Analysis of Partitioned Global Address Space Programs

by Amir Kamil

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor K. Yelick
Research Advisor

(Date)

* * * * * * *

Professor S. Graham
Second Reader

(Date)

Abstract

The introduction of multi-core processors by the major microprocessor vendors has brought parallel programming into the mainstream. Analysis of parallel languages is critical both for safety and optimization purposes. In this report, we consider the specific case of languages with barrier synchronization and global address space abstractions. Two of the fundamental problems in the analysis of parallel programs are to determine when two statements in a program can execute concurrently, and what data can be referenced by each memory location. We present an efficient interprocedural analysis algorithm that conservatively computes the set of all concurrent statements, and improve its precision by using context-free language reachability to ignore infeasible program paths. In addition, we describe a pointer analysis using a hierarchical machine model, which distinguishes between pointers that can reference values within a thread, within a shared memory multiprocessor, or within a network of processors. We then apply the analyses to two clients, data race detection and memory model enforcement. Using a set of five benchmarks, we show that both clients benefit significantly from the analyses.

1 Introduction

The introduction of multi-core processors by the major microprocessor manufacturers marks a dramatic shift in software development: parallelism for laptops and desktop machines will no longer be hidden within a micro-architecture, but will be exposed to higher level software. Most parallel code for small-scale machines is written using a shared memory programming model. Though the high end is dominated by message passing, there are currently several efforts to provide a global address space abstraction across shared and distributed memory machines, in which each thread can access data on any other thread. The global address space languages include Unified Parallel C (UPC) [13, 48], Co-Array Fortran (CAF) [45], Titanium [53, 25] (based on Java [23]), and the HPCS languages under development by Cray (Chapel [16]), IBM (X10 [42]), and Sun (Fortress [3]). Analysis of these parallel languages is critical both for safety and optimization purposes.

We present two analyses for the specific case of *partitioned global address space* (PGAS) programs with barrier synchronization. In such programs, the memory space is partitioned into multiple localities. While each thread can access data anywhere in the global address space, access to its own locality is much faster than elsewhere. Programs written in the PGAS model exploit this by concentrating each thread's computation on data within its own locality.

We first introduce an *interprocedural concurrency analysis*, which captures information about the potential concurrency between statements in a program. We construct a *concurrency graph* representation of a program and present a simple algorithm that uses it to determine the set of all potentially concurrent expressions in a program. This analysis proves too conservative, however, and we improve its precision by performing a context-free language analysis on a modified form of the concurrency graph. We prove the correctness of both analyses and show that their total running times are quadratic in the size of the input program.

We also present a pointer analysis that is designed for a PGAS setting. The analysis takes into account a hierarchical machine model, in which a pointer may be valid only within a subset of processors in the hierarchy. For example, a pointer may refer to data only within a single thread, or to data associated with any threads within an SMP node, or to any thread in the machine. We develop a model language, *Ti*, for presenting our pointer analysis and provide a type system for the language. *Ti* has the essential features of any PGAS language: the ability to create references to data, share references with other machines in the system, and dereference them for either read or write access. *Ti* also has a hierarchical machine model, a feature that is also available (although less general) in many PGAS languages.

We implement the analyses in a compiler for the Titanium language [53], a single program, multiple data global address space dialect of Java that runs on most parallel and distributed memory machines. The analyses can be used by many clients, including locality inference [33] and data sharing inference [34]. In this report, we consider two clients in particular. We first use them to perform *data race analysis* [36], which can be used to report potential program errors to application programmers. We then use the analyses to enforce a *sequentially consistent memory model* [30], a stronger and more intuitive memory model than is normally provided by parallel languages. Using a set of test programs, we demonstrate that both clients benefit significantly from the analyses.

2 Background

We implement the analyses in a compiler for the Titanium programming language, and modify two clients to leverage the results. In this section, we present an overview of Titanium and the features relevant to the analyses. We also discuss how the two clients can take advantage of the analyses in order to produce better results.

2.1 Titanium

Titanium is a dialect of Java and contains most of the features of Java 1.4. In addition, it provides language features for parallel and scientific programming. These include multidimensional arrays and index spaces, immutable classes, region-based memory allocation [2, 22], C++-style templates, operator overloading, and barrier synchronization. In this report, we discuss only those features that are relevant to our analyses.

The Titanium compiler does not use the Java Virtual Machine model. Instead, the end target is assembly code. For portability, Titanium is first translated into C and then compiled into an executable. In addition to generating C code to run on each processor, the compiler generates calls to a runtime layer based on GASNet [10], a lightweight communication layer that exploits hardware support for direct remote reads and writes when possible. Titanium runs on a wide range of platforms including uniprocessors, shared memory machines, distributed-memory clusters of uniprocessors or SMPs, and a number of specific supercomputer architectures (Cray X1, Cray T3E, SGI Altix, IBM SP, Origin 2000, and NEC SX6).

2.1.1 Textually Aligned Barriers

Titanium uses a different model of parallelism than Java. Instead of having dynamically created threads, Titanium is a *single program, multiple data* (SPMD) language, so the number of threads is fixed at program startup and all threads execute the same code image.

Like many SPMD languages, Titanium has a *barrier* construct that forces threads to wait at the barrier until all threads have reached it. Aiken and Gay introduced the concept of *structural correctness* to enforce that all threads execute the same number of barriers, and developed a static analysis that determines whether or not a program is structurally correct [1, 21]. The following code is not structurally correct:

```
if (Ti.thisProc() % 2 == 0)
  Ti.barrier(); // even ID threads
else
  ; // odd ID threads
```

Titanium provides a stronger guarantee of *textually aligned barriers*: not only do all threads execute the same number of barriers, they also execute the same *textual* sequence of barriers. Thus, both the above structurally incorrect code and the following structurally correct code are erroneous in Titanium:

```
if (Ti.thisProc() % 2 == 0)
  Ti.barrier(); // even ID threads
else
  Ti.barrier(); // odd ID threads
```

The fact that Titanium barriers are textually aligned is central to our concurrency analysis: not only does it guarantee that code before and after each barrier cannot run concurrently, it also guarantees that code immediately following two different barriers cannot execute simultaneously.

Titanium's type system ensures that barriers are textually aligned by making use of *single-valued* expressions [1]. Such expressions provably evaluate to the same value for all threads[1], and include the following:

- compile-time constants

[1] In the case of single-valued expressions of reference type, the result is not the same but is *replicated and coherent*. See the Titanium language reference for details [25].

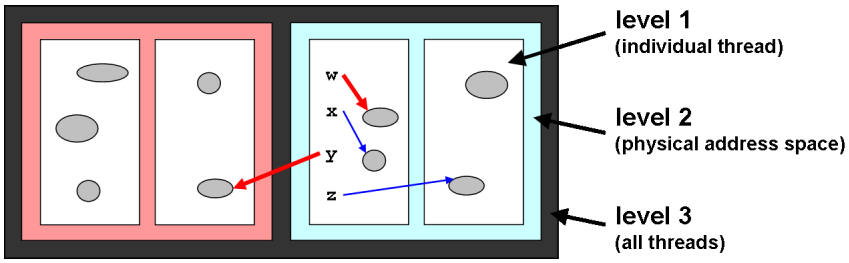

Fig. 1: The Titanium thread hierarchy. The thin, blue arrows signify local pointers, while the thick, red arrows designate global pointers. Global pointers may point to local data.

- program arguments
- certain library functions, such as `Ti.numProcs()`, which returns the total number of threads
- expressions that are combinations of the above

Other expressions such as those involving references and method calls can also be single-valued, the details of which can be found in the Titanium reference manual [25].

Barrier alignment can only be violated if different threads take different program paths, and any of those paths contain a barrier. Titanium statically prevents this by requiring path forks, including conditionals, loops, and dynamically dispatched method calls, to be conditioned on single-valued expressions if any of the branches contains a barrier. This guarantees that all threads take the same branch and therefore execute the same barriers. The examples above are erroneous: they each have branches with barriers but `Ti.thisProc() % 2 == 0` is not single-valued, so not all threads take the same branch. If the condition was replaced by the single-valued expression `Ti.numProcs() % 2 == 0`, then both examples would become legal.

In addition to the existing barriers in a program, our concurrency analysis also exploits single-valued expressions to determine which conditional branches can run concurrently. The analysis does not insert any new barriers, and it ignores the lock-based `synchronized` construct of Java, which is rarely used in Titanium programs.

2.1.2 Memory Hierarchy

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 using load and store instructions, or they may be in distinct address spaces, in which case the global access must be translated into communication through the GASNet communication layer.

In addition to dereferencing, communication between threads can be done through the one-to-all *broadcast* and the all-to-all *exchange* operations. Local and 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, as shown in Figure 1:

- **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 programmer can restrict a variable to only point within a physical address space (level 2) by qualifying it with the `local` keyword. In Figure 1, the thin (blue) pointers are local, while the thick (red) pointers are global. Titanium allows downcasts between global and local, but they only succeed if the actual location referenced is within the same physical address space as the executing thread. In Figure 1, a downcast of `w` to local would succeed, since the

referenced memory is on the same thread, while a downcast of y would fail. Our analysis takes advantage of existing downcasts 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 both race detection and sequential consistency enforcement, since references to level 1 values on different threads cannot be to the same location. Sequential consistency enforcement can also benefit from the distinction between levels 2 and 3 by using cheaper barriers if a memory access on one thread can only conflict with an access from another thread in the same physical address space, though we do not use this here. Locality analysis can also benefit from this distinction, as shown in §B.4.1.

Theoretically, we could perform a two-level analysis twice to obtain a three-level analysis, but as shown in §B.3.3, the three-level analysis we have implemented is much more efficient. Thus, we would like a pointer analysis that accounts for all levels in the hierarchical distribution of Titanium threads. Our analysis is general enough to handle an arbitrary number of levels, which could be useful for systems and languages with more than three levels.

2.2 Applications

We evaluate our analyses by applying them to two clients, static race detection and enforcement of sequential consistency. Both clients require concurrency and pointer analyses in order to be effective.

2.2.1 Static Race Detection

In parallel programs, a *data race* occurs when multiple threads access the same memory location, at least one of the accesses is a write, and the accesses can occur concurrently [36]. Data races often correspond to programming errors and potentially result in non-deterministic runtime behavior. Concurrency analysis can determine which accesses can occur concurrently, and pointer analysis [4] can determine which accesses are to the same location.

2.2.2 Sequential Consistency

For a sequential program, compiler and hardware transformations must not violate data dependencies: the order of all pairs of conflicting accesses must be preserved. Two memory accesses *conflict* if they access the same memory location and at least one of them is a write. The execution model for parallel programs is more complicated, since each thread executes its own portion of the program asynchronously and there is no predetermined ordering among accesses issued by different threads to shared memory locations. A memory consistency model defines the memory semantics and restricts the possible execution order of memory operations.

Titanium's memory consistency model is defined in the language specification [25]. Here are some informal properties of the Titanium model.

1. **Locally sequentially consistent:** All reads and writes issued by a given thread must appear to that thread to occur in exactly the order specified. Thus, dependencies within a thread must be observed.

2. **Globally consistent at synchronization events:** At a global synchronization event such as a barrier, all threads must agree on the values of all the variables. At a non-global synchronization event, such as entry into a critical section, the thread must see all previous updates made using that synchronization event.

Henceforth, we will refer to the Titanium memory consistency model as the *relaxed model*.

A simpler memory model, *sequential consistency*, is the most intuitive for the programmer. The sequential consistency model states that a parallel execution must behave as if it were an interleaving of the serial executions by individual threads, with each individual execution sequence preserving the program order [30].

An easy way to enforce sequential consistency is to insert memory barriers after each shared memory access. This forbids all reordering of shared memory operations, preventing optimizations such as prefetching and code motion and resulting in an unacceptable performance penalty. Various techniques, such as *cycle detection* [44, 29], have been proposed to minimize the number of barriers, or *delay set*, required to enforce sequential consistency. Our prior work with Su and Yelick, however, has shown that the delay set can be precisely approximated by inserting memory barriers around each memory access that may be part of a race condition [27].

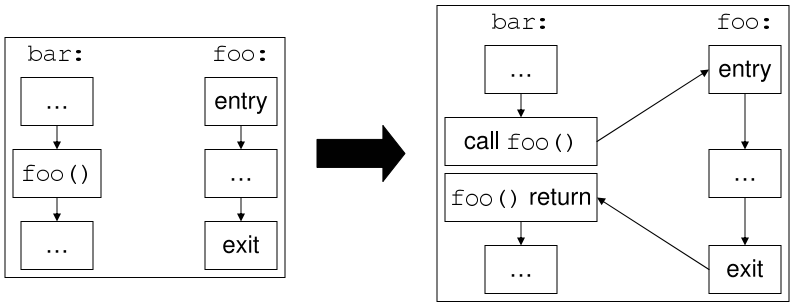

Fig. 2: Construction of the interprocedural control flow graph of a program from the individual method flow graphs.

3 Concurrency Analysis

A precise knowledge of the set of concurrent statements in parallel programs is fundamental to many analyses and optimizations. In this section, we develop a basic analysis to determine this set. We then improve on its results by only considering program paths that can occur at runtime.

3.1 Analysis Background

Our concurrency analyses do not operate directly on Titanium program's source code, but on a graph representation of a reduced form of the program, in order to simplify both the theory and implementation of the analyses.

3.1.1 Intermediate Language

We operate on an *intermediate language* that allows the full semantics of Titanium but is simpler to analyze. In particular, we rewrite dynamic dispatches as conditionals. A call `x.foo()`, where `x` is of type `A` in the class hierarchy

```
class A {
  void foo() { ... }
}

class B extends A {
  void foo() { ... }
}
```

gets rewritten to

```
if ([type of x is A])
  x.A$foo();
else if ([type of x is B])
  x.B$foo();
```

We also rewrite `switch` statements and conditional expressions (... `?` ... `:` ...) as conditional `if ... else ...` statements.

3.1.2 Control Flow Graphs

The concurrency algorithms are whole-program analyses that operate over a *control flow graph* that represents the flow of execution in a program. Nodes in the graph correspond to expressions in the program, and a directed edge from one expression to another occurs when the target can execute immediately after the source.

The Titanium compiler produces an intraprocedural control flow graph for each method in a program. We modify each of these graphs to model transfer of control between methods by splitting each method invocation node into a

Algorithm 3.3.
ConcurrencyGraph(P : program) : graph

1. Let G be the interprocedural control flow graph of P, as described in §3.1.2.
2. For each conditional C in P {
3. If C is not a single conditional:
4. Add a cross edge for C in G.
5. } // End for (2).
6. For each barrier B in P:
7. Delete the node for B and its adjacent edges from G.
8. Return G.

Fig. 3: Algorithm 3.3 computes the concurrency graph of a program by inserting cross edges into its control flow graph and deleting all barriers.

call node and a return node. The incoming edges of the original node are attached to the call node, and the outgoing edges to the return node. An edge is added from the call node to the target method's entry node, and from the target method's exit node to the return node. Figure 2 illustrates this procedure. We also add edges to model interprocedural control flow due to exceptions.

3.2 Basic Analysis

Titanium's structural correctness allows us to develop a simple graph-based algorithm for computing concurrent expressions in a program. The algorithm specifically takes advantage of Titanium's textually aligned barriers and single-valued expressions. The following definitions are useful in developing the analysis:

Definition 3.1 (Single Conditional). A *single conditional* is a conditional guarded by a single-valued expression.

Since a single-valued expression provably evaluates to the same result on all threads, every thread is guaranteed to take the same branch of a single conditional. A single conditional thus may contain a barrier, since all threads are guaranteed to execute it, while a non-single conditional may not.

Definition 3.2 (Cross Edge). A *cross edge* in a control flow graph connects the end of the first branch of a conditional to the start of the second branch.

Cross edges do not provide any control flow information, since the second branch of a conditional does not execute immediately after the first branch. They are, however, useful for determining concurrency information, as shown in Theorem 3.4.

In order to determine the set of concurrent expressions in a program, we construct a *concurrency graph* G of the program P by inserting cross edges in the interprocedural control flow graph of P for every non-single conditional and deleting all barriers and their adjacent edges. Algorithm 3.3 in Figure 3 illustrates this procedure. The algorithm runs in time $O(n)$, where n is the number of statements and expressions in P, since it takes $O(n)$ time to construct the control flow graph of a program. The control flow graph is very sparse, containing only $O(n)$ edges, since the number of expressions that can execute immediately after a particular expression e is constant. Since at most n cross edges are added to the control flow graph and at most $O(n)$ barriers and adjacent edges are deleted, the resulting graph G is also of size in $O(n)$.

The concurrency graph G allows us to determine the set of concurrent expressions using the following theorem:

Theorem 3.4. *Two expressions* a *and* b *in* P *can run concurrently only if one is reachable from the other in the concurrency graph* G.

In order to prove Theorem 3.4, we require the following definition:

Definition 3.5 (Code Phase). The *code phase* of a barrier is the set of expressions that may execute after the barrier but before hitting another barrier, including itself[2].

```
B1:  Ti.barrier();
L1:  int i = 0;
L2:  int j = 1;
L3:  if (Ti.thisProc() < 5)
L4:    j += Ti.thisProc();
L5:  if (Ti.numProcs() >= 1) {
L6:    i = Ti.numProcs();
B2:    Ti.barrier();
L7:    j += i;
L8:  } else { j += 1; }
L9:  i = broadcast j from 0;
B3:  Ti.barrier();
L10: j += i;
```

Code Phase	Statements
B1	L1, L2, L3, L4, L5, L6, L8, L9
B2	L7, L9
B3	L10

Fig. 4: The set of code phases for an example program.

Algorithm 3.7.
ConcurrentExpressions(P : program) : set

1. Let $concur \leftarrow \emptyset$.
2. Let $G \leftarrow$ **ConcurrencyGraph**(P) [Algorithm 3.3].
3. For each access a in P {
4. Do a depth first search on G starting from a.
5. For each expression b reached in the search:
6. Insert (a, b) into $concur$.
7. } // End for (3).
8. Return $concur$.

Fig. 5: Algorithm 3.7 computes the set of all pairs of concurrent expressions in a given program.

Figure 4 shows the code phases of an example program. Since each code phase is preceded by a barrier, and each thread must execute the same sequence of barriers, each thread executes the same sequence of code phases. This implies the following:

Lemma 3.6. *Two expressions* a *and* b *in* P *can run concurrently only if they are in the same code phase.*

Using Lemma 3.6, we can prove Theorem 3.4. Details are in §A.1.1.

By Theorem 3.4, in order to determine the set of all pairs of concurrent expressions, it suffices to compute the pairs of expressions in which one is reachable from the other in the concurrency graph G. This can be done efficiently by performing a depth first search from each expression in G. Algorithm 3.7 in Figure 5 does exactly this. The running time of the algorithm is dominated by the depth first searches, each of which takes $O(n)$ time, since G has at most n nodes and $O(n)$ edges. At most n searches occur, so the algorithm runs in time $O(n^2)$.

3.3 Feasible Paths

Algorithm 3.7 computes an over-approximation of the set of concurrent expressions. In particular, due to the nature of the interprocedural control flow graph constructed in §3.1.2, the depth first searches in Algorithm 3.7 can follow *infeasible paths*, paths that cannot structurally occur in practice. Figure 6 illustrates such a path, in which a method is entered from one context and exits into another.

In order to prevent infeasible paths, we follow the procedure outlined by Reps [40]. We label each method call edge and corresponding return edge with matching parentheses, as shown in Figure 6. Each path then corresponds to a string of parentheses composed of the labels of the edges in the path. A path is then infeasible, if in its corresponding string, an open parenthesis is closed by a non-matching parenthesis.

[2] A statement can be in multiple code phases, as is the case for a statement in a method called from multiple contexts.

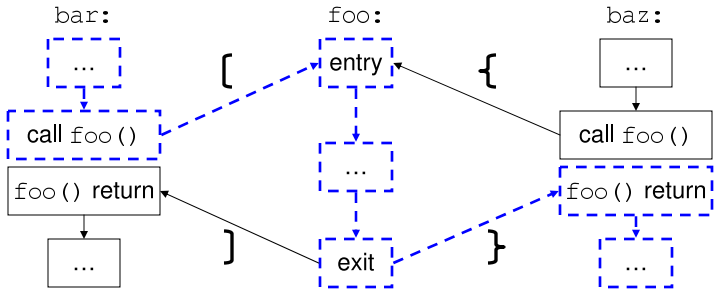

Fig. 6: Interprocedural control flow graph for two calls to the same function. The dashed path is infeasible, since `foo()` returns to a different context than the one from which it was called. The infeasible path corresponds to the unbalanced string "[}".

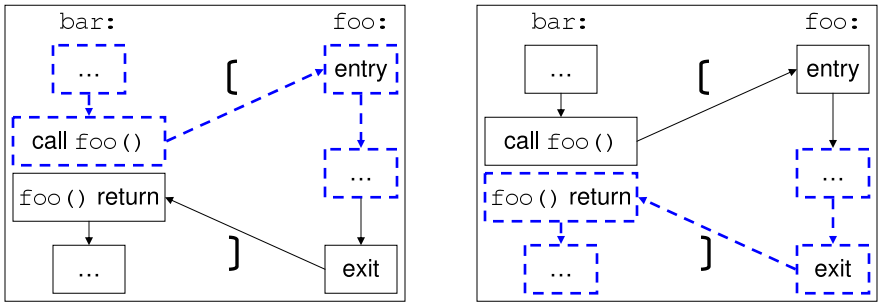

Fig. 7: Feasible paths that correspond to unbalanced strings. The dashed path on the left corresponds to a method call that has not yet returned, and the one on the right corresponds to a path that starts in a method call that returns.

It is not necessary that a path's string be balanced in order for it to be feasible. In particular, two types of unbalanced strings correspond to feasible paths:

- A path with unclosed parentheses. Such a path corresponds to method calls that have not yet finished, as shown in the left side of Figure 7.
- A path with closing parentheses that follow a balanced prefix. Such a string is allowed since a path may start in the middle of a method call and corresponds to that method call returning, as shown in the right side of Figure 7.

Determining the set of nodes reachable[3] using a feasible path is the equivalent of performing context-free language (CFL) reachability on a graph using the grammar for each pair of matching parentheses $(_\alpha$ and $)_\alpha$. CFL reachability can be performed in cubic time for an arbitrary grammar [40]. Algorithm 3.7 takes only quadratic time, however, and we desire a feasibility algorithm that is also quadratic. In order to accomplish this, we develop a specialized algorithm that modifies the concurrency graph G and the standard depth first search instead of using generic CFL reachability.

At first glance, it appears that a method must be revisited in every possible context in which it is called, since the context determines which open parentheses have been seen and therefore which paths can be followed. However, as shown in §A.1.2, the set of expressions that can be executed in a method call is the same regardless of context. This implies that the set of nodes reachable along a feasible path in a program's control flow graph is also independent of the context of a method call, with two exceptions:

- If a method call can complete, then the nodes after the call are reachable from a point before the call.

[3]In this section, we make no distinction between *reachable* and *reachable without hitting a barrier*. The latter reduces to the former if all barrier nodes are removed from each control flow graph.

Algorithm 3.8.
ComputeBypasses(P : program, $G_1, \ldots, G_k$: intraprocedural flow graph) : set

1. Let $change \leftarrow true$.
2. Let $marked \leftarrow \emptyset$.
3. While $change = true$ {
4. $change \leftarrow false$.
5. Set $visited(u) \leftarrow false$ for all nodes u in $G_1, \ldots, G_k$.
6. For each method f in P {
7. If $f \notin marked$ and $CanReach(entry(f), exit(f), G_f, marked)$ {
8. $marked \leftarrow marked \cup \{f\}$.
9. $change \leftarrow true$.
10. } // End if (7).
11. } // End for (6).
12. } // End while (3).
13. Return $marked$.

14. Procedure $CanReach(u, v$: vertex, G : graph, $marked$: method set) : boolean:
15. Set $visited(u) \leftarrow true$.
16. If $u = v$:
17. Return $true$.
18. Else If u is a method call to function g and $g \notin marked$:
19. Return $false$.
20. For each edge $(u, w) \in G$ {
21. If $visited(w) = false$ and $CanReach(w, v, G, marked)$:
22. Return $true$.
23. } // End for (20).
24. Return $false$.

Fig. 8: Algorithm 3.8 uses each method's intraprocedural control flow graph (G_i) to determine if its exit is reachable from its entry.

- If no context exists, such as in a search that starts from a point within a method f, then all nodes that can be reached from the return node of any method call to f are reachable.

The second case above can easily be handled by visiting a node twice: once in *some* context, and again in no context. The first case, however, requires adding bypass edges to the control flow graph.

3.3.1 Bypass Edges

Recall that the interprocedural control flow graph was constructed by splitting a method call into a call node and a return node. An edge was then added from the call node to the target method's entry, and another from the target's exit to the return node. If the target's exit is reachable (or for our purposes, reachable without hitting a barrier) from the target's entry, then adding a *bypass edge* that connects the call node directly to the return node does not affect the transitive closure of the graph.

Computing whether or not a method's exit is reachable from its entry is not trivial, since it requires knowing whether or not the exits of each of the methods that it calls are reachable from their entries. Algorithm 3.8 in Figure 8 computes this by continually iterating over all the methods in a program, marking those that can complete through an execution path that only calls previously marked methods, until no more methods can be marked. In the first iteration of loop 3, it only marks those methods that can complete without making any calls, or equivalently, those methods that can complete using only a single stack frame. In the second iteration, it only marks those that can complete by calling only methods that don't need to make any calls, or equivalently, those methods that can complete using only two stack

Algorithm 3.9.
FeasibleSearch(v : vertex, G : graph) : set

1. Let $visited \leftarrow \emptyset$.
2. Let $s \leftarrow \emptyset$.
3. Call $FeasibleDFS(v, G, s, visited)$.
4. Return $visited$.

5. Procedure $FeasibleDFS(v$: vertex, G : graph, s : stack, $visited$: set):
6. If $s = \emptyset$ {
7. If $no\_context\_mark(v)$ return.
8. Set $no\_context\_mark(v) \leftarrow true$.
9. } // End if (6).
10. Else {
11. If $context\_mark(v)$ return.
12. Set $context\_mark(v) \leftarrow true$.
13. } // End else (10).
14. $visited \leftarrow visited \cup \{v\}$
15. For each edge $(v, u) \in G$ {
16. Let $s' \leftarrow s$.
17. If $label(v, u)$ is a close symbol and $s' \neq \emptyset$ {
18. Let $o \leftarrow pop(s')$.
19. If $label(v, u)$ does not match o:
20. Skip to next iteration of 15.
21. } // End if (17).
22. Else if $label(v, u)$ is an open symbol:
23. Push $label(v, u)$ onto s'.
24. Call $FeasibleDFS(u, G, s')$.
25. } // End for (15).

Fig. 9: Algorithm 3.9 computes the set of nodes reachable from the start node through a feasible path.

frames. In general, a method is marked in the ith iteration if it can complete using i, and no less than i, stack frames[4]. As shown in §A.1.3, Algorithm 3.8 marks all methods that can complete using any number of stack frames.

Algorithm 3.8 requires quadratic time to complete in the worst case. Each iteration of loop 3 visits at most n nodes. Only k iterations are necessary, where k is the number of methods in the program, since at least one method is marked in all but the last iteration of the loop. The total running time is thus $O(kn)$ in the worst case. In practice, only a small number of iterations are necessary[5], and the running time is closer to $O(n)$.

After computing the set of methods that can complete, it is straightforward to add bypass edges to the concurrency graph G: for each method call c, if the target of c can complete, add an edge from c to its corresponding method return r. This can be done in time $O(n)$.

3.3.2 Feasible Search

Once bypass edges have been added to the graph G, a modified depth first search can be used to find feasible paths. A stack of open but not yet closed parenthesis symbols must be maintained, and an encountered closing symbol must match the top of this stack, if the stack is nonempty. In addition, as noted above, the modified search must visit each node twice, once in no context and once in *some* context. Algorithm 3.9 in Figure 9 formalizes this procedure, and we prove that it does not follow infeasible paths in §A.1.4.

[4]Note that just because a method only requires a fixed number of stack frames doesn't mean that it can complete. A method may contain an infinite loop, preventing it from completing at all, or barriers along all paths through it, preventing it from completing without executing a barrier. Algorithm 3.8 does not mark such methods.

[5]Even on the largest example we tried (>45,000 lines of user and library code, 1226 methods), Algorithm 3.8 required only five iterations to converge.

Algorithm 3.10.
FeasibleConcurrentExpressions(P : program) : set

1. Let $G \leftarrow$ **ConcurrencyGraph**(P) [Algorithm 3.3].
2. For each method f in P {
3. Construct the intraprocedural flow graph G_f of f.
4. For each barrier B in f {
5. Delete B from G_f.
6. } // End for (4).
7. } // End for (2).
8. Let $bypass \leftarrow$ **ComputeBypasses**$(P, G_1, \ldots, G_k)$ [Algorithm 3.8].
9. For each method call and return pair c, r in P {
10. If the target f of c, r is in $bypass$:
11. Add an edge (c, r) to G.
12. } // End for (9).
13. For each expression a in P {
14. Let $visited \leftarrow$ **FeasibleSearch**(a, G) [Algorithm 3.9].
15. For each expression $b \in visited$:
16. Insert (a, b) into $concur$.
17. } // End for (13).
18. Return $concur$.

Fig. 10: Algorithm 3.10 computes the set of all concurrent expressions that can feasibly occur in a given program.

Since G contains bypass edges and Algorithm 3.9 visits each node both in some context and in no context, it finds all nodes that can be reachable in a feasible path from the source. Since it visits each node at most twice, it runs in time $O(n)$.

3.3.3 Feasible Concurrent Expressions

Putting it all together, we can now modify Algorithm 3.7 to find only concurrent expressions that are feasible. As in Algorithm 3.7, the concurrency graph G must first be constructed. Then the intraprocedural flow graphs of each method must be constructed, Algorithm 3.8 used to find the methods that can complete without hitting a barrier, and the bypass edges inserted into G. Then Algorithm 3.9 must be used to perform the searches instead of a vanilla depth first search. Algorithm 3.10 in Figure 10 illustrates this procedure.

The setup of Algorithm 3.10 calls Algorithm 3.8, so it takes $O(kn)$ time. The searches each take time in $O(n)$, and at most n are done, so the total running time is in $O(kn + n^2) = O(n^2)$, quadratic as opposed to the cubic running time of generic CFL reachability.

4 Pointer Analysis

Given a program, it is useful to know the locations that each variable and memory location can reference. We would like to produce a points-to analysis [4] in order to produce this information.

4.1 Analysis Background

We define a machine[6] 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.

[6]Throughout this report, we will use *machine* interchangeably with *thread*.

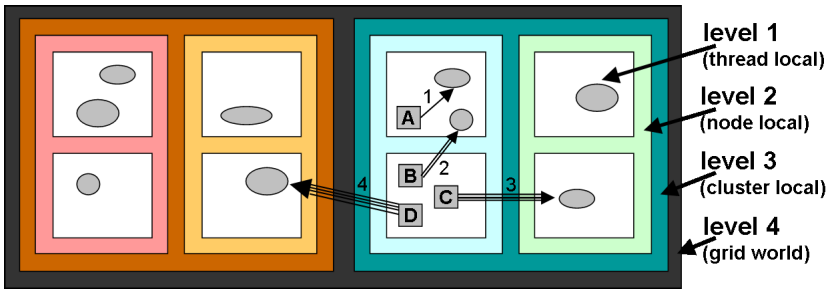

Fig. 11: A possible machine hierarchy with four levels. The width of arrows and their labels indicate the hierarchy distance between the endpoints.

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

Fig. 12: The syntax of the *Ti* language.

$$
expand(\tau, n) \equiv \begin{cases} \texttt{ref}_{max(m,n)}\ \tau' & \text{if } \tau = \texttt{ref}_m \tau' \\ \tau & \text{otherwise} \end{cases}
$$

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

Fig. 13: 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. 14: Type checking rules.

4.1.1 Machine Structure

Consider a set of machines arranged in an arbitrary hierarchy with the machines as leaves, such as that of Figure 11. A *machine* corresponds to a single execution stream within a parallel program, and for the purposes of our analysis, we ignore physical address spaces. Each machine has a corresponding *machine number*. The *depth* of the hierarchy is the number of levels it contains. The *distance* between machines is equal to the 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.

4.1.2 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 [33]. 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 $[1, h]$.

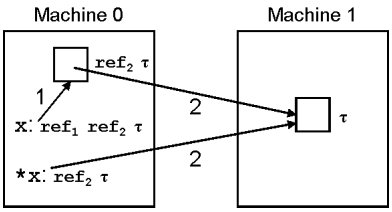

Fig. 15: Dereferences may require width expansion. The arrow labels correspond to pointer widths.

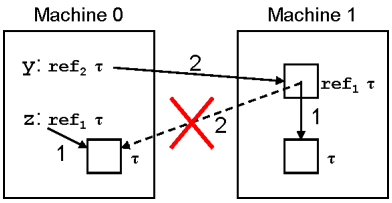

Fig. 16: The assignment $y \leftarrow z$ is forbidden, since the location referred to by y can only hold pointers of width 1 but requires a pointer of width 2 to refer to z.

The syntax of *Ti* is summarized in Figure 12. 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)
- 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. Each allocation site has 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 ($\texttt{transmit}\ e_1\ \texttt{from}\ e_2$). The expression $\texttt{transmit}\ e_1\ \texttt{from}\ e_2$ evaluates e_1 on machine e_2 and transmits the result to all 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 14. 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 15, where x on machine 0 refers to a location on machine 0 that refers to a 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, such as 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 16. 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, [33], 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 13.

The operational semantics of *Ti* are provided in §B.1.

4.2 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.

4.2.1 Semantic Domains

We use the following semantic domains in our analysis:

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)

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

$m \in M$	(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)

We define $hier(m, m')$ to be the distance between two machines m and m'.

4.2.2 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 $CS = (G+Id) \to \mathcal{P}(G)$.

4.2.3 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:

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

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

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

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

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

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 in σ_A is subsumed by the set in σ'_A for each element in $(A_m + Id)$. 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 $\mathrm{O}(h^2 \cdot |P|^2)$.

4.2.4 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. 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, provided in §B.1.

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.

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

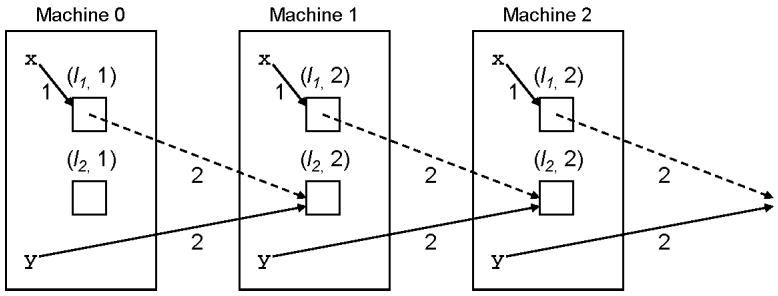

Fig. 17: The assignment $x \leftarrow y$ on machine 0 results in the abstract location $(l_2, 2)$ being added to the points-to set of $(l_1, 1)$, as shown by the first dashed arrow. The assignment on machine 1 results in the abstract location $(l_2, 2)$ being added to the points-to set of $(l_1, 2)$, 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 17. 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. In §B.2, we show that 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

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

4.2.5 Algorithm

The set of inference rules, instantiated over all the expressions in a program and applied in some arbitrary order[7], 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 $O(|P|)$ to apply it[8], where P is the input program. Since the lattice over AS has height in $O(h^2 \cdot |P|^2)$, it takes time in $O(h^2 \cdot |P|^3)$ to compute the fixed point of F. The running time of the analysis is thus cubic in the size of the input program and quadratic in the height of the machine hierarchy.

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

[8]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.

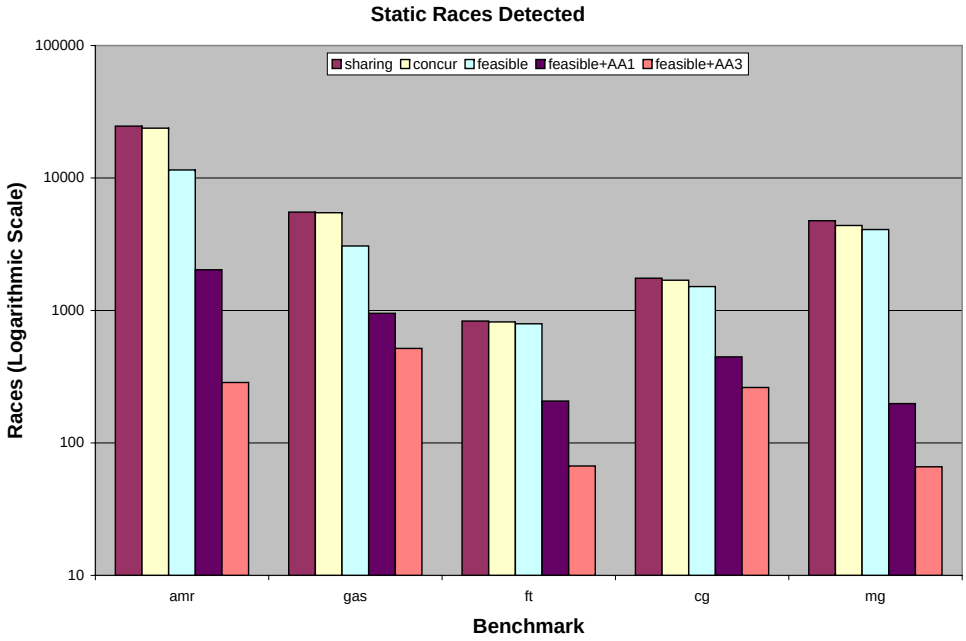

Fig. 18: Number of data races detected at compile-time.

5 Evaluation

We evaluate our concurrency and pointer analyses using two clients: static race detection and enforcing sequential consistency at the language/compiler level. We use the following set of benchmarks for our evaluation:

- **`amr`** [51] (7581 lines) Chombo adaptive mesh refinement suite [5] in Titanium.
- **`gas`** [9] (8841 lines): Hyperbolic solver for a gas dynamics problem in computational fluid dynamics.
- **`ft`** [17] (1192 lines): NAS Fourier transform benchmark [7] in Titanium.
- **`cg`** [17] (1595 lines): NAS conjugate gradient benchmark [7] in Titanium.
- **`mg`** [17] (1952 lines): NAS multigrid benchmark [7] in Titanium.

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.

5.1 Static Race Detection

Using our concurrency and pointer analyses, we built a compile-time data race analysis into the Titanium compiler. 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 18 compares the effectiveness of five levels of race detection:

- **sharing**: Type-based alias analysis and Liblit and Aiken's sharing inference [34] are used to detect potential races.
- **concur**: Our basic concurrency analysis (§3.2) is used to eliminate non-concurrent races.

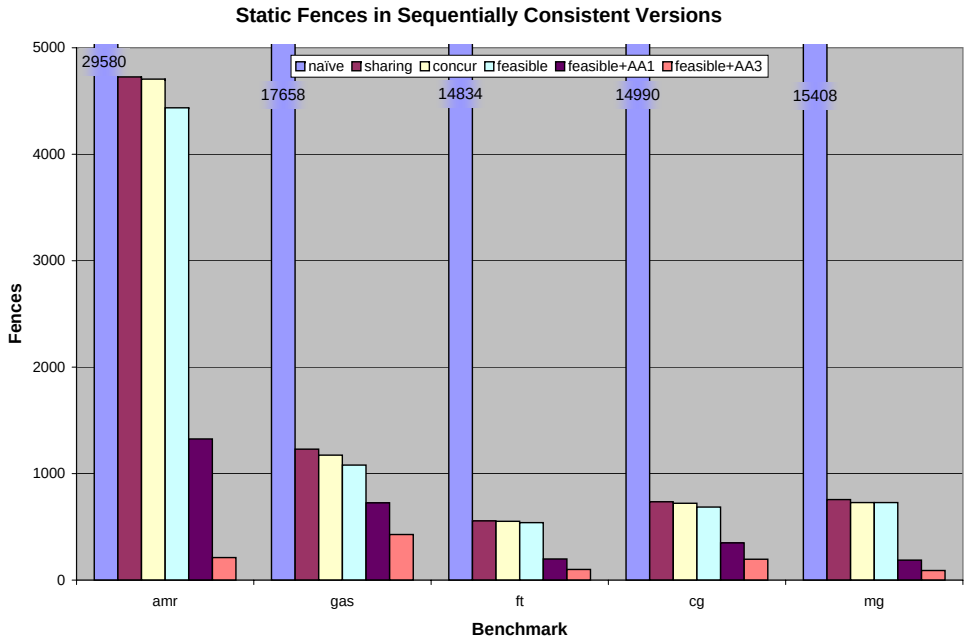

Fig. 19: Number of memory barriers generated at compile-time.

- **feasible**: Our feasible paths concurrency analysis (§3.3) is used to eliminate non-concurrent races.
- **feasible+AA1**: A single-level pointer analysis is used to eliminate false aliases.
- **feasible+AA3**: A three-level pointer analysis is used to eliminate false aliases.

The results show that the concurrency and pointer analyses can eliminate most of the races reported by our detector. None of the benchmarks benefit significantly from the basic concurrency analysis, but the feasible paths version significantly reduces the number of races found in two of the benchmarks. The addition of pointer analysis removes most of the remaining races, with a three-level analysis providing significant benefits over a one-level analysis.

5.2 Sequential Consistency

In order to enforce sequential consistency in Titanium, we insert memory barriers where required in an input program. These memory barriers can be expensive to execute at runtime, potentially costing an entire roundtrip latency for a remote memory access. The memory barriers also prevent code motion, so they directly preclude many optimizations from being performed. The static number of memory barriers generated provides a rough estimate for the amount of optimization prevented, but the affected code may actually be unreachable at runtime or may not be significant to the running time of a program. We therefore additionally measure the dynamic number of memory barriers hit at runtime, which more closely estimates the performance impact of the inserted memory barriers. Finally, we measure the actual running time of each benchmark on two platforms:

- **`jacquard.nersc.gov`**: An Opteron cluster running Linux, with two processors per node clocked at 2.2GHz and an Infiniband network. Benchmarks were run on four nodes, using one thread per node.
- **`bassi.nersc.gov`**: A Power5 cluster running AIX, with eight processors per node clocked at 1.9GHz and an IBM Federation network. Benchmarks were run on one node, using four threads, each in its own physical address space.

Execution time is compared to a version of each benchmark compiled using Titanium's default relaxed memory model.

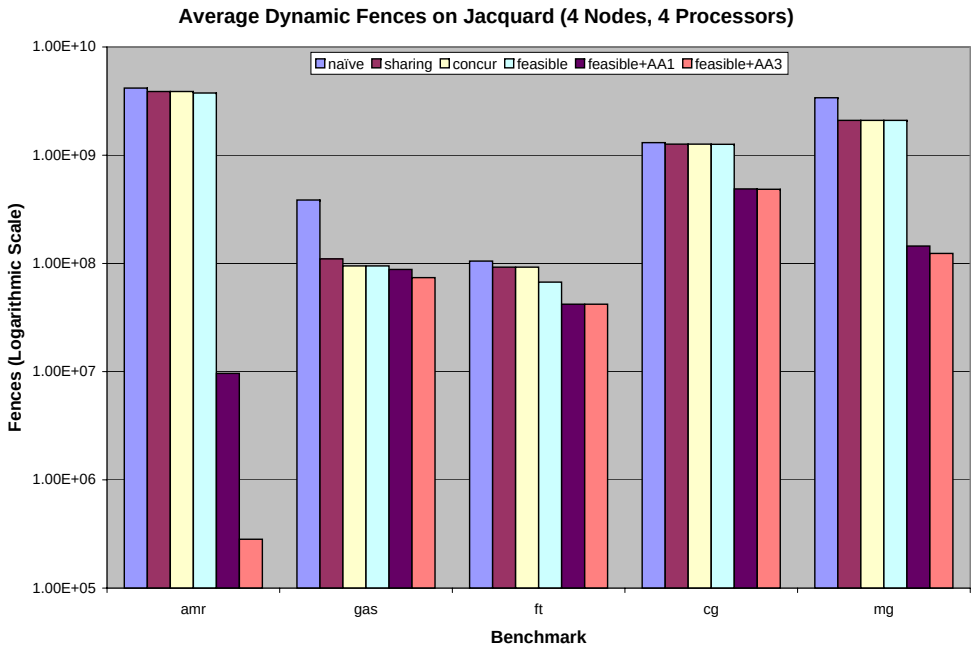

Fig. 20: Average number of memory barriers executed at runtime. Benchmarks were run on **`jacquard.nersc.gov`** using four processors.

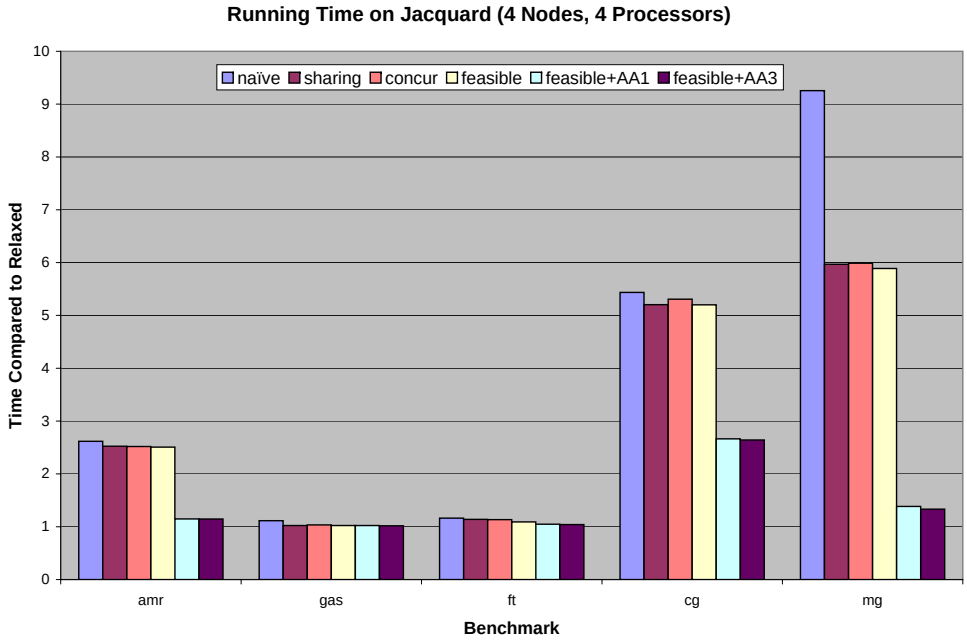

Fig. 21: Execution time on **`jacquard.nersc.gov`** using four processors, compared to a relaxed consistency version of the code.

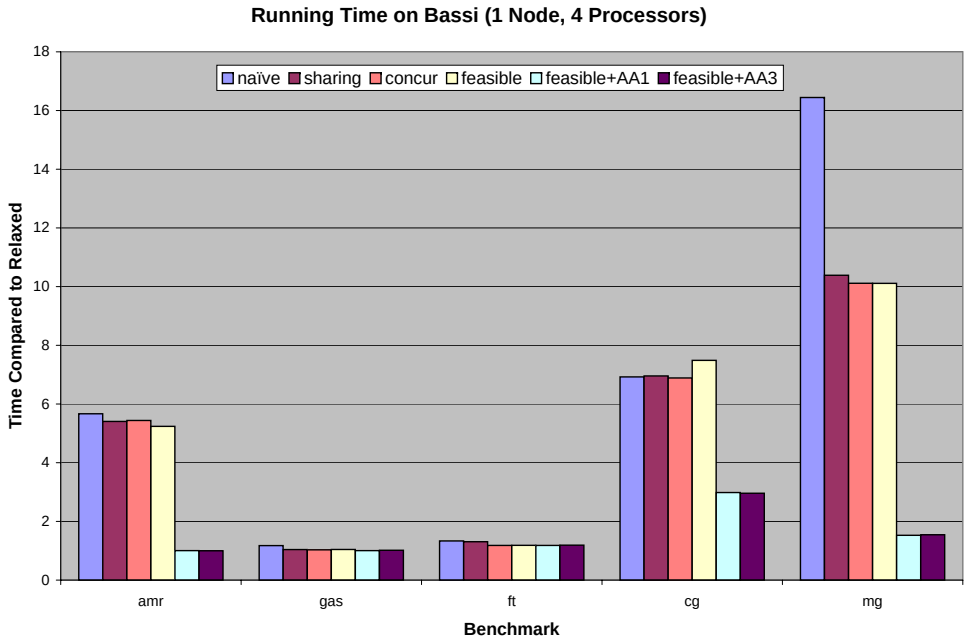

Fig. 22: Execution time on **`bassi.nersc.gov`** using four processors, compared to a relaxed consistency version of the code.

Figure 19 compares the number of memory barriers generated for each program using the five different levels of analysis above, with an additional base level of analysis:

- **naïve**: Fences are inserted around all heap accesses.

Figure 20 compares the resulting dynamic counts at runtime, and Figures 21 and 22 show the execution time on the two test platforms.

The results show that our analysis, at its highest precision, is very effective in reducing the numbers of both static and dynamic memory barriers. In three of the benchmarks, nearly all runtime memory barriers are eliminated, and in another, the number of memory barriers hit is reduced by a large fraction. Using the most precise analysis, all but one of the benchmarks perform nearly as well as their relaxed consistency versions.

6 Related Work

There is an extensive literature on compiler and runtime optimizations for parallel machines, including automatically parallelized programs and optimization of data parallel programs, which in their pure form have a sequential semantics. This includes work on concurrency analysis, race detection, pointer analysis, and enforcement of sequential consistency. This report itself is an extension of previous work in conjunction with Su and Yelick [27, 28].

6.1 Concurrency Analysis and Race Detection

An extensive amount of work on concurrency analysis has been done for both languages with dynamic parallelism and SPMD programs. Duesterwald and Soffa presented a data flow analysis to compute the *happened-before* and *happened-after* relation for program statements [19]. Their analysis is for detecting races in programs based on the Ada rendezvous model [49]. Masticola and Ryder developed a more precise non-concurrency analysis for the same set of programs [35]. The results are used for debugging and optimization. Jeremiassen and Eggers developed a static analysis for barrier synchronization for SPMD programs with non-textual barriers and used the information to reduce

false sharing on cache-coherent machines [26]. Their analysis doesn't take advantage of barrier alignment or single-valued expressions, so it isn't as precise as ours.

Others besides Duesterwald and Soffa and Masticola and Ryder have developed tools for race detection. Flanagan and Freund presented a static race detection tool for Java based on type inference and checking [20]. Boyapati and Rinard developed a type system for Java that guarantees that a program is race-free [11]. Tools such as Eraser [43] and TRaDe [15] detect races at runtime instead of statically. Other static and dynamic race detection schemes have also been developed [6, 14, 18, 38, 50].

Our work differs from previous work in that we develop an analysis specifically for SPMD programs with textual barriers. This allows our analysis to be both sound and precise. In addition, our analysis takes advantage of single-valued expressions, which no previous analysis does.

6.2 Pointer Analysis

The language and type system we presented here are generalizations of those described by Liblit and Aiken [33]. 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 [34].

Pointer analysis was first described by Andersen [4], and later extended by others to parallel programs. Rugina and Rinard developed a thread-aware alias analysis for the Cilk multithreaded programming language [41] that is both flow-sensitive and context-sensitive. Others such as Zhu and Hendren [54] and Hicks [24] have developed flow-insensitive versions for multithreaded languages. However, none of these analyses consider hierarchical, distributed machines.

6.3 Sequential Consistency

The memory consistency issue arises in a language with an explicitly parallel semantics and some type of shared address space. The class of such languages includes Java, UPC, Titanium, and Co-Array Fortran, some of the languages proposed in the recent HPCS effort, as well as shared memory language extensions such as POSIX Threads and OpenMP [13, 37, 39, 53].

Shasha and Snir provided some of the foundational work in enforcing sequential consistency from a compiler level when they introduced the idea of *cycle detection* [44]. However, that work was designed for general MIMD parallelism, limited to straight-line code, and was not designed as a practical static analysis. Midkiff and Padua outlined some of the implementation techniques that could violate sequential consistency and developed some static analysis ideas, including a concurrent static single assignment form in a paper by Lee et al. [31]. As part of the Pensieve project, Lee and Padua exploited properties of fences and synchronization to reduce the number of delays in cycle detection [32]. The project also includes a Java compiler that takes a memory model as input [47]. More recently, Sura et al. have shown that cooperating escape, thread structure, and delay set analyses can be used to provide sequential consistency cheaply in Java [46]. Our work differs from theirs in two primary ways: 1) we take advantage of some of the synchronization paradigms, such as barriers, that exist in SPMD programs, and 2) our machine targets include distributed memory architectures where the cost of a memory fence is essentially that of a round-trip communication across the network.

The earliest implementation work on cycle detection was by Krishnamurthy and Yelick for the restricted case of SPMD programs [29]. That was done in a simplified subset of the Split-C language and introduced a polynomial time algorithm for cycle detection in SPMD programs. They also used synchronization analysis to reduce the number of fences, but their source language did not have the restriction that barriers must match textually and they did not take advantage of single conditionals. At compile time, they generated two versions of the code, one assuming the barriers line up and the other one not. At runtime, they switched between the two versions depending on how the barriers were executed. Our approach does not suffer the same runtime overhead and code bloat that exists in theirs. In addition, their compiler used only a simple type-based alias analysis.

There has also been work done in the area of reducing the number of fences required to enforce sequential consistency. Liblit, Aiken, and Yelick developed a type system to identify shared data accesses in Titanium programs [34],

and for sequential consistency, they only insert a fence at each shared data access identified. Based on our experimental results in §5, our technique is a significant improvement over theirs in terms of static fence count, dynamic fence count, and running time of the generated programs.

7 Conclusion

The global address space abstraction is a powerful programming model for shared memory machines, distributed memory machines, and hybrid mixtures of the two. Prior work has shown the expressive value of these languages [51, 12, 17] and the performance benefits of being able to directly read and write remote memory [8]. In this report, we presented critical program analyses for concurrency and pointers, necessary for detecting many types of program errors and enabling optimizations.

We introduced a graph representation of parallel programs with textually aligned barriers and two different concurrency analyses. The first was a basic concurrency analysis that uses barriers and single-valued expressions, and the second a more complex one that only explores those execution paths across function calls that can occur in practice. In addition, we presented a pointer analysis for languages with a hierarchical machine model that matches the current trend for building large-scale systems.

We also implemented two clients of the analyses in a compiler for the Titanium language. In the first, programs were analyzed at compile-time to report potential data races. The second was to leverage the analyses to minimize the cost of guaranteeing a sequentially consistent execution of a parallel program. Both clients benefited significantly from the analyses, with the former reporting far fewer false data races than without them, and the performance of the latter approaching that of the default relaxed memory model.

The increasing use of software-exposed parallelism will make parallel languages more common, and the kinds of analysis presented in this report will be critical to performing optimizations and detecting errors. The ability to perform such analyses may affect a language designer's choice of programming model semantics. Simpler programming models, such as those that prohibit races, use synchronous communication, or ensure a strong memory model, may be feasible if accurate analyses can be developed to enable optimizations while ensuring a stronger semantics. The hierarchical nature of machines at the high end is also increasing, and while three levels is the most exposed in any of the current PGAS languages, we expect that the desire for control over data layout and increasing complexity of networks are likely to result in more levels of hierarchy in these machines. Our results indicate that providing a simpler programming model and exposing the memory hierarchy within the language can balance the desire of programmers for both simplicity and high performance.

References

[1] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1998.

[2] A. Aiken and D. Gay. Memory management with explicit regions. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, June 1998.

[3] 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.

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

[5] Applied Numerical Algorithms Group (ANAG). Chombo. `http://seesar.lbl.gov/ANAG/software.html`.

[6] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 382–400, New York, NY, USA, 2000. ACM Press.

[7] 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.

[8] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, 2006. Also available as Lawrence Berkeley National Lab Tech Report LBNL-59207.

[9] 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.

[10] D. Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, University of California, Berkeley, November 2002.

[11] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002. ACM Press.

[12] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity Analysis of the UPC Language. In *IPDPS*, 2004.

[13] 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.

[14] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 298–309, New York, NY, USA, 1998. ACM Press.

[15] M. Christiaens and K. De Bosschere. TRaDe, a topological approach to on-the-fly race detection in Java programs. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, April 2001.

[16] Cray Inc. *Chapel Specification 0.4*, Feb. 2005.

[17] 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.

[18] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 85–96, New York, NY, USA, 1991. ACM Press.

[19] E. Duesterwald and M. Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *Symposium on Testing, analysis, and verification*, Victoria, British Columbia, October 1991.

[20] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232, New York, NY, USA, 2000. ACM Press.

[21] D. Gay. *Barrier Inference*. PhD thesis, University of California, Berkeley, May 1998.

[22] D. Gay and A. Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, June 2001.

[23] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[24] 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.

[25] 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.

[26] T. Jeremiassen and S. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, August 1994.

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

[28] 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.

[29] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computations*, 1996.

[30] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[31] J. Lee, S. Midkiff, and D. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proceedings of 1999 ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, May 1999.

[32] J. Lee and D. Padua. Hiding relaxed memory consistency with compilers. In *Parallel Architectures and Compilation Techniques*, Barcelona, Spain, September 2001.

[33] 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.

[34] B. Liblit, A. Aiken, and K. Yelick. Type systems for distributed data sharing. In *International Static Analysis Symposium*, San Diego, California, June 2003.

[35] S. Masticola and B. Ryder. Non-concurrency analysis. In *Proceedings of the Fourth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, May 1993.

[36] 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.

[37] R. Numwich and J. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

[38] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, New York, NY, USA, 2003. ACM Press.

[39] OpenMP specifications. http://www.openmp.org.

[40] T. Reps. Program analysis via graph reachability. In *ILPS '97: Proceedings of the 1997 international symposium on Logic programming*, pages 5–19, Cambridge, MA, USA, 1997. MIT Press.

[41] 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.

[42] V. Saraswat. *Report on the Experimental Language X10, Version 0.41*. IBM Research, Feb. 2006.

[43] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[44] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[45] Silicon Graphics. CF90 co-array programming manual. Technical Report SR-3908 3.1, Cray Computer, 1994.

[46] Z. Sura, X. Fang, C. Wong, S. Midkiff, and D. Padua. Compiler techniques for high performance sequentially consistent Java programs. In *Proceedings of the 2005 ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, Chicago, Illinois, June 2005.

[47] Z. Sura, C. Wong, X. Fang, J. Lee, S. Midkiff, and D. Padua. Automatic implementation of programming language consistency models. In *Proceedings of the 15th Workshop on Workshop on Languages and Compilers for Parallel Computing*, College Park, Maryland, July 2002.

[48] The UPC Consortium. *UPC Language Specifications, Version 1.2*, May 2005.

[49] United States Department of Defense. Reference manual for the Ada programming language. Technical Report ANSI/MIL-STD-1815A, Washington, D.C., January 1983.

[50] C. von Praun and T. R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 115–128, New York, NY, USA, 2003. ACM Press.

[51] T. Wen and P. Colella. Adaptive mesh refinement in titanium. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[52] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.

[53] 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.

[54] 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.

A Concurrency Analysis Details

In this appendix, we provide soundness proofs for the concurrency algorithms in §3. In addition, we discuss some of the optimizations our implementation performs in order to reduce the running time of the analyses.

A.1 Soundness

We prove the theorems given in §3, and the correctness of the concurrency analyses.

A.1.1 Theorem 3.4

In order to prove Theorem 3.4, we need to first prove Lemma 3.6:

Proof (of Lemma 3.6). Suppose a and b are never in the same code phase. Then they are always preceded by two different barriers. Consider arbitrary occurrences of a and b in any program execution in which they both occur. (If one or both don't occur, then they trivially don't run concurrently.) Let B_a and B_b be the barriers preceding a and b, respectively. Since every thread executes the same set of barriers, either B_a precedes B_b or B_b precedes B_a. Since a occurs after B_a but before any other barrier, and b occurs after B_b but before any other barrier, this implies that a and b are separated by a barrier. Thus, a and b cannot run concurrently, since a barrier prevents the code before it and after it from executing concurrently. □

We can now prove Theorem 3.4:

Proof (of Theorem 3.4). Suppose a and b can run concurrently. By Lemma 3.6, a and b must be in the same code phase S. By Definition 3.5, there must be program flows from the initial barrier B_S to a and b that do not go through barriers. There are three cases:

Case 1: There is a program flow from a to b in S. This means the control flow graph of the program must contain a path from the node for a to the node for b that does not pass through a barrier. Since G contains all nodes and edges of the control flow graph except those corresponding to barriers, it also contains such a path, so b is reachable from a.

Case 2: There is a program flow from b to a in S. This case is analogous to the one above.

Case 3: There is no program flow either from a to b or from b to a in S. Since there is a flow from B_S to a and from B_S to b, a and b must be in different branches of a conditional C. Since only one branch of a single conditional can run, C must be a non-single conditional in order for a and b to run concurrently. Without loss of generality, let a be in the first branch, and b be in the second. Since C is non-single, it cannot contain a barrier, and the end of the first branch is reachable in G from a without hitting a barrier. Similarly, b is reachable from the beginning of the second branch without executing a barrier. Since G contains a cross edge from the first branch of C to the second, this implies that there is a path from a to b in G that does not pass through a barrier. □

A.1.2 Method Contexts

The feasible paths algorithm in §3.3 only visits each method in a single context. The following implies that this, in addition to visiting each method in no context, is sufficient:

Theorem A.1. *Ignoring the effect of the arguments, the set of expressions that may be executed in a call to a method* f *is the same regardless of the context in which* f *is called.*

Proof (by Induction Over the Function Call Depth).
Base case: The execution of f makes no method calls. Then the call to f can execute at most those expressions that are contained in f and reachable from its entry regardless of the calling context.
Inductive step: The execution of f makes method calls. By the inductive hypothesis[9], each method call in f can transitively execute the same expressions independent of the context. In addition, the call to f can execute exactly those expressions that are contained in f and reachable from its entry. The call to f thus can execute the same set of expressions regardless of context. □

A.1.3 Algorithm 3.8

We prove the correctness of Algorithm 3.8, used by the feasible paths analysis.

Theorem A.2. *Algorithm 3.8 marks all methods that can complete using any number of stack frames.*

Proof. Suppose there are some methods that can complete but that Algorithm 3.8 does not find. Out of these methods, let f be the one that can complete with the minimum number of stack frames j. In order for f to require j frames to complete, there must be an execution path through f that only calls methods that require at most $j-1$ frames to complete. These methods must all be marked, since f is the minimum method that isn't marked. Let i be the iteration in which the last of these methods is marked. Since a method is marked in this iteration, loop 3 will iterate at least once more. Since f now has a path in which it only calls marked methods, f will be marked in the $(i+1)$th iteration. This is a contradiction, so Algorithm 3.8 marks all methods that can complete. □

[9] In order for induction be be applicable, the function call depth in f must be finite. It is reasonable to assume that this is always the case, since in practice, an infinite function call depth is impossible due to finite memory limits.

A.1.4 Algorithm 3.9

We now prove that Algorithm 3.9, which is used by the feasible paths analysis, only follows feasible paths.

Theorem A.3. *Algorithm 3.9 does not follow any infeasible paths.*

Proof. Consider an arbitrary infeasible path p. In order for p to be infeasible, the labels along p must form a string in which an open parenthesis $(_\alpha$ is closed by a non-matching parenthesis $)_\beta$. Consider the execution of Algorithm 3.9 on this path. An open parenthesis is pushed onto the the stack s when it is encountered, so before any close parentheses are encountered, the top of the stack is the most recently opened parenthesis. A close parenthesis causes the top of the stack to be popped, so in general, the top of the stack is the most recently opened parenthesis that has not yet been closed. Now consider s when the label $)_\beta$ is reached. The symbol $(_\alpha$ must be on the top of s, since $)_\beta$ closes it. But Algorithm 3.9 checks the top of the stack against the newly encountered label, and since they don't match, it does not proceed along p. □

A.2 Optimizations

We perform a few optimizations to decrease the running time of our implementation.

- **Dynamic method calls**: As shown in §3.1.1, we rewrite dynamic method calls as conditionals. In order to maintain correctness, a conditional branch must be added for each possible target of the call. Our implementation uses pointer analysis to determine the possible targets, reducing the number of required branches.
- **Private data**: Private data can only be accessed by the thread that created them. Since a race condition requires simultaneous access by multiple threads, accesses to private data cannot occur in a race condition. Our implementation ignores all such accesses.
- **Graph compaction**: The control flow graph contains many nodes that are irrelevant to our analysis. Since our concurrency analyses are quadratic in the number of graph nodes, it is beneficial to remove these nodes before the analysis. Our experiments show that on average, about 95% of nodes can be removed.

B Pointer Analysis Details

In this appendix, we elaborate on the pointer analysis in §4. We provide the operational semantics for *Ti* and prove the soundness of the most complicated inference rule in §4.2.4. In addition, we discuss some implementation details of the analysis and some more applications that can benefit from it.

B.1 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.

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'. As in §4.2.4, 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.

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

B.2 Soundness

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 soundness 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. 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, $a_{2'}^m$ must be added to the points-to set of a_1^m. For consistency, $a_{2'}^m$ must also be added to the points-to set of any abstract location

$a^m_{1'} \sqsubseteq a^m_1$, since any of the concrete locations corresponding to $a^m_{1'}$ can point to any of the concrete locations corresponding to $a^m_{2'}$.

Thus, the abstract location $a^m_{2'} = (l_2, max(n^m_1, n^m_2))$ must be added to the points-to set of any $a^m_{1'} = (l_1, n^m_{1'})$ such that $n^m_{1'} \leq n^m_1$.

- On a machine m', where $hier(m, m') > n^m_1$. The set of concrete locations corresponding to $a^{m'}_1$ all reside on machines a distance of $n^m_{1'} = hier(m, m')$ away from machine m. Thus, $a^{m'}_1 \sqsubseteq a^m_{1'}$, where $a^m_{1'} = (l_1, n^m_{1'})$.

  Now suppose $n^m_2 < n^m_{1'}$. Then all the concrete locations corresponding to $a^{m'}_2$ reside at a distance of $n^m_{1'}$ from machine m, so that $a^{m'}_2 \sqsubseteq a^m_{2'}$, where $a^m_{2'} = (l_2, n^m_{1'})$. Thus, the source locations can be soundly approximated by $a^m_{2'}$.

  Suppose instead that $n^m_2 \geq n^m_{1'}$. Then m and m' agree on $a^{m'}_2 = a^m_2$, so the source locations correspond to a^m_2.

  In either case, some of the concrete locations corresponding to $a^m_{1'}$ can now point to some of the concrete locations corresponding to $a^m_{2'} = (l_2, max(n^m_{1'}, n^m_2))$. Soundness can be maintained, though precision lost, if the analysis assumes that any concrete location corresponding to $a^m_{1'}$ can point to any concrete location corresponding to $a^m_{2'}$. Thus, $a^m_{2'}$ should be added to the points-to set of $a^m_{1'}$.

  Now consider an abstract location $a^m_{1''} = (l_1, n^m_{1''})$, where $n^m_{1''} < n^m_{1'}$. All concrete locations represented by $a^m_{1''}$ reside less than a distance of $n^m_{1'}$ away from m. Since all concrete locations corresponding to $a^{m'}_1$ reside at a distance of $n^m_{1'}$ from m, the abstract locations $a^m_{1''}$ and $a^{m'}_1$ do not intersect. Thus, none of the concrete locations in $a^m_{1''}$ are targeted by the assignment, so its points-to set does not need to be updated.

  Thus, the abstract location $a^m_{2'} = (l_2, max(n^m_{1'}, n^m_2))$ must be added to the points-to set of each $a^m_{1'} = (l_1, n^m_{1'})$ such that $n^m_{1'} > n^m_1$.

Summarizing over all possibilities, we obtain the rule that the abstract location $a^m_{2'} = (l_2, max(n^m_{1'}, n^m_1, n^m_2))$ must be added to the points-to set of any $a^m_{1'} = (l_1, n^m_{1'})$. This corresponds exactly to the update rule provided in §4.2.4.

B.3 Implementation

We have implemented a prototype of the pointer analysis in the Titanium compiler. For evaluation purposes, we implemented three variants of the analysis, with one, two, and three levels of hierarchy. The single-level analysis combines all three levels and cannot be used for either locality or sharing inference. In two-level analysis, level 1 remains separate while levels 2 and 3 are combined. Level 1 must be separate in order to perform sharing analysis, and this separation still allows locality inference, though with less precision than combining levels 1 and 2. Finally, the three-level analysis separates all three levels, providing the most precise results.

B.3.1 Titanium Features

The *Ti* language is much simpler than Titanium, and certain Titanium features require special treatment:

- **types**: Objects in Titanium have types, so the corresponding abstract locations are also typed.
- **fields**: Objects can have multiple fields, so an abstract location must have points-to sets for each of its fields.
- **arrays**: Arrays can have multiple entries. For simplicity, the analysis makes no attempt to distinguish between the different entries of an array.
- **method calls**: Methods may have parameters, return values, and a `this` value. The analysis considers each of these to be variables, and the result of a method call is the set of abstract locations corresponding to its return variable.
- **dynamic dispatch**: A method call on an object may dispatch to different targets at runtime. The analysis can compute a conservative but precise estimate of the possible dispatch targets by examining the types of the abstract locations corresponding to the source object.

- **native code**: Native methods are handled conservatively for the most part. However, the analysis assumes that a native method does not violate type safety, and that it does not modify the fields of an object in certain ways. Native library methods are treated specially by the analysis if they violate these assumptions.

B.3.2 Optimizations

A handful of optimizations were applied to the pointer analysis, focusing on increasing functionality over efficiency. Execution time can likely be improved drastically by using binary decision diagrams [52].

B.3.2.1 Reachability

Titanium is to a large extent backwards compatible with Java, providing most of its language features and much of its library. The typical Titanium program uses only a small portion of the Java library, so analyzing the entire library is unnecessary. The pointer analysis implementation only analyzes those methods that are reachable from the program entry point and static initializers. It does so by marking the `main()` method and static initializers reachable, and the rest of the methods unreachable. When the analysis encounters a call to an unreachable method, it makes the method reachable and proceeds to analyze it. This is continued until a fixed point is reached.

B.3.2.2 Lazy Creation of Abstract Locations

Theoretically, the pointer analysis requires $A \cdot h$ abstract locations, where A is the number of allocation sites and h is the number of levels in the analysis. However, if a particular thread-local abstract location is never leaked beyond its creator thread, the analysis never uses the wider versions of the location. The implementation takes advantage of this fact by only creating the wider counterparts on demand if the thread-local version is leaked.

B.3.3 Performance

Though our implementation is not as optimized as possible, its performance still demonstrates some interesting results. As expected, the reachability optimization is very effective, decreasing execution time by an average of almost 70%. The performance difference between one, two, and three levels of hierarchy is nonexistent, with all three averaging 0.85 seconds on a test benchmark when run on a 2.4GHz Pentium 4. This validates our decision to allow an arbitrary number of levels in the analysis, since execution time would increase linearly with the number of levels if a two-level analysis was used multiple times instead.

B.4 Applications

The pointer information computed in §4.2 can be applied to multiple analyses and optimizations for parallel programs. We show how to apply it to two clients, locality inference and sharing inference.

B.4.1 Locality Inference

Pointer information can be used to infer an upper bound on the width of a particular reference or expression. All referents of an expression must be contained within its width, so if an expression e of reference type evaluates to the abstract set S, an upper bound on its width is:

$$w_{upb} = max\{n \mid (l, n) \in S\}$$

A reference is *local* if it can only be to the same physical address space as the source thread. In the two-level pointer analysis, a reference is local if its width is bounded from above by 1, while in the three-level analysis, it is local if its width is bounded by 2.

B.4.2 Sharing Inference

An object in a parallel program is *private* if it is never leaked beyond its source thread. A reference is private if it can only refer to private objects. As described in §B.3.2, if an abstract location can be leaked, our pointer analysis implementation creates wide versions of it. Thus, an abstract location must be private if it has no wide counterparts, and a variable or expression is private if it evaluates to an abstract set that only contains private locations. Note that this inference is independent of the number of levels in the analysis hierarchy, as long as level 1 is separate from the rest of the levels in the analysis.