Concurrency Analysis for Sequential Consistency in Titanium

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

May 17, 2005

1 Introduction

As the limits of uniprocessor machines are being approached, application writers and system vendors alike have been turning to multiprocessor machines for performance. The major CPU manufacturers all have recently or will shortly introduce chips with multiple cores. Such systems, along with traditional multiprocessor machines, allow all processors to simultaneously access shared memory. This requires a *memory consistency model* to be defined, which determines in what order the memory updates on one processor appear to the other processors.

The memory consistency model can be specified at the level of a programming language, which is crucial for languages that can be used on a wide variety of hardware. Language designers have traditionally been very reluctant to use the simplest model, *sequential consistency*, in which memory operations appear to occur in the order specified in the original program. This reluctance is due to a perception that such a model incurs prohibitive performance penalties, since it prevents reordering of operations and requires memory fences to be inserted in order to force the underlying hardware to respect ordering. Language designers instead have used complicated models [15, 19] that aren't well-understood by programmers, or worse, ill-defined [8]. This is very problematic for programmers, since many common techniques such as spin-locks and presence bits depend on the details of the memory consistency model in order to function correctly.

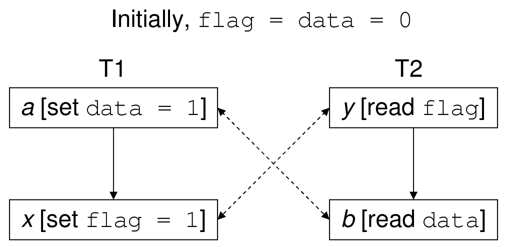
Various techniques have been proposed in order to decrease the cost of sequential consistency. In this paper, we present an interprocedural concurrency analysis for the Titanium programming language that can increase the precision of one such technique, *cycle detection*. We present both a basic algorithm and a modified one that only considers program execution paths that can occur in practice and prove that both algorithms are correct. We then apply these algorithms to a set of benchmarks, showing that they are effective in reducing the number of fences required to enforce sequential consistency in most of the benchmarks.

2 Background

2.1 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.

Among the various models, *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 [17]. For example, for the accesses $\{x, y, a, b\}$ in figure 1, the behavior in which b reads the value 1 and y 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.



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	$y \Rightarrow b \Rightarrow a \Rightarrow x$

Figure 1: A cycle consisting of four 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.

An easy way to enforce sequential consistency is to insert memory fences after each shared memory access. This forbids all reordering of shared memory operations, which prevents optimizations such as prefetching and code motion, resulting in an unacceptable performance penalty. Various techniques can be used to minimize the number of fences, or *delay set*, required to enforce sequential consistency [17, 7, 11].

2.1.1 Cycle Detection

Computing the minimal delay set for an arbitrary parallel program is an intractable NP-hard problem [7]. Krishnamurthy and Yelick proposed a polynomial time algorithm based on *cycle detection* for analyzing SPMD programs [7] such as Titanium. The analysis uses a graph where the nodes represent shared memory accesses. There are two types of edges in the graph: *program edges* and *conflict edges*. Program edges reflect the program order: there is a directed program edge from u to v if u can execute before v. Conflict edges are undirected edges between accesses that conflict: there is a conflict edge between u and v if u and v can access the same memory location and at least one of them is a write.

The goal of cycle detection is to check each program edge to see if it needs a fence to enforce its order. Given the program edge (u, v), if there is no local dependency between u and v, v could execute before u. If this reordering is observable by another thread, then sequential consistency is violated. In that case, a fence must be inserted between u and v to ensure that u always executes before v. Figure 1 gives one example of this. There is no local dependency on T1, but if the two writes on T1 were reordered, then the following execution order would be possible: $x \Rightarrow y \Rightarrow b \Rightarrow a$. This results in (y, b) reading the values $(1, 0)$, which means that the reordering on T1 is observable on T2. A fence must be placed between a and x to prevent such reordering.

Kirshnamurthy and Yelick [7] show that given a program edge (u, v), if there is a path from v to u where the first and last edge are conflict edges, and the intermediate edges are program edges, then the program edge (u, v) belongs to the minimal delay set and a fence must be placed between u and v to prevent reordering. The path together with the program edge (u, v) forms a *critical cycle*.

Concurrency information can be used to increase the precision of cycle detection. As shown in appendix A, a conflict edge in which the corresponding memory accesses cannot run concurrently can be ignored.

2.2 Titanium

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 [1], 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).

Titanium is a *single program, multiple data* (SPMD) language, which means that all threads execute the same code image. In addition, Titanium has the following unique features that our analysis relies on:

1. A call to a *barrier* in Titanium causes the calling thread to wait until all other threads have executed the same *textual*

instance of the barrier call. The code in the example below is not allowed because not all the threads will hit the same textual barrier. The Titanium compiler checks statically that all the barriers are lined up correctly [4].

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

Since a barrier forces threads to wait until all threads have reached the barrier, it prevents code before and after the barrier from running concurrently.

2. A *single-valued* expression evaluates to the same value for all threads. With programmer annotation and compiler inference, the Titanium compiler statically determines which expressions are single. Single-valued expressions are used to ensure that barriers line up: the above code is erroneous since `Ti.thisProc() % 2 == 0` is not single-valued.

Titanium's memory consistency model is defined in the language specification [5]. 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.

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

2.2.1 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.

2.2.2 Control Flow Graphs

A *control flow graph* 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. 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.

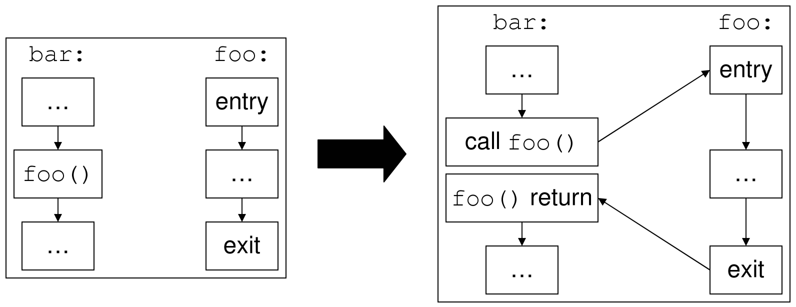


Figure 2: Construction of the interprocedural control flow graph of a program from the individual method flow graphs.

2.3 Program Analysis Correctness

Program analyses generally approximate the set of events that can actually occur during an execution. The concurrency analysis that we present is a *may analysis*, in that it finds events that may, but are not guaranteed to, occur in practice. Such an analysis is *sound* if it finds all events that may occur, and *precise* if it only finds events that will actually occur. In general, an analysis is considered correct if it is sound.

3 Concurrency Analysis

Concurrency information can be used to reduce the set of conflict edges for a program, and the resulting number of fences required to enforce sequential consistency. We present an algorithm for computing pairs of memory accesses that can run concurrently in a Titanium program, using barriers and single-valued expressions. Our algorithm makes use of the following definitions:

Definition 3.1 (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 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 accesses in a program, we construct a graph representation G of the program P as follows:

Algorithm 3.3 (P : program**).**
```
1. Let $G$ be the interprocedural control flow graph of $P$, as described in §2.2.2.
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. Return $G$.
```

Algorithm 3.3 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, the resulting graph G is also of size $O(n)$.

The graph G allows us to determine the set of concurrent accesses as follows:

```
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

Figure 3: The set of code phases for an example program.

Theorem 3.4. *Two memory accesses* a *and* b *in* P *can run concurrently only if one is reachable from the other in* G *along a path that does not pass through a barrier.*

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

Definition 3.5 (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 3 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 memory accesses* 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.

□

Now we can 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 is a super-graph of the control flow graph, it also contains such a path, so b is reachable from a without passing through a barrier.

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

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

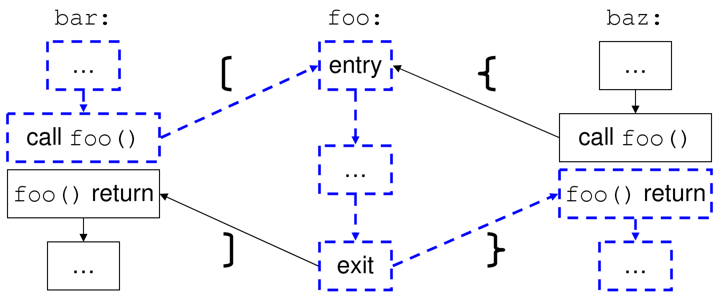


Figure 4: Interprocedural control flow graph for two calls to the same function. The dashed path is infeasible, since `foo()` returns to a different context than the one from which it was called. The infeasible path corresponds to the unbalanced string "[}".

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. □

The following algorithm then determines the set of all concurrent accesses $concur$:

Algorithm 3.7 (P : program**).**

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

By theorem 3.4, algorithm 3.7 correctly computes the set of all concurrent accesses. The running time of algorithm 3.7 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 m searches occur, where m is the number of memory accesses in P, so the algorithm runs in time O(mn).

4 Feasible Paths

Algorithm 3.7 computes an over-approximation of the set of concurrent accesses. In particular, due to the nature of the interprocedural control flow graph constructed in §2.2.2, the depth first searches in algorithm 3.7 can follow *infeasible paths*, paths that cannot actually occur in practice. Figure 4 illustrates such a path, in which a method is entered from one context and exits into another.

In order to prevent infeasible paths, we label each method call edge and corresponding return edge with matching parentheses, as shown in figure 4. Each path then corresponds to a string of parentheses composed of the labels of the edges in the path. The following determines if a path is feasible:

Theorem 4.1 ([16]). *A path is 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 are allowed by theorem 4.1:

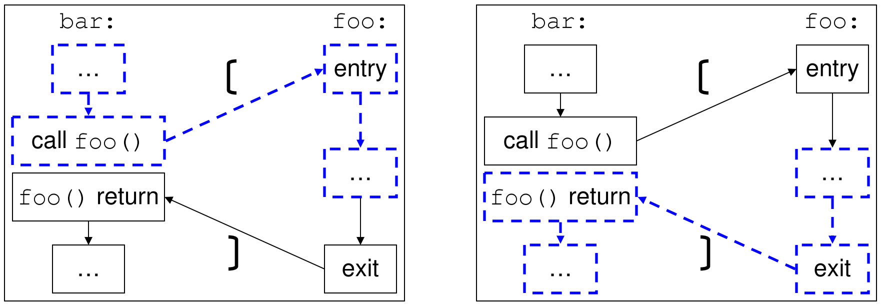


Figure 5: Feasible paths that correspond to unbalanced strings. The dashed path on the left corresponds to a method call that has not yet returned, and the one on the right corresponds to a path that starts in a method call that returns.

- 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 5.
- A path with close 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 5.

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

$$\begin{aligned} S &\rightarrow L\,R \\ L &\rightarrow S\,M \mid S\,)_\alpha \mid \epsilon \\ R &\rightarrow M\,R \mid (_\alpha\,R \mid \epsilon \\ M &\rightarrow (_\alpha\,M\,)_\alpha \mid M\,M \mid \epsilon, \end{aligned}$$

for each pair of matching parentheses $(_\alpha$ and $)_\alpha$. CFL reachability can be performed in cubic time for an arbitrary grammar [16].

Instead of performing generic CFL reachability, we modify the input graph G and the standard depth first search to obtain a fast algorithm for finding nodes reachable through a feasible path.

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 paths can be followed. However, the following implies that it is only necessary to visit the method in a single context:

Theorem 4.2. *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 exactly 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:

[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 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.

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.

4.1 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 from the target's entry, then it is always safe to add a *bypass edge* that connects the call node directly to the return node.

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. The following algorithm does so, operating over the intraprocedural control flow graphs of each method:

Algorithm 4.3 (P : program, $G_1, \ldots, G_k$: intraprocedural flow graph**).**

```
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 $false$.
```

Algorithm 4.3 continually iterates over all the method 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].

[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 ever completing. Algorithm 4.3 does not mark such methods.

Theorem 4.4. *Algorithm 4.3 marks all methods that can complete using any number of stack frames.*

Proof. Suppose there are some methods that can complete but that algorithm 4.3 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 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 4.3 marks all methods that can complete. □

Algorithm 4.3 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, much fewer than k 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 interprocedural control flow graph G: for each method call c, it the target of c can complete, add an edge from c to its corresponding method return r. This can be done in time $O(n)$.

4.2 Feasible Search

Now that bypass edges have been added to the graph G, a modified depth first search can be used to find feasible paths. The algorithm is as follows:

Algorithm 4.5 (v : vertex, G : graph**).**

```
 1. Let $s \leftarrow \emptyset$.
 2. Call $FeasibleDFS(v, G, s)$.

 3. Procedure $FeasibleDFS(v$ : vertex, $G$ : graph, $s$ : stack):
 4.   If $s = \emptyset$ {
 5.     If $no\_context\_mark(v)$ return.
 6.     Set $no\_context\_mark(v) \leftarrow true$.
 7.   } // End if (4).
 8.   Else {
 9.     If $context\_mark(v)$ return.
10.     Set $context\_mark(v) \leftarrow true$.
11.   } // End else (8).
12.   For each edge $(v, u) \in G$ {
13.     Let $s' \leftarrow s$.
14.     If $label(v, u)$ is a close symbol and $s' \neq \emptyset$ {
15.       Let $o \leftarrow pop(s')$.
16.       If $label(v, u)$ does not match $o$:
17.         Skip to next iteration of 12.
18.     } // End if (14).
19.     Else if $label(v, u)$ is an open symbol:
20.       Push $label(v, u)$ onto $s'$.
21.     Call $FeasibleDFS(u, G, s)$.
22.   } // End for (12).
```

Theorem 4.6. *Algorithm 4.5 does not follow any infeasible paths.*

Proof. Consider an arbitrary infeasible path p. By theorem 4.1, 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 4.5 on this path. An open

[5]Even on the largest example we tried (>45,000 lines of user and library code, 1226 methods), algorithm 4.3 required only five iterations to converge.

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 4.5 checks the top of the stack against the newly encountered label, and since they don't match, it does not proceed along p. $\square$

Since G contains bypass edges and algorithm 4.5 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).

4.3 Feasible Concurrent Accesses

We can now modify algorithm 3.7 to find only concurrent accesses that are feasible.

Algorithm 4.7 (P : program**).**

```
 1. Let $G \leftarrow$ Algorithm 3.3$(P)$.
 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.     Delete $B$ from $G$.
 7.   } // End for (4).
 8. } // End for (2).
 9. Let $bypass \leftarrow$ Algorithm 4.3$(P, G_1, \ldots, G_k)$.
10. For each method call and return pair $c, r$ in $P$ {
11.   If the target $f$ of $c, r$ is in $bypass$:
12.     Add an edge $(c, r)$ to $G$.
13. } // End for (10).
14. For each access $a$ in $P$ {
15.   Use Algorithm 4.5 to do a search on $G$ starting from $a$.
16.   For each access $b$ reached in the search:
17.     Insert $(a, b)$ into $concur$.
18. } // End for (14).
19. Return $concur$.
```

The setup of algorithm 4.7 calls algorithm 4.3, so it takes O(kn) time. The searches each take time O(n), and at most m are done, so the total running time is O$(kn + mn)$.

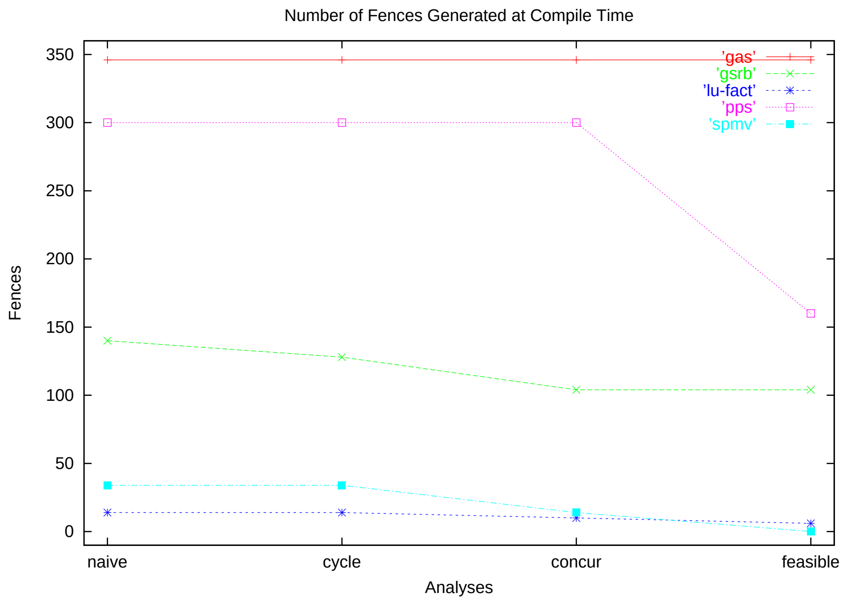
5 Evaluation

As discussed previously, enforcing sequential consistency can result in a large cost to performance. We evaluate the effectiveness of our algorithm by measuring the number of fences generated at compile time and executed at runtime.

5.1 Benchmarks

We use the following benchmarks to evaluate our analysis:

- **`gas`** (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`** (3673 lines) : Poisson equation solver.



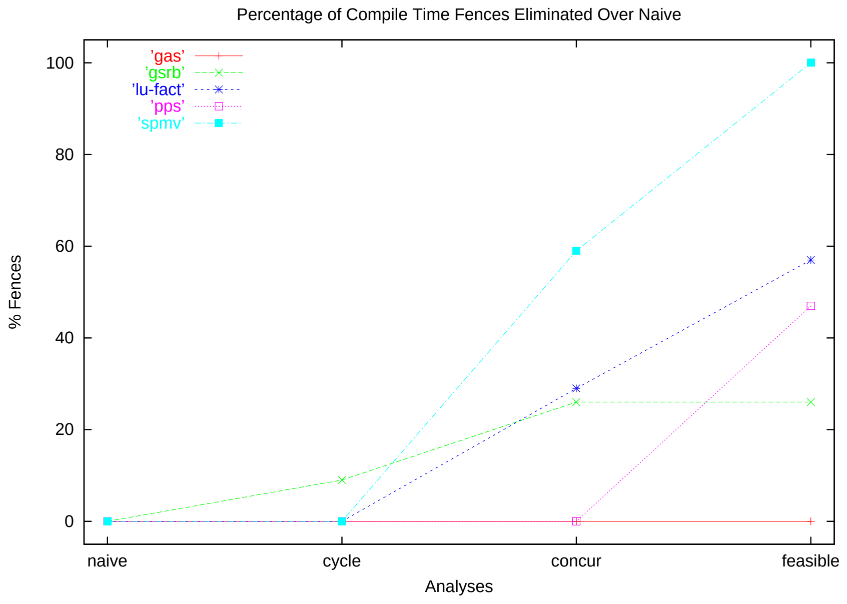


Figure 6: Number of fences statically generated for each program, and percentage of fences reduced over **naive**.

- **`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.

5.2 Fence Counts

In order to enforce sequential consistency, we insert memory fences where required in an input program. These fences can be expensive to execute at runtime, potentially costing an entire roundtrip latency for a remote access. The fences also prevent code motion, so they directly preclude many optimizations from being performed [10]. The static number of fences 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 fences hit at runtime, which more closely estimates the performance impact of the inserted fences.

Figure 6 shows the number of fences generated for each program using different levels of analysis:

- **naive**: fences are inserted around every heap access that has a conflict
- **cycle**: fences are inserted around every heap access that has a conflict edge that occurs in a cycle (§2.1.1)
- **concur**: same as **cycle**, but with only those conflict edges found by our basic concurrency analysis (§3)
- **feasible**: same as **cycle**, but with only those conflict edges found by our feasible paths concurrency analysis (§4)

Figure 7 shows the resulting dynamic counts at runtime.

The results show that our analysis, at its highest precision, is very effective in reducing the numbers of both static and dynamic fences. In three of the benchmarks, nearly all runtime fences are eliminated, and in another, the number of fences hit is reduced by a large fraction. In only one benchmark, **`gas`**, is our analysis ineffective.

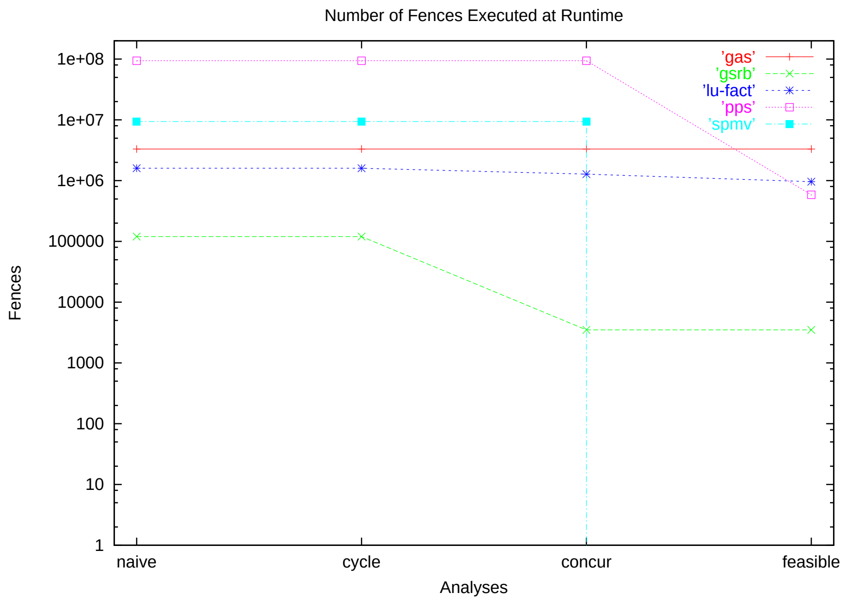
It is interesting to note that eliminating infeasible paths is effective in three of the four benchmarks for which our analysis is useful, and that cycle detection on its own has very little effect on the number of fences in any benchmark. It should also be noted that most of the remaining fences are due to imprecision in our supporting analyses, such as the inability of our alias analysis to distinguish array indices. Even so, we believe our analysis reduces the number of fences enough to nearly match the performance of Titanium's relaxed model.

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. 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 [2, 20, 13, 14].

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* [17]. 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 [9]. In more recent work as part of the Pensieve project, Lee and Padua exploit properties of fences and synchronization to reduce the number of delays in cycle detection [10]. The project also includes a Java compiler that takes a memory model as input [18]. 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 [7]. 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



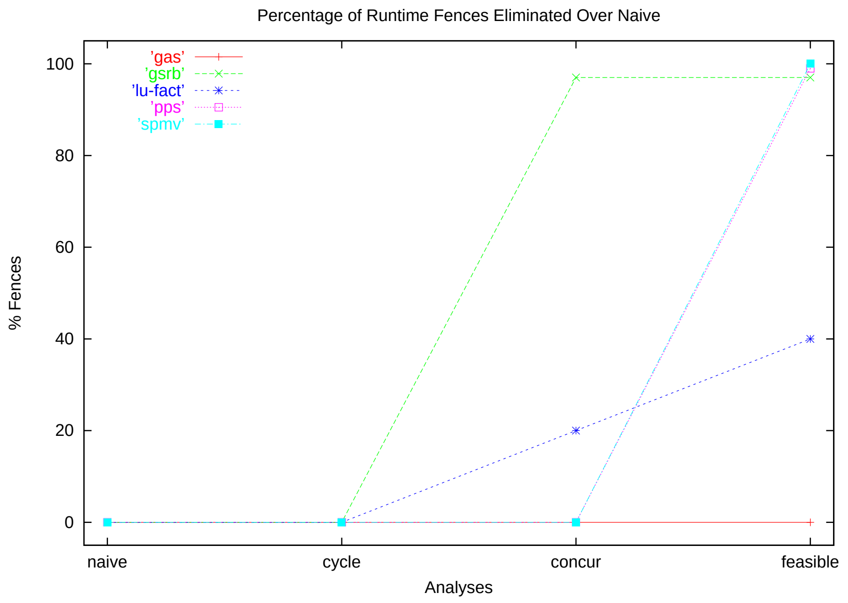


Figure 7: Number of fences hit at runtime for each program, and percentage of fences reduced over **naive**.

language did not have the restriction that barriers must match textually and they did not take advantage of single conditionals. At compile time, they generate two versions of the code, one assuming the barriers line up and the other one not. At runtime, they switch between the two versions depending on how the barriers are executed. Our approach does not suffer the same runtime overhead and code bloat that exists in theirs.

Several other parallel analyses have been developed that do not directly address memory consistency issues. Jeremiassen and Eggers develop a static analysis for barrier synchronization [6] for non-textual barriers. With textual barriers, our analysis is more precise in finding memory accesses that cannot run concurrently. Duesterwald and Soffa use data flow analysis to compute the happened-before and happened-after relation for program statements. The information is used in detecting data races [3]. Masticola and Ryder develop non-concurrency analysis to identify pairs of statements in a parallel program that cannot run concurrently. The results are used for debugging and optimization [12].

7 Conclusion

As shared memory multiprocessors have become more common, the issue of which memory consistency model to use has gained importance. This paper provides evidence that, with the proper set of compiler analyses, the intuitive model of sequential consistency can be provided without sacrificing much performance.

The contribution of this paper is a concurrency analysis that can be used to increase the precision of the existing cycle detection algorithm for the Titanium language. We presented both a basic analysis and a more complex one that only explores those execution paths that can occur in practice. We experimented with several benchmark programs and showed that the analyses were able to eliminate a large fraction, if not most, of the fences required to guarantee sequential consistency in all but one example.

While the number of fences generated and executed in a program provides some measure of the cost of sequential consistency, it remains to be seen to what extent these fences affect a program's running time. In particular, the fences may prevent certain optimizations that result in large performance gains. In the future, we plan to explore the effects of the remaining fences on important communication optimizations to determine if the cost is indeed negligible.

Acknowledgments

I would like to thank Jimmy Su, who helped me a great deal both in developing the concurrency algorithms and in implementing them.

References

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

[2] 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.

[3] 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.

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

[5] 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.

[6] T. Jeremiassen and S. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *Parallel Architectures and Compilation Techniques*, Montreal, Canada, August 1994.

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

[8] W. Kuchera and C. Wallace. The UPC memory model: Problems and prospects. In *18th International Parallel and Distributed Processing Symposium, 2004*, April 2004.

[9] 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.

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

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

[12] S. Masticola and B. Ryder. Non-concurrency analysis. In *Principles and practice of parallel programming*, San Diego, California, May 1993.

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

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

[15] W. Pugh. Fixing the Java memory model. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 89–98, New York, NY, USA, 1999. ACM Press.

[16] 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.

[17] 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.

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

[19] K. Yelick, D. Bonachea, and C. Wallace. A proposal for a UPC memory consistency model, v1.1. Technical Report LBNL-54983, Lawrence Berkeley National Lab, 2004.

[20] 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.

A Concurrency and Conflicts

Suppose two memory accesses a and b conflict. We show that if a and b can never run concurrently, it is possible to remove the resulting conflict edge since it can never take part in a cycle that violates sequential consistency.

Theorem A.1. *Let* a *and* b *be two memory accesses in a program, and* C *a cycle containing the conflict edge* (a,b)*. If* a *and* b *cannot run concurrently, then reordering* a *with another access*[6] *does not violate sequential consistency with respect to the accesses in* C *in any execution of the program.*

Proof. We prove this for a cycle consisting of four accesses in two threads where a is the first access in thread 1 and b is the second access in thread 2, as in figure 1 (the proof can be generalized to arbitrary cycles). Let x and y be the other two conflicting accesses in C, in thread 1 and 2 respectively. Consider an arbitrary execution in which the accesses in C occur. Since a and b cannot run concurrently, either a must complete before b or b must complete before a.

Case 1: a *occurs before* b. Sequential consistency can only be violated if y sees the effect of x, but b does not see the effect of a. In all other cases, execution corresponds to a valid sequentially consistent ordering, as shown in the table in figure 1. But since a occurs before b, b always sees the effect of a, so sequential consistency is preserved regardless of the order of a and x.

Case 2: b *occurs before* a. In order to enforce that b occur before a, there must be a synchronization point between b and a in the execution stream of each thread. Since accesses aren't moved across such points, y must occur before it and x must occur after it. This means that y must complete before x and therefore does not see its effect. Since y does not see the effect of x and b does not see the effect of a, the execution is sequentially consistent independent of the order of a and x. □

[6] We assume that accesses are never moved across synchronization points.