Type Systems for Distributed Data Sharing [⋆]

Ben Liblit, Alex Aiken, and Katherine Yelick

University of California, Berkeley
Berkeley, CA 94720-1776

Abstract. Parallel programming languages that let multiple processors access shared data provide a variety of sharing mechanisms and memory models. Understanding a language's support for data sharing behavior is critical to understanding how the language can be used, and is also a component for numerous program analysis, optimization, and runtime clients. Languages that provide the illusion of a global address space, but are intended to work on machines with physically distributed memory, often distinguish between different kinds of pointers or shared data. The result can be subtle rules about what kinds of accesses are allowed in the application programs and implicit constraints on how the language may be implemented. This paper develops a basis for understanding the design space of these sharing formalisms, and codifies that understanding in a suite of type checking/inference systems that illustrate the trade-offs among various models.

1 Introduction

Parallel, distributed, and multithreaded computing environments are becoming increasingly important, but such systems remain difficult to use and reason about. Data sharing (the ability of multiple threads to hold references to the same object) is one source of difficulty. Programming languages such as Java, Titanium, and UPC offer facilities for sharing data that vary in subtle, often implicit, but semantically significant ways.

We take a type-based approach to characterizing data sharing in distributed programming environments. This paper makes four principal contributions:

- We show that there is an essential difference between distributed pointers, which may be either *local* (within one address space) or *global* (across address spaces) and the patterns of access to data, which may be either *private* (used by one processor) or *shared* (used by many processors) [Section 2]. Earlier efforts have not clearly distinguished these two concepts [Section 4].
- We show that there is more than one notion of data sharing, and that various sharing models can be captured in a suite of type systems [Sections 3.1 through 3.3].
- We show that type qualifier inference can automatically add detailed sharing information to an otherwise unannotated program, and that such an approach can be used with realistic distributed programming languages [Section 3.4].

[⋆] This research was supported in part by NASA under NAG2-1210; NSF under EIA-9802069, CCR-0085949, and ACI-9619020; DOE under DE-FC03-01ER25509; and an NDSEG fellowship. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

$$\begin{array}{lll lll}
i & ::= & \text{integer literal} & e & ::= & i \mid x \mid f\ e \mid \uparrow_{\texttt{shared}} e \mid \uparrow_{\texttt{private}} e \mid \downarrow e \\
x & ::= & \text{program variable} & & \mid & e\ \texttt{;}\ e \mid e \texttt{ := } e \mid \langle e, e\rangle \mid \texttt{@1}\ e \mid \texttt{@2}\ e \\
f & ::= & \text{function name} & & \mid & \texttt{widen}\ e \mid \texttt{transmit}\ e
\end{array}$$

Fig. 1. Common grammar for expressions

– We report on the results of adding sharing inference to a real compiler for a parallel language, which highlights the strengths of our approach. We also present unexpected results on the effect of the underlying memory consistency model on program performance. Our experience in this area may be of independent interest [Section 5].

2 Background

Parallel applications with a distributed address space have two distinct notions of data: whether a pointer is *local* or *global* and whether an object is *private* or *shared*. Previous work has not brought out the distinction between these two ideas (see Section 4). Our primary thesis is that these should be separate concepts. This section explains what these two ideas are and why they are distinct.

Figure 1 introduces a small data manipulation language of pointers and pairs. This language extends one used by Liblit and Aiken [21] with new features to capture data sharing behavior. For the sake of brevity, we omit a detailed formal semantics and soundness proof. The semantics is unchanged from prior work, as our extensions here serve only to restrict which programs are admitted by the type system without changing how accepted programs behave at run time. Soundness is addressed briefly at the close of Section 3.3, and follows as a straightforward adaptation of a more complete proof previously published elsewhere for a related system [22].

A base expression may be an integer literal (i) or a named variable from some predefined environment (x). Function calls ($f\ e$) similarly assume that the function (f) is predefined. A sequencing operator ($e\ \texttt{;}\ e$) provides ordered evaluation. The language has no facilities for defining functions, new types, or recursion. These features are unnecessary to our exposition, though our techniques readily extend to realistic languages.

For our purposes, the essential aspects of the language are data structures (modeled by pairs) and pointers. Data may be combined into pairs ($\langle e, e\rangle$), which are unboxed (held directly as an immediate, flat-structured value, with no intervening levels of indirection). Unboxed pairs and integers may be stored in memory using the allocation operators ($\uparrow_\delta e$), which return boxed values stored in the local processor's memory (i.e., allocation is always on the local processor). The subscript δ states whether a boxed value is to be *shared* or *private*. Informally, a private value can be accessed only by the processor that allocated it. Thus, "$\uparrow_{\texttt{private}}\langle 5, 7\rangle$" produces a pointer to a private memory cell holding the pair (5, 7). In contrast, shared values can be examined and manipulated by multiple processors across the entire system.

If x holds the value of $\uparrow_{\texttt{private}}\langle 5,7\rangle$, then "$\downarrow x$" is the unboxed pair $\langle 5,7\rangle$. Each pair selection operator ($\texttt{@1}\ e$ or $\texttt{@2}\ e$) accepts a pointer to a boxed pair, and produces an offset pointer to the first or second component of the pair respectively. Thus, if x holds the value of $\uparrow_{\texttt{private}}\langle 5,7\rangle$, then "$\texttt{@2}\ x$" yields a pointer to the second component within the boxed pair, and "$\downarrow \texttt{@2}\ x$" yields the unboxed value 7.

The left operand of an assignment $e \texttt{ := } e$ must be a pointer; the value given on the right is placed in the memory location named by the pointer given on the left. The explicit treatment of boxing and offset pointers allows us to model the update of components of data structures. For example, the expression

$$(\texttt{@2}\ \uparrow_{\texttt{private}}\langle 5,7\rangle) \texttt{ := } 9$$

modifies the allocated pair from $\langle 5,7\rangle$ to $\langle 5,9\rangle$.

The remaining constructs model distributed memory systems. Allocation is local: $\uparrow_\delta e$ always produces a *local* pointer. A local pointer names memory local to a single processor, and corresponds in practice to a simple memory address. It does not make sense to transfer local pointers between processors; for that we need a *global* pointer, which names any location in the entire distributed system. This is akin to a $(processor, address)$ pair, where *processor* uniquely identifies one processor in the complete distributed system and *address* is a memory address within that processor's local address space. Local pointers are preferred when data is truly local, as global pointers may be both larger as well as slower to use. The widen operator ($\texttt{widen}\ e$) provides coercions from local to global pointers. For clarity we make these coercions explicit. In practice, coercions would be inserted automatically by the compiler.

The transmission operator ($\texttt{transmit}\ e$) models sending a value to some other machine. Transmission does not implicitly dereference pointers or serialize interlinked data structures. If e evaluates to an unboxed tuple, then both components of the tuple are sent; if e evaluates to a pointer, then just a single pointer value (not the pointed-to data) is sent. We intentionally leave the communication semantics unspecified; `transmit` might correspond to a network broadcast, a remote procedure invocation, or any other cross-machine data exchange. The important invariant is that $\texttt{transmit}\ e$ must produce a representation of e that can be used safely on remote processors. For this reason, it is typical to transmit widened values. Transmitting a local pointer without widening is forbidden, as a local pointer value is meaningless on any but the originating processor.

2.1 Representation Versus Sharing

Consider the expression

$$x \texttt{ := } \uparrow_{\texttt{private}} e$$

The referent of x is supposed to be private: no other processor should ever have access to the value of e. Thus, all pointers to e, including x, can use the cheaper local pointer representation. In a single-threaded program, all data is both local and private.

Now consider

$$x \texttt{ := } \texttt{transmit}\ (\texttt{widen}\ (\uparrow_{\texttt{shared}} e))$$

Here the value of e is declared to be `shared` and a global pointer to this value is transmitted to a remote processor (or processors), where it is stored in a variable x.

For example, if `transmit` broadcasts the value computed on processor 0, then each running processor receives a pointer to processor 0's value of e and stores this in its own instance of the variable x. On each remote processor, then, x is a global pointer to the same piece of shared data.

Finally, consider the following minor variation on the last example:

$$y \mathtt{:=} \uparrow_{\mathtt{shared}} e;$$
$$x \mathtt{:=} \mathtt{transmit}\ (\mathtt{widen}\ (y))$$

As before, x is a global pointer that points to shared data. But what is y? It points to shared data, but y is on the same processor as e. Thus, y should be a local pointer to shared data.

It is this last case that distinguishes `local`/`global` from `shared`/`private`. The distinction is that `local`/`global` determines the representation of pointers, while the `shared`/`private` determines how data is used. As illustrated in the examples above, a local pointer may point either to shared or private data. Just having a local pointer does not tell us whether concurrent access from other processors is possible.

While `local`/`global` is distinct from `shared`/`private`, these two concepts are not quite orthogonal. In particular, global pointers to private data are problematic: What could it mean for a processor to hold a pointer to data that is supposed to be private to another processor? As we show in Section 3, there are multiple possible answers to this question.

2.2 Uses of Sharing Information

In later sections we show how to statically determine which data is private to a single processor versus shared by multiple processors. Such information can support a number of clients. *Autonomous garbage collection* can reclaim private data as a strictly local operation without coordinating with other processors [33]. *Data location management* can be important when hardware constraints make shared memory a limited resource [26]. *Cache coherence overhead* [1] can be avoided for private data that only one processor ever sees. *Race condition detection* [17, 24, 32] need never consider races on private data. *Program/algorithm documentation* can be augmented by compiler validation of programmers' claims. *Consistency model relaxation* allows more aggressive optimizations on data that other processors cannot see [25]. *Synchronization elimination* boosts performance for private monitors that can never come under contention [3, 7]. *Security* mandates careful treatment of private data in distributed systems without mutual trust.

Each of these clients depends upon identifying data accessed by only one processor. Typically, this processor is the one in whose local memory the data lives; the data is accessed by way of local pointers. However, local pointers alone do not suffice, because of the possibility that a global pointer may alias the same location. Furthermore, even if the data itself is only referenced by local pointers, transitive reachability is still a concern: if we have a global pointer to a local pointer to a local pointer to the value 5, the memory cell containing 5 could still be accessed by a remote processor via a sequence of dereference operations that widen the local pointers to global on each dereference.

$$\tau ::= \texttt{int} \mid \langle \tau, \tau \rangle \mid \texttt{boxed}\, \omega\, \delta\, \tau$$
$$\omega ::= \texttt{local} \mid \texttt{global}$$
$$\delta ::= \texttt{shared} \mid \texttt{mixed} \mid \texttt{private}$$

(a) Grammar for types

$$\texttt{shared} < \texttt{mixed} \qquad \texttt{private} < \texttt{mixed}$$
$$\texttt{boxed}\, \omega\, \delta\, \tau \leq \texttt{boxed}\, \omega\, \delta'\, \tau \iff \delta \leq \delta'$$
$$\langle \tau_1, \tau_2 \rangle \leq \langle \tau_1', \tau_2' \rangle \iff \tau_1 \leq \tau_1' \wedge \tau_2 \leq \tau_2'$$

(b) Subtyping relations

Fig. 2. Common properties of all type systems

Again, pointer representations are not the same as data sharing patterns, and the latter cannot be trivially deduced from the former.

In Section 3 we show that some clients (in particular, autonomous GC and security) require stronger privacy guarantees than others. This suggests a solution based not on a single definition of privacy, but rather on a family of alternatives from which each client can select according to its needs.

3 Type Systems

Figure 2 presents the types and the subtyping relation used in the following subsections. The basic types are unboxed integers `int` and unboxed pairs $\langle \tau, \tau \rangle$. Pointers `boxed` $\omega\ \delta\ \tau$ to values of type τ also carry qualifiers ω and δ, which respectively range over $\{\texttt{local}, \texttt{global}\}$ and $\{\texttt{shared}, \texttt{mixed}, \texttt{private}\}$.

A local pointer may be widened into an equivalent global pointer. However, local and global pointers have distinct physical representations and are manipulated using very different machine-level operations. Therefore, widening is a coercion, not a subtyping relation, which is why `widen` is included as an explicit operator in the language in Figure 1. On the other hand, sharing qualifiers merely dictate which remote operations are permissible and which are forbidden. In general, shared and private pointers can have identical physical representation and can be manipulated (where allowed at all) by identical machine-level operations.

However, neither coercion nor simplistic subtyping is appropriate between shared and private pointers. Shared pointers have functionality that private pointers do not: they can be widened to `global` and used at a distance. Private pointers have unique functionality of their own: they admit aggressive optimization that could violate language invariants if observed by remote processors.

Furthermore, any sound type system must ensure that sharing qualifiers are consistent across aliasing: local and global pointers may address the same location simultaneously, but no location may ever be considered to be both shared and private.

There are, however, good reasons to allow code that operates on either shared or private data. For example, consider the type of “`this`” in the `Object()` constructor of a Java-like language. As observed earlier, there is no coercion from `private` to `shared` or `shared` to `private`. If `this` is `shared` in the constructor then it can never be called to construct a private object, while if `this` is `private`, then no shared object may ever be built. Since every other constructor for every other class ultimately calls `Object()`, any restrictions introduced here affect the entire system.

Reuse of constructor code, then, requires polymorphism. We find in Section 5.1 that basic utility code is also reused in many contexts and therefore imposes similar requirements on the type system. In this paper we use subtyping polymorphism with a "`mixed`" sharing qualifier.[1] A `mixed` datum may be either `shared` or `private`; code that manipulates `mixed` data may do so only in ways sound for both. The subtyping relation is defined in Figure 2(b). Note that subtyping does not cross pointers; this restriction is necessary to avoid the well-known unsoundness problems that would result with subtyping updatable references.

3.1 Late Enforcement

If we define "private" to mean neither read nor written by a remote processor, then global pointers to private data may be freely created, but they cannot be dereferenced. This section presents such a *late enforcement* system.

Typing judgments "$A \vdash e : \tau$" are read "In environment A, it is provable that expression e has type τ." The auxiliary expand and pop functions defined in Figure 3 describe how types are transformed or constrained by widening and cross-processor communication. Observe that type expansion recursively descends into pairs but never crosses pointers; we expand only the immediate value directly being communicated across processor boundaries. Integers and global pointers are the same everywhere. Local pointers expand to global only at the topmost level of a type (expand), but are banned from appearing within expanding pairs (pop).

(Global pointers are, in general, larger than local pointers. Therefore, expanding local pointers to global pointers inside a pair would change the size and layout of the pair. This is undesirable in practice for languages where each named structure type (class) must have a single, consistent layout. An alternative would be to allow deep local pointers to remain the same size, but mark them as invalid for use by the remote processor. This possibility is explored in greater detail elsewhere [21], and is omitted here for simplicity.)

Types for integers, variables, and function applications are completely standard. Sequencing, pair construction, and subtyping are also given in the typical manner:

$$\frac{}{A \vdash i : \texttt{int}} \qquad \frac{A(x) = \tau}{A \vdash x : \tau} \qquad \frac{A(f) = \tau \rightarrow \tau' \quad A \vdash e : \tau}{A \vdash f\, e : \tau'}$$

$$\frac{A \vdash e : \tau \quad A \vdash e' : \tau'}{A \vdash e \,\texttt{;}\, e' : \tau'} \qquad \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2}{A \vdash \langle e_1, e_2 \rangle : \langle \tau_1, \tau_2 \rangle}$$

$$\frac{A \vdash e : \tau \quad \tau \leq \tau'}{A \vdash e : \tau'}$$

Shared or private allocation creates an appropriately qualified local pointer:

[1] An alternative is to use parametric polymorphism, which is more expressive, but subtype polymorphism is a little simpler to explain.

$$\begin{aligned}
\mathit{expand}(\texttt{boxed}\ \omega\ \delta\ \tau) &\equiv \texttt{boxed}\ \texttt{global}\ \delta\ \tau \\
\mathit{expand}(\langle \tau_1, \tau_2 \rangle) &\equiv \langle \mathit{pop}(\tau_1), \mathit{pop}(\tau_2) \rangle \\
\mathit{expand}(\texttt{int}) &\equiv \texttt{int} \\
\\
\mathit{pop}(\texttt{boxed}\ \texttt{global}\ \delta\ \tau) &\equiv \texttt{boxed}\ \texttt{global}\ \delta\ \tau \\
\mathit{pop}(\langle \tau_1, \tau_2 \rangle) &\equiv \langle \mathit{pop}(\tau_1), \mathit{pop}(\tau_2) \rangle \\
\mathit{pop}(\texttt{int}) &\equiv \texttt{int}
\end{aligned}$$

Fig. 3. Supporting functions for late enforcement. Notice that pop is not defined for local pointers.

$$\frac{A \vdash e : \tau \qquad \delta \in \{\texttt{shared}, \texttt{private}\}}{A \vdash \uparrow_\delta e : \texttt{boxed}\ \texttt{local}\ \delta\ \tau}$$

Notice late enforcement places no restrictions on the type of data to be boxed. One may, for example, create a pointer to shared data containing embedded pointers to private data.

Dereferencing of local pointers is standard, and is allowed regardless of the sharing qualifier. For global pointers, we only allow dereferencing if the pointed-to data is known to be shared, and apply pointer widening to the result:

$$\frac{A \vdash e : \texttt{boxed}\ \texttt{local}\ \delta\ \tau}{A \vdash \downarrow e : \tau} \qquad \frac{A \vdash e : \texttt{boxed}\ \texttt{global}\ \texttt{shared}\ \tau}{A \vdash \downarrow e : \mathit{expand}(\tau)}$$

This is the first instance in which late enforcement restricts program behavior: private data may not be read across a global pointer. Private data is only visible to the owning processor, by way of local pointers.

Assignment is similar, and represents the second instance in which late enforcement restricts program behavior. A local pointer may be used to modify both shared and private data, while a global pointer may only touch shared data. To enforce that the assigned value embed no local pointers, global assignment carries an additional requirement that the type being assigned be preserved by type expansion:

$$\frac{A \vdash e : \texttt{boxed}\ \texttt{local}\ \delta\ \tau \qquad A \vdash e' : \tau}{A \vdash e \mathrel{\texttt{:=}} e' : \tau}$$

$$\frac{\begin{array}{c} A \vdash e : \texttt{boxed}\ \texttt{global}\ \texttt{shared}\ \tau \\ A \vdash e' : \tau \qquad \mathit{expand}(\tau) = \tau \end{array}}{A \vdash e \mathrel{\texttt{:=}} e' : \tau}$$

Widening directly applies the type expansion function to coerce local pointers into their global equivalents. Transmission across processor boundaries requires that type expansion be the identity, just as for global assignment; typically, one would transmit a value that had just been widened:

$$\frac{A \vdash e : \tau}{A \vdash \texttt{widen}\ e : \mathit{expand}(\tau)} \qquad \frac{A \vdash e : \tau \qquad \mathit{expand}(\tau) = \tau}{A \vdash \texttt{transmit}\ e : \mathit{expand}(\tau)}$$

Selection propagates the sharing qualifier through pointer displacement in the obvious manner:

$$\frac{A \vdash e : \texttt{boxed}\ \omega\ \delta\ \langle \tau_1, \tau_2 \rangle \qquad n \in \{1, 2\}}{A \vdash \texttt{@}n\ e : \texttt{boxed}\ \omega\ \delta\ \tau_n}$$

Design Implications In the rules given above, it is only at the point of an actual dereference or assignment that we add restrictions to implement late enforcement. Any program that tries to read or write private or mixed data via a global pointer fails to type check. In conjunction with the $expand$ function, these rules implicitly cover the case of global pointers to local pointers as well. Suppose that p is a local pointer to private data, and that q is a global pointer to a shared location containing p. Then:

$$\begin{aligned} p &: \texttt{boxed}\ \texttt{local}\ \texttt{private}\ \tau \\ q &: \texttt{boxed}\ \texttt{global}\ \texttt{shared}\ \texttt{boxed}\ \texttt{local}\ \texttt{private}\ \tau \\ \downarrow q &: \texttt{boxed}\ \texttt{global}\ \texttt{private}\ \tau \end{aligned}$$

Dereferencing q does not yield local pointer p, but rather an equivalent pointer widened to global. Since p points to private data, the widened pointer also points to private data and cannot be dereferenced or assigned through.

In general, the late enforcement system forbids undesirable behavior only if a private value is actually used by a remote processor. A global pointer to private data may be created, copied, sent to other processors, placed into data structures, compared to other pointers, and so on, but the memory location named by the pointer cannot be examined or modified.

We know of two situations in which the type `boxed global private` τ could be desirable. First, this pointer may be embedded in a tuple containing both shared and private components. The shared portions of the data structure may be accessed remotely, while the private portions are not. Type checking such a program requires that we allow global pointers to private data to be formed and manipulated, provided that they are never actually used.

Second, although a global pointer is conservatively assumed to address remote memory, a global pointer may address local memory as well. Real distributed languages typically allow dynamically checked conversion of global pointers back to local. A global pointer to private data, then, might be converted back to a local pointer to private data, whereupon it could be used freely.

Applicability and Limitations If we intend to use sharing qualifiers to make data location management decisions, the weak guarantees offered by late enforcement are sufficient. When memory is reserved using $\uparrow_{\texttt{private}}$ or $\uparrow_{\texttt{shared}}$, we can use the subscripted qualifier to choose a suitable region of memory. Global pointers may escape to distant processors, but the memory occupied by private data is never examined or modified remotely. Thus, private data can reside in memory that is not network-addressable. Sharable memory, which may be slower or more scarce, is reserved for shared data.

Several other clients can make use of late enforcement guarantees. Distributed cache coherence need not be maintained for data that is never examined remotely. Race condition detection systems need not concern themselves with global pointers to private data, since these can never create races. Similarly, any sequence of operations on private data may be reordered or optimized quite freely even under the strictest of consistency models, because changes to intermediate states are never observable by other processors. Treating lock acquisition as a dereference, private locks can be eliminated at compile time.

However, late enforcement is too late for other applications. Certain languages may be unable to autonomously garbage collect using late enforcement. Security concerns may also be unsatisfied by exposed pointers to private data. Our remaining type systems incrementally impose stricter enforcement policies to accommodate these concerns.

3.2 Export Enforcement

In some languages, late enforcement is too weak to support autonomous garbage collection. As suggested earlier, many distributed programming environments support checked conversion of global pointers back to local. In such a system, the following sequence of actions could take place:

1. Processor A creates private data and sends its address to remote processor B. Processor B now holds a global pointer to this private data.
2. Processor A destroys all of its own references to the private data. The global pointer held by processor B is now the only live reference to this private data.
3. Some time later, the processor B sends the global private pointer back to processor A.
4. Processor A uses a checked conversion to recover a local pointer to its own private data, and subsequently dereferences that pointer.

Autonomous garbage collection requires that any live data have at least one live local reference. If processor A were to autonomously garbage collect between steps 2 and 3, the private data would seem unreachable and the memory it occupies would be reclaimed. Therefore, if a language allows narrowing casts from global to local, late enforcement cannot be used to support autonomous garbage collection.

We modify the late enforcement system as follows. One possible source of global pointers to private data is the initial environment (A). We impose a well-formedness requirement on the initial environment, stipulating that `boxed global private` τ not appear within any part of the type for any variable or function. For compound expressions, the chief source of global pointers is the `widen` operator, which relies upon $expand$ to expand local pointers to global. Figure 4 gives a revised $expand$ function that only produces global pointers to shared data. Notice that the new version is not defined for pointers to private or mixed data. Thus, `transmit` can no longer send pointers to private data across processor boundaries. The `widen` coercion is similarly restricted, as are global assignment and global dereferencing. Given a well-formed starting environment, the revised $expand$ function ensures that no expression has type `boxed global private` τ.

The new $expand$ function guarantees that the only pointers exported are those to shared data. However, observe that when a pointer of type `boxed` ω `shared` τ is expanded, there are no restrictions on the type τ of the pointed-to data. In particular,

$$\begin{aligned} expand(\texttt{boxed}\ \omega\ \texttt{shared}\ \tau) &\equiv \texttt{boxed}\ \texttt{global}\ \texttt{shared}\ \tau \\ expand(\langle \tau_1, \tau_2 \rangle) &\equiv \langle pop(\tau_1), pop(\tau_2) \rangle \\ expand(\texttt{int}) &\equiv \texttt{int} \end{aligned}$$

Fig. 4. Revised type expansion function supporting export enforcement. The subordinate pop function is unchanged from Figure 3.

one may freely transmit a pointer to a shared memory cell which, in turn, points to private data: $\texttt{boxed}\ \omega\ \texttt{shared}\ (\texttt{boxed}\ \omega'\ \texttt{private}\ \tau')$ is an identity for $expand$. Thus, this *export enforcement* type system restricts only the actual, immediate values being exported. It does not extend transitively beyond the first level of pointers. This is sufficient to support autonomous garbage collection, as it guarantees that no remote processor can hold the only live reference to any piece of private data.

This approach does not eliminate the need to manage memory used by objects which genuinely are shared. Rather, it complements distributed garbage collection techniques (e.g. stubs and scions [30]) by identifying a private subset of data which can be collected aggressively using simpler, traditional, purely local collection algorithms.

3.3 Early Enforcement

In an untrusted environment, the address at which private data is stored may itself be sensitive information. Security concerns may mandate that no private address ever escape the owning processor. Neither late nor export enforcement can provide that kind of protection. The vulnerability can be seen in the type checking rule for global dereference, which requires that the pointed-to data be shared. In an untrusted environment, a remote processor that willfully disregards this restriction may be able to transitively walk across pointers and ultimately reach private data. Global assignment is similarly vulnerable. Runtime checks could detect such misbehavior, but static (compile-time) assurances may be a more attractive option.

For complete control over private addresses, we refine export enforcement to additionally require that no private data be transitively reachable from shared memory. For variables and functions, we extend the well-formedness requirements on initial environments in the obvious manner. For compound expressions, the only change required is in the type checking rules for allocation. Late and export enforcement allowed either shared or private boxing of any type τ. For early enforcement, we impose an additional restriction on shared allocation:

$$\frac{A \vdash e : \tau}{A \vdash \uparrow_{\texttt{private}} e : \texttt{boxed}\ \texttt{local}\ \texttt{private}\ \tau}$$

$$\frac{A \vdash e : \tau \qquad allShared(\tau)}{A \vdash \uparrow_{\texttt{shared}} e : \texttt{boxed}\ \texttt{local}\ \texttt{shared}\ \tau}$$

The new $\mathit{allShared}$ predicate holds if and only if all pointers directly embedded within a type are shared:

$$\begin{aligned} \mathit{allShared}(\texttt{boxed}\ \omega\ \delta\ \tau) &\equiv (\delta = \texttt{shared}) \\ \mathit{allShared}(\langle \tau_1, \tau_2 \rangle) &\equiv \mathit{allShared}(\tau_1) \wedge \mathit{allShared}(\tau_2) \\ \mathit{allShared}(\texttt{int}) &\equiv \text{true} \end{aligned}$$

Thus, no pointer to private data may ever be placed in shared memory. If we require that the initial environment (A) obey similar restrictions, then in general no private storage is transitively reachable from shared or global memory. The universe of shared data is transitively closed. A consequence of this is that the sharing constraint in the global dereference and assignment rules is always trivially satisfied: because all data transitively reachable from a global pointer must be shared, it is impossible for a malicious remote processor to disregard the sharing constraint and transitively reach private data.

Applicability and Trade-offs Clearly, export enforcement is more restrictive than late enforcement, and early enforcement is more restrictive still. By accepting fewer programs, export and early enforcement allows us to make progressively stronger guarantees about the meaning of "private" in those programs that do type check. Thus, early enforcement can also support all late enforcement clients, such as race detectors or reordering optimizers, as well as autonomous garbage collection. The effectiveness of some clients may be reduced, though, as early enforcement treats some data as shared which late or export enforcement could have taken as private.

The set of programs accepted under each system is a strict subset of those accepted under the one before. We have adapted an earlier proof of `local`/`global` soundness [22] to show that the late enforcement system is sound, from which the soundness of the other two systems follows. The additional requirements imposed by each system (e.g. no access to private data by way of a global pointer) are enforced directly by the type checking rules; correctness of these restrictions is verifiable by direct inspection of the type checking rules for the corresponding operations.

3.4 Type Inference

The type systems described above are easily converted from type checking to type inference. We illustrate late enforcement inference here. Inference for export and early enforcement is similar, but has been omitted for the sake of brevity.

We assume that the program is known to type check disregarding sharing qualifiers. The `local`/`global` qualifiers can also be inferred [21], and one may wish to infer all qualifiers simultaneously. For simplicity we assume here that `local`/`global` inference has already taken place separately.

Our type inference rules produce a system of constraints that must be solved. Rules are given in Figure 5; additional sharing constraints arise from the expand function as defined in Figure 6. For clarity of presentation, the rules use several abbreviations:

1. Constraint sets are not explicitly propagated up from subexpressions; the complete constraint set is the union of all sets of constraints induced by all subexpressions.

$$\frac{}{A \vdash i : \texttt{int}} \qquad \frac{A(x) = \tau}{A \vdash x : \tau}$$

$$\frac{A(f) = \tau \to \tau' \quad A \vdash e : \tau'' \quad \tau'' \leq \tau}{A \vdash f\, e : \tau'} \qquad \frac{A \vdash e : \tau}{A \vdash \uparrow_\delta e : \texttt{boxed}\ \texttt{local}\ \delta\, \tau}$$

$$\frac{A \vdash e : \texttt{boxed}\ \texttt{local}\ \delta\, \tau}{A \vdash \downarrow e : \tau} \qquad \frac{\begin{array}{c}A \vdash e : \texttt{boxed}\ \texttt{global}\ \texttt{shared}\ \tau \\ \mathit{expand}(\tau, \tau')\end{array}}{A \vdash \downarrow e : \tau'}$$

$$\frac{A \vdash e : \tau \quad A \vdash e' : \tau'}{A \vdash e\, \texttt{;}\, e' : \tau'} \qquad \frac{\begin{array}{c}A \vdash e : \texttt{boxed}\ \texttt{local}\ \delta\, \tau \\ A \vdash e' : \tau' \qquad \tau' \leq \tau\end{array}}{A \vdash e \texttt{:=} e' : \tau}$$

$$\frac{\begin{array}{c}A \vdash e : \texttt{boxed}\ \texttt{global}\ \texttt{shared}\ \tau \\ A \vdash e' : \tau' \quad \tau' \leq \tau \quad \mathit{expand}(\tau, \tau)\end{array}}{A \vdash e \texttt{:=} e' : \tau} \qquad \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2}{A \vdash \langle e_1, e_2 \rangle : \langle \tau_1, \tau_2 \rangle}$$

$$\frac{A \vdash e : \texttt{boxed}\ \omega\ \delta\ \langle \tau_1, \tau_2 \rangle \quad n \in \{1, 2\}}{A \vdash \texttt{@}n\, e : \texttt{boxed}\ \omega\ \delta\ \tau_n} \qquad \frac{A \vdash e : \tau \quad \mathit{expand}(\tau, \tau')}{A \vdash \texttt{widen}\ e : \tau'}$$

$$\frac{A \vdash e : \tau \quad \mathit{expand}(\tau, \tau)}{A \vdash \texttt{transmit}\ e : \tau}$$

Fig. 5. Type inference rules for late enforcement

2. A nontrivial rule hypothesis such as

$$e : \texttt{boxed}\ \texttt{global}\ \texttt{shared}\ \tau$$

   should be read as an equality constraint

$$e : \tau_0 \qquad \tau_0 = \texttt{boxed}\ \texttt{global}\ \texttt{shared}\ \tau$$

3. All constraint variables are fresh.

Any solution to the constraints induced by these rules gives a valid typing for the program. We note that setting all possible variables to `shared` always produces one legitimate solution. Thus, languages that assume all data to be shared are safe, albeit

$$\begin{aligned} \mathit{expand}(\texttt{boxed}\ \omega\ \delta\ \tau, \texttt{boxed}\ \omega'\ \delta'\ \tau') &\equiv \{\delta = \delta', \tau = \tau'\} \\ \mathit{expand}(\langle \tau_1, \tau_2 \rangle, \langle \tau_1', \tau_2' \rangle) &\equiv \mathit{expand}(\tau_1, \tau_1') \cup \mathit{expand}(\tau_2, \tau_2') \\ \mathit{expand}(\texttt{int}, \texttt{int}) &\equiv \emptyset \end{aligned}$$

Fig. 6. Additional constraints induced by supporting functions. We assume `local`/`global` qualifiers have already been assigned; these functions cover only the additional sharing constraints.

$$\begin{tikzcd}
 & \delta_1 & & \delta_2 & \\
\texttt{private} & & \delta & & \texttt{shared}
\end{tikzcd}$$

Fig. 7. Constraint graph requiring choice between shared and private. An arrow $x \rightarrow y$ encodes the constraint $x \leq y$

overly conservative. Because our sharing qualifier lattice has no $\perp$, there is no least solution. Rather, we are interested in the "best" solution, defined as the one having the largest number of `private` qualifiers. This maximally-private solution may be computed efficiently as follows:

1. Assume that initially we have a static typing showing what is a pointer, pair, or integer, as well as which pointers are local or global.
2. Using the equivalences in Figure 2(b), expand type constraints $\tau = \tau'$ and $\tau \leq \tau'$ to obtain the complete set of sharing constraints.
3. Identify the set of qualifier constants that are transitive lower bounds of each qualifier variable. Collect the set $\mathcal{S}$ of all constraint variables that have either `shared` or `mixed` as a transitive lower bound. These variables cannot be `private`.
4. For each sharing qualifier variable δ not in $\mathcal{S}$, set $\delta =$ `private`. This may cause `private` to appear as a transitive lower bound on a variable where it was not present earlier.
5. For each sharing qualifier δ, let d be the least upper bound of its constant transitive lower bounds. Let $\delta = d$ in the final solution.

The meat of this algorithm devolves to graph reachability flowing forward along constraint edges from nodes representing each of the three type qualifiers. The solution, therefore, is computable in time linear with respect to the number of sharing qualifiers in the fully typed program [16, 19]. As `local`/`global` inference is also linear, this gives language designers great flexibility. One may expose all of these type choices in the source language, or one may present a simpler source-level model augmented by fast compiler-driven type inference to fill in the details.

The critical feature of this algorithm is that it first identifies all qualifiers that cannot possibly be private, then binds all other variables to private, and lastly chooses between shared and mixed for the variables that could not be made private. This strategy maximizes the number of `private` qualifiers in exchange for driving some other qualifiers to `mixed` instead of `shared`. In the example in Figure 7, our algorithm binds δ to `private`, which means that δ_1 can be `private` as well but δ_2 must be `mixed`. An alternative is to set δ to `shared`, which would drive δ_1 to `mixed` but allow δ_2 to be `shared`. In either case, some δ_n must be `mixed`. Our algorithm resolves such choices in favor of maximizing the number of variables bound `private`, as this is most useful to the clients of interest.

4 Related Work

The static type systems of previous proposals have not dealt with `local`/`global` and `shared`/`private` in a general way. As a result, they are all either less general than is possible, unsafe, or both. Lack of generality prevents programmers from enforcing that data should be private, which makes it more difficult to reason about program correctness and results in missed opportunities for optimization. Lack of safety exhibits itself in unsafe (often implicit) casts among pointer types that impede optimization, or in under-specified semantics where optimization may change program behavior in unexpected ways.

One group of languages guarantees safety but has no facility for declaring private heap data. In these languages the stack is private but the entire heap must be treated as potentially shared. Java, Olden [9], and Titanium (prior to this work) [20] are in this category. For these languages, our techniques provide a basis for automatically inferring private heap data. We also believe it is important for programmers to be able to declare private data explicitly, as knowledge of what data is private is critical in understanding parallel and distributed programs.

Jade is a safe language that distinguishes local from global pointers, and which allows some heap data to be private. This private data can never be transitively reachable from the shared heap [31], which corresponds to our early enforcement system (Section 3.3). Our results show that where security is not a primary concern, significantly more flexible sharing of data is possible, allowing for more data to be statically identified as private and thereby making privacy-dependent analyses more effective.

EARTH-C explicitly offers both local/global and shared/private type qualifiers. Local/global may be inferred [36], but shared/private must be given explicitly [18]. Our approach shows that shared/private is amenable to inference as well, operating either fully automatically or to augment programmer directives. The broader EARTH-C project has also clearly demonstrated the value of identifying local private data to drive analyses such as redundant read/write removal and communication optimization [37].

Among the unsafe (C-derived) languages, AC [10], PCP [8], and UPC [11] offer shared and private data. However, their type systems do not distinguish the addresses of private data from narrowed global pointers to shared data. In effect, these languages offer only `global shared` and `local mixed`. Although private data exists at run time, the static type system cannot identify it in a useful manner, and many of the clients listed in Section 2.2 cannot be deployed.

Also in the C family, CC++ [12] and Split-C [14] do not directly address the notion of private data. This may mean that all data is presumed shared, but it is difficult to know exactly what semantics are required, especially with regard to code reordering and other aggressive optimizations. Cilk explicitly treats all memory as shared, and states that the programmer is responsible for understanding the underlying memory model provided by the host hardware [34]. We believe that sharing inference can support aggressive optimization without the added burden of under-specified semantics.

Our type systems are similar to escape analysis. Previous research has focused on identifying data that does not escape a stack frame or a thread of execution [3, 6, 7, 13, 35]. The early enforcement system may be thought of as identifying data that does not escape a local address space. Considered in this light, the late enforcement system is unusual: escape of addresses is permitted, provided that the data referenced by an escaped address is never actually examined or modified from a distance. This is more permissive than escape analysis, yet it is strong enough to support certain traditional escape analysis clients, such as synchronization removal.

To our knowledge, only one earlier study takes an approach that is similar to late enforcement. The "thread-local analysis" presented by Aldrich et al. [3] defines *multithreaded objects* as objects that escape from one thread and are also written to by a (conservatively defined) distinct thread. An escaped object that is never written to need not be considered multithreaded. This is similar in spirit to late enforcement: a globally reachable piece of data that is not actually accessed remotely need not be considered shared. The question of whether something akin to late enforcement can be applied directly to stack and thread escape analyses warrants further study.

5 Experimental Findings

We have added sharing qualifiers to Titanium, an experimental Java dialect for high performance parallel computing [20]. Unqualified references are assumed to be `shared`; programmers may declare references as `private` or `mixed` subject to validation by the type checker. Stronger (more private) qualifiers are added automatically using type inference. To highlight the bounds of the design space, both late and early enforcement are available. (Export enforcement would yield performance results between these two.) Our inference engine is based on the `cqual` qualifier inference engine [15].

Our benchmarks are single-program multiple-data (*SPMD*) codes with no explicit sharing qualifiers. All benchmarks are designed for execution on distributed memory multiprocessors, and reflect the scientific focus of SPMD programming. The applications include Monte Carlo integration (`pi`), sorting (`sample-sort`), dense linear algebra (`lu-fact`, `cannon`), a Fourier transform (`3d-fft`), particle methods (`n-body`, `particle-mesh`, `ib` [28]), and solvers for computational fluid dynamics (`gsrb`, `pps` [4], `amr` [29], `gas` [5]).

Whole-program sharing inference including the Java class library (roughly an additional 16,000 lines) takes no more than one second on a 1.3 GHz Pentium 4 Linux workstation. As the inference algorithm itself is linear, it should scale well to much larger code bases.

We studied several of the sharing based analyses and optimizations in Section 2.2. Detailed results appear in a companion report, which also provides additional details on incorporating sharing analysis into a complete programming language [23]. Here we focus on three areas: the static prevalence of various inferred types; dynamic tallies of shared versus private allocation to support data location management; and the performance impact of sharing inference on consistency model relaxation.

5.1 Static Metrics

Table 1 shows the number of static reference declaration sites in each benchmark: places where some sharing qualifier could syntactically appear. The Titanium stack is trivially

Table 1. Benchmark sizes and relative counts of inferred qualifiers.

benchmark	**lines**	**sites**	**late**			**early**		
			shared	**mixed**	**private**	**shared**	**mixed**	**private**
`pi`	56	12	25%	0%	75%	25%	0%	75%
`sample-sort`	321	73	38%	1%	60%	38%	1%	60%
`lu-fact`	420	150	54%	3%	43%	54%	3%	43%
`cannon`	518	162	36%	2%	61%	36%	2%	61%
`3d-fft`	614	191	37%	1%	63%	37%	1%	63%
`n-body`	826	113	76%	1%	23%	76%	1%	23%
`gsrb`	1090	281	48%	1%	51%	48%	1%	51%
`particle-grid`	1095	201	83%	$< 1\%$	16%	84%	0%	16%
`pps`	3673	551	41%	5%	54%	44%	6%	50%
`ib`	3777	1094	56%	1%	43%	58%	1%	41%
`amr`	5206	1353	57%	1%	42%	59%	1%	40%
`gas`	8841	1699	50%	$< 1\%$	50%	51%	$< 1\%$	49%

private, so we exclude local variables and tabulate only heap data. Although whole program inference was used, we include here only those sites appearing in the benchmark application code (not in libraries). For each style of enforcement, we show the fraction of static references inferred as `shared`, `mixed`, and `private`.

In our review of related work, we saw that several distributed languages either have only a private stack or else have no notion of private data at all. We believe that this is an important omission. Regardless of which system is used, we consistently identify large amounts of private heap data in all benchmarks of all sizes. The largest benchmark, `gas`, has private data at half of all declaration sites. Other benchmarks range from 16% to 75%, and overall 46% of all sites in all benchmarks are inferred `private`. This is encouraging news for analysis clients which may want to exploit such information. It also reinforces the need for inference: it is unlikely that any human programmer could correctly place all such qualifiers by hand, or maintain such qualifiers over time.

A small number of `mixed` qualifiers appear in nearly every benchmark. In many cases, `mixed` is found in utility code shared by distinct parts of the application; parametric polymorphism could certainly be used here instead of subtyping. Elsewhere we find code, not isolated within methods, that performs complex operations on either shared or private data based on complex run time control decisions. The `mixed` qualifier works well here, whereas method-based parametric polymorphism would be difficult to apply without nontrivial code factoring. In both cases, the code is clearly polymorphic with respect to sharing, and without a polymorphic type system significantly more data would be forced to be `shared`. Polymorphism may be more important to the overall system than the small `mixed` counts would suggest.

When heap data is shared by several processors, the obvious choice is to address it using global pointers; it is not clear that local pointers to shared data are needed. However, other than `pi`, all programs show heavy use of local pointers to shared data: 24%–53% of all shared heap data is addressed by local pointers, and these numbers remain high (42% for `amr`, 32% for `gas`) even for the largest benchmarks. A local shared pointer often represents the locally allocated portion of some larger, distributed

Table 2. Kilobytes allocated in shared or private memory. We omit gsrb and ib due to unrelated Titanium bugs which prevent them from running to completion.

benchmark	**late**				**early**			
	shared		**private**		**shared**		**private**	
pi	74	(75%)	25	(25%)	74	(75%)	25	(25%)
sample-sort	3306	(5%)	66453	(95%)	3347	(5%)	67843	(95%)
cannon	8771	(60%)	5768	(40%)	8771	(60%)	5768	(40%)
3d-fft	4755	(52%)	4328	(48%)	4755	(52%)	4328	(48%)
n-body	368	(< 1%)	101700	(100%)	368	(< 1%)	101700	(100%)
particle-grid	9511	(99%)	123	(1%)	9513	(99%)	123	(1%)
pps	19459	(26%)	55360	(74%)	60518	(81%)	14302	(19%)
amr	36455	(88%)	4841	(12%)	40990	(99%)	306	(1%)
gas	2587611	(55%)	2157523	(45%)	2587866	(55%)	2157267	(45%)

data structure. Each processor retains a local pointer to the data that it created, and may use that pointer for efficient access to the data which it "owns". Earlier work has demonstrated that use of local pointers is critical to performance [21]; if a quarter to half of all statically shared data references were forced to be global, performance can only suffer.

5.2 Data location management

Shared memory may be a scarce or costly resource. We have instrumented each benchmark to tally the number of shared and private allocations over the course of an entire run. Table 2 gives these totals, in bytes, for each of late and early enforcement. Observe that we see slight differences between the two enforcement schemes even on small benchmarks which reported identical results in Table 1. This is because that earlier table examined only application code and excluded libraries, whereas these allocation counts apply to the entire program. Slight differences in inference results for library code are visible here as slight differences in allocation counts for late versus early enforcement.

Overall, we see wide variation between benchmarks, ranging from 99% of allocations shared (particle-grid) to nearly 100% of allocations private (n-body). We have examples at both extremes among both the large and small benchmarks. Our largest benchmark, gas, is also the most memory intensive, and we find that 45% of allocated bytes can be placed in private memory.

Most byte counts do not vary appreciably between late and early enforcement, though amr sees an 11% shift. The most dramatic shift is found in pps: late enforcement allows 74% private allocation, while early enforcement drops that to merely 19%. In Table 1 we observe that pps shows a relatively large difference in static private declaration counts as well. Clearly those differences encompass data structures which account for a preponderance of pps's runtime memory consumption. When running on machines with costly shared memory, pps stands to benefit greatly from data location management guided by sharing inference.

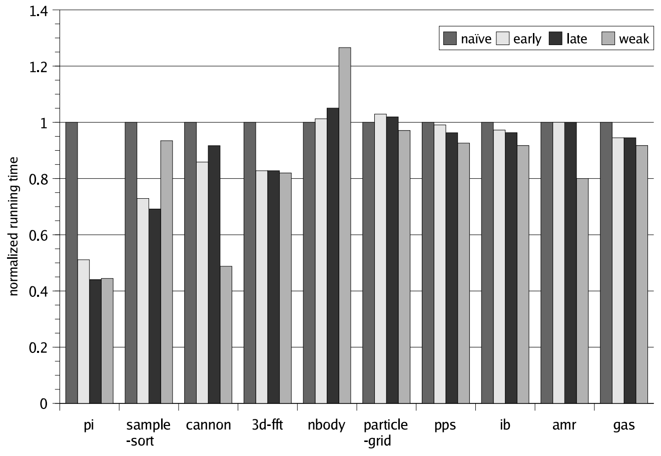

Fig. 8. Performance cost of sequential consistency. We omit `lu-fact` and `gsrb` due to unrelated Titanium bugs which prevent them from running to completion.

5.3 Consistency Model Relaxation

Titanium uses a fairly weak consistency model, which allows the compiler or hardware to reorder memory operations [20]. A stronger model would be a more attractive programming target if it did not unacceptably harm performance. As suggested in Section 2.2, we can use sharing inference to allow private data accesses to be reordered while ensuring the stronger semantics at the language level. We have implemented such an optimization for a sequentially consistent variant of Titanium.

Figure 8 presents benchmark running times using each of four configurations. *Naïve* uses no inference: all data is assumed shared, and sequential consistency is enforced everywhere. *Early* and *late* enforce sequential consistency except where private data is inferred using the corresponding type system. *Weak* is the weak consistency model used in Titanium, which is an upper bound on the speedup from allowing reordering.

Because the benchmarks have very different raw performance, we present the running times normalized by the running time of the naïve implementation of sequential consistency. Measurements were taken on an SMP Linux workstation with four Pentium III, 550 MHz CPU's and 4GB of DRAM.

The large speedup for weak `pi` confirms that sequential consistency is costly if bluntly applied. Sharing inference is able to identify enough private data, though, to erase that penalty in the late and early variants. Hand inspection shows that sharing inference for `pi` is perfect: all data in the main computational loop is inferred `private` and no restrictions are needed on optimizations to enforce sequential consistency. The early, late, and weak versions if `pi` yield identical machine code; apparent performance here are measurement noise.

For most of the other benchmarks, there is only modest improvement between the naïve implementation and the weak consistency model, so the potential speedup from sharing inference is limited. This defies conventional wisdom, which says that sequential consistency is too expensive. There are two potential sources of inefficiency in the sequentially consistent versions: lost optimization opportunities (e.g., loop transformations) and additional memory fences between load and store instructions. Neither of these appear to be significant in our benchmarks. This highlights a limitation of our experimental environment: neither the Titanium compiler nor the Pentium hardware is taking advantage of weak consistency's looser requirements to significantly boost performance over sequential consistency.

Among the larger benchmarks, `cannon`, `3d-fft`, and `amr` show the largest performance gap between the naïve and weak models. These, then, stand to benefit the most from sharing inference. In `3d-fft`, inference (either late or early) is able to nearly match the weak model. Modest benefits are seen in `cannon`, where the larger slowdown is only partly offset by inference. Late and early enforcement yield identical results for `cannon`; the difference between the late and early slowdown factors is measurement noise.

The results for `amr` are interesting. None of the key performance-critical data structures can be inferred private using our current system. Like many SPMD programs, `amr` has an alternating-phase structure: all processors exchange boundary information, then each processor updates its own local portion of the shared grid, then all processors communicate again, and so on. Data is shared widely during `amr`'s communication phase, but we would like to treat that same data as private during local computation phases. These phases are delimited by global barrier operations, so no processor looks at another processors' data while the local computations are taking place. For sharing inference to be effective here, it would need to allow for a limited form of flow sensitivity keyed to these phases. Because the structure of barriers is quite regular in practice [2], we believe such an extension of our techniques should be feasible.

Observe that two benchmarks, `n-body` and `particle-grid` exhibit unexpected speedups under naïve sequential consistency. Because the direct penalty of sequential consistency here is so small, measurement noise due to secondary effects (such as cache alignment and code layout) becomes more noticeable.

6 Conclusions

We have presented a general approach to describing the data sharing behavior of distributed programming languages, and codified that approach in a suite of type systems. Early enforcement resembles earlier work on escape analysis. Export and late enforcement are unusual in enforcing privacy closer to the point of use, rather than at the point of escape, allowing them to identify more private data in exchange for weaker guarantees as to what "private" actually means. We have considered these type systems in light of the optimizations they would permit, and present experimental data on two such optimizations: data layout management and consistency model relaxation.

The approach is conducive to efficient type qualifier inference, and can be adapted to suit realistic languages. Our survey of related languages suggests that most fall into two categories: those which under-specify the behavior of shared data, and those which equate shared with global and private with local. Our approach points out that other combinations, such as mixed pointers and local shared pointers, have a role to play.

References

1. A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture (ISCA'95)*, pages 2–13, June 1995.
2. A. Aiken and D. Gay. Barrier inference. In *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 342–354, San Diego, California, January 19–21, 1998.
3. J. Aldrich, E. G. Sirer, C. Chambers, and S. J. Eggers. Comprehensive synchronization elimination for Java. *Science of Computer Programming*, to appear. Also published as University of Washington Technical Report UW-CSE-00-10-01, October 2000.
4. G. T. Balls. *A Finite Difference Domain Decomposition Method Using Local Corrections for the Solution of Poisson's Equation*. PhD thesis, Department of Mechanical Engineering, University of California at Berkeley, 1999.
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. B. Blanchet. Escape analysis for object oriented languages. Application to Java. In OOPSLA [27], pages 20–34.
7. J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In OOPSLA [27], pages 35–46.
8. E. D. Brooks, III. PCP: A parallel extension of C that is 99% fat free. Technical Report UCRL-99673, Lawrence Livermore National Laboratory, Sept. 1988.
9. M. C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Department of Computer Science, Princeton University, June 1996.
10. W. W. Carlson and J. M. Draper. Distributed data access in AC. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 39–47, Santa Barbara, California, July 1995. IDA Supercomputing Research Center.
11. W. W. Carlson, J. M. Draper, D. E. 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, May 13 1999.
12. K. M. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. *Lecture Notes in Computer Science*, 757:124–144, 1993.
13. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In OOPSLA [27], pages 1–19.
14. D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In IEEE, editor, *Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, pages 262–273, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1993. IEEE Computer Society Press.
15. J. Foster. `cqual`. Available at `<http://bane.cs.berkeley.edu/cqual>`, Nov. 2001.
16. J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1–4, 1999. *SIGPLAN Notices*, 34(5), May 1999.
17. D. P. Helmbold and C. E. McDowell. Computing reachable states of parallel programs. *ACM SIGPLAN Notices*, 26(12):76–84, Dec. 1991.
18. L. J. Hendren, X. Tang, Y. Zhu, S. Ghobrial, G. R. Gao, X. Xue, H. Cai, and P. Ouellet. Compiling C for the EARTH multithreaded architecture. *International Journal of Parallel Programming*, 25(4):305–338, Aug. 1997.

19. F. Henglein and J. Rehof. The complexity of subtype entailment for simple types. In *Proceedings, Twelth Annual IEEE Symposium on Logic in Computer Science*, pages 352–361, Warsaw, Poland, 29 June–2 July 1997. IEEE Computer Society Press.
20. P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, and K. Yelick. Titanium language reference manual. Technical Report CSD-01-1163, University of California, Berkeley, Nov. 2001.
21. B. Liblit and A. Aiken. Type systems for distributed data structures. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 199–213, Boston, Massachusetts, January 19–21, 2000.
22. B. Liblit and A. Aiken. Type systems for distributed data structures. Technical Report CSD-99-1072, University of California, Berkeley, Jan. 2000. Available at `<http://sunsite.berkeley.edu/TechRepPages/CSD-99-1072>`.
23. B. Liblit, A. Aiken, and K. Yelick. Data sharing analysis for Titanium. Technical Report CSD-01-1165, University of California, Berkeley, Nov. 2001. Available at `<http://sunsite.berkeley.edu/TechRepPages/CSD-01-1165>`.
24. J. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. *ACM SIGPLAN Notices*, 28(12):129–139, Dec. 1993.
25. S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, Software, pages II–105–II–113, University Park, Penn, Aug. 1990. Penn State U. Press. CSRD TR#993, U. Ill.
26. Myricom Inc. *The GM Message Passing System*, July 18 2000. Version 1.1.
27. *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, Denver, CO, Oct. 1999. ACM Press.
28. C. S. Peskin and D. M. McQueen. A three-dimensional computational method for blood flow in the heart. I. Immersed elastic fibers in a viscous incompressible fluid. *Journal of Computational Physics*, 81(2):372–405, Apr. 1989.
29. G. Pike, L. Semenzato, P. Colella, and P. N. Hilfinger. Parallel 3D adaptive mesh refinement in Titanium. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, Mar. 1999.
30. D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In H. Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, ILOG, Gentilly, France, and INRIA, Le Chesnay, France, Sept. 1995. Springer-Verlag.
31. M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, May 1998.
32. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.
33. B. Steensgaard. Thread-specific heaps for multi-threaded programs. In T. Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, Oct. 2000. ACM Press.
34. Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.3.1 Reference Manual*, June 24 2000.
35. J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In OOPSLA [27], pages 187–206.
36. Y. Zhu and L. Hendren. Locality analysis for parallel C programs. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):99–114, Feb. 1999.
37. Y. Zhu and L. J. Hendren. Communication optimizations for parallel C programs. *ACM SIGPLAN Notices*, 33(5):199–211, May 1998.