Hierarchical Pointer Analysis for Distributed Programs

Amir Kamil
Computer Science Division, University of California, Berkeley
kamil@cs.berkeley.edu

April 14, 2006

1 Introduction

Many distributed, parallel programming languages provide the programmer the illusion of a global, shared address space. In such a language, a pointer on a thread can reference either memory located in the same physical address space as the thread or memory located in a separate physical address space. In a distributed program, it is desirable for a pointer analysis to distinguish pointers within a physical address space and pointers between distinct address space.

The Titanium programming language [7] is a high performance dialect of Java designed for distributed machines. It is a *single program, multiple data* (SPMD) language, so all threads execute the same code image. In addition, Titanium has a global address space abstraction, so that any thread can directly access memory on another thread. At runtime, two threads may share the same physical address space, in which case such an access is done directly, or they may be in distinct address spaces, in which case the global access must be translated into some form of communication. In the former case, the access is much faster, so it is useful to know statically whether or not an access is within the same physical address space or across distinct address spaces.

Since threads can share a physical address space, they are arranged in the following three-level hierarchy:

- **Level 1**: an individual thread
- **Level 2**: threads within the same physical address space
- **Level 3**: all threads

The Titanium type system distinguishes between levels 2 and 3 of the hierarchy in order to determine whether or not an access crosses address spaces, but it does not separate levels 1 and 2. As we will see in Section 5.1, the distinction between levels 1 and 2 is important for inter-thread alias analysis. Thus, we would like a pointer analysis that accounts for the hierarchical distribution of Titanium threads.

In the rest of this paper, we first generalize the problem to a hierarchy with an arbitrary depth. We then present a small language that incorporates such a hierarchy and provide both a type system and operational semantics for the language. We proceed to define a pointer analysis on the language. Finally, we describe some example applications of the pointer analysis.

2 Background

Consider a set of machines arranged in an arbitrary hierarchy, such as that of Figure 1. A *machine* corresponds to a single execution stream within a distributed program, and for the purposes of our analysis, we ignore physical address spaces. Each machine has a corresponding *machine number*. The *depth* of the hierarchy is the number of levels it contains. The *distance* between machines is equal to the lowest level of the hierarchy in which they are in the same subtree. A pointer on a machine m has a corresponding *width*, and it can only refer to locations on machines whose distance from m is no more than the pointer's width.

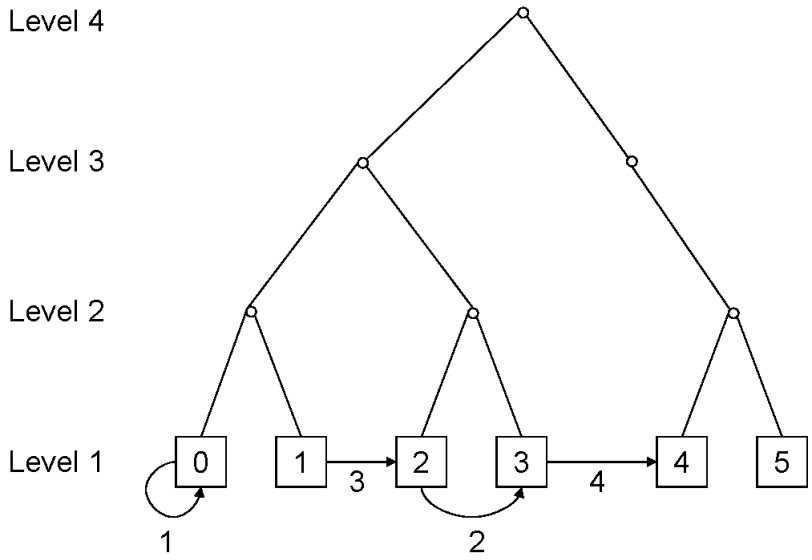

Figure 1: A possible machine hierarchy with four levels. Labels on arrows indicate the hierarchy distance between the endpoints.

$$
\begin{array}{lll}
n & ::= \text{integer literals} & \\
\tau & ::= int \mid \texttt{ref}_n \tau & \text{(types)} \\
e & ::= n \mid x \mid \texttt{new}_l\ \tau \mid *\, e \mid \texttt{convert}(e, n) & \\
 & \quad \mid \texttt{transmit}\ e_1\ \texttt{from}\ e_2 \mid e_1; e_2 & \\
 & \quad \mid x := e \mid e_1 \leftarrow e_2 & \text{(expressions)}
\end{array}
$$

Figure 2: The syntax of the *Ti* language.

3 Language

Our analysis is formalized using a simple language, called *Ti* , that illustrates the key features of the analysis. *Ti* is a generalization of the language used by Liblit and Aiken in their work on locality inference [5]. Like Titanium, *Ti* uses a SPMD model of parallelism, so that all machines execute the same program text. The height of the machine hierarchy is known statically, and we will refer to it as h from here on out. References thus can have any width in the range $[1, h]$.

3.1 Syntax

The syntax of *Ti* is summarized in Figure 2. Types can be integers or reference types. The latter are parameterized by a width n, in the range $[1, h]$. Expressions in *Ti* consist of the following

- integer literals
- variables. We assume a fixed set of variables of predefined type. We also assume that variables are thread-private.
- reference allocations. The expression $\texttt{new}_l\ \tau$ allocates a memory cell of type τ and returns a reference to the cell. Each allocation site has a unique label l.
- dereferencing

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash \texttt{new}_l\ \tau : \texttt{ref}_1\ \tau}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{ref}_n\ \tau}{\Gamma \vdash *e : expand(\tau, n)}$$

$$\frac{\Gamma \vdash e : \texttt{ref}_n\ \tau}{\Gamma \vdash \texttt{convert}(e, m) : \texttt{ref}_m\ \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{transmit}\ e_1\ \texttt{from}\ e_2 : expand(\tau, h)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1; e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma(x) = \tau}{\Gamma \vdash x := e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{ref}_n\ \tau \quad \Gamma \vdash e_2 : \tau \quad robust(\tau, n)}{\Gamma \vdash e_1 \leftarrow e_2 : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{ref}_n\ \tau \quad n < m}{\Gamma \vdash e : \texttt{ref}_m\ \tau}$$

Figure 3: Type checking rules.

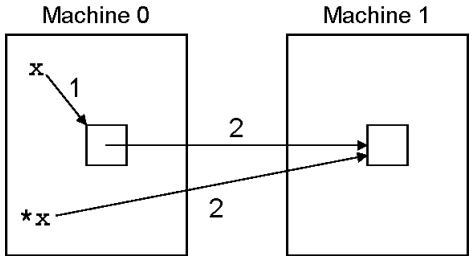

Figure 4: Dereferences may require width expansion. The labels correspond to pointer widths.

- type conversions. Only the width parameter of a reference type is subject to conversion.
- communication. The expression `transmit` e_1 `from` e_2 evaluates e_1 on machine e_2 and transmits the result to all other machines.
- sequencing
- assignment to variables
- assignment through references. In $e_1 \leftarrow e_2$, e_2 is written into the location referred to by e_1.

For simplicity, *Ti* does not have conditional statements, as they are not integral to the analysis.

3.2 Type System

The type checking rules for *Ti* are summarized in Figure 3, and are similar to those in [5].

The allocation expression $\texttt{new}_l\ \tau$ produces a reference type $\texttt{ref}_1\ \tau$ with width 1, since the allocated memory is guaranteed to be on the machine that is performing the allocation. Pointer dereferencing is more problematic, however. Consider the situation in figure 4, where x on machine 0 refers to a location on machine 0 that refers to a location on machine 1. This

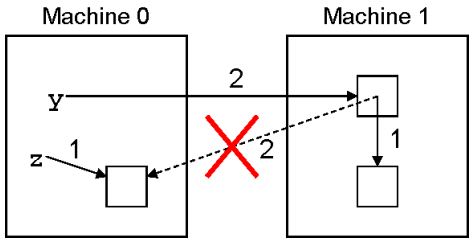

Figure 5: The assignment $y \leftarrow z$ is forbidden, since the location referred to by y can only hold pointers of width 1 but requires a pointer of width 2 to refer to z.

$$
\begin{aligned}
expand(\texttt{ref}_m \tau, n) &\equiv \texttt{ref}_{max(m,n)}\ \tau \\
expand(\tau, n) &\equiv \tau \quad \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
robust(\texttt{ref}_m \tau, n) &\equiv false \quad \text{if } m < n \\
robust(\tau, n) &\equiv true \quad \text{otherwise}
\end{aligned}
$$

Figure 6: Type manipulating functions.

implies that x has type $\texttt{ref}_1\ \texttt{ref}_2\ \tau$. The result of $*x$ should be a reference to the location on machine 1, so it must have type $\texttt{ref}_2\ \tau$. In general, a dereference of a value of type $\texttt{ref}_a\ \texttt{ref}_b\ \tau$ produces a value of type $\texttt{ref}_{max(a,b)}\ \tau$.

The `transmit` expression is another interesting case. If the value to be communicated is an integer, then the resulting type is still an integer. If the value is a reference, however, the result must be promoted to the maximum width h, since the relationship between source and destination is not statically known.

The typing rule for the assignment through reference expression is also nontrivial. Consider the case where y has type $\texttt{ref}_2\ \texttt{ref}_1\ \tau$, as in Figure 5. Should it be possible to assign to y with a value of type $\texttt{ref}_1\ \tau$? Such a value must be on machine 0, but the location referred to by x is on machine 1. Since that location holds a value of type $\texttt{ref}_1\ \tau$, it must refer to a location on machine 1. Thus, the assignment should be forbidden. In general, an assignment to a reference of type $\texttt{ref}_a\ \texttt{ref}_b\ \tau$ should only be allowed if $a \leq b$.

There is also a subtyping rule that allows for implicit widening of a reference. Subsumption is only allowed for the top-level width of a reference.

As in [5], we define an $expand$ function and a $robust$ predicate to facilitate type checking. The $expand$ function widens a type when necessary, and the $robust$ predicate determines when it is legal to assign to a reference. They are shown in Figure 6.

3.3 Operational Semantics

In this section we present the operational semantics of *Ti* . We ignore concurrency in defining the semantics, since it is not essential to our analysis.

We use the following semantic domains:

$$
\begin{array}{rr}
M & \text{(the set of machines)} \\
A & \text{(the set of local addresses)} \\
Id & \text{(the set of identifiers)} \\
Var = M \times Id & \text{(the set of variables)} \\
L & \text{(the set of labels)} \\
T & \text{(the set of all types)} \\
G = L \times M \times A & \text{(the set of global addresses)} \\
V = n + G & \text{(the set of values)} \\
Store = (G + Var) \rightarrow V & \text{(the contents of memory)} \\
Exp & \text{(the set of all expressions)}
\end{array}
$$

We use the following conventions for naming elements of the above domains:

$$
\begin{array}{rr}
m \in M & \text{(a machine)} \\
v \in V & \text{(a value)} \\
\sigma \in Store & \text{(a memory snapshot)} \\
a \in A & \text{(a local address)} \\
l \in L & \text{(a label)} \\
g = (l, m, a) \in G & \text{(a global address)} \\
e \in Exp & \text{(an expression)}
\end{array}
$$

Judgements in our operational semantics have the form $< e, m, \sigma > \Downarrow < v, \sigma' >$, which means that expression e executed on machine m in a global state σ evaluates to the value v and results in the new state σ'. We use the notation $\sigma[g := v]$ to denote the function $\lambda x.$ `if` $x = g$ `then` v `else` $\sigma(x)$.

The rules for integer and variable expressions are trivial.

$$\frac{}{< n, m, \sigma > \Downarrow < n, \sigma >} \qquad \frac{\sigma(x) = v}{< x, m, \sigma > \Downarrow < v, \sigma >}$$

For allocations, we introduce a special $null$ value to represent uninitialized pointers. The result on an allocation is an address on the local machine that is guaranteed to not already be in use.

$$\frac{(l, m, a) \text{ is fresh} \quad \sigma' = \sigma[(l, m, a) := null]}{< new_l\ \tau, m, \sigma > \Downarrow < (l, m, a), \sigma' >}$$

The rule for dereferencing is simple, except that it is illegal to dereference a $null$ pointer.

$$\frac{< e, m, \sigma > \Downarrow < g, \sigma' > \quad g \neq null \quad v = \sigma'(g)}{< *e, m, \sigma > \Downarrow < v, \sigma' >}$$

The rule for variable assignment is also simple.

$$\frac{< e, m, \sigma > \Downarrow < v, \sigma' > \quad \sigma'' = \sigma'[x := v]}{< x := e, m, \sigma > \Downarrow < v, \sigma'' >}$$

The rule for assignment through a reference is the combination of a dereference and a normal assignment.

$$\frac{< e_1, m, \sigma > \Downarrow < g, \sigma_1 > \quad < e_2, m, \sigma_1 > \Downarrow < v, \sigma_2 > \quad g \neq null \quad \sigma' = \sigma_2[g := v]}{< e_1 \leftarrow e_2, m, \sigma > \Downarrow < v, \sigma' >}$$

The rule for sequencing is as expected.

$$\frac{< e_1, m, \sigma > \Downarrow < v_1, \sigma_1 > \quad < e_2, m, \sigma_1 > \Downarrow < v_2, \sigma_2 >}{< e_1; e_2, m, \sigma > \Downarrow < v_2, \sigma_2 >}$$

The type conversion expression makes use of the $hier$ function, which returns the hierarchical distance between two machines. The conversion is only allowed if that distance is no more than the target type.

$$\frac{< e, m, \sigma > \Downarrow < g = (l, m', a), \sigma' > \quad hier(m, m') \leq n}{< \texttt{convert}(e, n), m, \sigma > \Downarrow < g, \sigma' >}$$

In the `transmit` operation, the expression is evaluated on the given machine.

$$\frac{< e_2, m, \sigma > \Downarrow < n, \sigma_2 > \quad n \in M \quad < e_1, n, \sigma_2 > \Downarrow < v, \sigma_1 >}{< \texttt{transmit}\ e_1\ \texttt{from}\ e_2, m, \sigma > \Downarrow < v, \sigma_1 >}$$

4 Pointer Analysis

Given a program, it is useful to know which variables and memory locations can refer to the same location. We would like to produce a points-to analysis [1] for *Ti* in order to answer this question. So that we can ignore any issues of concurrency and also for efficiency, our analysis is flow-insensitive. We only define the analysis on the single machine m – since *Ti* is SPMD, the results are the same for all machines.

4.1 Concrete Domain

Since our analysis is flow-insensitive, we need not determine the concrete state at each point in a program. Instead, we define the concrete state over the whole program. Since we are doing pointer analysis, we are only interested in reference values, and since a location can contain different values over the lifetime of the program, we must compute the set of all possible values for each location on machine m. The concrete state is thus a member of the domain $CS = (G + Id) \rightarrow \mathcal{P}(G)$.

4.2 Abstract Domain

For our abstract semantics, we define an *abstract location* to correspond to the abstraction of a concrete memory location. Abstract locations are defined relative to a particular machine m. An abstract location relative to machine m is a member of the domain $A_m = L \times h$ – it is identified by both a hierarchy width and an allocation site. The elements of A_m are ordered by the following relation:

$$(l, n_1) \sqsubseteq (l, n_2) \iff n_1 \leq n_2$$

The lattice thus has height in $\mathrm{O}(h)$.

We define $R \subset \mathcal{P}(A_m)$ to be the subset of $\mathcal{P}(A_m)$ that contains no redundant elements. An element S is *redundant* if:

$$\exists x \in S.\ \exists y \in S.\ x \sqsubseteq y\ \wedge\ x \neq y$$

An elements of $S \in R$ can be represented by an n-digit vector u, where $n = |A_m|$ and the digits are in the range $[0, h]$. The vector is defined as follows:

$$u(i) = j \text{ if } \exists j.\ (l_i, j) \in S$$
$$0 \text{ otherwise}$$

We define the following ordering on elements of R:

$$S_1 \sqsubseteq S_2 \iff \forall x \in S_1.\ \exists y \in S_2.\ x \sqsubseteq y$$

In the vector representation, the following is an equivalent ordering:

$$S_1 \sqsubseteq S_2 \iff \forall i \in [1, |A_m|].\ u_1(i) \leq u_2(i)$$

The relation induces a lattice with minimal element corresponding to $u_\perp(i) = 0$, and a maximal element corresponding to $u_\top(i) = h$. The maximal chain between $\perp$ and $\top$ is derived by increasing a single vector digit at a time by 1, so the chain has height $(h+1) \cdot |A_m|$. The height of the lattice is thus in $\mathrm{O}(h \cdot |A_m|)$.

We now define a Galois connection between $\mathcal{P}(G)$ and R as follows:

$$\beta_m((l, m', a)) = \{(l, hier(m, m'))\}$$
$$\gamma_m(S) = \{(l, m', a) \mid (l, n) \in S \ \wedge \ hier(m, m') \leq n\}$$

Finally, we abstract the concrete domain CS to the following abstract domain:

$$AS = (A_m + Id) \to R$$

We define an ordering over elements of AS as follows:

$$\sigma_A \sqsubseteq \sigma'_A \iff \forall x \in (A_m + Id).\ \sigma_A(x) \sqsubseteq \sigma'_A(x)$$

The resulting lattice has height in $\mathrm{O}(h \cdot |A_m| \cdot (|A_m| + |Id|))$. Since the number of abstract locations and identifiers is limited by the size of the input program P, the height is in $\mathrm{O}(h \cdot |P|^2)$.

4.3 Abstract Inference Rules

For each expression in *Ti* , we provide inference rules for how the expression updates the abstract state σ_A. The judgements are of the form $\sigma_A \ \vdash \ e \ : \ S, \sigma'_A$, which means that expression e in abstract state σ_A can refer to the abstract locations S and results in the modified abstract state σ'_A. Most of the rules are derived directly from the operational semantics of the language.

The rules for integer and variable expressions are straightforward. Neither updates the abstract state, and the latter returns the abstract locations in the points-to set of the variable.

$$\frac{}{\sigma_A \ \vdash \ n \ : \ \emptyset, \sigma_A} \qquad \frac{\sigma_A(x) = S}{\sigma_A \ \vdash \ x \ : \ S, \sigma_A}$$

An allocation returns the abstract location corresponding to the allocation site, with width 1.

$$\frac{S = \{(l, 1)\}}{\sigma_A \ \vdash \ new_l \ \tau \ : \ S, \sigma_A}$$

The rule for dereferencing is similar to the operational semantics rule, except that all source abstract locations are simultaneously dereferenced.

$$\frac{\sigma_A \ \vdash \ e \ : \ S', \sigma'_A \quad S = \{a \mid \exists b \in S'.\ a \in \sigma'_A(b)\}}{\sigma_A \ \vdash \ *e \ : \ S, \sigma'_A}$$

The rule for sequencing is also analogous to its operational semantics rule.

$$\frac{\sigma_A \ \vdash \ e_1 \ : \ S_1, \sigma'_A \quad \sigma'_A \ \vdash \ e_2 \ : \ S_2, \sigma''_A}{\sigma_A \ \vdash \ e_1; e_2 \ : \ S_2, \sigma''_A}$$

The rule for variable assignment merely copies the source abstract locations into the points-to set of the target variable.

$$\frac{\sigma_A \ \vdash \ e \ : \ S, \sigma'_A \quad \sigma''_A = \sigma'_A[x := \sigma'_A(x) \sqcup S]}{\sigma_A \ \vdash \ x := e \ : \ S, \sigma''_A}$$

The type conversion expression can only succeed if the result is within the specified hierarchical distance, so it narrows all abstract locations that are outside that distance.

$$\frac{\sigma_A \ \vdash \ e \ : \ S', \sigma'_A \quad S = \{a = (l, m) \mid \exists k.\ (l, k) \in S' \ \wedge \ m = min(k, n)\}}{\sigma_A \ \vdash \ \texttt{convert}(e, n) \ : \ S, \sigma'_A}$$

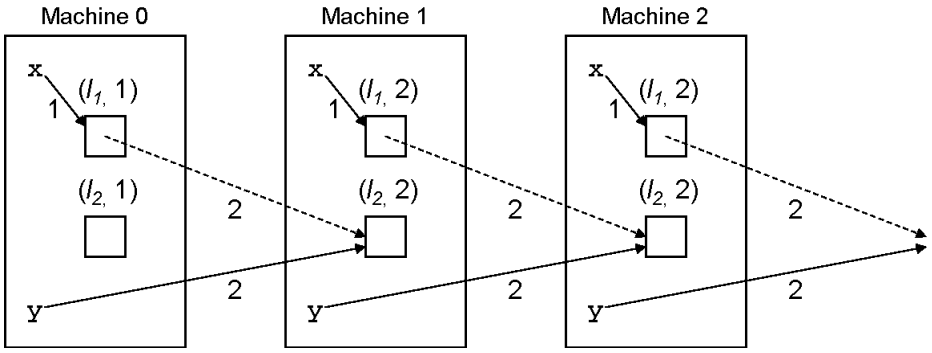

Figure 7: The assignment $x \leftarrow y$ on machine 0 results in the abstract location $(l_2, 2)$ being added to the points-to set of $(l_1, 1)$, as shown by the first dashed arrow. The assignment on machine 1 results in the abstract location $(l_2, 2)$ being added to the points-to set of $(l_1, 2)$, as shown by the second dashed arrow. The assignment must also be accounted for on the rest of the machines. (Abstract locations in the figure are with respect to machine 0.)

The SPMD model of parallelism in *Ti* implies that the source expression of the `transmit` operation evaluates to the same abstract locations on the source machine, relative to the source machine, as it would on the destination machine, relative to the destination machine. These locations then need to be maximally widened to correspond to the result on the destination machine.

$$\frac{\sigma_A \vdash e_2 : S_2, \sigma'_A \quad \sigma'_A \vdash e_1 : S_1, \sigma''_A \quad S = \{(l,h) \mid \exists m.\ (l,m) \in S_1\}}{\sigma_A \vdash \texttt{transmit}\ e_1\ \texttt{from}\ e_2 : S, \sigma''_A}$$

The rule for assignment through references is the most interesting. Suppose an abstract location $a_2 = (l_2, 2)$ is assigned into an abstract location $a_1 = (l_1, 1)$, as in Figure 7. Of course, we have to add a_2 to the points-to set of a_1. In addition, since *Ti* is SPMD, we have to account for the effect of the same assignment on a different machine. Consider the assignment on machine m', where $hier(m, m') = 2$. The location a_1 relative to m corresponds to a location $a'_1 = (l_1, 2)$ relative to m'. The location a_2 can correspond to a concrete location on m', so its abstraction can be $a'_2 = (l_2, 1)$ relative to m'. But it can also correspond to a concrete location on m'' where $hier(m, m'') = hier(m', m'') = 2$, so its abstraction can also be $a''_2 = (l_2, 2)$. But since $a'_2 \sqsubseteq a''_2$, it is sufficient to assume that a_2 corresponds to a''_2 on m'. From the point of view of m' then, the abstract location $(l_2, 2)$ should be added to the points-to set of the location $(l_1, 2)$.

In general, whenever an assignment occurs from (l_2, n_2) to (l_1, n_1), we have to update not only the points-to set of (l_1, n_1) but the sets of all locations corresponding to label l_1 and of any width. The proper update is to add the location $(l_2, max(n_1, n_2, w))$ to the points-to set of each location (l_1, w), as reflected in the assignment rule[1].

$$\frac{\sigma_A \vdash e_1 : S_1, \sigma'_A \quad \sigma'_A \vdash e_2 : S_2, \sigma''_A \quad \sigma'''_A = update(\sigma''_A, S_1, S_2)}{\sigma_A \vdash e_1 \leftarrow e_2 : S_2, \sigma'''_A}$$

The update function is defined as follows:

$$\begin{aligned} update(\sigma, S_1, S_2)(a_1 = (l_1, n_1)) = \sigma(a_1) \\ \sqcup \{(l_2, n_2) \mid (l_1, n'_1) \in S_1 \wedge (l_2, n'_2) \in S_2 \wedge \neg(\exists a''_1 = (l_1, n''_1) \in S_1.\ n''_1 > n'_1) \\ \wedge \neg(\exists a''_2 = (l_2, n''_2) \in S_2.\ n''_2 > n'_2) \wedge n_2 = max(n_1, n'_1, n'_2)\} \end{aligned}$$

4.4 Algorithm

The set of inference rules, instantiated over all the expressions in a program and applied in some arbitrary order[2], composes a function $F : AS \rightarrow AS$. Only the two assignment rules affect the input state σ_A, and in both rules, the output consists of a

[1] The proof of this is omitted for brevity.

[2] Since the analysis is flow-insensitive, the order of application is not important.

Program	Types	Intra-Machine	Inter-Machine
`gas`	2482	779	223
`gsrb`	512	187	18
`lu-fact`	490	177	1
`pps`	7026	1827	26
`spmv`	443	177	0

Table 1: Race detection results for some Titanium programs.

least upper bound operation involving the input state. As a result, F is an increasing function, and the least fixed point of F, $F_0 = \sqcup_n F^n(\lambda x.\, \emptyset)$, is the analysis result.

The function F has a rule for each program expression, so it takes time in $\mathrm{O}(|P|)$ to apply it, where P is the input program. Since the lattice over AS has height in $\mathrm{O}(h \cdot |P|^2)$, it takes time in $\mathrm{O}(h \cdot |P|^3)$ to compute the fixed point of F. The running time of the analysis is thus cubic in the size of the input program and proportional to the height of the machine hierarchy.

5 Applications

The pointer information computed in Section 4 can be applied to multiple analyses and optimizations for parallel programs. In this section, we take a look at two clients: alias analysis and locality inference.

5.1 Alias Analysis

Pointer information can be used to determine which expressions alias each other in a program. Two expressions e_1 and e_2 can alias each other on machine m if, given the fixed point state F_0, abstract inference judgements $F_0 \vdash e_1 : S_1, F_0$ and $F_0 \vdash e_2 : S_2$ exist, and S_1 and S_2 satisfy the following:

$$\exists a_1 = (l_1, n_1) \in S_1.\, \exists a_2 = (l_2, n_2) \in S_2.\, l_1 = l_2$$

This is because the concretization of the abstract locations (l_1, n_1) and (l_1, n_2) both must contain the concrete locations (l_1, m, a) for any a.

The pointer information can also determine inter-machine aliasing of expressions. An expression e_1 on machine m can alias an expression e_2 on some other machine m' if the resulting abstract sets S_1 and S_2 satisfy the following:

$$\exists a_1 = (l_1, n_1) \in S_1.\, \exists a_2 = (l_2, n_2) \in S_2.\, l_1 = l_2 \ \wedge\ (n_1 > 1 \ \vee\ n_2 > 1)$$

In particular, on m' such that $hier(m, m') = 2$, if $n_1 > 1$, then the concretizations of a_1 on m and a_2 on m' both contain the concrete locations (l_1, m', a) for any a, but if $n_2 > 1$, they both must contain the concrete locations (l_1, m, a). In either case, the expressions can alias across the two machines.

A prototype alias analysis has been built into the Titanium compiler, using a two-level hierarchy. The analysis can be used in conjunction with concurrency analysis [4] to statically detect race conditions. Table 1 shows the number of races detected in some sample programs when only types are used to distinguish expressions, when intra-machine alias analysis is used, and when inter-machine analysis is applied. Using inter-machine analysis reduces the number of false positives detected by a large margin.

5.2 Reference Width Inference

Liblit and Aiken produced a constraint-based analysis for inferring the minimal width for the type of a variable or expression [5]. Pointer information can be used instead of constraints. In particular, if an expression e of reference type evaluates to the abstract set S, then its minimal width is:

$$w_{min} = max\{n \mid \exists a = (l, n) \in S\}$$

Work is under way to implement width inference in the Titanium compiler.

6 Related Work

The language and type system we presented here are generalizations of those described by Liblit and Aiken [5]. They defined a two-level hierarchy and used it to to infer locality information about pointers.

The pointer analysis we presented here is a generalization and formalization of the analysis sketched in a previous paper [3]. That analysis is similar to a two-level version of our hierarchical analysis, but the abstraction is quite different. Only the abstraction of the `transmit` operation was described in that paper, though a full implementation was done. For this paper, we spent much effort in attempting to extend the implementation to a three-level version of the abstraction presented here. However the amount of changes required were more extensive than we expected due to the differences in the abstraction used, and we were unable to finish the implementation.

Others have also tackled the problem of parallel pointer analysis. Rugina and Rinard develop a thread-aware alias analysis for the Cilk multithreaded programming language [6] that is both flow-sensitive and context-sensitive. Others such as Zhu and Hendren [8] and Hicks [2] have developed flow-insensitive versions for multithreaded languages. However, none of these analyses consider hierarchical, distributed machines.

7 Conclusion

The global address space abstraction over a distributed machine is quite powerful, allowing pointers to refer to locations in separate physical address spaces. In this paper, we presented a simple language that encodes a hierarchical distribution in its type system and a pointer analysis for that language. The analysis can be used for both intra-thread and inter-thread alias analysis, as well as type inference on the pointers in a program.

In the future, we plan on completing the implementation of a three-level pointer analysis for the Titanium programming language. The resulting pointer information will be used to increase the precision of existing analyses and effectiveness of compiler optimizations. We expect the pointer information to be very useful, as suggested by the two-level race detection results presented in this paper.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] J. Hicks. Experiences with compiler-directed storage reclamation. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 95–105, New York, NY, USA, 1993. ACM Press.

[3] A. Kamil, J. Su., and K. Yelick. Making sequential consistency practical in Titanium. In *Supercomputing 2005*, November 2005.

[4] A. Kamil and K. Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In *18th International Workshop on Languages and Compilers for Parallel Computing*, October 2005.

[5] B. Liblit and A. Aiken. Type systems for distributed data structures. In *Principles of Programming Languages*, Boston, Massachusetts, January 2000.

[6] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 77–90, New York, NY, USA, 1999. ACM Press.

[7] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *Workshop on Java for High-Performance Network Computing*, Stanford, California, February 1998.

[8] Y. Zhu and L. J. Hendren. Communication optimizations for parallel C programs. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 199–211, New York, NY, USA, 1998. ACM Press.