Concurrency Analysis for Parallel Programs with Textually Aligned Barriers

Amir Kamil Katherine Yelick

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

Abstract. A fundamental problem in the analysis of parallel programs is to determine when two statements in a program may run concurrently. This analysis is the parallel analog to control flow analysis on serial programs and is useful in detecting parallel programming errors and as a precursor to semantics-preserving code transformations. We consider the problem of analyzing parallel programs that access shared memory and use barrier synchronization, specifically those with textually aligned barriers and single-valued expressions. We present an intermediate graph representation for parallel programs and an efficient interprocedural analysis algorithm that conservatively computes the set of all concurrent statements. We improve the precision of this algorithm by using context-free language reachability to ignore infeasible program paths. We then apply the algorithms to static race detection and enforcing a sequentially consistent execution in the Titanium programming language and show that both can benefit from the concurrency information provided.

1 Introduction

As the rate of scaling of uniprocessor machines slows down, application writers and system vendors alike have been turning to multiprocessor machines for performance. Most major CPU manufacturers have products or plans for chips with multiple cores, so that parallelism once hidden within the micro-architecture will now be exposed to the assembly language and, in all likelihood, to application level software. Such systems are modeled after SMP multiprocessors and allow all processors to simultaneously access shared memory. In addition, for large-scale parallel machines there is increasing interest in global address space languages, which give programmers the illusion of a shared memory machine on top of distributed memory machines and clusters. Analysis and optimization of parallel shared memory code is increasingly important in both of these settings.

In this paper we introduce an *interprocedural concurrency analysis* for programs with barrier synchronization, which captures information about the potential concurrency between statements in a program. The analysis is done for the Titanium language [29], a single program, multiple data global address space variation of Java that runs on most parallel and distributed memory machines. We first construct a *concurrency graph* representation of a program, taking advantage of two features of the Titanium language parallel execution model: *textual barrier alignment*, which statically guarantees that all threads reach the same textual sequence of barriers, and *single-valued* expressions, which provably evaluate to the same value on all threads [1]. We then present a simple algorithm that uses the concurrency graph to determine the set of all 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.

Concurrency analysis can be used to improve the quality of other analyses and to enable optimizations. To demonstrate the usefulness of our concurrency analysis, we apply it to two client problems. The first is data race analysis, which can be used to report potential program errors to application programmers. The second is *memory consistency model enforcement*, which can be used to provide a stronger and more intuitive memory model while still allowing the compiler and hardware to reorder memory operations in many instances. In related work with Su [15], we demonstrate that memory model enforcement can have a significant negative impact on optimizations as well, but that this effect is mitigated when combined with our concurrency analysis. In this paper, we focus on the foundations of the concurrency analysis problem: how it can be performed efficiently and be made accurate enough to effectively increase the precision of both clients on a set of application benchmarks.

2 Motivation

Concurrency information is useful for many program analyses and optimizations. We focus on two clients that stand to benefit from this information: static race detection and enforcing sequential consistency.

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 [21]. Data races often correspond to programming errors and potentially result in non-deterministic runtime behavior. Concurrency analysis can be used to statically detect races at

compile-time [11,12], particularly when combined with alias analysis [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 memory 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.

Among the various models, *sequential consistency* [17] 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 [25]. For example, for the accesses $\{x, y, a, b\}$ in Figure 1, the behavior in which y reads the value 1 and b reads the value 0 is not sequentially consistent, since it does not reflect an interleaving in which the order of the individual execution sequences is preserved.

In order to enforce sequential consistency, *memory barriers* must be inserted to prevent reordering of memory operations by the compiler or architecture. Memory barriers prevent optimizations such as prefetching and code motion, and can result in an unacceptable performance penalty [19]. The *cycle detection* algorithm computes the minimal set of memory barriers needed to enforce sequential consistency [25,16]. Cycle detection can benefit from concurrency information, since it can ignore pairs of memory operations that cannot run concurrently [26,15].

3 Titanium Background

Titanium is a dialect of Java, but 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 [6], 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 (CLUMPS), and a number of specific supercomputer architectures (Cray X1, Cray T3E, SGI Altix, IBM SP, Origin 2000, and NEC SX6). Instead of having dynamically created threads as in Java, Titanium is a *single program, multiple data* (SPMD) language, so all threads execute the same code image.

3.1 Textually Aligned Barriers

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 [1,13]. The following code is not structurally correct:

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

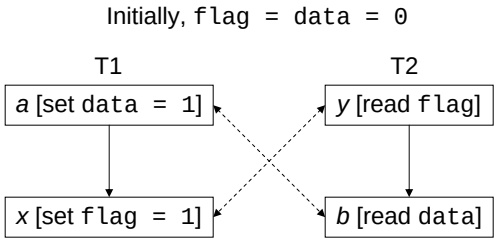
Titanium provides a stronger guarantee of *textually aligned barriers*: not only do all threads execute the same number of barriers, they also execute the same *textual* sequence of barriers. Thus, both the above structurally incorrect code and the following structurally correct code are erroneous in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

In order to enforce that a program correctly align barriers, Titanium makes use of *single-valued* expressions [1]. Such expressions evaluate to the same value for all threads, and a combination of programmer annotation and compiler inference is used to statically determine which expressions are single-valued. A conditional may only contain a barrier, or a call to a method with a barrier, if it is guarded by a single-valued expression: the above code is erroneous since `Ti.thisProc() % 2 == 0` is not single-valued. Our concurrency analysis also exploits such expressions and conditionals to determine which conditional branches can run concurrently.

Our concurrency analysis operates on the existing barriers in a program – no additional barriers are inserted. The analysis also ignores the lock-based `synchronized` construct of Java, since it is rarely used in Titanium programs.



y sees effect of x	b sees effect of a	possible sequential order
yes	yes	$a \Rightarrow x \Rightarrow y \Rightarrow b$
yes	no	none
no	yes	$a \Rightarrow y \Rightarrow b \Rightarrow x$
no	no	$y \Rightarrow b \Rightarrow a \Rightarrow x$

Fig. 1. A program fragment consisting of four memory accesses in two threads. The solid edges correspond to order in the execution stream of each thread, and the dashed edges are conflicts. Of the four possible results of thread 1 visible to thread 2, the second is illegal since it does not correspond to an overall execution sequence in which operations are not reordered within a thread.

3.2 Memory Model

Titanium's memory consistency semantics are a *relaxed model* similar to Java's, providing few ordering guarantees. In order to guarantee sequential consistency, memory barriers must be inserted into a program to enforce order.

3.3 Intermediate Language

In this paper, we will 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 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.

3.4 Control Flow Graphs

The algorithms in this paper 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 call node into a 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.

4 Concurrency 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 4.1 (Single Conditional). A *single conditional* is a conditional guarded by a single-valued expression.

Since a single-valued expression 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 4.2 (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

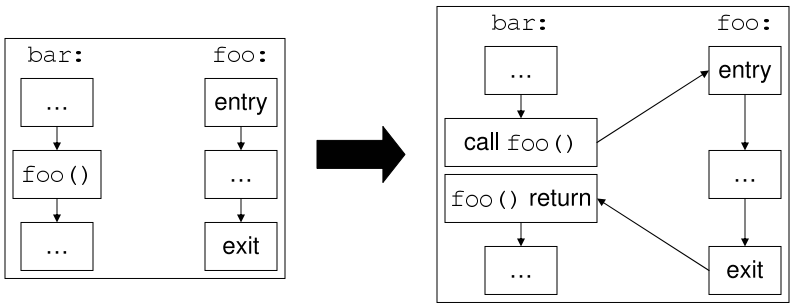


Fig. 2. Construction of the interprocedural control flow graph of a program from the individual method flow graphs.

Algorithm 4.3.
ConcurrencyGraph(P : program) : graph

1. Let G be the interprocedural control flow graph of P, as described in §3.4.
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 B from G.
8. Return G.

Fig. 3. Algorithm 4.3 computes the concurrency graph of a program by inserting cross edges into its control flow graph and deleting all barriers.

determining concurrency information, as shown in Theorem 4.4.

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 4.3 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 $O(n)$.

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

Theorem 4.4. *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 4.4, we require the following definition:

Definition 4.5 (Code Phase). For each barrier in a program, its *code phase* is the set of statements that can execute after the barrier but before hitting another barrier, including itself[1].

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

Proof. Suppose a and b are not in the same code phase. Then they are preceded by two different barriers B_a and B_b. 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.) 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. □

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

```
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();
LA: j += i;
```

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

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

Now we can prove Theorem 4.4:

Proof (of Theorem 4.4). Suppose a and b can run concurrently. By Lemma 4.6, a and b must be in the same code phase S. By Definition 4.5, 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 is 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 from either a to b or 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. □

By Theorem 4.4, in order to determine the set of all 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 4.7 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 $O(n^2)$.

5 Feasible Paths

Algorithm 4.7 computes an over-approximation of the set of concurrent expressions. In particular, due to the nature of the interprocedural control flow graph constructed in §3.4, the depth first searches in Algorithm 4.7 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 [23]. 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.

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.

Algorithm 4.7.
ConcurrentExpressions(P : program) : set

1. Let $concur \leftarrow \emptyset$.
2. Let $G \leftarrow$ **ConcurrencyGraph**(P) [Algorithm 4.3].
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 4.7 computes the set of all concurrent expressions in a given program.

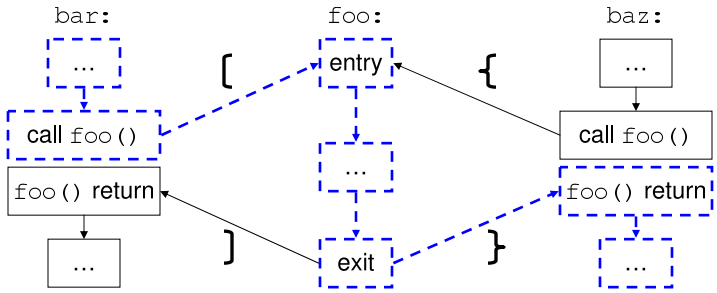


Fig. 6. Interprocedural control flow graph for two calls to the same function. The dashed path is infeasible, since `foo()` returns to a different context than the one from which it was called. The infeasible path corresponds to the unbalanced string "[}".

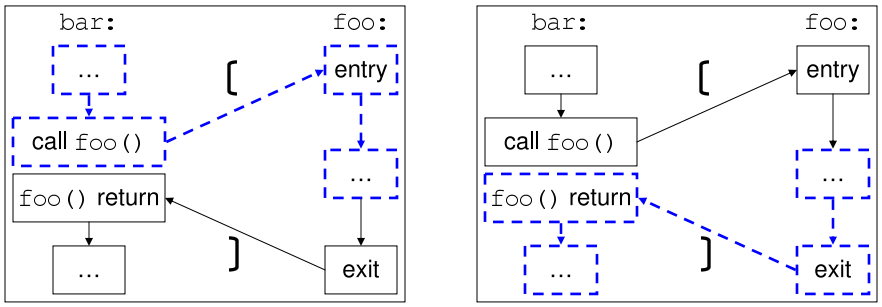


Fig. 7. Feasible paths that correspond to unbalanced strings. The dashed path on the left corresponds to a method call that has not yet returned, and the one on the right corresponds to a path that starts in a method call that returns.

Determining the set of nodes reachable[2] 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 [23]. Algorithm 4.7 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, the following implies that it is only necessary to visit the method in a single context:

Theorem 5.1. *Assuming nothing about the arguments, the set of expressions that can be executed in a call to a method* f *is the same regardless of the context in which* f *is called.*

Proof (by Induction).
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[3], 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. □

Since the set of expressions that can be executed in a method call is the same regardless of context, 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:

- The nodes reachable following the method call. If the method call can complete, then the nodes after a method call are reachable from a point before the method call.
- When no context exists, such as in a search that starts from a point within a method f. Then all nodes that are reachable following any method call to f are reachable.

[2] 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.

[3] 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.

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.

5.1 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 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 5.2 in Figure 8 does so 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 only calling methods that don't need to make any calls, or equivalently, those methods that can complete using only two stack 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].

Theorem 5.3. *Algorithm 5.2 marks all methods that can complete using any number of stack frames.*

Proof. Suppose there are some methods that can complete but that Algorithm 5.2 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 was the minimum method that wasn't marked. Since f requires j frames, at least one of the methods called must require $j-1$ frames and thus was marked in the $(j-1)$th iteration of loop 3 above. Loop

[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 5.2 does not mark such methods.

Algorithm 5.2.
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 5.2 uses each method's intraprocedural control flow graph to determine if its exit is reachable from its entry.

3 will thus iterate at least once more, and since f now has a path in which it only calls marked methods, f will be marked, which is a contradiction. Thus Algorithm 5.2 marks all methods that can complete. □

Algorithm 5.2 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 $O(n)$.

5.2 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, it 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 5.4 in Figure 9 formalizes this procedure.

Theorem 5.5. *Algorithm 5.4 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 5.4 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 5.4 checks the top of the stack against the newly encountered label, and since they don't match, it does not proceed along p. □

Since G contains bypass edges and Algorithm 5.4 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 $O(n)$.

[5] Even on the largest example we tried (>45,000 lines of user and library code, 1226 methods), Algorithm 5.2 required only five iterations to converge.

5.3 Feasible Concurrent Expressions

Putting it all together, we can now modify Algorithm 4.7 to find only concurrent expressions that are feasible. As in Algorithm 4.7, the concurrency graph G must first be constructed. Then the intraprocedural flow graphs of each method must be constructed, Algorithm 5.2 used to find the methods that can complete without hitting a barrier, and the bypass edges inserted into G. Then Algorithm 5.4 must be used to perform the searches instead of a vanilla depth first search. Algorithm 5.6 in Figure 10 illustrates this procedure.

The setup of Algorithm 5.6 calls Algorithm 5.2, so it takes $O(kn)$ time. The searches each take time $O(n)$, and at most n are done, so the total running time is $O(kn + n^2) = O(n^2)$, quadratic as opposed to the cubic running time of generic CFL reachability.

6 Evaluation

We evaluate our concurrency analysis 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:

- **`gas`** [5] (8841 lines): Hyperbolic solver for a gas dynamics problem in computational fluid dynamics.
- **`gsrb`** (1090 lines): Nearest neighbor computation on a regular mesh using red-black Gauss-Seidel operator. This computational kernel is often used within multigrid algorithms or other solvers.
- **`lu-fact`** (420 lines): Dense linear algebra.
- **`pps`** [4] (3673 lines): Parallel Poisson equation solver using the domain decomposition method in an unbounded domain.
- **`spmv`** (1493 lines): Sparse matrix-vector multiply.

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.

6.1 Static Race Detection

Using our concurrency analysis and a thread-aware alias analysis, 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 alias and concurrency analyses ensure that

Algorithm 5.4.
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 5.4 computes the set of nodes reachable from the start node through a feasible path.

Algorithm 5.6.
FeasibleConcurrentExpressions(P : program) : set

1. Let $G \leftarrow$ **ConcurrencyGraph**(P) [Algorithm 4.3].
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 5.2].
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 5.4].
15. For each expression $b \in visited$:
16. Insert (a, b) into $concur$.
17. } // End for (13).
18. Return $concur$.

Fig. 10. Algorithm 5.6 computes the set of all concurrent expressions that can feasibly occur in a given program.

no false negatives occur.) We therefore consider a race detector that reports the fewest races to be the most effective.

Table 1. Number of data races detected by the **base** level of analysis.

Benchmark	Races Detected
`gas`	1410
`gsrb`	33
`lu-fact`	7
`pps`	80
`spmv`	15

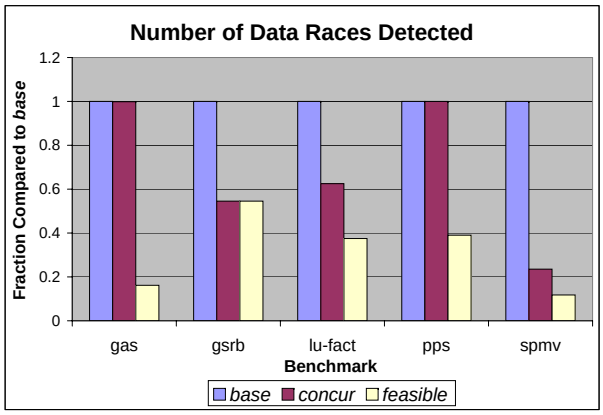


Fig. 11. Fraction of data races detected at compile-time compared to **base**.

Figure 11 compares the effectiveness of three levels of race detection:

- **base**: only alias analysis is used to detect potential races
- **concur**: our basic concurrency analysis (§4) is used to eliminate non-concurrent races
- **feasible**: our feasible paths concurrency analysis (§5) is used to eliminate non-concurrent races

For reference, the number of races detected by the **base** analysis is reported in Table 1.

The results show that the addition of concurrency analysis can eliminate most of the races reported by our detector. Two of the benchmarks do not benefit at all from the basic concurrency analysis, but all benefit considerably from the feasible paths analysis. The concurrency analysis should be of significant help to users of our race detector by weeding out many false positives.

6.2 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.

Table 2. Number of static and dynamic barriers required by the **base** level of analysis.

Benchmark	Static Memory Barriers	Dynamic Memory Barriers
`gas`	346	3.3M
`gsrb`	128	120K
`lu-fact`	14	1.6M
`pps`	286	94M
`spmv`	34	9.4M

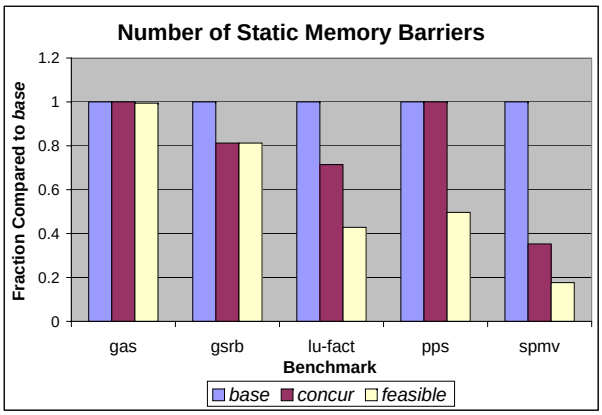


Fig. 12. Fraction of memory barriers generated at compile-time compared to **base**.

Figure 12 compares the number of memory barriers generated for each program using different levels of analysis:

- **base**: cycle detection is used to determine the minimal number of memory barriers

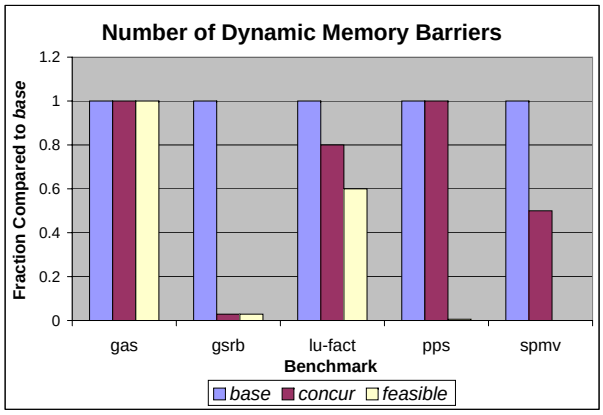


Fig. 13. Fraction of memory barriers executed at runtime compared to **base**.

- **concur**: our basic concurrency analysis (§4) is additionally used to eliminate memory barriers for pairs of non-concurrent memory accesses
- **feasible**: our feasible paths concurrency analysis (§5) is additionally used to eliminate memory barriers for pairs of non-concurrent memory accesses

Figure 13 compares the resulting dynamic counts at runtime. For reference, the number of static and dynamic memory barriers required by the **base** level of analysis is show in Table 2.

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. In only one benchmark, `gas`, is our analysis ineffective: while it does reduce the number of concurrent pairs detected, it does not significantly reduce the number of memory accesses that are a member of *some* pair (134 under **base** compared to 124 under **feasible**), preventing cycle detection from benefiting from the analysis.

It is interesting to note that eliminating infeasible paths is effective in three of the four benchmarks for which our analysis is useful. It should also be noted that most of the remaining memory barriers are due to imprecision in our supporting analyses, such as the inability of our alias analysis to distinguish array indices. Even so, our analysis significantly reduces the number of memory barriers required for enforcing sequential consistency.

7 Related Work

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 [11]. Their analysis is for detecting races in programs based on the Ada rendezvous model [27]. Masticola and Ryder developed a more precise non-concurrency analysis for the same set of programs [20]. 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 [14]. They used the information to reduce false sharing on cache-coherent machines.

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 [12]. Boyapati and Rinard developed a type system for Java that guarantees that a program is race-free [7]. Tools such as Eraser [24] and TRaDe [9] detect races at runtime instead of statically. Other static and dynamic race detection schemes have also been developed [28,3,10,8,22].

The concept of sequential consistency was first defined by Lamport [17]. 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* for general parallel programs [25]. Krishnamurthy and Yelick presented a practical cycle detection analysis for the restricted case of SPMD programs [16]. They also used concurrency analysis to reduce the number of memory barriers, but their non-textual barriers forced them to generate both an optimized and an unoptimized version of the code and to switch between them at runtime depending on how the barriers lined up. 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. [18]. More recently, Sura et al. used cooperating escape, thread structure, and delay set analyses to provide sequential consistency cheaply in Java [26].

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, unlike that of Krishnamurthy and Yelick, and precise. In addition, our analysis takes advantage of single-valued expressions, which no previous analysis does.

We presented a more abstract version of our concurrency analysis and its application to sequential consistency in a previous paper [15]. That analysis was slightly less precise, fol-

lowed infeasible program paths, and would have been much more difficult to modify to ignore them.

8 Conclusion

In this paper, we made several contributions to the foundation of parallel program analysis, specifically the concurrency analysis problem of determining whether two statements can execute concurrently. 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. We experimented with several benchmark programs using two client problems, data race detection and enforcing a sequentially consistent execution. Our experiments showed that the analyses were able to eliminate a large fraction of the false positives reported by a race detector in all programs and most of the memory barriers required to provide sequential consistency in all but one program. We believe the efficiency and precision of our concurrency analysis make it a very useful tool in analyzing parallel programs with textually aligned barriers.

In addition to aiding in optimizations and helping to detect parallel programming 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. Our analysis is one piece of a larger picture on the kinds of parallelism constructs and synchronization operations for which accurate concurrency analyses can be developed.

Acknowledgments

We would like to thank Jimmy Su, who helped us a great deal both in developing the concurrency algorithms and in implementing them. We would also like to thank the Titanium group for their valuable support.

References

1. A. Aiken and D. Gay. Barrier inference. In *Principles of Programming Languages*, San Diego, California, January 1998.
2. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
3. 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.
4. G. T. Balls. *A Finite Difference Domain Decomposition Method Using Local Corrections for the Solution of Poisson's Equation*. PhD thesis, Department of Mechanical Engineering, University of California at Berkeley, 1999.
5. M. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64–84, May 1989. Lawrence Livermore Laboratory Report No. UCRL-97196.
6. D. Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, University of California, Berkeley, November 2002.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. D. Gay. *Barrier Inference*. PhD thesis, University of California, Berkeley, May 1998.
14. T. Jeremiassen and S. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *Parallel Architectures and Compilation Techniques*, Montreal, Canada, August 1994.
15. A. Kamil, J. Su., and K. Yelick. Making sequential consistency practical in Titanium. In *Supercomputing 2005*, November 2005. To appear.
16. A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computations*, 1996.
17. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

18. 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.
19. J. Lee and D. Padua. Hiding relaxed memory consistency with compilers. In *Parallel Architectures and Compilation Techniques*, Barcelona, Spain, September 2001.
20. S. Masticola and B. Ryder. Non-concurrency analysis. In *Principles and practice of parallel programming*, San Diego, California, May 1993.
21. R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.
22. 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.
23. 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.
24. 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.
25. 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.
26. Z. Sura, X. Fang, C. Wong, S. Midkiff, and D. Padua. Compiler techniques for high performance sequentially consistent Java programs. In *Principles and Practice of Parallel Programming*, Chicago, Illinois, June 2005.
27. United States Department of Defense. Reference manual for the Ada programming language. Technical Report ANSI/MIL-STD-1815A, Washington, D.C., January 1983.
28. 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.
29. 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.