Abstraction Analysis and Certified Flow and Context Sensitive Points-to Relation for Distributed Programs

Mohamed A. El-Zawawy

College of Computer and Information Sciences, Al-Imam M. I.-S. I. University
Riyadh, Kingdom of Saudi Arabia
and
Department of Mathematics, Faculty of Science, Cairo University
Giza 12613, Egypt
maelzawawy@cu.edu.eg

Abstract. This paper presents a new technique for pointer analysis of distributed programs executed on parallel machines with hierarchical memories. One motivation for this research is the languages whose global address space is partitioned. Examples of these languages are Fortress, X10, Titanium, Co-Array Fortran, UPC, and Chapel. These languages allow programmers to adjust threads and data layout and to write to and read from memories of other threads.

The techniques presented in this paper have the form of type systems which are simply structured. The proposed technique is shown on a language which is the *while* langauge enriched with basic commands for pointer manipulations and also enriched with commands vital for distributed execution of programs. An abstraction analysis that for a given statement calculates the set of function abstractions that the statement may evaluate-to is introduced in this paper. The abstraction analysis is needed in the proposed pointer analysis. The mathematical soundness of all techniques presented in this paper are discussed. The soundness is proved against a new operational semantics presented in this paper.

Our work has two advantages over related work. In our technique, each analysis result is associated with a correctness proof in the form of type derivation. The hierarchical memory model used in this paper is in line with the hierarchical character of concurrent parallel computers.

Keywords: abstraction analysis, certified code, flow and context sensitive points-to relation, distributed programs, semantics of programming languages, operational semantics.

1 Introduction

The need for distributed programs [20,21] for graphics processors and desktops was magnified by the creation of multi-core processors which therefore greatly affected the software development. Large parallel systems were built using multi-core processors. These systems have memory that uses message passing and

B. Murgante et al. (Eds.): ICCSA 2012, Part IV, LNCS 7336, pp. 83–99, 2012.

that is cache-coherent shared, direct-access (DMA or RDMA) distributed, and hierarchical [17,18]. An appropriate address space model for machines equipped with multi-core processors is the partitioned global model (PGAS). Examples of DPLs, distributed programming languages that program machines equipped with multi-core processors, that use PGAS are Unified Parallel C (UPC), X10, Chapel, and Titanium which is based on Java.

The objective of pointer analysis [10,8,13,9] is to calculate for every variable at each program point of the program the set of addresses to which the variable may point. Pointer analysis of DPLs is a complicated problem as these languages allow pointers to shared states. A common query concerning pointer analysis of DPLs is whether the access of a given pointer can be proven to be restricted to a specific region of the memory hierarchy. Such information is useful in many directions including the following: (a) pointer representation – fewer bits are needed to represent pointers with restricted domains; (b) improving performance of software caching systems – coherence protocols may be limited to a proper subset of processors; (c) allowing data race to ignore pointers that access private data of a thread; (d) identifying pointers that access data on chip multiprocessor and hence need ordering fences [23].

The algorithmic style is the typical manner to analyze distributed program. This manner relies on data-flow analyses and works on control-flow graphs – intermediate representations of programs. The type-systems manner is an alternative style for analyzing distributing programs and it works on the syntactical structure of programs [10,8,13,9]. This latter manner is the convenient approach to analyze distributed programs [2,16,27] when justifications for the correctness of analysis results are required to be delivered together with analysis results. These communications are required to clarify the way the analysis results were obtained. Certified code is an application that requires machine-checkable, simple, and deliverable justifications. Justifications provided by the type-systems manner have the form of type derivations which are user-friendly. The type systems approach has the advantage of relatively simple inference rules.

In previous work [10,8,13,9,11,6], we have shown that the type-systems approach is indeed a convenient framework for achieving pointer analysis of *while* languages enriched with pointer and parallel constructs. In this paper, we show that this approach extends also to the complex problem of pointer analysis of distributed programs. Many factors cause the complexity of the problem; (a) the use of shared memory in distributed programs allows pointing to these locations and (b) the interference between shared memory concept also complicates the problem when the studied model langauge contains important real-constructs like functions.

This paper presents a new approach for pointer analysis of distributed programs executed on hierarchical memories. The proposed technique has the form of a type system that is simply structured. The presented method is shown on a toy language which is the *while* langauge enriched with basic commands for pointer manipulations and also enriched with commands vital for distributed execution of programs. The execution model adopted in this paper is the

single program multiply data (*SPMD*) model. In this model the same program is executed on different machines storing different data. Although the language used is simple, it is powerful enough to study distributing and pointer concepts. An abstraction analysis that for a given statement calculates the set of function abstractions that the statement may evaluate to is introduced in this paper. The abstraction analysis is needed in the proposed pointer analysis. The mathematical soundness of all techniques presented in this paper are discussed. The soundness is proved against a new operational semantics presented in this paper.

Motivation

Figure 1 presents a motivating example of our work. The program of the figure consists of two parts; the first part (lines 1 and 2) introduces definitions for statements S_1 and S_2, and the second part (lines 3, 4, 5 and 6) is the code that uses these definitions. We suppose that the program is executed on a distributed system that has two machines labeled m_1 and m_2. We also assume that each of the machines has local registers (say x and y) and has local addresses (say $\{a, b\}$ for m_1 and $\{c, d\}$ for m_2). Therefore the set of global variables, $gVar$, is $\{(m_1, x), (m_1, y), (m_2, x), (m_2, y)\}$ and the set of global addresses, $gAdrrs$, is $\{(1, m_1, a), (1, m_1, b), (2, m_1, a), (2, m_1, b), (1, m_2, a), (1, m_2, b), (2, m_2, a), (2, m_2, b)\}$. Loc denotes the union set of global variables and addresses. We consider the parallelism mode *SPMD*. Our paper presents a new technique for analyzing the pointer content of programs like this one. The proposed technique has the advantage, over any existing technique, of associating each analysis result with a correctness proof. The analysis results of the example program are shown in Figure 2 and Figure 3 for machines m_1 and m_2, respectively.

1. $S_1 = x;$
2. $S_2 = y;$
3. $x := new_1;$
4. $y := new_2;$
5. $y := transmit S_1 from 2;$
6. $x := convert(S_2, 2);$

Fig. 1. A motivating example

Contributions

Contributions of this paper are the following:

1. A new abstraction analysis that calculates the set of abstractions that a statement may evaluate-to.
2. A new pointer analysis technique, that is context and flow-sensitive, for distributed programs.

Program point	Pointer information for m_1
the first point	$\{l \mapsto \emptyset \mid l \in Loc\}$
the point between 3 & 4	$\{(m_1, x) \mapsto \{(1, \{m_1\})\}, l \mapsto \emptyset \mid l \neq (m_1, x)\}$
the point between 4 & 5	$\{(m_1, x) \mapsto \{(1, \{m_1\})\}, (m_1, y) \mapsto \{(2, \{m_1\})\},$ $l \mapsto \emptyset \mid l \notin \{(m_1, x), (m_1, y)\}\}$
the point between 5 & 6	$\{(m_1, x) \mapsto \{(1, \{m_1\})\}, (m_1, y) \mapsto \{(2, \{m_1\}), (1, \{m_2\})\},$ $l \mapsto \emptyset \mid l \notin \{(m_1, x), (m_1, y)\}\}$
the last point	$\{(m_1, x) \mapsto \{(1, \{m_1, m_2\})\}, (m_1, y) \mapsto \{(2, \{m_1\}), (1, \{m_2\})\},$ $l \mapsto \emptyset \mid l \notin \{(m_1, x), (m_1, y)\}\}$

Fig. 2. Results of pointer analysis of the program in Figure 1 on the machine m_1

Program point	Pointer information for m_2
the first point	$\{l \mapsto \emptyset \mid l \in Loc\}$
the point between 3 & 4	$\{(m_2, x) \mapsto \{(1, \{m_2\})\}, l \mapsto \emptyset \mid l \neq (m_2, x)\}$
the point between 4 & 5	$\{(m_2, x) \mapsto \{(1, \{m_2\})\}, (m_2, y) \mapsto \{(2, \{m_2\})\},$ $l \mapsto \emptyset \mid l \notin \{(m_2, x), (m_2, y)\}\}$
the point between 5 & 6	$\{(m_2, x) \mapsto \{(1, \{m_2\})\}, (m_2, y) \mapsto \{(2, \{m_2\}), (1, \{m_2\})\},$ $l \mapsto \emptyset \mid l \notin \{(m_2, x), (m_2, y)\}\}$
the last point	$\{(m_2, x) \mapsto \{(1, \{m_1, m_2\})\}, (m_1, y) \mapsto \{(2, \{m_1\}), (1, \{m_2\})\},$ $l \mapsto \emptyset \mid l \notin \{(m_2, x), (m_2, y)\}\}$

Fig. 3. Results of pointer analysis of the program in Figure 1 on the machine m_2

3. A context insensitive variation for our main context and flow-sensitive pointer analysis.
4. A new operational-semantics for distributed programs on a hierarchy memory model.

Organization

The rest of the paper is organized as follows. Section 2 presents our hierarchy memory model. The language that we use to present our analyses and an operational semantics to its constructs are also presented in Section 2. Our main analyses are introduced in Section 3. These analyses include an abstraction analysis, a pointer analysis that is context and flow-sensitive for distributed programs, and a context insensitive variation of the previous analysis. Related and future work are briefed in Section 4.

2 Memory Model, Language, and Operational Semantics

Memory models of parallel computers are typically hierarchical [20,21]. In these models each process is often assigned local stores. Caches and local addresses are examples of hierarchies inside processors. The Cell game processor provides

an example where each SPE processor is assigned a local store that is accessible by other SPEs via operations for memory move. It is possible to increase levels of partitions via partitioning a computational grid memory into clusters which in turn can be partitioned into nodes (Figure 4).

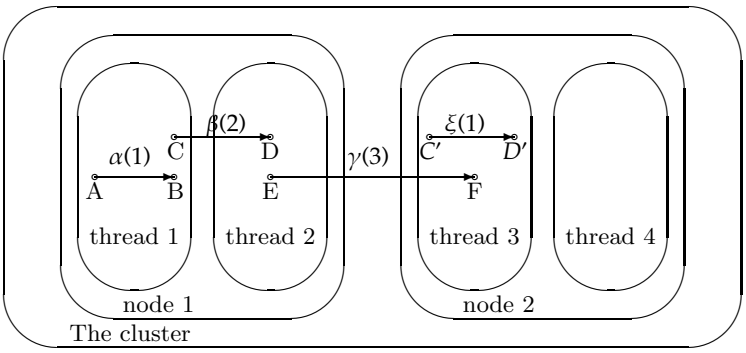

Fig. 4. Grid memory

A memory of two-levels hierarchy is a typical choice for most PGAS [2,16,27]. The two levels are a local one devoted to a particular thread and a shared one serving all threads. The basic idea in PGAS languages is to assign each pointer the memory-hierarchy level that the pointer may reference. The memory model in Figure 4 is a three-levels hierarchy (a cluster, a node, and a thread). For example in Figure 4, pointers α and ξ are thread pointers and they can reference addresses on thread 1 and 4, respectively. However pointers β and γ are node and cluster pointers, respectively. While β can reference addresses on node 1, γ can reference addresses on the cluster. Such domains of pointers can be represented by assigning each pointer a number (width) like edge labels in Figure 4. Clearly the higher the width of the pointer, the more expensive to manipulate the pointer. The manipulation costs of pointers include representation costs, access costs, and dereference costs. The direction in hardware research is to increase hierarchy levels. Therefore it is extremely important for software to benefit from the hierarchy [20,21,2].

We use a simple language [17,18], named $While_d$, to present the results of this paper. $While_d$ is the well-known *while* language enriched with pointer, parallelism, and function constructions [20,21]. The model of parallelism used in $While_d$ is SPMD which means that the same code is executed on all machines. We assume that h denotes the hierarchy height of memory. Consequently, widths of pointers are in the interval [1,h]. Figure 5 presents the syntax of $While_d$. The language uses a fixed set of variables, *lVar*, each of which is machine-private. According to the $While_d$ syntax, each program consists of a sequence of definitions followed by a statement ($Defs : S$). The first part of program, *Defs*, is assignments of names to statements. These names can be used in S, the

$$
\begin{aligned}
\mathit{name} &::= \text{'string of characters'}.\\
S \in \mathit{Stmts} &::= n \mid \mathit{true} \mid \mathit{false} \mid x \mid S_1\ i_{op}\ S_2 \mid S_1\ b_{op}\ S_2 \mid *S \mid \mathit{skip} \mid \mathit{name} \mid x := S \mid S_1 \leftarrow S_2 \mid\\
&\quad S_1; S_2 \mid \mathit{if}\ S\ \mathit{then}\ S_t\ \mathit{else}\ S_f \mid \mathit{while}\ S\ \mathit{do}\ S_t \mid \lambda x.S \mid S_1 S_2 \mid \mathit{letrec}\ x = S\ \mathit{in}\ S' \mid\\
&\quad \mathit{new}_l \mid \mathit{convert}\ (S, n) \mid \mathit{transmit}\ S_1\ \mathit{from}\ S_2.\\
\mathit{Defs} &::= (\mathit{name} = S); \mathit{Defs} \mid \varepsilon.\\
\mathit{Program} &::= \mathit{Defs} : S.
\end{aligned}
$$

where
$x \in lVar$, an infinite set of variables, $n \in \mathbb{Z}$ (integers), $i_{op} \in \mathbb{I}_{op}$ (integer-valued binary operations), and $b_{op} \in \mathbb{B}_{op}$ (Boolean-valued binary operations).

Fig. 5. The programming language $while_d$

second part of program. For each program of the langauge $While_d$, our semantics assigns a function fd of the domain *Function-defs*:

$$fd \in \mathit{Function\text{-}defs} = \text{'strings of characters'} \to \mathit{Stmts}.$$

The map fd is supposed to link each name with its definition. The map fd is built using the following inference rules:

$$\frac{}{\varepsilon : fd \rightsquigarrow fd}\ (fd_1) \qquad \frac{\mathit{Defs} : fd[\mathit{name} \mapsto S] \rightsquigarrow fd'}{(\mathit{name} = S); \mathit{Defs} : fd \rightsquigarrow fd'}\ (fd_2)$$

Therefore for a program $Defs : S$, we calculate $Defs : \emptyset \rightsquigarrow fd$ using the above inference rules to construct fd.

We define the meaning of $While_d$ statements by their operational semantics, i .e., by introducing transition relations $\rightsquigarrow_m$; between pairs of statements and states and pairs of values and states. Different components used in the inference rules of the semantics are introduced in the following definition.

Definition 1. *1. The set of global variables, denotes by gVar, is defined as* $gVar = \{(m, x) \mid m \in M, x \in lVar\}$.
2. The set of global addresses, denotes by gAddrs, is defined as $gAddrs = \{g = (l, m, a) \mid l \in L, m \in M, a \in lAddrs\}$.
3. $loc \in Loc = gAddrs \cup gVar$.
4. $v \in Values = \mathbb{Z} \cup gAddrs \cup \{true, false\} \cup \{\lambda x.S \mid S \in Stmt\}$.
5. $\delta \in States = Loc \longrightarrow Values$.

The symbols M, W, and lAddrs denote the set of machines labels (integers), the set of width $\{1, \ldots, h\}$*, and the set of local addresses located on each single machine, respectively. The set of labels of allocation sites is denoted by L.*

The semantics produces judgments of the form $(S, \delta) \rightsquigarrow_m (v, \delta')$. The judgement means that executing the statement S on the machine m and in the state δ results

in the value v and modifying the state δ to become δ'. The notation $\delta[x \mapsto v]$ denotes the map $\lambda y.$ if $y = x$ then v else $\delta(y)$.

The inference rules of our semantics are as follows:

$$(n,\delta) \rightsquigarrow_m (n,\delta) \qquad (true,\delta) \rightsquigarrow_m (true,\delta) \qquad (false,\delta) \rightsquigarrow_m (false,\delta) \qquad (x,\delta) \rightsquigarrow_m (\delta(x),\delta)$$

$$(\lambda x.S,\delta) \rightsquigarrow_m (\lambda x.S,\delta)\,(abs) \qquad \frac{(S_1,\delta) \rightsquigarrow_m (n_1,\delta'') \quad (S_2,\delta'') \rightsquigarrow_m (n_2,\delta')}{(S_1\ i_{op}\ S_2,\delta) \rightsquigarrow_m \begin{cases}(n_1\ i_{op}\ n_2,\delta'), & n_1\ i_{op}\ n_2 \in \mathbb{Z};\\ abort, & \text{otherwise.}\end{cases}}\,(int\text{-}stmt)$$

$$\frac{(S_1,\delta) \rightsquigarrow_m (b_1,\delta'') \quad (S_2,\delta'') \rightsquigarrow_m (b_2,\delta')}{(S_1\ b_{op}\ S_2,\delta) \rightsquigarrow_m \begin{cases}(b_1\ b_{op}\ b_2,\delta'), & b_1\ b_{op}\ b_2 \text{ is a Boolean value;}\\ abort, & \text{otherwise.}\end{cases}}\,(bo\text{-}stmt)$$

$$\frac{(S,\delta) \rightsquigarrow_m (g,\delta')}{(*S,\delta) \rightsquigarrow_m \begin{cases}(\delta'(g),\delta'), & g \in gAddrs;\\ abort, & \text{otherwise.}\end{cases}}\,(de\text{-}ref) \qquad (skip,\delta) \rightsquigarrow_m (0,\delta)$$

$$\frac{(S,\delta) \rightsquigarrow_m abort}{(x := S,\delta) \rightsquigarrow_m abort} \qquad \frac{(S,\delta) \rightsquigarrow_m (v,\delta')}{(x := S,\delta) \rightsquigarrow_m (v,\delta'[x \mapsto v])} \qquad \frac{(S_1,\delta) \rightsquigarrow_m abort}{(S_1;S_2,\delta) \rightsquigarrow_m abort}$$

$$\frac{(S_1,\delta) \rightsquigarrow_m abort \text{ or for } v \notin gAddrs.\ (S_1,\delta) \rightsquigarrow_m (v,\delta'')}{(S_1 \leftarrow S_2,\delta) \rightsquigarrow_m abort}\,(\leftarrow_1)$$

$$\frac{(S_1,\delta) \rightsquigarrow_m (v,\delta'') \quad (S_2,\delta'') \rightsquigarrow_m abort}{(S_1 \leftarrow S_2,\delta) \rightsquigarrow_m abort}\,(\leftarrow_2)$$

$$\frac{(S_1,\delta) \rightsquigarrow_m (g,\delta'') \quad (S_2,\delta'') \rightsquigarrow_m (v,\delta''') \quad g \in gAddrs}{(S_1 \leftarrow S_2,\delta) \rightsquigarrow_m (v,\delta'''[g \mapsto v])}\,(\leftarrow_3) \qquad \frac{(S_1,\delta) \rightsquigarrow_m abort}{(S_1;S_2,\delta) \rightsquigarrow_m abort}$$

$$\frac{(S_1,\delta) \rightsquigarrow_m (v_1,\delta'') \quad (S_2,\delta'') \rightsquigarrow_m (v_2,\delta')}{(S_1;S_2,\delta) \rightsquigarrow_m (v_2,\delta')} \qquad \frac{(S_1,\delta) \rightsquigarrow_m (v_1,\delta'') \quad (S_2,\delta'') \rightsquigarrow_m abort}{(S_1;S_2,\delta) \rightsquigarrow_m abort}$$

$$\frac{(S,\delta) \rightsquigarrow_m (true,\delta'') \quad (S_t,\delta'') \rightsquigarrow_m abort}{(if\ S\ then\ S_t\ else\ S_f,\delta) \rightsquigarrow_m abort} \qquad \frac{(S,\delta) \rightsquigarrow_m (true,\delta'') \quad (S_t,\delta'') \rightsquigarrow_m (v,\delta')}{(if\ S\ then\ S_t\ else\ S_f,\delta) \rightsquigarrow_m (v,\delta')}$$

$$\frac{(S,\delta) \rightsquigarrow_m (false,\delta'') \quad (S_f,\delta'') \rightsquigarrow_m abort}{(if\ S\ then\ S_t\ else\ S_f,\delta) \rightsquigarrow_m abort} \qquad \frac{(S,\delta) \rightsquigarrow_m (false,\delta'') \quad (S_f,\delta'') \rightsquigarrow_m (v,\delta')}{(if\ S\ then\ S_t\ else\ S_f,\delta) \rightsquigarrow_m (v,\delta')}$$

$$\frac{(S,\delta) \rightsquigarrow_m abort}{(if\ S\ then\ S_t\ else\ S_f,\delta) \rightsquigarrow_m abort} \qquad \frac{(S,\delta) \rightsquigarrow_m abort}{(while\ S\ do\ S_t,\delta) \rightsquigarrow_m abort}$$

$$\frac{(S,\delta) \rightsquigarrow_m (false,\delta'')}{(while\ S\ do\ S_t,\delta) \rightsquigarrow_m (skip,\delta)} \qquad \frac{(S,\delta) \rightsquigarrow_m (true,\delta'') \quad (S_t,\delta'') \rightsquigarrow_m abort}{(while\ S\ do\ S_t,\delta) \rightsquigarrow_m abort}$$

$$\frac{(S,\delta) \rightsquigarrow_m (true,\delta'') \quad (S_t,\delta'') \rightsquigarrow_m (v'',\delta'') \quad (while\ S\ do\ S_t,\delta'') \rightsquigarrow_m (v',\delta')}{while\ S\ do\ S_t : (\delta,p) \rightsquigarrow_m (v',\delta')}$$

$$\frac{(S,\delta) \rightsquigarrow_m (true,\delta'') \quad (S_t,\delta'') \rightsquigarrow_m (v'',\delta'') \quad (while\ S\ do\ S_t,\delta'') \rightsquigarrow_m abort}{(while\ S\ do\ S_t,\delta) \rightsquigarrow_m abort}$$

$$\frac{(S_1,\delta) \rightsquigarrow_m (\lambda x.S_1',\delta'') \quad (S_1'[S_2/x],\delta'') \rightsquigarrow_m (v,\delta')}{(S_1S_2,\delta) \rightsquigarrow_m (v,\delta')}\,(appl)$$

$$\frac{(S,\delta) \rightsquigarrow_m (v,\delta'') \quad (S'[v/x],\delta'') \rightsquigarrow_m (v',\delta')}{(letrec\ x = S\ in\ S',\delta) \rightsquigarrow_m (v',\delta')}\ (letrec) \qquad \frac{a \in lAddrs \quad a \text{ is fresh on } m}{(new_l,\delta) \rightsquigarrow_m ((l,m,a),\delta[(l,m,a) \mapsto null])}$$

$$\frac{(S,\delta) \rightsquigarrow_m (g = (l,m',a),\delta') \quad \text{hdist}(m,m') \leq n}{(convert(S,n),\delta) \rightsquigarrow_m (g,\delta')}\ (conv) \qquad \frac{(fd(name),\delta) \rightsquigarrow_m v,\delta')}{(name,\delta) \rightsquigarrow_m (v,\delta')}\ (name)$$

$$\frac{(S_2,\delta) \rightsquigarrow_m (m',\delta'') \quad m' \in M \quad (S_1,\delta'') \rightsquigarrow_{m'} (v,\delta')}{(transmit\ S_1\ from\ S_2,\delta) \rightsquigarrow_m (v,\delta')}\ (trans)$$

Some comments on the inference rules are in order. The rules for integer and Boolean statements (*(int-stmt)* and *(bo-stmt)*) and the rule for abstraction are trivial. The rule *(de-ref)* makes sure that the value being dereferenced is indeed a global address. The rules $(\leftarrow_1), (\leftarrow_2)$, and $(\leftarrow_3)$ treat the assignment through references. The rules make sure that S_1 is computable and it is a global addresses, otherwise the execution aborts. Every allocation site is assigned a label (the subscription l of new_l). Theses labels simplify the problem of pointer analysis. The allocation statement allocates a fresh local address on the machine m and initializes the address to *null*. The rule $(conv)$ makes it clear that the statement $convert(e,n)$ changes pointer widths. A function, hdist, is used in the rule $(conv)$ to calculate the distance between machines. Inventing such a function is easy. According to this rule, the change of the width is only allowed if the distance between machines is within the provided range, n. The rule $(trans)$ clarifies that the statement *transmit* S_1 *from* S_2 amounts to evaluating the statement S_1 on the machine S_2 and then sending the value to other machines.

Function abstractions and applications are treated in rules (abs) and $(appl)$, respectively. The rule $(appl)$ asks for S_1 to evaluate to an abstraction, say $\lambda x.S_1'$. The value of the application is then the result of substituting v (value of S_2) for x in S_1'. Clearly, the rule applies *call by value* rather than *call by name*. The formal semantics of *letrec* statement amounts to the application specified in the rule $(letrec)$. The intuition of this statement is that it is a well known tool for expressing function recursion.

$$\frac{\theta : \{1,2,\ldots,|M|\} \to M \quad (S,\delta) \rightsquigarrow_{\theta(1)} (v_1,\delta_1) \rightsquigarrow_{\theta(2)} (v_2,\delta_2) \rightsquigarrow_{\theta(3)} \ldots \rightsquigarrow_{\theta(|M|)} (v_{|M|},\delta_{|M|})}{(Defs : S,\delta) \rightsquigarrow_M (v_{|M|},\delta_{|M|})}\ (main\text{-}sem)$$

The rule $(main\text{-}sem)$ provides the semantics of running the program $Defs : S$ on our distributed systems. This rules gives an approximal simulation for executing the program $Defs : S$ using the parallelism model SPMD.

3 Pointer Analysis

This section presents a pointer analysis [10,8,13,9] for the langauge $while_d$. The proposed technique is both flow and context-sensitive. An adaptation of our technique, towards a flow-sensitive and context-insensitive technique, is also presented. The analysis is presented first for single machines (of set M) and then

a rule that joins results of different machines is presented. The analysis has the form of a type system that assigns to each program point a type capturing points-to information. Types assigned to program points are constructed in a post-type derivation process that starts with the empty set as the type for the program's first point. The derivation of calculating the post type serves as a correctness proof for the analysis results. Such proofs are required in applications like certified code or proof-carrying code. These proofs make the results of analysis certified. Therefore in these applications each pointer analysis is assigned with a proof (type derivation) for the correctness of the analysis results [23].

Now we give intuition of Definition 2. Towards abstracting concrete memory addresses, the concept of abstract address is introduced in the following definition. An abstract address is a pair of an allocation site and a set of machines (subset of M). An order relation is defined on the set of abstract addresses. By this relation, an abstract location a_1 is included in another abstract location a_2 if the two addresses have the same location component and the machines set of a_1 is included in that of a_2. A set of abstract addresses is *compact* if it does not contain two distinct addresses with the same location. The set of all *compact* subsets of $Addrs_a$ is denoted by $\mathcal{C}$ (Definition 2.3). A Hoare ordering style on the set $\mathcal{C}$ is introduced in Definition 2.4. Types, *Pointer-types*, of our proposed type system for pointer analysis have the form of maps from $Addrs_a \cup lVar$ to $\mathcal{C}$. A point-wise ordering on the set *Pointer-types* is presented in Definition 2.6.

For each of the statements assigned names in the *Defs* part of programs built in the language $while_d$, the function fe (Definition 2.7) stores a pointer effect (type). These pointer effects are calculated by rules (fe_1) and (fe_2) introduced below. The pointer effects capture pointers that executing a statement may introduce to an empty memory. Not far from the reader expectations, pointer effects are important for studying context-sensitive pointer analysis of our langauge.

A concrete global address $g = (l, m, a)$ is abstracted by an abstract address (l', ms), denoted by $g = (l, m, a) \models (l', ms)$, if $l = l'$ and $m \in ms$. A concrete state δ is of a pointer type p, denoted by $\delta \models p$, if for every $x \in dom(\delta)$, then if $\delta(x)$ is a global address then $\delta(x)$ is abstracted by an abstract address in $p(x)$.

Definition 2. *1.* $Addrs_a = L \times \mathcal{P}(M)$.

2. $\forall (l_1, ms_1), (l_2, ms_2) \in Addrs_a.\ (l_1, ms_1) \leq (l_2, ms_2) \overset{\text{def}}{\Longleftrightarrow} l_1 = l_2$ *and* $ms_1 \subseteq ms_2$.

3. $\mathcal{C} = \{S \in \mathcal{P}(Addrs_a) \mid (l_1, ms_1), (l_2, ms_2) \in S \Longrightarrow l_1 \neq l_2\}$.

4. $\forall S_1, S_2 \in \mathcal{C}.\ S_1 \ll S_2 \overset{\text{def}}{\Longleftrightarrow} \forall x \in S_1.\ \exists y \in S_2.\ x \leq y$.

5. $p \in$ *Pointer-types* $= Addrs_a \cup lVar \to \mathcal{C}$.

6. $\forall p_1, p_2 \in$ *Pointer-types*. $p_1 \sqsubseteq p_2 \overset{\text{def}}{\Longleftrightarrow} \forall x \in dom(p_1).\ p_1(x) \ll p_2(x)$.

7. $fe \in$ *Function-effects* $=$ *'strings of characters'* $\to$ *Pointer-types.*

Definition 3. *The concretization and abstraction maps between concrete and abstract addresses are defined as follow:*

$$Con : Addrs_a \to \mathcal{P}(gAddrs) : (l, ms) \mapsto \{(l, m, a) \in gAddrs \mid m \in ms\}.$$

$$Abst : gAddrs \to Addrs_a : (l, m, a) \mapsto (l, \{m\}).$$

Definition 4. – *A concrete address* $g = (l, m, a)$ *is said to be abstracted by an abstract address* $a = (l', ms)$*, denoted by* $g \models a$*, iff* $l = l'$ *and* $m \in ms$*.*
– *A concrete state* δ *is said to be of type* p*, denoted by* $\delta \models p$*, iff*
- $\forall x \in lVar.\ \delta(x) \in gAddrs \Longrightarrow \exists\ (l, ms) \in p(x).\ \delta(x) \models (l, ms)$*, and*
- $\forall g \in gAddrs.\ \delta(g) \in gAddrs \Longrightarrow ((\forall a.\ g \models a).\ \exists\ a' \in p(a)).\ \delta(g) \models a'$*.*

It is not hard to see that the set of pointer types form a complete lattice. Calculating joins in this lattice is an ease exercise which we leave for the interested reader to do.

3.1 Abstraction Analysis

An analysis that for a given statement calculates the set of abstractions that the given statement may evaluate to is required for our pointer analysis. For example, the statement $\mathit{if}\ b > 0\ \mathit{then}\lambda x.u\ \mathit{else}\ \lambda y.v$ may evaluate to the abstraction $\lambda x.u$ or to the abstraction $\lambda y.v$ depending on the value of b. This section presents an abstraction analysis, an analysis calculating the set of abstractions that a statement may evaluate to. The analysis is achieved via the following set of inference rules:

$$\frac{}{n : abs \twoheadrightarrow abs}\ (n) \qquad \frac{S_2 : abs \twoheadrightarrow abs'}{S_1 \leftarrow S_2 : abs \twoheadrightarrow abs'}\ (:=^p) \qquad \frac{fd(name) : abs \twoheadrightarrow abs'}{name : abs \twoheadrightarrow abs'}\ (name^{abs})$$

$$\frac{S_t : abs \twoheadrightarrow abs_t \quad S_f : abs \twoheadrightarrow abs_f}{\mathit{if}\ S\ \mathit{then}\ S_t\ \mathit{else}\ S_f : abs \twoheadrightarrow abs_t \cup abs_f}\ (\mathit{if}^p) \qquad \frac{S_2 : abs \twoheadrightarrow abs'}{S_1; S_2 : abs \twoheadrightarrow abs'}\ (seq^p)$$

$$\frac{S_1 : abs \twoheadrightarrow abs'' \quad \forall(\lambda x.S_i) \in abs''.\ S_i[S_2/x] : abs \twoheadrightarrow abs_i}{S_1 S_2 : abs \twoheadrightarrow \cup_i abs_i}\ (app_n^p)$$

$$\frac{}{\lambda x.S : abs \twoheadrightarrow abs \cup \{\lambda x.S\}}\ (abs^p) \qquad \frac{S : abs \twoheadrightarrow abs'}{\mathit{while}\ S\ \mathit{do}\ S_t : abs \twoheadrightarrow abs'}\ (whl^p)$$

$$\frac{(\lambda x.S')S : abs \twoheadrightarrow abs'}{\mathit{letrec}\ x = S\ \mathit{in}\ S' : abs \twoheadrightarrow abs'}\ (letrec^p) \qquad \frac{S_1 : abs \twoheadrightarrow abs'}{\mathit{transmit}\ S_1\ \mathit{from}\ S_2 : abs \twoheadrightarrow abs'}\ (trans^p)$$

Some comments on the inference rules are in order. Since integer statements $(n, S_1\ i_{op}\ S_2)$ do not evaluate to abstractions the rule (n) does not change the input set of abstractions, abs. Other statements that do not evaluate to abstractions (like assignment statement) have inference rules similar to (n). For a given statement S, we use the inference rules above to find abs' such that $S : \emptyset \twoheadrightarrow abs'$. The set abs' contains abstractions that S may evaluate to.

It is straightforward to prove the following lemma:

Lemma 1. *Suppose that* $(S, \delta) \rightarrow (v, \delta')$ *and* $S : \emptyset \twoheadrightarrow abs'$*. If* v *is an abstraction, then* $v \in abs'$*.*

3.2 Context and Flow Sensitive Pointer Analysis

Now we are ready to present the inference rules for our type system for pointer analysis of distributed programs [20,21,2,16,27]. The pointer analysis treated by the following rules is of type flow and context-sensitive. The rules illustrate how different statements update a pointer type, p. Judgement produced by the type system have the form $S : p \rightarrow_m (A, p')$; A contains abstractions for all concrete addresses that may result from executing the statement S in a concrete state of type p. Moreover, if the execution ended then the resulting concrete memory state is of type p'.

$$\frac{}{n : p \rightarrow_m (\emptyset, p)} \qquad \frac{}{x : p \rightarrow_m (p(x), p)}\,(x^p) \qquad \frac{S : p \rightarrow_m (A, p')}{{*S} : p \rightarrow_m (\sup\{p'(a) \mid a \in A\}, p')}\,(*:=^p)$$

$$\frac{}{skip : p \rightarrow_m (\emptyset, p)} \qquad \frac{S : p \rightarrow_m (A, p')}{x := S : p \rightarrow_m (A, p'[x \mapsto A])}\,(:=^p)$$

$$\frac{S_1 : p \rightarrow_m (A_1, p_1) \quad S_2 : p_1 \rightarrow_m (A_2, p_2)}{\begin{array}{l} S_1 \leftarrow S_2 : p \rightarrow_m \\ (A_2, p_2[(l_1, ms_1') \mapsto p_2((l_1, ms_1')) \sqcup \{(l_2, ms_1' \cup ms_1 \cup ms_2) \mid (l_1, ms_1) \in A_1, (l_2, ms_2) \in A_2\}]) \end{array}}\,(\leftarrow^p)$$

$$\frac{S_1 : p \rightarrow_m (A'', p'') \quad S_2 : p'' \rightarrow_m (A, p')}{S_1; S_2 : p \rightarrow_m (A', p')}\,(seq^p) \qquad \frac{S_t : p \rightarrow_m (A, p') \quad S_f : p \rightarrow_m (A, p')}{if\ S\ then\ S_t\ else\ S_f : p \rightarrow_m (A, p')}\,(if^p)$$

$$\frac{S_1 : \emptyset \twoheadrightarrow abs \neq \emptyset \quad \forall \lambda x.S_1'' \in abs.\ S_1''[S_2/x] : p \rightarrow_m (A, p')}{S_1 S_2 : p \rightarrow_m (A, p')}\,(appl^p)$$

$$\frac{fd(name) : p \rightarrow_m (A, p')}{name : p \rightarrow_m (A, p')}\,(name^p) \qquad \frac{S_t : p \rightarrow_m (A, p)}{while\ S\ do\ S_t : p \rightarrow_m (A, p)}\,(whl^p) \qquad \frac{S : p \rightarrow_m (A, p')}{\lambda x.S : p \rightarrow_m (A, p')}\,(abs^p)$$

$$\frac{(\lambda x.S')S : p \rightarrow_m (A, p')}{letrec\ x = S\ in\ S' : p \rightarrow_m (A, p')}\,(letrec^p) \qquad \frac{}{new_l : p \rightarrow_m (\{(l, \{m\})\}, p)}\,(new^p)$$

$$\frac{S : p \rightarrow_m (A, p')}{convert\ (S, n) : p \rightarrow_m (\{(l, \{m' \in ms \mid hdist(m, m') \leq n\}) \mid (l, ms) \in A\}, p')}\,(convert^p)$$

$$\frac{S_2 : p \rightarrow_m (A'', p'') \quad S_1 : p'' \rightarrow_m (A, p')}{transmit\ S_1\ from\ S_2 : p \rightarrow_m (\{(l, M) \mid (l, ms) \in A\}, p')}\,(trans^p)$$

$$\frac{p_1' \leq p_1 \quad S : p_1 \rightarrow_m (A, p_2) \quad A \subseteq A' \quad p_2 \leq p_2'}{S : p_1' \rightarrow_m (A', p_2')}\,(csq^p)$$

$$\frac{Defs : \emptyset \rightsquigarrow fd \quad S : p \rightarrow_m (A', p')}{Defs : S : p \rightarrow_m (A', p')}\,(prg^p)$$

Some comments on the rules are in order. The statements $true, false, S_1\ i_{op}\ S_2$ and $S_1\ b_{op}\ S_2$ have inference rules similar to that of n. The rules for local variables, integer, and Boolean statements are clear; the pointer pre-type does not get updated. The allocation rule, (new^p), results

in the abstract location consists of the allocation site and the number of the machine on which the allocation is taking place. The de-referencing rule, $(*:=^p)$, first calculates the set of addresses that S may evaluate to. Then the rule sums the contents of all these addresses. The sequencing rule, (seq^p), is similar to the corresponding semantics rule. The assignment rule, $(:=^p)$, copies the abstract addresses, resulted from evaluating the right hand side, into the points-to set of the variable x. As the conversion statement only succeeds if S evaluates to a global address on a machine that is within distance n from m. The rule $(convert^p)$ ignores all abstract addresses that are not within the distance n.

This paper considers the parallelism model SPMD (applied in many languages) [2,16,27]. In this model, the *transmit* statement succeeds only if the target statement (S_1) evaluates to abstract addresses that have same cite labels on the current and target machines. Since the distance between these machines is not known statically, the abstract addresses calculated by the rule $(trans^p)$ assumes maximum possible distance. For the statement $S_1 \leftarrow S_2$, assignment via references, we illustrate the following example. Let S_1 be a variable r whose points-to set contains the abstract address $a_1 = (l_1, ms_1)$. Let S_2 be a variable s whose points-to set contains the abstract address $a_2 = (l_2, ms_2)$. Then processing the rule must include augmenting the points-to set of a_1 with a_2. However this is not all; since our parallelism [20,21] model is SPMD, this assignment has to be considered on the other machines, as well. Hence the set $\{(l_2, ms'_1 \cup ms_1 \cup ms_2)\}$ is added to the points-to set of all addressers (l_1, ms'_1) whose first component is l_1.

The recursion rule, $(letrec^p)$, is similar to the corresponding semantics rule. The abstraction rule, (abs^p), is straightforward. The function application rule, $(appl^p)$, uses the abstraction analysis presented earlier to calculate the set of abstractions that S_1 may evaluate-to. For each of the calculated abstractions, the rule does the application and achieve the pointer analysis. Finally the obtained post-pointer-types for all abstractions are summed in the post-pointer-type of the application statement.

The following rule calculates the pointer information for running a statement S on all the machine in M using the SPMD model.

$$\frac{\forall m \in M.\ S : \sup\{p, p_j \mid j \neq i\} \rightarrow_m (A_m, p_m)}{S : p \rightarrow_M (\cup_i A_i, \sup\{p_1, \ldots, p_n\})}\ (main\text{-}pt)$$

The following theorem proves the soundness of our pointer analysis on each single machine of our $|M|$ machines. The soundness of the analysis for the whole distributed system is inferred in the next corollary.

Theorem 1. *Suppose that* $(S, \delta) \rightsquigarrow_m (v, \delta')$, $S : p \rightarrow_m (A', p')$, *and* $\delta \models p$. *Then*

1. $v \in gAddrs \Longrightarrow v$ *is abstracted by some* $a \in A'$, *and*
2. $\delta' \models p'$.

Proof. The proof is by structure induction on the type derivation. Some cases are detailed below.

- The case of (x^p): in this case $v = \delta(x), \delta' = \delta, A' = p(x)$, and $p' = p$. If $\delta(x)$ is an address, then it is abstracted by some $a \in p(x)$ because $\delta \models p$. Clearly in this case $\delta' \models p'$.
- The case of $(:=^p)$: in this case there exists S'', δ'' and p'' such that $S = x := S'', (S'', \delta) \rightsquigarrow_m (v, \delta'')$, and $S'' : p \rightarrow_m (A', p'')$. Moreover, in this case, $\delta' = \delta''[x \mapsto v]$ and $p' = p''[x \mapsto A']$. Therefore by induction hypothesis on $S'' : p \rightarrow_m (A', p'')$, we conclude that $\delta'' \models p''$ and that if $v \in gAddrs$ then v is abstracted by some $a \in A'$. These two results together with definitions of p' and δ' imply that $\delta' \models p'$ which completes the proof of this case.
- The case of $(* :=^p)$: in this case there exist $g = (l, m, a) \in gAddrs, S''$, and A'' such that $S = *S'', (S'', \delta) \rightsquigarrow_m (g, \delta')$, and $S'' : p \rightarrow_m (A'', p')$. Moreover, in this case, $A' = \sup\{p'(a) \mid a \in A''\}$ and $v = \delta'(g)$. Hence by induction hypothesis on $S'' : p \rightarrow_m (A'', p')$, we conclude that g is abstracted by some $a = (l, ms) \in A''$ and that $\delta' \models p'$. Hence $m \in ms$. Now we suppose that $\delta'(g) \in gAddrs$ and prove that $\delta'(g)$ is abstracted by some element in A'. Since $\delta' \models p', \delta'(g)$ is abstracted by some element in $p'(a) \subseteq A'$.
- The case of $(\leftarrow^p)$: in this case there exist $g \in gAddrs, S_1, S_2, p_1, p_2, A_1$, and A_2 such that $S = S_1 \leftarrow S_2, S_1 : p \rightarrow_m (A_1, p_1), S_2 : p_1 \rightarrow_m (A_2, p_2), (S_1, \delta) \rightsquigarrow_m (g, \delta'')$, and $(S_2, \delta'') \rightsquigarrow_m (v', \delta''')$. Moreover, in this case, $\delta' = \delta'''[g \mapsto v'], A' = A_2$, and $p' = p_2[(l_1, ms_1') \mapsto p_2((l_1, ms_1')) \sqcup \{(l_2, ms_1' \cup ms_1 \cup ms_2) \mid (l_1, ms_1) \in A_1, (l_2, ms_2) \in A_2\}]$. Hence by induction hypothesis on $S_1 : p \rightarrow_m (A_1, p_1)$ we conclude that g is abstracted by some $(l_1, ms_1) \in A_1$ and that $\delta'' \models p_1$. Again by induction hypothesis on $S_2 : p_1 \rightarrow_m (A_2, p_2)$, we conclude that v is abstracted by some $(l_2, ms_2) \in A_2$ and that $\delta''' \models p_2$. It remains to show that $\delta' \models p'$. By definitions of δ' and p' it is enough to show that if v is a concrete address, then the image of any abstraction of g under p' contains an abstraction to v. But any abstraction of g has allocation site l_1 and an abstraction of v exists in A_2. Therefore in p' all images of abstract addresses with allocation sites l_1 are augmented with all abstract addresses of A_2 with allocation site l_2. This augmentation guarantees the required.
- The case of $(appl^p)$: in this case there exist S_1 and S_2 such that $S = S_1 S_2$, $(S_1, \delta) \rightsquigarrow_m (\lambda x.S_1', \delta''), (S_2, \delta'') \rightsquigarrow_m (v', \delta''')$, and $(S_1'[v'/x], \delta''') \rightsquigarrow_m (v, \delta')$. Moreover, in this case, $A' = A, S_1 : \emptyset \twoheadrightarrow abs \neq \emptyset$ and $\forall \lambda x.S_1'' \in abs(S_1''[S_2/x] : p \rightarrow_m (A, p'))$. By Lemma 1, $\lambda x.S_1' \in abs$. Hence $S_1'[S_2/x] : p \rightarrow_m (A, p')$. Therefore by induction hypothesis on $\lambda x.S_1' \in abs$, we conclude that $\delta' \models p'$ and that if $v \in gAddrs$ then v is abstracted by some $a \in A'$.
- The case of $(convert^p)$: in this case there exists S' such that $S = convert\ (S', n), (S, \delta) \rightsquigarrow_m (g = (l, m', a), \delta'), \mathrm{hdist}(m, m') \leq n$, and $S' : p \rightarrow_m (A, p')$. Moreover, in this case, $A' = \{(l, \{m' \in ms \mid hdist(m, m') \leq n\}) \mid (l, ms) \in A\}$. By induction hypothesis on $S' : p \rightarrow_m (A, p')$, we conclude that $\delta' \models p'$ and that g is abstracted by some $a = (l', ms') \in A$. As $\mathrm{hdist}(m', m') = 0 \leq n, m' \in ms'$, and $a = (l', ms') \in A$, we conclude that g is abstracted by some element in A which completes the proof for this case.
- The case of $(trans^p)$: in this case there exist S_1 and S_2 such that $S = transmit\ S_1\ from\ S_2, S_2 : p \rightarrow_m (A'', p''), S_1 : p'' \rightarrow_m (A, p'), (S_2, \delta) \rightsquigarrow_m$

$(m', \delta''), m' \in M$, and $(S_1, \delta'') \rightsquigarrow_{m'} (v, \delta')$. Moreover, in this case, $A' = \{(l, M) \mid (l, m) \in A\}$. By induction hypothesis on S_1 and S_2, we conclude that $\delta' \models p'$ and that if $v \in gAddrs$ then it is abstracted by some $a = (l', ms') \in A$. We assume that $v = (l, m, a) \in gAddrs$. Then $(l, ms) \in A$, for some ms that contains m. We conclude that $(l, M) \in A'$. But this last element abstracts v. This completes the proof for this case.

The following corollary follows from Theorem 1 by using the semantics and pointer rules (*main-sem*) and (*main-pt*), respectively. It is noticeable that subscriptions of arrows in Theorem 1 and Corollary 1 are different.

Corollary 1. *Suppose that* $(S, \delta) \rightsquigarrow_M (v, \delta')$, $S : p \rightarrow_M (A', p')$ *and* $\delta \models p$. *Then*

1. $v \in gAddrs \Longrightarrow v \in A$, *and*
2. $\delta' \models p'$.

3.3 Flow Sensitive and Context Insensitive Pointer Analysis

Letting the following rules replace their corresponding ones in the type system of pointer analysis presented above results in a flow sensitive and context insensitive pointer analysis.

$$\frac{name \in dom(fe)}{name : p \rightarrow_m \sup\{p, fe(name)\}}\ (name_1^{ci})$$

$$\frac{}{\varepsilon : fe \rightsquigarrow_m fe}\ (fe_1) \qquad \frac{S : \emptyset \rightarrow p \quad Defs : fe[name \mapsto p] \rightsquigarrow_m fe'}{(name = S); Defs : (fd, fe) \rightsquigarrow_m (fd', fe')}\ (fe_2)$$

$$\frac{Defs : \emptyset \curvearrowright_m fd \quad Defs : \emptyset \rightsquigarrow_m fe \quad S : p \rightarrow_m p'}{Defs : S : p \rightarrow_m p'}\ (prg^{ci})$$

4 Related and Future Work

The langauge that is studied in the current paper is a generalization of those described in [17,18]. The work in [17] introduces a flow-insensitive pointer analysis for programs sharing memory and running on parallel machines that are hierarchical. Beside making the results vague (inaccurate), the insensitivity of the analysis in [17] forces the pointer analysis to ignore the distributivity of targeted programs which is a major drawback. Also the analysis in [17] does not treat context-sensitivity, which is a basic aspect in real-life programming. The current paper overcomes these drawbacks. Using a two-level hierarchy, in [19], constraint-based analyses to calculate sharing properties and locality information of pointers are introduced. These constraint-based analyses are extensions of the earlier work in [18].

Pointer analysis, that dates back to work in [1], was extended to cover parallel programs. In [23] a pointer analysis that is flow-sensitive, context-sensitive, and

thread-aware is introduced for the Cilk multithreaded programming language. Another pointer analysis that is a mix of flow-sensitive and flow-insensitive for multithreaded languages is introduced in [8]. Examples of flow-insensitive pointer analyses for multithreaded programs are [15,28]. Probabilistic points-to analysis is studied in [4,9]. In [4], using the algorithmic style, the probability that a points-to relationship holds is represented by a quantitative description. On the other hand [9] uses type systems to provide an analysis that very suitable to certified codes or proof carrying software. However pointer analysis, that provides a proof for each pointer analysis, for distributed programs running on hierarchical machines are not considered by any of these analyses.

The importance of distributed programs makes analyzing them the focus of much research acclivities [20,21,2,16,27]. The concurrent access of threads of a multi-threaded process to a physically distributed memory may result in data racing bugs [5,22]. In [5] a scheme, called DRARS, is introduced for avoidance and replay of this data race. On DSM or multi-core systems, DRARS assists debugging parallel programs. An important issue to many distributed systems applications is the capturing and examining of the concurrent and causal relationships. In [25], an inclusive graph, POG, of the potential behaviors of systems is produced via an analysis that considers the source code of each process. In [26] and on the model of message sending of distributed programs, the classical problems of algorithmic decidability and satisfiability decidability are studied. In this study, communicating through buffers are used to represent distributed programs.

Associating the result of each pointer analysis with a correctness proof is important and required by applications like certified code or proof carrying code. One advantage of the work presented in this paper over any other related work is the constructions of these proofs. The proofs constructed in our approach have the form of a type derivation and this adds to the value of using type systems. Examples of other analyses that have the form of type systems are [10,8,13,9].

Mathematical domains and maps between domains can be used to mathematically represent programs and data structures. This representation is called denotational semantics of programs [3,14,24]. One of our directions for future research is to translate concepts of data and program slicing to the side of denotational semantics [12,7]. Doing so provide a good tool to mathematically study in deep heap slicing. Then obtained results can be translated back to the side of programs and data structures.

References

1. Amme, W., Zehendner, E.: A/D Graphs - a Data Structure for Data Dependence Analysis in Programs with Pointers. In: Böszörme'nyi, L. (ed.) ACPC 1996. LNCS, vol. 1127, pp. 229–230. Springer, Heidelberg (1996)
2. Barpanda, S.S., Mohapatra, D.P.: Dynamic slicing of distributed object-oriented programs. IET Software 5(5), 425–433 (2011)
3. Cazorla, D., Cuartero, F., Ruiz, V.V., Pelayo, F.L.: A denotational model for probabilistic and nondeterministic processes. In: Lai, T.-H. (ed.) ICDCS Workshop on Distributed System Validation and Verification, pp. E41–E48 (2000)

4. Chen, P.-S., Hwang, Y.-S., Ju, R.D.-C., Lee, J.K.: Interprocedural probabilistic pointer analysis. IEEE Trans. Parallel Distrib. Syst. 15(10), 893–907 (2004)
5. Chiu, Y.-C., Shieh, C.-K., Huang, T.-C., Liang, T.-Y., Chu, K.-C.: Data race avoidance and replay scheme for developing and debugging parallel programs on distributed shared memory systems. Parallel Computing 37(1), 11–25 (2011)
6. El-Zawawy, M., Daoud, N.: New error-recovery techniques for faulty-calls of functions. Computer and Information Science 4(3) (May 2012)
7. El-Zawawy, M.A.: Semantic spaces in Priestley form. PhD thesis, University of Birmingham, UK (January 2007)
8. El-Zawawy, M.A.: Flow Sensitive-Insensitive Pointer Analysis Based Memory Safety for Multithreaded Programs. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2011, Part V. LNCS, vol. 6786, pp. 355–369. Springer, Heidelberg (2011)
9. El-Zawawy, M.A.: Probabilistic pointer analysis for multithreaded programs. ScienceAsia 37(4) (December 2011)
10. El-Zawawy, M.A.: Program optimization based pointer analysis and live stack-heap analysis. International Journal of Computer Science Issues 8(2) (March 2011)
11. El-Zawawy, M.A.: Dead code elimination based pointer analysis for multithreaded programs. Journal of the Egyptian Mathematical Society (January 2012), doi:10.1016/j.joems.2011.12.011
12. El-Zawawy, M.A., Jung, A.: Priestley duality for strong proximity lattices. Electr. Notes Theor. Comput. Sci. 158, 199–217 (2006)
13. El-Zawawy, M.A., Nayel, H.A.: Partial redundancy elimination for multi-threaded programs. IJCSNS International Journal of Computer Science and Network Security 11(10) (October 2011)
14. Guo, M.: Denotational semantics of an hpf-like data-parallel langguage model. Parallel Processing Letters 11(2/3), 363–374 (2001)
15. Hicks, J.: Experiences with compiler-directed storage reclamation. In: FPCA, pp. 95–105 (1993)
16. Seragiotto Jr., C., Fahringer, T.: Performance analysis for distributed and parallel java programs with aksum. In: CCGRID, pp. 1024–1031. IEEE Computer Society (2005)
17. Kamil, A., Yelick, K.A.: Hierarchical Pointer Analysis for Distributed Programs. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 281–297. Springer, Heidelberg (2007)
18. Liblit, B., Aiken, A.: Type systems for distributed data structures. In: POPL, pp. 199–213 (2000)
19. Liblit, B., Aiken, A., Yelick, K.A.: Type Systems for Distributed Data Sharing. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 273–294. Springer, Heidelberg (2003)
20. Lindberg, P., Leingang, J., Lysaker, D., Khan, S.U., Li, J.: Comparison and analysis of eight scheduling heuristics for the optimization of energy consumption and makespan in large-scale distributed systems. The Journal of Supercomputing 59(1), 323–360 (2012)
21. Onbay, T.U., Kantarci, A.: Design and implementation of a distributed teleradiography system: Dipacs. Computer Methods and Programs in Biomedicine 104(2), 235–242 (2011)
22. Park, C.-S., Sen, K., Hargrove, P., Iancu, C.: Efficient data race detection for distributed memory parallel programs. In: Lathrop, S., Costa, J., Kramer, W. (eds.) SC, p. 51. ACM (2011)

23. Rugina, R., Rinard, M.C.: Pointer analysis for multithreaded programs. In: PLDI, pp. 77–90 (1999)
24. Schwartz, J.S.: Denotational Semantics of Parallelism. In: Kahn, G. (ed.) Semantics of Concurrent Computation. LNCS, vol. 70, pp. 191–202. Springer, Heidelberg (1979)
25. Simmons, S., Edwards, D., Kearns, P.: Communication analysis of distributed programs. Scientific Programming 14(2), 151–170 (2006)
26. Toporkov, V.V.: Dataflow analysis of distributed programs using generalized marked nets. In: DepCoS-RELCOMEX, pp. 73–80. IEEE Computer Society (2007)
27. Truong, H.L., Fahringer, T.: Soft Computing Approach to Performance Analysis of Parallel and Distributed Programs. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 50–60. Springer, Heidelberg (2005)
28. Zhu, Y., Hendren, L.J.: Communication optimizations for parallel c programs. J. Parallel Distrib. Comput. 58(2), 301–332 (1999)