Hierarchical Pointer Analysis for Distributed Programs

Amir Kamil Katherine Yelick

Computer Science Division, University of California, Berkeley
{*kamil,yelick*}@*cs.berkeley.edu*

Abstract. We present a new pointer analysis for use in shared memory programs running on hierarchical parallel machines. The analysis is motivated by the partitioned global address space languages, in which programmers have control over data layout and threads and can directly read and write to memory associated with other threads. Titanium, UPC, Co-Array Fortran, X10, Chapel, and Fortress are all examples of such languages. The novelty of our analysis comes from the hierarchical machine model used, which captures the increasingly hierarchical nature of modern parallel machines. For example, the analysis can distinguish between pointers that can reference values within a thread, within a shared memory multiprocessor, or within a network of processors. The analysis is presented with a formal type system and operational semantics, articulating the various ways in which pointers can be used within a hierarchical machine model. The hierarchical analysis has several applications, including race detection, sequential consistency enforcement, and software caching. We present results of an implementation of the analysis, applying it to data race detection, and show that the hierarchical analysis is very effective at reducing the number of false races detected.

1 Introduction

The introduction of multi-core processors marks a dramatic shift in software development: parallel software will be required for laptops, desktops, gaming consoles, and graphics processors. These chips are building blocks in larger shared and distributed memory parallel systems, resulting in machines that are increasingly hierarchical and use a combination of cache-coherent shared memory, partitioned memory with (remote) direct memory access (DMA or RDMA), and message passing. The partitioned global address space (PGAS) model is a natural fit for programming these machines, and languages that use it include Unified Parallel C (UPC) [7, 26], Co-Array Fortran (CAF) [25], Titanium [28, 12] (based on Java [10]), Chapel [8], X10 [24], and Fortress [1]. In all of these languages, pointers to shared state is permitted, and a fundamental question is whether a given pointer can be proven to access data in only a limited part of the machine hierarchy. Some applications of this are: 1) a pointer that accesses data that is private to a single thread cannot be involved in a data race; 2) a pointer that accesses data within a chip multiprocessor may require memory fences to ensure ordering, but those fences only need to make data visible within the chip level; 3) pointer limits may inform a software caching system that coherence protocols may be restricted to a subset of processors; 4) a pointer with a limited domain may use fewer bits in its representation, since only a fraction of the total address space is accessible.

In this paper we introduce a pointer analysis that is designed for a hierarchical setting. Our analysis allows for an arbitrarily deep hierarchy, such as the abstract machine model in Fortress, although in this paper we apply it to the three-level model of Titanium. In Titanium, a pointer may refer to data only within a single thread, or to data associated with any threads within a SMP node, or to any thread in the machine.

We develop a model language, *Ti*, for presenting our analysis and give both a type system and operational semantics for the language. *Ti* has the essential features of any global address space language: the ability to create references to data, share data with other machines in the system through references, and dereference them for either read or write access. *Ti* also has a hierarchical machine model, which is general enough to cover all of the existing PGAS languages. We implement our analysis in the context of the full Titanium language and then apply the analysis, in conjunction with an existing concurrency analysis [15], towards race detection, and show that it greatly reduces the number of false races detected on five application benchmarks. In previous work we demonstrated some of the other applications of pointer analysis in Titanium [14], but without the generality of the hierarchical analysis presented here.

2 Background

In this section, we describe some machines and languages that use a hierarchical memory model and discuss the aspects of Titanium that are relevant to the pointer analysis.

2.1 Hierarchical Memory

Parallel machines are often built with hierarchical memory systems, with local caches or explicitly managed local stores associated with each process. For example, partitioned global address space (PGAS) languages may run on shared memory, distributed memory machines or hybrids, with the language runtime providing the illusion of shared memory through the use of wide pointers (that store both a processor node number and an address), distributed arrays, and implicit communication to access such data. Hierarchies also exist within processors in the form of caches and local stores. For example, the Cell game processor has a local store associated with each of the *SPE* processors, which can be accessed by other SPEs through memory move (DMA) operations. Additional levels of partitioning are also possible, such as partitioning memory in a computational grid into clusters, each of which is partitioned into nodes, as in Figure 1.

Most PGAS languages use a two level abstraction of memory, where data is either local to a thread or shared by all, although Titanium uses three levels and Fortress has an arbitrary number. In many PGAS languages, pointers are restricted in what they can reference. In Figure 1, pointers A, B, and C are examples of pointers that can only refer to thread-local, node-local, and cluster-local locations, respectively, while D can point anywhere in the grid. The *width* of a pointer specifies what locations it can reference, with a higher width allowing further locations, as shown by the edge labels in Figure 1. Wider pointers consume more space and are more expensive to manipulate and access. For example, thread-local and node-local pointers could be represented

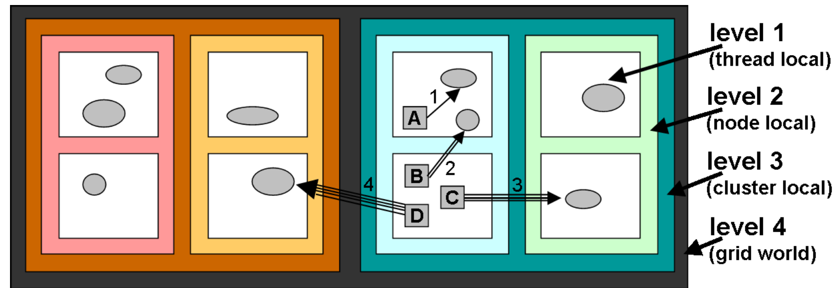

Fig. 1. A possible machine hierarchy with four levels. The width of arrows and their labels indicate the hierarchy distance between the endpoints.

simply by an address, while a cluster-local pointer contains an address and a node number. Wider pointers also have added costs to dereference, even if they happen to refer to nearby data; the pointer must be checked to see whether it is local, and coherence traffic or fences may be required to ensure the data is consistent with that viewed by other threads. The trend in hardware is towards more levels of hierarchy, and towards high costs between levels. Thus, software that can take advantage of the hierarchy is increasingly important.

2.2 Titanium

The Titanium programming language [28] 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 communication through the GASNet communication layer [6].

In addition to dereferencing, communication between threads can be done through the one-to-all *broadcast* and the all-to-all *exchange* operations. Program variables, including static variables, are not shared between threads, so they cannot be used for communication.

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

In the Titanium type system, variables are implicitly *global*, meaning that they can point to a location on any thread (level 3). A variable can be restricted to only point within a physical address space (level 2) by qualifying it with the `local` keyword. Downcasts between global and local are allowed and only succeed if the actual location referenced

is within the same physical address space as the executing thread. Our analysis takes advantage of existing such casts in a program in determining what variables must reference data in the same address space.

The Titanium type system does not separate levels 1 and 2 of the hierarchy. The distinction between 1 and 2 is important for many applications, such as race detection [21], data sharing analysis [17], and sequential consistency enforcement [14], since references to level 1 values on different threads cannot be to the same location. Other applications such as data locality inference [16] can benefit from the distinction between levels 2 and 3. Though we could perform a two-level analysis twice to obtain a three-level analysis, we show in §4.5 that the three-level analysis we have implemented is much more efficient.

3 Analysis Background

We define a machine[1] hierarchy and a simple language as the basis of our analysis. This allows the analysis to be applied to languages besides Titanium, and it avoids language constructs that are not crucial to the analysis. While the language we use is SPMD, the analysis can easily be extended to other models of parallelism, though we do not do so here.

3.1 Machine Structure

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 parallel program. 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 level of the hierarchy containing their least common ancestor. 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

3.2 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 [16]. 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. References thus can have any width in the range $[1, h]$.

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 (n)

[1] Throughout this paper, we will use *machine* interchangeably with *thread*.

n ::= integer literals
x ::= variables
τ ::= int | $\texttt{ref}_n\ \tau$ (types)
e ::= n | x | $\texttt{new}_l\ \tau$ | $*e$ | $\texttt{convert}(e, n)$
| $\texttt{transmit}\ e_1\ \texttt{from}\ e_2$ | $e_1; e_2$
| $x := e$ | $e_1 \leftarrow e_2$
(expressions)

Fig. 2. The syntax of the *Ti* language.

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

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

Fig. 3. Type manipulating functions.

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

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

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

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

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

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

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

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

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

Fig. 4. Type checking rules.

- variables (x). We assume a fixed set of variables of predefined type. We also assume that variables are machine-private.
- reference allocations ($\texttt{new}_l\ \tau$). The expression $\texttt{new}_l\ \tau$ allocates a memory cell of type τ and returns a reference to the cell. In order to facilitate the pointer analysis in §4, each allocation site is given a unique label l.
- dereferencing ($*e$)
- type conversions ($\texttt{convert}(e, n)$), which widen or narrow the width of an expression, converting its type from $\texttt{ref}_m\ \tau$ to $\texttt{ref}_n\ \tau$.
- communication ($\texttt{transmit}\ e_1\ \texttt{from}\ e_2$). In $\texttt{transmit}\ e_1\ \texttt{from}\ e_2$, machine e_2 evaluates the expression e_1 and sends the result to the other machines.
- sequencing ($e_1; e_2$)
- assignment to variables ($x := e$)
- assignment through references ($e_1 \leftarrow e_2$). 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. Since the analysis is flow-insensitive, conditionals are not essential to it.

The type checking rules for *Ti* are summarized in Figure 4. The rules for integer literals, variables, sequencing, and variable assignments are straightforward.

The allocation expression $\texttt{new}_l\ \tau$ produces a reference type $\texttt{ref}_1\ \tau$ of 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 5, where x on machine 0 refers to a location on machine 0 that refers to a

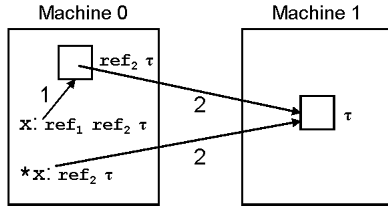

Fig. 5. Dereferences may require width expansion. The arrow labels correspond to pointer widths.

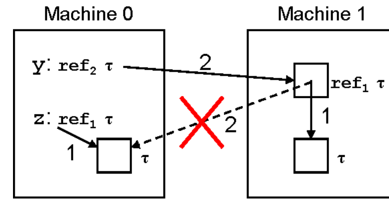

Fig. 6. The assignment $y \leftarrow z$ is forbidden, since the location referred to by y can only hold pointers of width 1 but requires a pointer of width 2 to refer to z.

location on machine 1. This 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 `convert` expression allows the top-level width of an expression to be up or downcast. Upcasts are rarely used due to the subtyping rule below. A programmer can use downcasts to inform the compiler that the reference is to data residing on a machine closer than the original width, and usually does so only after a dynamic check that this is the case. The resulting type is the same as the input expression, but with the provided top-level width.

In the `transmit` expression, 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 6. 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 the approach of Liblit and Aiken, [16], 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. These functions are shown in Figure 3.

3.3 Concrete Operational Semantics

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

We use the following semantic domains and naming conventions for their elements:

M (the set of machines)
$H = \{1, ..., h\}$ (the set of possible widths)
A (the set of local addresses)
Id (the set of identifiers)
N (the set of integer literals)
$Var = M \times Id$ (the set of variables)
L (the set of allocation site labels)
T (the set of all types)
$G = L \times M \times A$ (the set of global addresses)
$V = N \cup G$ (the set of values)
$Store = (G \cup Var) \rightarrow V$ (the contents of memory)
Exp (the set of all expressions)

$m \in M$ (a machine)
$v \in V$ (a value)
$\sigma \in Store$ (a memory state)
$a \in A$ (a local address)
$l \in L$ (a label)
$g = (l, m, a) \in G$ (a global address)
$e \in Exp$ (an expression)

Judgments in our operational semantics have the form $\langle e, m, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$, 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 $\lambda x.\ \texttt{if}\ x = g\ \texttt{then}\ v\ \texttt{else}\ \sigma(x)$.

The rules for integer and variable expressions are trivial.

$$\frac{}{\langle n, m, \sigma \rangle \Downarrow \langle n, \sigma \rangle} \qquad \frac{}{\langle x, m, \sigma \rangle \Downarrow \langle \sigma(x), \sigma \rangle}$$

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

$$\frac{}{\langle new_l\ \tau, m, \sigma \rangle \Downarrow \langle (l, m, a), \sigma[(l, m, a) := null] \rangle}\ (a \text{ is fresh on } m)$$

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

$$\frac{\langle e, m, \sigma \rangle \Downarrow \langle g, \sigma' \rangle \quad g \neq null}{\langle *e, m, \sigma \rangle \Downarrow \langle \sigma'(g), \sigma' \rangle}$$

The rule for variable assignment is also simple.

$$\frac{\langle e, m, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle x := e, m, \sigma \rangle \Downarrow \langle v, \sigma'[x := v] \rangle}$$

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

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

The rule for sequencing is as expected.

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

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.

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

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

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

4 Abstract Interpretation

We now present a pointer analysis for the *Ti* language. 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 memory location and variable on machine m. The concrete state thus maps each memory location and variable to a set of memory locations, and it is 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 an allocation site and a hierarchy width. An element a_1 of A_m is subsumed by another element a_2 if a_1 and a_2 have the same allocation site, and a_2 has a higher width than a_1. The elements of A_m are thus ordered by the following relation:

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

The ordering thus has height h.

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

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

In other words, S is redundant if it contains two related elements of A_m, such that one subsumes the other.

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

$$u(i) = \begin{cases} j & \text{if } (l_i, j) \in S, \\ 0 & \text{otherwise.} \end{cases}$$

The vector has a digit for each allocation site, and the value of the digit is the width of the abstract location in S corresponding to the site, or 0 if there is none.

We use the following Hoare ordering on elements of R:

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

The element S_1 is subsumed by S_2 if every element in S_1 is subsumed by some element in S_2. In the vector representation, the following is an equivalent ordering:

$$S_1 \sqsubseteq S_2 \iff \forall i \in \{1, ..., |L|\}.\, u_1(i) \leq u_2(i)$$

In this representation, S_1 is subsumed by S_2 if each digit in S_1 is no more than the corresponding digit in S_2. The ordering relation induces a lattice with minimal element corresponding to $u_\bot(i) = 0$, and a maximal element corresponding to $u_\top(i) = h$. The maximal chain between $\bot$ and $\top$ is derived by increasing a single vector digit at a time by 1, so the chain, and therefore the lattice, has height $h \cdot |L| + 1$.

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

$$\gamma_m(S) = \big\{(l, m', a) \bigm| (l, n) \in S \ \wedge \ hier(m, m') \leq n\big\}$$
$$\alpha_m(C) = \sqcap\big\{S \bigm| C \sqsubseteq \gamma_m(S)\big\}$$

The concretization of an abstract location (l, n) with respect to machine m is the set of all concrete locations with the same allocation site and located on machines that are at most n away from m. The abstraction with respect to m of a concrete location (l, m', a) is an abstract location with the same allocation site and width equal to the distance between m and m'.

Finally, we abstract the concrete domain CS to the following abstract domain, which maps abstract locations and variables to *points-to sets* of abstract locations:

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

An element σ_A of AS is subsumed by σ'_A if the points-to set of each abstract location and variable in σ_A is subsumed by the corresponding set in σ'_A. The elements of AS are therefore ordered 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 $\mathrm{O}(h \cdot |L| \cdot (|A_m| + |Id|)) = \mathrm{O}(h \cdot |L| \cdot (h \cdot |L| + |Id|))$. Since the number of allocation sites and identifiers is limited by the size of the input program P, the height is in $\mathrm{O}(h^2 \cdot |P|^2)$.

4.3 Abstract Semantics

For each expression in *Ti*, we provide inference rules for how the expression updates the abstract state σ_A. The judgments are of the form $\langle e, \sigma_A \rangle \Downarrow \langle S, \sigma'_A \rangle$, 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. As in §3.3, we use the notation $\sigma[g := v]$ to denote the function $\lambda x.\ \texttt{if}\ x = g\ \texttt{then}\ v\ \texttt{else}\ \sigma(x)$. 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.

$$\frac{}{\langle n, \sigma_A \rangle \Downarrow \langle \emptyset, \sigma_A \rangle} \qquad \frac{}{\langle x, \sigma_A \rangle \Downarrow \langle \sigma_A(x), \sigma_A \rangle}$$

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

$$\frac{}{\langle new_l\ \tau, \sigma_A \rangle \Downarrow \langle \{(l, 1)\}, \sigma_A \rangle}$$

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

$$\frac{\langle e, \sigma_A \rangle \Downarrow \langle S, \sigma'_A \rangle}{\langle *e, \sigma_A \rangle \Downarrow \langle \bigcup_{b \in S} \sigma'_A(b), \sigma'_A \rangle}$$

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

$$\frac{\langle e_1, \sigma_A \rangle \Downarrow \langle S_1, \sigma'_A \rangle \quad \langle e_2, \sigma'_A \rangle \Downarrow \langle S_2, \sigma''_A \rangle}{\langle e_1; e_2, \sigma_A \rangle \Downarrow \langle S_2, \sigma''_A \rangle}$$

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

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

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.

$$\frac{\langle e, \sigma_A \rangle \Downarrow \langle S, \sigma'_A \rangle}{\langle \texttt{convert}(e, n), \sigma_A \rangle \Downarrow \langle \{(l, min(k, n)) \mid (l, k) \in S\}, \sigma'_A \rangle}$$

The SPMD model of parallelism in *Ti* implies that the source expression of the `transmit` operation evaluates to abstract locations with the same labels on both the source and destination machines. The distance between the source and destination machines, however, is not statically known, so the resulting abstract locations must be assumed to have the maximum width.

$$\frac{\langle e_2, \sigma_A \rangle \Downarrow \langle S_2, \sigma'_A \rangle \quad \langle e_1, \sigma'_A \rangle \Downarrow \langle S_1, \sigma''_A \rangle}{\langle \texttt{transmit}\ e_1\ \texttt{from}\ e_2, \sigma_A \rangle \Downarrow \langle \{(l, h) \mid (l, m) \in S_1\}, \sigma''_A \rangle}$$

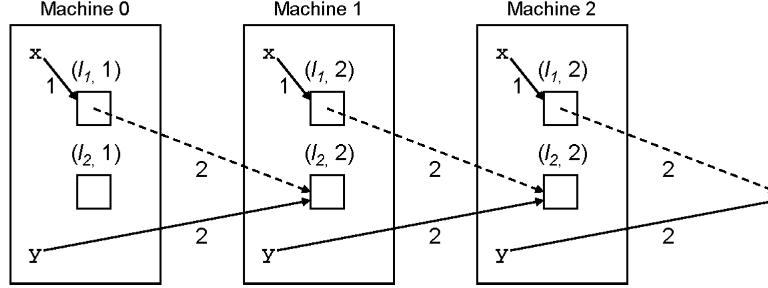

Fig. 7. The assignment $x \leftarrow y$ on machine 0 results in the abstract location $(l_2, 2)$ being added to the points-to set of $(l_1, 1)$, as shown by the first dashed arrow. The assignment on machine 1 results in the abstract location $(l_2, 2)$ being added to the points-to set of $(l_1, 2)$, 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 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. As we show below, the proper update is to add the location $(l_2, max(n'_1, n_1, n_2))$ to the points-to set of each location (l_1, n'_1). The rule is then

$$\frac{\langle e_1, \sigma_A \rangle \Downarrow \langle S_1, \sigma'_A \rangle \quad \langle e_2, \sigma'_A \rangle \Downarrow \langle S_2, \sigma''_A \rangle}{\langle e_1 \leftarrow e_2, \sigma_A \rangle \Downarrow \langle S_2, update(\sigma''_A, S_1, S_2) \rangle},$$

with $update$ defined as

$$\begin{aligned} & update(\sigma, S_1, S_2) = \\ & \quad \lambda(l_1, n'_1) : L \times H \; . \\ & \qquad \sigma((l_1, n'_1)) \sqcup \big\{ (l_2, max(n'_1, n_1, n_2)) \,\big|\, (l_1, n_1) \in S_1 \;\wedge\; (l_2, n_2) \in S_2 \big\}. \end{aligned}$$

4.4 Soundness

An abstract intepretation is sound if the abstraction and concretization functions are monotonic and form a Gallois connection, and the abstract inference rules for each operation is correct. The former condition was shown in §4.2.

Most of the abstract inference rules are derived directly from the operational semantics, so their correctness is obvious. The rule for assignment through a reference, however, is nontrivial, so we prove its correctness here.

Let a_i^m represent the abstract location a_i with respect to machine m. Let n^m represent a width n with respect to m.

Consider an assignment $e_1 \leftarrow e_2$. Let m be the reference machine for the analysis. Without loss of generality, assume that e_1 evaluates to the lone abstract location $a_1^m = (l_1, n_1^m)$, and that e_2 evaluates to $a_2^m = (l_2, n_2^m)$. Consider the execution of this assignment on the following machines:

– On machines m' such that $hier(m, m') \leq n_1^m$. This implies that the $(n_1^m - 1)$th ancestor of each m' in the machine hierarchy is the same as that of m. As a result, abstract locations of width at least n_1 are the same with respect to both m and m'. In particular, $a_1^{m'} = a_1^m$, so the assignment on any machine can target any concrete location in a_1^m.
  Now suppose $n_2^m < n_1^m$. Then the $a_2^{m'}$ are not equivalent for all machines m'. However, note that $a_2^{m'}$ contains the concrete locations (l_2, m', a) for any a. Considering the assignment on all machines m', the concrete locations in a_1^m can receive any of the source concrete locations (l_2, m', a) for all m' and a. This set of source locations corresponds exactly to the abstract location $a_{2'}^m = (l_2, n_1^m)$.
  Suppose instead that $n_2^m \geq n_1^m$. Then the machines m' all agree on the set $a_2^{m'} = a_2^m$. Thus, regardless of which machine the assignment is executed on, the source locations correspond exactly to a_2^m.
  In either case, any of the concrete locations corresponding to a_1^m can now point to any of the concrete locations corresponding to $a_{2'}^m = (l_2, max(n_1^m, n_2^m))$. To capture this in the abstract inference, it is sufficient to add $a_{2'}^m$ to the points-to set of a_1^m. For consistency, $a_{2'}^m$ should also be added to the points-to set of any abstract location $a_{1'}^m \sqsubseteq a_1^m$, since any of the concrete locations corresponding to $a_{1'}^m$ can point to any of the concrete locations corresponding to $a_{2'}^m$.
  Thus, it is sufficient to add the abstract location $a_{2'}^m = (l_2, max(n_1^m, n_2^m))$ to the points-to set of any $a_{1'}^m = (l_1, n_{1'}^m)$ such that $n_{1'}^m \leq n_1^m$.
– On a machine m', where $hier(m, m') > n_1^m$. The set of concrete locations corresponding to $a_1^{m'}$ all reside on machines a distance of $n_{1'}^m = hier(m, m')$ away from machine m. Thus, $a_1^{m'} \sqsubseteq a_{1'}^m$, where $a_{1'}^m = (l_1, n_{1'}^m)$.
  Now suppose $n_2^m < n_{1'}^m$. Then all the concrete locations corresponding to $a_2^{m'}$ reside at a distance of $n_{1'}^m$ from machine m, so that $a_2^{m'} \sqsubseteq a_{2'}^m$, where $a_{2'}^m = (l_2, n_{1'}^m)$. Thus, the source locations can be soundly approximated by $a_{2'}^m$.
  Suppose instead that $n_2^m \geq n_{1'}^m$. Then m and m' agree on $a_2^{m'} = a_2^m$, so the source locations correspond to a_2^m.
  In either case, some of the concrete locations corresponding to $a_{1'}^m$ can now point to some of the concrete locations corresponding to $a_{2'}^m = (l_2, max(n_{1'}^m, n_2^m))$. Soundness can be maintained, though precision lost, if the analysis assumes that any concrete location corresponding to $a_{1'}^m$ can point to any concrete location corresponding to $a_{2'}^m$. Thus, $a_{2'}^m$ should be added to the points-to set of $a_{1'}^m$.
  Now consider an abstract location $a_{1''}^m = (l_1, n_{1''}^m)$, where $n_{1''}^m < n_{1'}^m$. All concrete locations represented by $a_{1''}^m$ reside less than a distance of $n_{1'}^m$ away from m. Since

all concrete locations corresponding to $a_1^{m'}$ reside at a distance of $n_{1'}^m$ from m, the abstract locations $a_{1''}^m$ and $a_1^{m'}$ do not intersect. Thus, none of the concrete locations in $a_{1''}^m$ are targeted by the assignment, so its points-to set does not need to be updated.
Thus, it is sufficient to add the abstract location $a_{2'}^m = (l_2, max(n_{1'}^m, n_2^m))$ to the points-to set of each $a_{1'}^m = (l_1, n_{1'}^m)$ such that $n_{1'}^m > n_1^m$.

Summarizing over all possibilities, we obtain the rule that the abstract location $a_{2'}^m = (l_2, max(n_{1'}^m, n_1^m, n_2^m))$ is to be added to the points-to set of any $a_{1'}^m = (l_1, n_{1'}^m)$. This corresponds exactly to the update rule provided in §4.3.

4.5 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 least upper bound operation involving the input state. As a result, F is a monotonically 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[3], where P is the input program. Since the lattice over AS has height in $\mathrm{O}(h^2 \cdot |P|^2)$, it takes time in $\mathrm{O}(h^2 \cdot |P|^3)$ to compute the fixed point of F.

In our implementation, we have found that the running time of the analysis varies little between one, two, and three levels of hierarchy. For the benchmarks in §5, a three-level analysis takes no more than 10% longer than a single-level analysis and less than 5% longer than a two-level analysis. Thus, the three-level analysis is far more efficient than running a two-level analysis twice.

5 Evaluation

The pointer information computed in §4 can be applied to multiple analyses and optimizations for parallel programs. We evaluate the pointer analysis by using it for race detection. In [13], we apply it as well to enforcement of sequential consistency and describe how it can be used to infer data locality and privacy.

We use the following set of benchmarks:

- **`amr`** [27] (7581 lines) Chombo adaptive mesh refinement suite [3] in Titanium.
- **`gas`** [5] (8841 lines): Hyperbolic solver for a gas dynamics problem in computational fluid dynamics.
- **`ft`** [9] (1192 lines): NAS Fourier transform benchmark [4] in Titanium.
- **`cg`** [9] (1595 lines): NAS conjugate gradient benchmark [4] in Titanium.
- **`mg`** [9] (1952 lines): NAS multigrid benchmark [4] in Titanium.

[2] Since the analysis is flow-insensitive, the order of application is not important.
[3] We ignore the cost of the join operations here. In practice, points-to sets tend to be small, so the cost of joining them can be neglected.

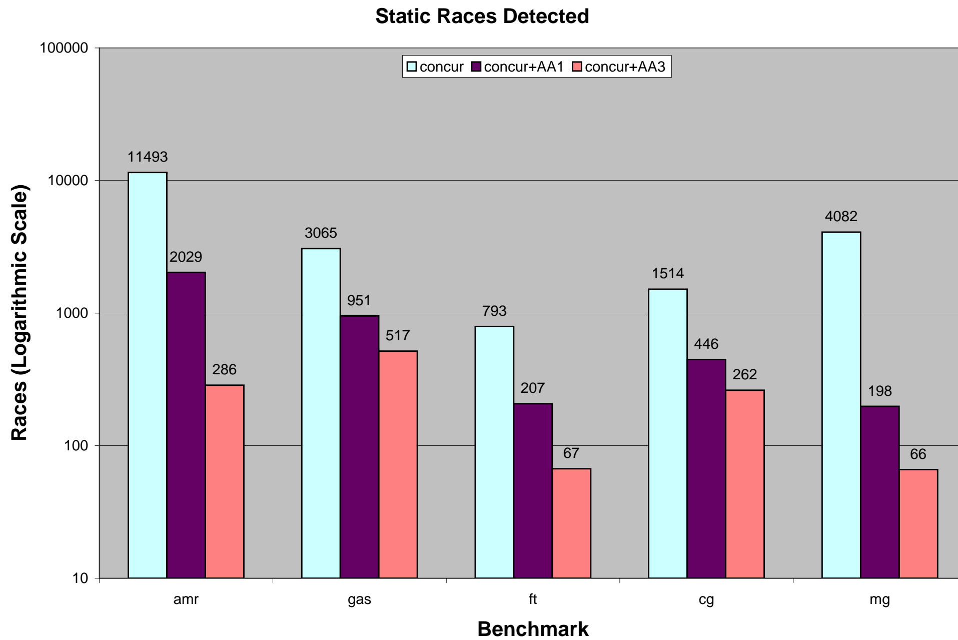

Fig. 8. Number of data races reported for different levels of analysis.

The line counts for the above benchmarks underestimate the amount of code actually analyzed, since all reachable code in the 37,000 line Titanium and Java 1.0 libraries is also processed.

A race condition occurs when two memory accesses can occur simultaneously on different threads, they can be to the same memory location, and at least one is a write. An existing concurrency analysis for Titanium [15] can conservatively determine which accesses are simultaneous. The pointer analysis can detect if two accesses may be to the same location by checking if they can operate on abstract locations whose concretizations with respect to different machines overlap. In a single-level analysis, all abstract locations with the same label overlap, while in a multi-level analysis, they do not overlap if they are both machine-local (i.e. have width 1). Thus, a multi-level analysis results in higher race detection precision than a single-level analysis.

Static information is generally not enough to determine with certainty that two memory accesses compose a race, so nearly all reported races are false positives. (The correctness of the concurrency and pointer analyses ensure that no false negatives occur.) We therefore consider a race detector that reports the fewest races to be the most effective

Figure 8 compares the effectiveness of three levels of race detection:

– **concur**: Our concurrency analysis[4] [15] is used to eliminate non-concurrent memory accesses. Sharing inference [17] is used to eliminate accesses to thread-private data.
– **concur+AA1**: A single-level pointer analysis is added to eliminate false aliases.
– **concur+AA3**: A three-level pointer analysis is added to eliminate false aliases.

The results show that the pointer analysis can eliminate most of the races reported by our detector. The addition of pointer analysis removes most of the races discovered by only using the concurrency analysis, with a three-level analysis providing significant benefits over a one-level analysis. However, the results are still not precise enough for production use. The pointer analysis does not currently distinguish between array indices, and since Titanium programs tend to make extensive use of arrays in their data structures, this results in a significant number of false aliases. However, the addition of an array index analysis [20, 19, 18, 22] should remove most of these false aliases, and consequently most of the false positives reported by the race detector.

6 Related Work

The language and type system we presented here are generalizations of those described by Liblit and Aiken [16]. They defined a two-level hierarchy and used it to produce a constraint-based analysis that infers locality information about pointers. Later with Yelick, they extended the language and type system to consider sharing of data, and they defined another constraint-based analysis to infer sharing properties of pointers [17].

Pointer analysis was first described by Andersen [2], and later extended by others to parallel programs. Rugina and Rinard developed a thread-aware alias analysis for the Cilk multithreaded programming language [23] that is both flow-sensitive and context-sensitive. Others such as Zhu and Hendren [29] and Hicks [11] have developed flow-insensitive versions for multithreaded languages. However, none of these analyses consider hierarchical, distributed machines.

The pointer analysis we presented here is a generalization and formalization of the analysis sketched in a previous paper [14]. 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 an almost complete implementation was done.

7 Conclusion

In this paper, we introduced a program analysis technique for pointers, which has applications in detecting program errors and enabling optimizations. The novelty of the analysis derives from its view of the machine as an arbitrary hierarchy of processors, with the analysis proving that the range of a pointer is limited to a given hierarchy.

Our analysis was presented on a small language, *Ti*, which decouples the analysis from specifics of the language. The type system allows for references of different

[4] The most precise analysis in [15] is used, which was labeled as *feasible* in that paper.

widths, corresponding to local and global pointers in PGAS languages. We demonstrated the analysis with an implementation in the Titanium language, a global address space language with three levels of hierarchy. Our results show that the multi-level analysis is significantly more accurate than one based on only a single level.

There are several potential clients of our analysis, and in this paper we presented one such client, a static race detection algorithm, which combined the pointer analysis with our existing concurrency analysis to detect races in Titanium programs. Even on relatively complicated benchmarks codes, our results show that the more accurate pointer analysis has a significant impact on the quality of the race analysis. Our results indicate the value of exposing the hierarchy within the language and compiler to balance the desire of programmers for both simplicity and high performance.

References

1. E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification, Version 0.866*. Sun Microsystem Inc., Feb. 2006.
2. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
3. Applied Numerical Algorithms Group (ANAG). Chombo. `http://seesar.lbl.gov/ANAG/software.html`.
4. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
5. M. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64–84, May 1989. Lawrence Livermore Laboratory Report No. UCRL-97196.
6. D. Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, University of California, Berkeley, November 2002.
7. W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.
8. Cray Inc. *Chapel Specification 0.4*, Feb. 2005.
9. K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2005.
10. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.
11. 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.
12. P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. Technical Report UCB/CSD-04-1163-x, University of California, Berkeley, September 2004.
13. A. Kamil. Analysis of Partitioned Global Address Space Programs. Master's thesis, University of California, Berkeley, December 2006.
14. A. Kamil, J. Su., and K. Yelick. Making sequential consistency practical in Titanium. In *Supercomputing 2005*, November 2005.

15. A. Kamil and K. Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, October 2005.
16. B. Liblit and A. Aiken. Type systems for distributed data structures. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2000.
17. B. Liblit, A. Aiken, and K. Yelick. Type systems for distributed data sharing. In *International Static Analysis Symposium*, San Diego, California, June 2003.
18. Y. Lin and D. A. Padua. Analysis of irregular single-indexed array accesses and its applications in compiler optimizations. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 202–218, London, UK, 2000. Springer-Verlag.
19. D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array-data flow analysis and its use in array privatization. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 2–15, New York, NY, USA, 1993. ACM Press.
20. D. E. Maydan, S. Amarsinghe, and M. S. Lam. Data dependence and data-flow analysis of arrays. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 434–448, London, UK, 1993. Springer-Verlag.
21. R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.
22. Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM Trans. Program. Lang. Syst.*, 24(1):65–109, 2002.
23. 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.
24. V. Saraswat. *Report on the Experimental Language X10, Version 0.41*. IBM Research, Feb. 2006.
25. Silicon Graphics. CF90 co-array programming manual. Technical Report SR-3908 3.1, Cray Computer, 1994.
26. The UPC Consortium. *UPC Language Specifications, Version 1.2*, May 2005.
27. T. Wen and P. Colella. Adaptive mesh refinement in titanium. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.
28. 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.
29. 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.