{z \mid z \in \{x, b\} \wedge z \text{ has weak-edges in } E\} \\ AAL & \Rightarrow AAL \cup \{\langle o_i, g\rangle \mid (x \to^{+} o_i \in E \vee b \to^{+} o_i \in E) \wedge \langle o_i, g\rangle \text{ is a weak-edge}\} \end{array}$$

4(i). $L_j : a.g = b$ (Strong update)

$$\begin{array}{ll} (N,E) & \Rightarrow (N, (E - \{y \to^{\mathrm{f,g}} z \mid a \to^{\mathrm{p}} y \in E \wedge y \to^{\mathrm{f,g}} z \in E\}) \cup \{y \to^{\mathrm{f,g}} x \mid b \to^{\mathrm{p}} x \in E \wedge a \to^{\mathrm{p}} y \in E\}) \\ MayWS & \Rightarrow MayWS \cup \{\langle o_i, g\rangle \mid a \to^{\mathrm{p}} o_i \in E\} \\ MustWS & \Rightarrow MustWS \cup \{\langle o_i, g\rangle \mid a \to^{\mathrm{p}} o_i \in E\} \\ AAL & \Rightarrow AAL \cup \{b\} \text{ // } \textit{if b has weak-edges.} \end{array}$$

4(ii). $L_j : a.g = b$ (Weak update)

$$\begin{array}{ll} (N,E) & \Rightarrow (N, E \cup \{y \to^{\mathrm{f,g}} x \mid b \to^{\mathrm{p}} x \in E \wedge a \to^{\mathrm{p}} y \in E\}) \\ MayWS & \Rightarrow MayWS \cup \{\langle o_i, g\rangle \mid a \to^{\mathrm{p}} o_i \in E\} \\ AAL & \Rightarrow AAL \cup \{a\} \text{ // } \textit{a might not be} \in \textit{AAL, yet} \\ AAL & \Rightarrow AAL \cup \{b\} \text{ // } \textit{if b has weak-edges.} \end{array}$$

(b) Impact on the points-to-graph PG and auxiliary data structures (only the updated ones are shown). For brevity, we omit the subscripts on the auxiliary data-structures *RS, MustWS, MayWS,* and *AAL*.

Fig. 4. Rules for intra-procedural analysis.

3.1.3 Intra-procedural flow-sensitive analysis. We now discuss our scheme 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 for completeness. For the ease of presentation, we now 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 0th argument *this*), conservatively, we include an edge $a_j \to^{\mathrm{p}} O_\top$ in E_i. Also, we add an edge $O_\top \to^{\mathrm{f},*} O_\top$ to indicate that all field edges from $O_\top$ will point to $O_\top$. The rest of the auxiliary data structures are initialized to empty.

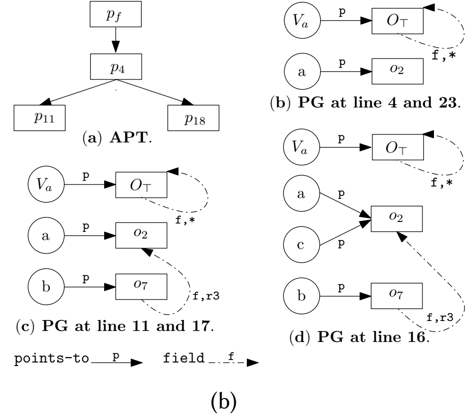
Statements and related operations. Fig. 4 shows how we update the APT, PG and auxiliary data structures on processing the different `X10` statements. For each statement $L : Stmt$, each transformation is shown as a transition of the form $X \Rightarrow X'$, where X is a data-structure before processing 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 (Fig. 4a). Further, we reset $AAL = MayWS = MustWS = RS = \Phi$ (not shown). Note: Only the `at-entry` instruction updates the APT.

```
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
```

(a)



(b)

Fig. 5. (a) Example synthetic X10 code. (b) Generated APT and points-to graph for code in Fig. 5a.

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, Fig. 4b – creates a new object node o_i and updates the points-to edges), we add $(o_i, \text{pOf}(L_j))$ to *POC*.

Copy statement. L_j:$a = b$: Besides updating the PG (Rule 2. Fig. 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. Fig. 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, Fig. 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. If b has weak-edges then b is added to *AAL*. 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. Fig. 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 *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 [19] and stop after reaching a fixed point (from the point of view of *APT* and *PG*).

Post analysis. Finally, we populate two 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 APT predecessors of n and that definition may reach a statement in n or one of its APT successors. An entry $k \in CAAL_n$ indicates that k is in the ambiguous-access-list of either n or one of its APT successors.

RS_{16}	$\{\langle o_7, r1\rangle\}$
RS_{21}	$\{\langle o_2, r1\rangle, \langle o_2, r2\rangle\}$
POC	$\{(o_2, p_f),$ $(o_7, p_4)\}$

(a)

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

(b)

$MayWS_4$	$\{\langle o_2, r1\rangle\}$
$MayWS_{11}$ = $MayWS_{18}$	$\{\langle o_2, r2\rangle,$ $\langle o_7, r1\rangle,$ $\langle o_7, r3\rangle\}$

(c)

$MustWS_4$	$\{\langle o_2, r1\rangle\}$
$MustWS_{11}$	$\{\langle o_2, r2\rangle, \langle o_7, r1\rangle,$ $\langle o_7, r3\rangle\}$
$MustWS_{18}$	$\{\langle o_2, r2\rangle, \langle o_7, r1\rangle\}$

(d)

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

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 $\langle i, j\rangle$, 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) $CAAL_n = AAL_y \cup_{\langle p,q\rangle \in aptChild(n)} CAAL_{\langle p,q\rangle}$; where $aptChild(n)$ returns all nodes rechable from n in the APT.

Example: Consider the example code shown in the Fig. 5a. Here, $L_p = \{p_4, p_{11}, p_{18}\}$ and $L_o = \{o_2, o_7\}$. Fig. 5b 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 $\langle 4, 22\rangle$. Fig. 6 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

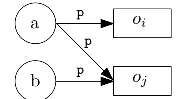
We now discuss our code generation scheme that uses the information (*APT*, *PG*, *RS*, *CRS*, *MayWS*, *POC*, and *CAAL*) computed in Section 3.1.3 to generate code which copies only the required data to the target places during a place-change operation. For the ease of explanation, we show the rules as a source-to-source translation scheme; the generated code is compiled by the current `X10` compiler.

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 Fig. 5a 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 Fig. 5a. 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: 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 by the `X10` compiler; `t3` will be copied instead. We 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, Fig. 7 shows a small snippet of `X10` code and its points-to graph (before line 2). At runtime line 3 may write to field $\langle o_i, r1\rangle$ or $\langle o_j, r1\rangle$ and hence, the field dereference at line 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.

```
1 at(p1){..//oi, oj created here
2  at(p2){
3    a.r1 = 9;
4    ... = b.r1; }}
```

(a) x10 code snippet



(b) *PG* at Line 2.

Fig. 7. An example to illustrate 'ambiguous' objects.

```
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 f1 = ∃ (o_i →^{f,g} o_l) ∈ PG_j.E;
6        bool f2 = f1 ∧ (o_l ∈ AOS_j) ;
7        if ¬f1 ∨ f2 then // Emit code to copy
8          if H.contains(o_i) then // o_i is
           referred to by a new name
9            x=H.get(o_i);
10         else
11           Set x to 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)// save
              mapping
```

(a) Emit the required code before the at-construct of the APT node $\langle j, k \rangle$.

```
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);
```

(b) Emit code to copy data from the temporaries.

```
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) || "();");
         H.put(o, T1);
```

(c) Emits code to create a "substitute" obj for o_i.

Fig. 8. Auxiliary functions

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 Fig. 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.

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: Fig. 8a 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 Fig. 8b, 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  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; }}}
```

Fig. 9. Optimized code for Fig. 5a.

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: Fig. 8b 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 7. 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 variables with the temporaries created. Section 7 discusses a further optimization for the statement `a=b`.

Code generation for our running example: For the example shown in the Fig. 5a, our code generation pass takes the computed data-structures (shown in Figures 5b and 6) and generates code as shown in Fig. 9.

As it can be seen, unlike the default `X10` compiler that copies the complete object (for example, in Fig. 5a 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 Fig. 9 see lines 4, 13, and 22), creates the required substitute objects therein (for example, in Fig. 9 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 7 for a further optimization in the generated code).

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

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

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

Fig. 10. Rules to translate a function-call instruction for inter-procedural analysis.

analysis [19]. 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 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 (Fig. 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 Fig. 10). 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 and OS of the function bar. (i) IS: 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. (ii) OS: 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

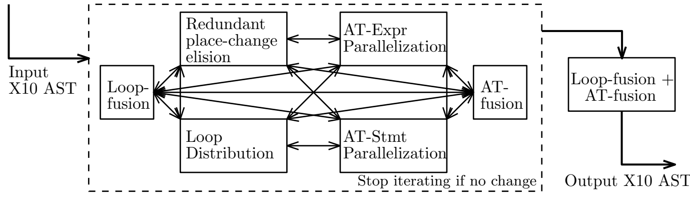


Fig. 11. Block diagram for AT-Pruning Fig.12

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*.

5 AT-PRUNING: REDUCING THE OVERHEADS OF PLACE CHANGE OPERATIONS

In this section, we propose new program transformation techniques to avoid redundant place-change operations and allow parallel execution of the remaining place-change operations – both of these together lead to reduction in the overheads resulting from the place-change operations. Though AT-Pruning is not directly related to communication optimization, these rules reduces the number of place-change operations, and thereby reduce the number of remote-communications. For the ease of presentation, in this section, we assume that the input code does not throw any exceptions. In Section 6 we show, how we handle code that may throw exceptions.

Fig. 11, shows the overall block-diagram of our proposed optimization phase AT-Pruning. This phase takes as input the Abstract Syntax Tree (AST) of the input X10 program and generates an optimized X10 AST. The optimization phase continuously invokes one of the seven mini-transformations (listed in Fig. 12), until no further change is possible. Note that except for the Loop+AT-Fusion rule, the rest can be applied in any order without impacting the final generated code. The rule Loop+AT-Fusion is invoked when no more program transformation is possible from other rules listed in Fig. 12. Applying the Loop+AT-Fusion rule on top of the generated transformed code helps improve the code generated from AT-Expr parallelization (rule [B(i)]). For efficiency, we apply these transformations in a bottom-up fashion in the AST. We specify the rules in the form $X \Rightarrow Y$ to indicate that code of the form X gets translated to Y. With each rule, we optionally mention (as conditions) the condition under which the rules can be invoked.

Removing redundant place-change operations. In Fig. 12, rule [A] removes redundant place-change operations, by generating code that invokes a single place-change operation per place (in contrast to one place-change operation per each point in the distribution `D`). The inner loop (over the place `p`) in the generated code iterates over the points assigned to the place `p`. Let `P` be the total number of available places and `K` be the points over the distribution `D` (as per the `X10` semantics the total points will be >= available places), then the total number of place-change operations by un-optimized and optimized code is `K` and `P`, respectively. Note that the loop in the input code iterates over the points in `D`, ordered by the place numbers, and this order is same as the order of the iterations of the generated code. This rule does not require any pre-conditions.

Parallelization of place-change operations. In Fig. 12, rules [B(i)] and [B(ii)] parallelizes the invocation of different place-change operations, thereby improving the overall code performance. In rule [B(i)], in the generated code, the place-change operations invoked inside a serial loop in the input loop are parallelized to be evaluated before the main loop. The values thus computed are stored in a temporary rail (array) `r1` (of size = number of points in `D`) and reused in the main loop. The dependencies between `S1`, `S2` and `E1` are computed using standard techniques [20]. Interestingly, say, 'p' be the number of places and 'k' be the size of `Distribution D`, then the

[A] Redundant place-change operation elision

(i) Serial place-change operations

```
for (i in D) {
  at(D(i)) { S1; }
}
```
⟹
```
for (p in place.Places()) {
  at(p) {
    for (i in D(p)) { S1; } } }
```

(ii) Parallel place-change operations

```
for (i in D) {
  async at(D(i)) { S1; }
}
```
⟹
```
for (p in place.Places()) {
  at(p) {
    for (i in D(p)) { async S1; }}}
```

[B] Parallelization of place-change operations

(i) AT-Expr parallelization

```
//E1 is independent of S1, S2 and other instances of E1
for (i in D) {
  S1; ... = at(D(i)) E1; //S1 has no at-expression
  S2;
}
```
⟹
```
Val r1 = new Rail[T](D.size());
finish for (i in D) {
    async r1(i) = at(D(i)) E1;
}
for (i in D)    {
  S1; ... = r1(i); S2;  }
```

(ii) AT-Stmt parallelization

```
at(p) { for (i in D(p)) { async S1; } }
```
⟹
```
async { at(p) {
    for (i in D(p)) { async S1; } } }
```

[C] Fusion

(i) Loop-fusion

```
//S1 and S2 are independent of each other
for (p in place.Places()) { at(p) S1; }
for (p in place.Places()) { at(p) S2; }
```
⟹
```
for (p in place.Places()) {
  at(p) S1;
  at(p) S2;
}
```

(ii) AT-fusion

```
at(p) S1; at(p) S2;
```
⟹
```
at(p) { S1; S2; }
```

(iii) Loop-fusion + AT-fusion

```
finish { for (i in D) {
  async ... = at(D(i)) E1; } }
finish { for (i in D) {
  async ... = at(D(i)) E2; } }
```
⟹
```
finish { for (i in D) {
  async{ t(i)=at(D(i)) {t1=new T();
    t1.f1 = E1; t1.f2 = E2; t1;};
  ... = t(i).f1; ... = t(i).f2; }
} }
```

[D] Loop-distribution

```
//S1 and S2 are independent of each other and at least one of them contains an at-statement
for (i in D) {
  S1; S2; }
```
⟹
```
for (i in D) { S1; }
for (i in D) { S2; }
```

Fig. 12. AT-Pruning rules. D: arbitrary distribution; S1, S2: arbitrary statements; E1, E2: arbitrary expressions

number of remote-communications performed by the code compiled using the synchronization-elimination and place-level strip-mining techniques of Barik et al. [6] are '2k' and 'p + k' respectively. In contrast, the rule [B(i)] leads to only 'k' remote-communications only.

In rule [B(ii)] the parallelization ensures that the place-change operation and serial loop are invoked inside a parallel task. This rule can only be invoked if the loop index variable `i` and the variable `D` are not read or modified anywhere after the at-statement.

Fusion. In Fig. 12, rules [C(i)], [C(ii)], and [C(iii)] show three variants of fusion techniques that we use to reduce the number of place-change operations. Rule [C(i)] is the standard loop fusion transformation applied to codes involving place-change operations. Rule [C(ii)] optimizes consecutive place-change operations (may have been present in the input code or generated as part of some prior transformation like the one using rule [C(i)]) into a single place-change operation. Rule [C(iii)] does both fusion of loops and place-change operations. It creates a temporary object to save the values of `E1`, and `E2`. Like rule [C(ii)], rule [C(iii)] is also used to additionally optimize code generated as part of the other translation rules (for example, rule [B(i)]). Even though the rule is shown for only for a sequence involving two `finish` statements (having a specific syntactic

```
for(i in D1){
 at(D1(i)){
  async S1;}
 S2;
}
```
(a) I/P code ⇒

```
for(i in D1){
 at(D1(i)){
  async S1;}}
for(i in D1){
 S2;}
```
(b) applying rule [D] ⇒

```
for(p in place.Places()){
 at(p){
  for(i in D1(p)){
   async S1; } } }
for (i in D1) { S2; }
```
(c) applying rule [A] ⇒

```
for (p in place.Places()) {
 async { at(p){
  for(i in D1(p)){async S1;}
 } } }
for (i in D1) { S2; }
```
(d) applying rule [B(ii)]

```
for(j in D2) {
 ...=at(D2(j)) ar1(j);
 ...=at(D2(j)) ar2(j);
}
```
(e) S1 expansion ⇒

```
Val r1=new Rail[T](D2.size);
Val r2=new Rail[T](D2.size);
finish { for(j in D2){
 async r1(j)=at(D2(j))ar1(j);}}
finish { for(j in D2){
 async r2(j)=at(D2(j))ar2(j);}}
for(j in D2) {
 ...=r1(j); ...=r2(j);}
```
(f) applying rule [B(i)] ⇒

```
Val r1=new Rail[T](D2.size);
Val r2=new Rail[T](D2.size);
finish{for (j in D2){async{
 t(j)=at(D2(j)){t1=new T();
   t1.f1 = ar1(j); t1.f2 = ar2(j);
   return t1; };
 r1(j)=t(j).f1; r2(j)=t(j).f2; }}}
for(j in D2){...=r1(j);...=r2(j);}
```
(g) applying rule [C(iii)]:

Fig. 13. Example showing the application of the transformation rules of AT-Pruning.

pattern), it is applicable to any number of consecutive `finish` statements (with that pattern). The variable `t` is an array of type `T` that can hold the values of `E1` and `E2`. Note that unlike other rules, the rule [C(iii)] does not commute and hence it is invoked at the end (Fig. 11). This rule [C(iii)] reduces the number of place-change operations from ‘n×K’ to K where ‘n’ is the number of such consecutive `finish` statements (with that pattern) and ‘K’ is the number of available points in the distribution `D`.

Loop-distribution. The rule [D] is the standard loop-distribution rule, applied only over statements when at least one of them contains an at-statement. This rule helps make the code amenable to be optimized using the prior rules ([A], [B], [C]).

Example. Fig. 13 shows an example on how the translation rules shown in Fig. 12 are applied on an `X10` snippet (Fig. 13(a)). Fig. 13(d) shows the code obtained after applying rules [D], [A] and [B(ii)]. If the statement `S1` in Fig. 13(a), expands to the code shown in Fig. 13(e) then Fig. 13(g) shows the translation of `S1`, after applying rules [B(i)] and [C(iii)]. Note that rule [B(i)] is applied twice on `S1` (Fig. 13(e)) for the two AT-Expr inside the loop. Because of applying rule [B(i)] twice, the new optimized code became amenable to applying rule [C(iii)]. It can be seen that, the input `X10` program is exposed to different `AT-Pruning` rules shown in Fig. 12 until the code reaches a state where no more rules can be applied.

6 AT-OPT AND AT-PRUNING WITH EXCEPTIONS

We now extend `AT-Opt` and `AT-Pruning` to handle code that may throw exceptions.

`AT-Opt` *in presence of exceptions. (a) In the 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) In the 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 Fig. 14, instead of the simple codes

AT-Opt

```
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 object 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=t2;...}
...
```

AT-Pruning

[B] Parallelization of place-change operations

(i) AT-Expr parallelization

```
//E1 is independent of S1, S2 and
//other instances of E1
for (i in D) {
  S1;  //S1 has no at-expression.
  ... = at(D(i)) E1;
  S2;
}
```
⟹
```
Val r1 = new Rail[T](D.size());
Val Ex = new Rail[Exception](D.size());
finish for (i in D) {
  async { try { r1(i) = at(D(i)) E1; }
  catch (Exception e) { Ex(i) = e; } }}
for (i in D) { S1;
  if (Ex(i)!=null) { throw Ex(i); }
  else { ... = r1(i); } S2; }
```

[C] Fusion

(i) Loop-fusion

```
//S1 and S2 are independent of each other
//S1 throws no exceptions
for(p in place.Places()){at(p) S1;}
for(p in place.Places()){at(p) S2;}
```
⟹
```
Var Ex:Exception;
for (p in place.Places()) {
  at(p) S1;
  try { at(p) if(!F) S2; }
  catch (Exception e) {Ex=e; F=true;}}
if(F) throw Ex;
```

(iii) Loop-fusion + AT-fusion

```
//First for-loop throws no exceptions
finish { for (i in D) {
  async {... = at(D(i)) E1;}}}
finish { for (i in D) {
  async {... = at(D(i)) E2;}}}
```
⟹
```
Val Ex = new Rail[Exception](D.size());
finish { for (i in D) {
  async { t(i) = at(D(i) { t1 = new T();
    t1.f1 = E1; try { t1.f2 = E2; }
    catch (Exception e) {Ex(i) = e;}t1;};
  ... = t(i).f1; ... = t(i).f2; } } }
finish for (i in D) { if(Ex(i)) {async throw Ex(i);}}
```

```
finish for (i in D) {
  async{try{ r1(i)=at(D(i)) E1;}
  catch (Exception e){Ex1(i)=e;}}}
finish for (i in D) {
  async{try{ r2(i)=at(D(i)) E2;}
  catch (Exception e){Ex2(i)=e;}}}
```
⟹
```
finish { for (i in D) {
  async { t(i) = at(D(i) { t1 = new T();
    try{t1.f1=E1;} catch (Exception e){Ex1(i)=e;}
    try{t1.f2=E2;} catch (Exception e){Ex2(i)=e;}
    t1;};
  ... = t(i).f1; ... = t(i).f2; } } }
```

[D] Loop-distribution

```
//S1 and S2 are independent of each other
//One of S1 and S2 contains an at-statement
//S2 throws no exceptions
for (i in D) { S1; S2; }
```
⟹
```
Val Ex = new Rail[Exception](D.size());
for (i in D) { try { S1;}
  catch (Exception e){Ex(i) = e;break;}}
for (i in D) if (Ex(i)==null) S2; else throw Ex(i);
```

Fig. 14. AT-Opt and AT-Pruning in presence of Exceptions

(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.

`AT-Pruning` *in presence of exceptions.* Except for the rules [A], [B(ii)] and [C(ii)], the other rules in Fig. 12 may alter the execution order of the statements and hence need to be handled in a special manner, in the presence of exceptions. Fig. 14 presents the modified rules.

Rule [B(i)]: If any exception is thrown during the evaluation of `E1` then we save the exceptions in an array and throw the first exception in the main loop (after executing `S1`). Rule [C(i)]: If an exception is thrown in any instances of `S2`, then (a) the exception is saved, (b) the latter instances of

S2 are not executed, and (c) the saved exception is thrown outside the loop. Rule [C(iii)]: We catch the exceptions thrown in the main loop and throw them outside the loop. We also do Loop- and AT-fusion in presence of try-catch statements (to handle the output of rule [B(i)]). Rule [D]: We catch any exception thrown, while executing the instances of S1 and execute only those instances of S2 whose corresponding instance of S1 executed and did not throw any exception. The exceptions thrown from S1 are captured in an array Ex and thrown before executing the corresponding S2.

7 DISCUSSION

We now discuss some interesting underlying points about our proposed optimization scheme.
(a) AT-Com **in the compiler.** Both AT-Opt and AT-Pruning are high-level optimizations that do not interfere negatively with other high-level optimizations. AT-Opt requires a pre-pass of expression simplification (to three-address code). AT-Opt and AT-Pruning should be invoked before any other high-level-optimization that may change the structure of at-constructs. AT-Opt and AT-Pruning can be applied in any order to get the combined improvements.
(b) **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.
(c) **Partitioning Aggregate Object fields.** Access to object field of type vectors, arrays, and rails are handled using an element insensitive approach. The reason for not using element sensitive approach is because of more complex analysis and additional data structures are required to keep track of each element of such data types. So, (i) AT-Opt treats vectors, arrays, and rails, along with all their constituent elements as a single object. Thus read/write to an element (or a sub-partition) is considered as read/write of that object. (ii) Further, each write to an element (or a sub-partition) of data type vectors, arrays, and rails is also considered as a read to those objects. In our experience, we have found that programs rarely access non-distributed array elements. Another possible way to handle this complex problem of partitioning aggregate data is by asking the programmer to specify the sub-regions that are shared; such an approach is used by Legion [7].
(d) **Transient and GlobalRef fields.** We ignore the fields declared transient (not required to be copied) or GlobalRef (no scope to optimize).
(e) **Code generation for substitute object.** During code generation (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 (Fig. 8a). We can further optimize this part by avoiding the creation of new substitute objects (altogether) and replacing the corresponding field dereferences with the temporaries. This optimization can be done only if the object under consideration is not passed to any function call (including as the this argument). Note: a dummy constructor is added for each user-defined type of the input X10 program to assist in creating the substitute objects.
(f) **Code generation for the statement a=b.** During code generation, 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 dereferenced/used in the body of the at-construct; likely dead-code. Hence, we eliminate the statement altogether.
(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.
(h) **Handling of at-expressions by** AT-Opt**.** They are handled in the same way as the at-statements.
(i) **Impact of** AT-Opt**.** Note that if all the fields of the reachable objects are scalars or distributed arrays, or most of the data accessible from shared objects is accessed inside the at-construct then the gains due to AT-Opt compared to X10 compiler would be minimal.

```
public class mst {
 val START = 0;
 val JOIN = 1;
 val CHANGE = 2;
 var adj_graph:Array[Long];
 var nodes:Long;
 val Infinity = Long.MAX_VALUE;
 var nSet:DistArray[node];
 var loadValue:long=0;
 var nval:DistArray[long];
 var R: Region; var D: Dist;
 var cagain:DistArray[boolean];
 var tagain:DistArray[boolean];
 var invagain:DistArray[boolean];
 var setCheck:DistArray[boolean];
 var checkflag:DistArray[boolean];

...
```

```
def setChildSignal():boolean {
 finish {
  for(i in this.D) {
   async at(this.D(i)) { //pco#1
    this.nSet(i).ss=this.nSet(i).tss;
    if(!(this.loadValue.equals(0))) {
     val pt:Long = i(0);
     this.nval(i)=this.weight(this.nval(i)+pt);
 } } } } /* finish */

 var retVal:boolean = false;
 for(i in this.D) {
  //pco#2
  var atVal:boolean=at(D(i)) this.setCheck(i);
  if(atVal) retVal = true;
 }
 return retVal; } /*setChildSignal*/ } /*mst*/
```

Fig. 15. IMSuite kernel: MST - data members and member functions.

(j) **Why flow-sensitive analysis?** AT-Opt uses flow-sensitive analysis to precisely identify object fields dereferenced across remote places. In our running example, for the place-change operation at line 4, the flow-insensitive approach will copy all fields (r1 and r2) of the object pointed by a, though only r1 is required during program execution.

(k) **Profitability of the rules of AT-Pruning.** The transformation rules discussed as part of AT-Pruning are invoked syntactically, and profitability is not explicitly evaluated. That is, a rule of the form $X \Rightarrow Y$ is invoked every time we encounter some code of the form X. Consequently, one may argue that some of the invocations may be redundant and even lead to performance deterioration. For example, if Rule [D] (Fig. 12) is invoked first, but it led to no actual optimization (say, because of issues related to dependencies) then the translation is redundant and in some cases lead to inefficient codes (executing the loop headers twice). However in our evaluation, we found that such cases did not lead to any visible performance deterioration.

(l) **Comparison against prior work.** Fig. 15 shows the code of the MST kernel from IMSuite. For brevity, we have shown all 16 data members and only the setChildSignal method of MST kernel. The setChildSignal method contains two place-change operations (pco#1 and pco#2) to check if any child can start processing in parallel. Fig. 16a and 16b show the optimized MST kernel generated by our proposed AT-Com and Barik et al. [6] work, respectively. For pco#1: AT-Com applies rule [A] of Fig. 12 (AT-Pruning), then AT-Opt precisely identifies the object fields (nset, nval, loadValue, and D) dereferenced at target places and emits the optimized code such that only required fields are copied during program execution. Whereas Barik et al. [6] use a simple scheme of scalar-replacement to reduce the amount of data communicated across places. In their optimized code, the scalar field localValue is saved to t6 and its occurrence are replaced by t6. But, their optimized code will still copy the complete this object because (i) nset and nval are non-scalar fields (Note that Barik et al. [6] scalar-replacement for array access can't be applied on distributed arrays because their elements are owned by target places), and (ii) an access of the this object during the function call this.weight.

For pco#2: AT-Com applies rule [B(i)] of Fig. 12, whereas synchronization-elimination and place-level strip-mining rules are applied by Barik et al. [6]. Though both the approaches execute the place-change operations in parallel, our technique leads to faster code (detailed in Section 5).

```
def setChildSignal():boolean {
 finish {
  for(p in place.Places()) {
   val t1=this.nSet; val t3=this.nval;
   val t2=this.loadValue; val t4=this.D;
   at(p) {
    val t5:mst = new mst();
    t5.nSet=t1; t5.nval=t3;
    t5.loadValue=t2; t5.D=t4;
    for(i in t5.D(p)) { async {
     t5.nSet(i).ss=t5.nSet(i).tss;
     if(!(t5.loadValue.equals(0))) {
      val pt:Long = i(0);
      t5.nval(i)=t5.weight(t5.nval(i)+pt);
 } } } } } } /* finish */

 var retVal:boolean = false;
 val r1=new Rail[boolean](this.D.size());
 finish for(i in this.D) {
  async r1(i)=at(D(i)) this.setCheck(i);
 }
 for(i in this.D) {
  var atVal:boolean=r1(i);
  if(atVal) retVal = true;
 }
 return retVal;
}
```

(a) AT-Com optimized code.

```
def setChildSignal():boolean {
 finish {
  for(i in this.D) {
   val t6 = this.loadValue;
   async at(this.D(i)) {
    this.nSet(i).ss=this.nSet(i).tss;
    if(!(t6.equals(0))) {
     val pt:Long = i(0);
     this.nval(i)=this.weight(this.nval
          (i)+pt);
 } } } } /* finish */

 var retVal:boolean = false;
 for(p in place.Places()) {
  val sd = this.D(p);
  val t7=new Rail[boolean](sd.size());
  finish at(p) async {
   var j:long = 0;
   for(i in sd) {
    val ind=j++;val v=this.setCheck(i);
    at(t7) async t7(ind)=v;
   } }
  var j:long = 0;
  for(i in sd) {
   if(t7(j++))
    retVal = true;
 } } /* for p in place.Places */
 return retVal;
}
```

(b) Code optimized by Barik et al. [6].

Fig. 16. Optimized MST kernel by (a) AT-Com and (b) Barik et al. [6]. The changed code is shown in red color.

(m) **Time and Space complexities:** The worst-case time-complexity of AT-Opt is bound by that of points-to analysis $O(N^3)$; where N is the size of the program. Similarly, the worst-case space-complexity is bound by the size of the points-to graph $O(N^2)$.

In case of AT-Pruning, the number of times the rules [A]-[D] in Fig. 12 can be applied on the input program is $O(N)$, and similarly these rules may add $O(N)$ new statements, which can lead to the invocation of the above rules at most $O(N)$ number of times. Each rule may take $O(N)$ time and hence the worst-case complexity of AT-Pruning is $O(N^2)$; this is assuming that the dependence analysis has been pre-computed and incrementally updated after each rule invocation. AT-Pruning does not use any auxiliary memory for its processing. Alternatively, we can invoke the complete dependence analysis (if the dependencies are not incrementally computed) before rules [B(i)], [C(i)], and [D] to compute dependencies between statements and the complexity of the dependence analysis will have to factored in.

Overall: We find that the time-complexity of AT-Com is $O(N^3)$ and the space-complexity is $O(N^2)$, where N is the program size.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Name	I/P	#at-constructs				serialized-data (GB)				AT-Com % reduction		AT-Opt % reduction		AT-Pruning % reduction	
		Base		AT-Com		Base	AT-Com	AT-Opt	AT-Pruning						
		Stat	Dyn	Stat	Dyn					c-m(I)	c-m(A)	c-m(I)	c-m(A)	c-m(I)	c-m(A)
BF	256	45	6,403	42	4,618	3.00	0.09	0.12	2.13	95.83	51.90	91.94	33.48	19.80	23.18
DST	256	101	15,660	93	7,755	7.39	0.40	0.50	3.53	96.76	64.73	88.58	31.23	35.96	46.07
BY	128	69	550,254	65	539,332	68.08	0.25	0.25	66.73	91.54	28.65	91.14	25.92	1.16	0.02
DR	256	39	489,944	37	489,434	120.07	0.38	0.39	119.70	96.02	40.05	95.80	39.42	0.02	0.39
DS	256	195	542,730	184	514,680	140.09	0.24	0.26	126.37	88.21	53.51	86.98	35.06	4.83	28.54
KC	256	121	84,178	110	72,449	0.32	0.15	0.16	0.30	34.42	22.01	21.97	7.75	22.53	15.99
DP	256	97	68,651	92	57,176	32.18	0.15	0.15	26.55	96.72	49.79	95.60	38.08	9.98	14.07
HS	256	130	400,477	124	7,777	1.01	0.01	0.22	0.01	92.75	90.87	22.05	5.56	91.98	90.85
LCR	256	48	197,639	45	66,824	0.47	0.08	0.09	0.16	86.05	84.46	22.75	8.61	80.61	82.77
MIS	256	68	18,814	62	15,244	8.94	0.13	0.13	7.19	96.67	52.19	88.55	42.78	16.22	21.09
MST	256	254	193,103	238	118,643	76.48	8.32	10.32	35.44	95.95	62.70	78.95	25.90	47.20	41.95
VC	256	86	5,126	80	2,066	2.27	0.10	0.12	0.76	97.07	65.05	86.39	30.68	59.78	37.05

Fig. 17. Characteristics of the IMSuite kernels. Abbreviations: c-m: cache misses; (I) - Intel; (A) - AMD.

8 IMPLEMENTATION AND EVALUATION

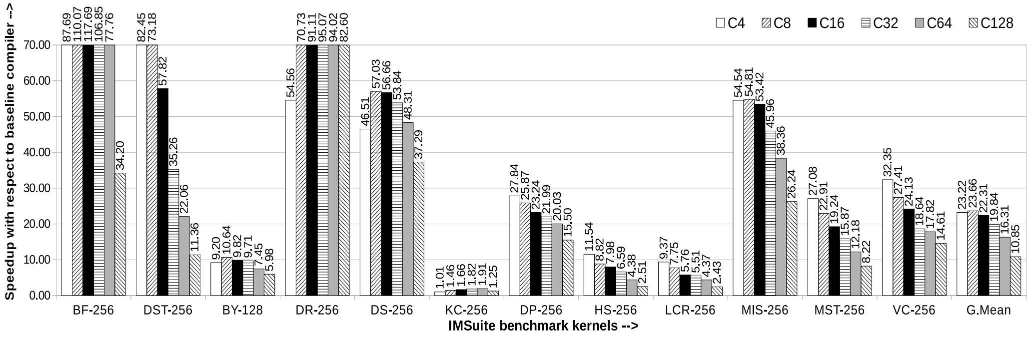
In this section, we evaluate our proposed optimization scheme AT-Com, on two different systems - a four-node Intel system, where each node has two Intel E5-2670 2.6GHz processors, 16 cores per processor, 64GB RAM per node, and 20MB cache-per core; and a two-node AMD system, where each node has an AMD Abu Dhabi 6376 processor, 16 cores per processor, 512GB RAM per node, and 2MB cache per core. We implemented AT-Opt and AT-Pruning, the constituent optimizations of AT-Com, in the x10v2.6.0 compiler x10c++ (C++ backend). Based on the ideas from the insightful paper of Georges et al. [12], we report the execution times by taking a geomean over thirty runs.

We evaluated AT-Com using 12 benchmark kernels from IMSuite [14]: 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 fields at a remote place (say p1), such that the object is not created in the remote place p1.

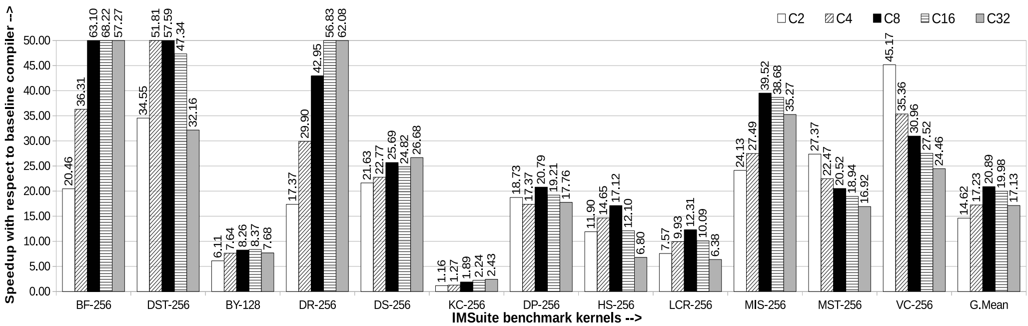
In Fig. 17, columns 2 to 4, show the chosen input sizes, and the number of remote-communications (number of at statements) during both compile-time and run-time, for the chosen input. For these benchmark kernels, we found that the maximum height of Abstract-place-tree was at most three. For all the benchmarks kernels, the chosen input size was the largest input such that on our 32-core Intel system, when the input program, compiled using the default compiler, 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.

8.1 Impact of AT-Com

In this section, we present the evaluation of the overall impact of AT-Com (uses both AT-Opt and AT-Pruning), by comparing against Base - the baseline version without any communication optimizations. In Fig. 17, columns 5 and 6 show the number of static and dynamic counts of the



(a) Intel system speedups; totalCores=128. Speedup=(exec-time using `Base` / exec-time using `AT-Com`).



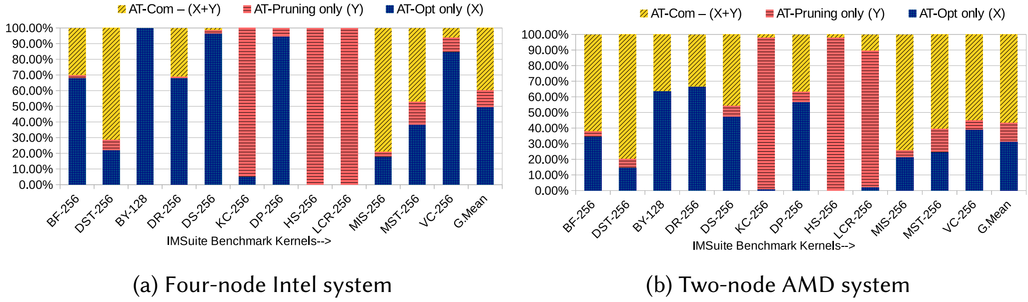
(b) AMD system speedups; totalCores=32. Speedup=(exec-time using `Base` / exec-time using `AT-Com`).

Fig. 18. Speedups for varying number of places (#P) and threads (#T). Config $C_i \equiv$ #P=i and #T=totalCores/i;

`at-constructs` in the `AT-Com` optimized code. It shows that `AT-Com` leads to a significant reduction in the number of static and dynamic `at-constructs`: 6.5% and 26%, respectively. Naturally, these reductions are obtained only from the `AT-Pruning` optimization (`AT-Opt` does not change the number of `at-constructs`). In Fig. 17, columns 7 and 8 report the amount of data (excluding some common meta-data and body of the `at-construct`) serialized during the execution of kernel programs, in the context of `Base` and `AT-Com`, respectively. As it can be seen, compared to `Base` the `AT-Com` optimized code leads to a large reduction in the amount of serialized data (6× to 583×). 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.

Fig. 18a and Fig. 18b show the speedups achieved by using `AT-Com`, on the four-node Intel system and AMD system, respectively, for varying number of places and threads. It can be seen that with respect to `Base`, `AT-Com` optimizer achieved significant speedups: geomean of 18.72× on the Intel system and 17.83× on the AMD system. It can be seen that the speedups for all benchmarks across the Intel and AMD systems are consistent, with the reduction in the number of `at-constructs` and the reduction in the amount of serialized data. For example, for the KC kernel, the relatively lower gains (though significant) is due to the low reduction in the number of dynamic `at-constructs` and the amount of serialized data. The exact speedup may vary, depending on the input program, input, and hardware (including the available cores, memory, cache and so on).

Compared to the rest of the kernels showing very high speedups, the speedups of kernels BY, HS and LCR look lower. This is because for these three kernels, the relative amount of time they spend in computation (compared to the amount of serialized data and number of place-change operations) is much higher (in contrast to kernels like BF, DR, DS, and so on).



(a) Four-node Intel system

(b) Two-node AMD system

Fig. 19. Individual impact of `AT-Opt` (say X) and `AT-Pruning` (say Y), and the combined impact (`AT-Com`–(X+Y)).

8.2 Impact of individual components of `AT-Com`

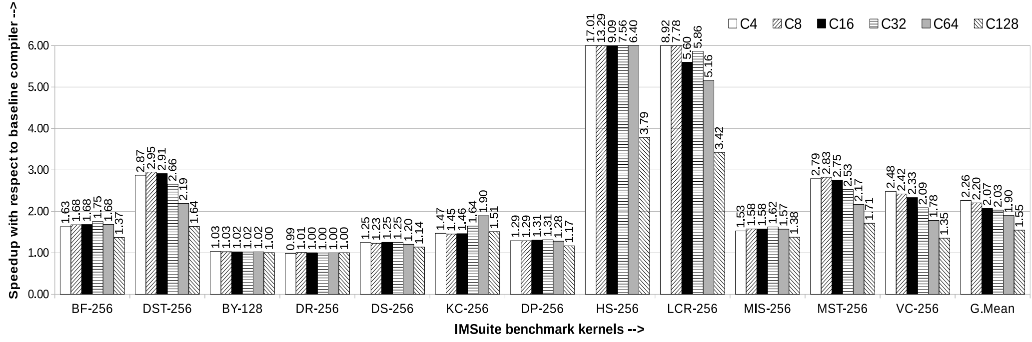
We have also studied the individual impacts of `AT-Opt` and `AT-Pruning` on the overall gains resulted due to `AT-Com`. Fig. 19 shows the individual contributions of `AT-Opt` and `AT-Pruning` to the overall speedups and the additional impact realized by the combination (`AT-Opt`+`AT-Pruning`). On average, we see that `AT-Opt` and `AT-Pruning` individually contribute 40.5% and 11% of the overall gains. The rest of the gains are realized because of the combination `AT-Opt + AT-Pruning`. This combined effect was very high especially for BF, DST, DR, MIS, MST, and VC.

Contributions of `AT-Opt`. In Fig. 17, column 9 reports the amount of data serialized during the execution of kernel programs, compiled by `AT-Opt`. Compared to `Base`, the `AT-Opt` optimized code leads to a reasonable reduction in the amount of serialized data (2× to 527×) and achieved large speedups: geomean of 9.30× on the four-node Intel system and 5.57× on the two-node AMD system; the detailed breakup is skipped for space.

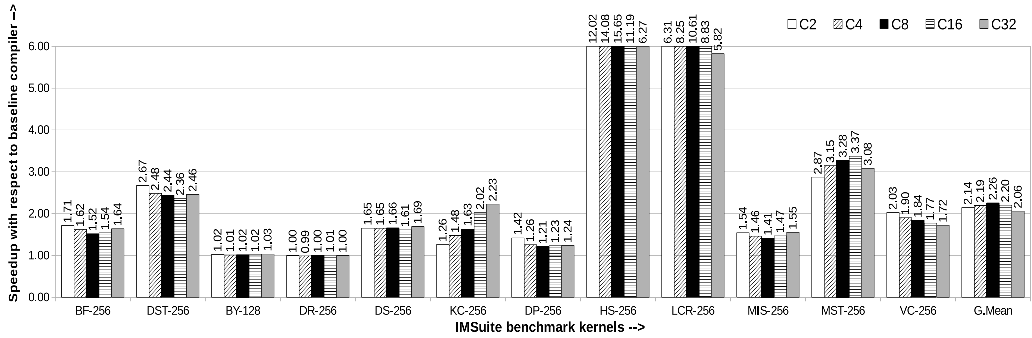
In Fig. 19, it can be seen that except for KC, HS and LCR, the contribution to speedups by `AT-Opt` across the Intel and AMD systems is consistently high. For kernels KC, HS and LCR, the contributions to speedups are not substantial. As discussed in the previous section, this is due to the amount of 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 Fig. 17). 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).

Contributions of `AT-Pruning`. In Fig. 17, columns 5 and 6 report the total number of remote-communications (`place`-calls) made during the execution of these kernel programs, in the context of `AT-Pruning`. As it can be seen, compared to `Base` the `AT-Pruning` transformed code leads to the reduction in remote-communication (place-change operations) across `places` (1.01× to 51.5×) and also results in the reduction in the amount of data serialized (1.01× to 101×; column 10, Fig. 17). Fig. 20a and Fig. 20b show the speedups achieved by using `AT-Pruning`, on the Intel and AMD systems, respectively, for varying number of places and threads. We see that compared to `Base`, the `AT-Pruning` optimizer achieved reasonable speedups: geomean 1.99× (Intel) and 2.17× (AMD).

Figures 19 and 20 show that except for BY and DR the contribution to speedups by `AT-Pruning` across both the systems are reasonable. The speedup depends on the maximum possible number of instances where `AT-Pruning` can be applied. The speedups are directly proportional to the reduction and parallel execution of place-change operations. For kernels BY and DR the speedups are not substantial. This is because in these kernels, the impact of `AT-Pruning` is quite low (columns 5 and 6). For HS and LCR, `AT-Pruning` leads to a large reduction in the place-change operations (> 100K number of place-change operations each). This in turn reduces the amount of serialized data considerably and leads to high speedups. For the rest of the kernels, `AT-Pruning` led to lesser reduction of place-change operations and this is reflected in the resulting speedups.



(a) Intel system speedups; totalCores=128. Speedup=(exec-time using `Base` / exec-time using `AT-Pruning`).



(b) AMD system speedups; totalCores=32. Speedup=(exec-time using `Base` / exec-time using `AT-Pruning`).

Fig. 20. Speedups for varying number of places (#P) and threads (#T). Config $C_i \equiv$ #P=i and #T=totalCores/i;

8.3 Evaluation on a Single-node (shared memory environment) setup

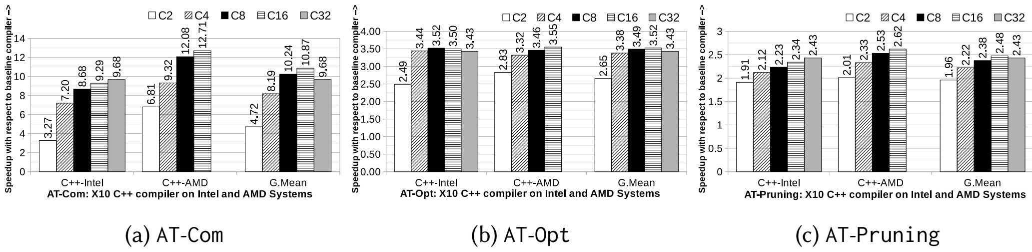
Fig. 21 shows the geomean speedups achieved by `AT-Com` on an Intel (32 cores) and an AMD (16 cores) single node system. We can see that `AT-Com` leads to significantly high speedups, even on a single node system: geomean 7.13× and 9.94× on the Intel and AMD system, respectively. The corresponding numbers for `AT-Opt` and `AT-Pruning` are 3.06×, 3.44× and 2.20×, 2.38×, respectively. Naturally, the speedups on the distributed-memory system are more because of the reduction in inter-node communication.

To understand the sources of the obtained speedups, we studied the cache behavior of the generated codes on the single node Intel and AMD systems. In Fig. 17, columns 11 - 16 report, the geomean % reduction (compared to `Base`) in cache-misses (for `AT-Com` 34-97% on the Intel system and 22-90% on 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: since `AT-Com` reduces the amount of memory usage, which in turn leads to less cache pollution and its impact is more visible on the Intel system with larger cache.

8.4 Impact of the design choices

We now discuss the emperical impact of two main design choices made as part of `AT-Opt`.

Inter-procedural analysis. For the benchmarks under consideration, we found that the inter-procedural component was essential for getting the reported large speedups. This is because, the intra-procedural analysis alone could impact only two benchmarks (BY and DP). Even there, the impacted `at-constructs` were not part of the main computation and hence led to no gains.



(a) AT-Com (b) AT-Opt (c) AT-Pruning

Fig. 21. GeoMean speedups for varying number of places (#P) and threads (#T) for single node Intel (#cores = 32) and AMD (#cores = 16). Config: $C_i \equiv$ #P=i and #T=#cores/i.

Conservative handling of ambiguous-objects. In the chosen kernels that met our selection requirements (Section 8) the impact due to ambiguous objects was non-existent. This is not unexpected, because in HPC codes, 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 Fig. 7 and the discussion thereof).

8.5 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; see Section 7 for a illustrative comparison. 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 were non-scalar fields (violates the requirement to perform scalar replacement [6]). This renders a comparison of AT-Opt against the scalar-replacement scheme of Barik et al. redundant. Similarly, the impact of rule [B(i)] of Fig. 12 and its counterpart rules of Barik et al. (synchronization-elimination and place-level strip-mining) was minimal and thus not detailed.

8.6 Overall summary of the evaluation

We have done a detailed evaluation and found that the actual 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. The factor (1) is impacted by AT-Pruning and factor (2) is the only one that is different between `Base` and AT-Opt optimized codes. However factor (1) impacts factor (2) and the impact of factor (2) can be felt on (4) as well. Since AT-Com reduces the factors (1) and (2) (consequently factor (4)), it leads to significant performance gains.

9 RELATED WORK

There have been many prior works [3, 5, 6, 10] 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, Fig. 7); 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 8).

Besides the scalar-replacement and program transformation techniques, 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-Com first and then these optimizations (that may change the structure of loops/at-construct).

There have been prior works [3, 10] that aim to optimize communication of fine-grain data by eliminating redundant communication, use of split-phase communication and coalescing. Similarly, Hiranandani et al. [16, 17] 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.[18] 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.

Sanz et al. [25] optimize the communication routines and block and cyclic distribution modules in Chapel [8] by performing aggregation for array assignments. Paudel et al. [21] 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.

Sharma et al. [27] perform affine loop optimization using modulo unrolling in Chapel [8] to access elements of array distributed across different locale (equivalent to places in X10). They propose a technique named zipper iteration which uses modulo unrolling to unroll loops, then gathers array elements from remote locale to a local buffer via one bulk get message and assign it to the current locale. Our rule [B(i)] of Fig. 12 can be used on top of it to further improve the performance. Similarly, Pellegrini et al. [22] perform precise data dependence analysis with the help of polyhedral model to restore the compiler analysis techniques to remove the communication overlap in MPI programs.

There have been many works on points-to and shape analysis [4, 11, 13, 23, 24, 29]. Chandra et al. [9] 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 places and their accessed objects.

10 CONCLUSION

In this paper, we present a novel scheme AT-Com to reduce the communication overheads by paying close attention to objects getting copied across *places* and number of place-change operations in X10 programs. We have implemented AT-Opt and AT-Pruning, the constituent optimizations of AT-Com, in the x10v2.6.0 compiler and evaluated the performance on two different systems (a four-node × 32-core Intel system and a two-node × 16-core AMD system). We show that AT-Com leads to significant gains in execution time with speedups of 18.72× and 17.83× on the Intel and AMD systems, respectively. Additionally, the experimental results show that the AT-Com optimized programs scale better than the baseline versions. Though we discussed AT-Com in the context of X10, we believe that it can be applied to other PGAS languages like Chapel, HJ, and so on.

ACKNOWLEDGMENTS

The work has been partially supported by the SERB core research grant numbered CRG/2018/002488.

REFERENCES

[1] S. Agarwal, R. Barik, V. K. Nandivada, R. K. Shyamasundar, and P. Varma. Static Detection of Place Locality and Elimination of Runtime Checks. In G. Ramalingam, editor, *APLAS*, pages 53–74, San Francisco, CA, USA, 2008.

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

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

[4] L. O. Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, University of Cophenhagen, 1994.

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

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

[7] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *SC*, pages 1–11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[8] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language . *Int. J. High Perform. Comput. Appl.*, 21:291–312, Aug 2007.

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

[10] W Chen, C Iancu, and K Yelick. Communication Optimizations for Fine-Grained UPC Applications. In *PACT*, pages 267–278, 2005.

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

[12] A Georges, D Buytaert, and L Eeckhout. Statistically Rigorous Java Performance Evaluation. In *OOPSLA*, pages 57–76, Oct, 2007.

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

[14] S. Gupta and V. K. Nandivada. IMSuite: A benchmark suite for simulating distributed algorithms. *Journal of Parallel and Distributed Computing*, 75:1 – 19, 2015.

[15] Habanero. Habanero Java. http://habanero.rice.edu/hj, Dec 2009.

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

[17] S Hiranandani, K Kennedy, and C Tseng. Compiler Optimizations for Fortran D on MIMD Distributed-memory Machines. In *SC*, pages 86–100, November, 1991.

[18] M Kandemir, P Banerjee, A Choudhary, J Ramanujam, and N Shenoy. A Global Communication Optimization Technique Based on Data-flow Analysis and Linear Algebra. *ACM Trans. Program. Lang. Syst.*, 21:1251–1297, Nov 1999.

[19] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., USA, 1997.

[20] V K Nandivada, J Shirako, J Zhao, and V Sarkar. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst.*, 35:1–48, 2013.

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

[22] S Pellegrini, T Hoefler, and T Fahringer. Exact Dependence Analysis for Increased Communication Overlap . In J L Träff, S Benkner, and J J Dongarra, editors, *Recent Advances in the Message Passing Interface*, pages 89–99, 2012.

[23] N. Rinetzky and S. Sagiv. Interprocedural Shape Analysis for Recursive Programs . In *CC*, pages 133–149, 2001.

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

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

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

[27] A. Sharma, D. Smith, J. Koehler, R. Barua, and M. Ferguson. Affine Loop Optimization Based on Modulo Unrolling in Chapel . In *PGAS*, pages 1–12, 2014.

[28] A. Thangamani and V. K. Nandivada. Optimizing Remote Data Transfers in X10 . In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018*, pages 1–15, Nov, 2018.

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