Making Sequential Consistency Practical in Titanium

Amir Kamil Jimmy Su Katherine Yelick*
Computer Science Division, University of California, Berkeley
{*kamil,jimmysu,yelick*}*@cs.berkeley.edu*

July 25, 2005

Abstract

The memory consistency model in shared memory parallel programming controls the order in which memory operations performed by one thread may be observed by another. The most natural model for programmers is to have memory accesses appear to take effect in the order specified in the original program. Language designers have been reluctant to use this strong semantics, called *sequential consistency*, due to concerns over the performance of memory fence instructions and related mechanisms that guarantee order. In this paper, we provide evidence for the practicality of sequential consistency by showing that advanced compiler analysis techniques are sufficient to eliminate the need for most memory fences and enable high-level optimizations. Our analyses eliminated over 97% of the memory fences that were needed by a naïve implementation, accounting for 87 to 100% of the dynamically encountered fences in all but one benchmark. The impact of the memory model and analysis on runtime performance depends on the quality of the optimizations: more aggressive optimizations are likely to be invalidated by a strong memory consistency semantics. We consider two specific optimizations–pipelining of bulk memory copies and communication aggregation and scheduling for irregular accesses–and show that our most aggressive analysis is able to obtain the same performance as the relaxed model when applied to two linear algebra kernels. While additional work on parallel optimizations and analyses is needed, we believe these results provide important evidence on the viability of using a simple memory consistency model without sacrificing performance.

*Also at Lawrence Berkeley National Laboratory.

SC|05 November 12-18, 2005, Seattle, Washington, USA

1 Introduction

A key question in the design of shared memory parallel languages is the memory consistency model: In what order are memory operations performed by one thread observed on another? For simplicity, one would like the operations to appear in the order specified in the original program, i.e., any reorderings performed by the compiler or hardware should not be observable. In practice, parallel language designers have been reluctant to use such a strong *sequentially consistent* semantics because memory operations are often overlapped and reordered for performance reasons, and requiring that all threads see the same memory order requires the insertion of expensive memory fence instructions. The problem is even worse for global address space languages, which use a shared memory abstraction on top of large-scale distributed memory hardware. In this setting, a read or write to a shared variable may require a network message, and a memory fence corresponds to waiting for a full roundtrip latency across the network. The memory model question is controversial across a wide range of languages, from Java™ threads and OpenMP to UPC and Titanium [25, 6, 35]. The relaxed semantics used in these languages are often ill-defined [17], and even when they have been made precise [26, 34], they are not well understood by application programmers. Programming techniques such as spin-locks and presence bits rely on the details of the memory consistency model, and while language designers may want to discourage such programming practice, leaving the behavior of these programs to a complicated or ill-defined semantics is troubling.

In this paper, we consider the question of whether a relaxed semantics is necessary, and whether a compiler can reasonably analyze parallel programs to ensure that no reorderings are observable while still allowing the compiler to perform important optimizations. Our study is done using the Titanium language [35], a single program, multiple data global address space variation of Java that runs on most parallel and distributed memory machines. We show that concurrency information can be used to reduce the number of fences required

to enforce sequential consistency and present a concurrency analysis that takes advantage of the unique semantics of Titanium. We augment this analysis with other analyses such as sharing inference [21], alias analysis, and cycle detection [29] to further reduce the number of fences. We perform a series of experiments with a Titanium compiler that produces sequentially consistent code, starting with a naïve implementation that places memory fences around all heap accesses and then using successively stronger analyses to prove that some of the fences are unnecessary and can be removed. As an upper bound measure of performance, we also include a version of the compiler that provides the relaxed memory semantics. Our experiments look at both static and dynamic memory fence counts and cover a range of programs from small benchmarks up to a full hyperbolic adaptive mesh refinement solver for gas dynamics. In addition, we look at the runtime performance impact on two linear algebra kernels, dense and sparse matrix vector multiplication, and show that the most aggressive analysis is needed to enable key optimizations, but with that analysis, the performance is identical to the relaxed model.

2 Background

2.1 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 [5], 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, so 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 [10].

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

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-valued. Single-valued expressions are used to ensure that barriers line up: a conditional may only contain 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.

Titanium's memory consistency model is defined in the language specification [13]. Here are some informal properties of the Titanium model.

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

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

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

2.2 Sequential Consistency

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

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 [18].

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 have been proposed to minimize the number of fences, or *delay set*, required to enforce sequential consistency.

Computing the minimal delay set for an arbitrary parallel program is an intractable NP-hard problem [29, 16]. Krishnamurthy and Yelick proposed a polynomial time algorithm based on *cycle detection* for analyzing SPMD programs [16] 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 [16] 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*.

3 Concurrency Analysis

Concurrency information can be used to reduce the set of conflict edges for a program. We show that a conflict edge in which the endpoints cannot run concurrently can be ignored, and we present an algorithm for computing concurrent accesses in Titanium.

3.1 Conflicts and Concurrency

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.

Theorem 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*[1] *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.

3.2 Finding Concurrent Accesses

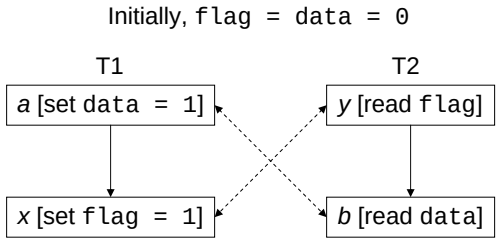
The goal of our concurrency analysis is to statically identify the pairs of memory accesses that can run concurrently. We rely on barriers and single-valued expressions in this analysis. Our algorithm makes use of the following definitions.

Definition 1 (Global Context): An expression or statement is in a *global context* if all threads are guaranteed to execute it at the same time, with respect to barriers.

Definition 2 (Single Conditional): A *single conditional* is a conditional or loop guarded by a single-valued expression and contained in a global context.

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.

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



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.

Definition 3 (Code Segment): Consider the set of barriers, entries into a branch of a single conditional, and exits from a branch of a single conditional. A *code segment* is a sequence of statements bounded by, and not containing, members of this set[2].

Since a barrier, single conditional entry, or single conditional exit occurs before each code segment and every thread executes the same sequence of barriers and branches of single conditionals, every thread executes the same sequence of code segments.

The set of code segments is easy to compute by performing a depth first search from each barrier, single conditional entry, and single conditional exit. The size of the resulting set is at most quadratic in the number of barriers and single conditionals. Figure 2 shows the code segments for an example program.

Using the code segments of a program, we construct a graph G in which there is a node per segment and the edges represent the control flow between segments, as shown in figure 2. Since barriers, single conditional entries, and single conditional exits separate code segments, each edge must correspond to one such construct.

Theorem 2: *Two code segments* A *and* B *can run concurrently only if one is reachable from the other in* G *along a path that does not include a barrier edge.*

Proof: Suppose A and B can run concurrently but are not reachable from one another in G. A and B cannot be the same code segment, since a segment is always self-reachable. Since each thread executes the same sequence of code segments, in order for A and B to run concurrently, there must be a program flow from A to B or from B to A that does not hit a barrier. Such a flow must consist solely of single conditional entries, single conditional exits, and other code segments. But if such a flow exists, there is a path in G that passes through only the edges corresponding to the entries and exits and the nodes corresponding to the code segments. Thus, A and B are reachable from one another in G through this path, which is a contradiction.

The set of concurrent code segments can easily be computed by removing barrier edges from G and performing a depth first search on the resulting graph G' from each segment. This takes time at most quadratic in the number of code segment.

Using the set of concurrent code segments, it is trivial to determine the set of concurrent memory accesses: two memory accesses may run concurrently only if they are in concurrent code segments.

3.3 Single-Threaded Code

Though all Titanium threads run the same program, it is still possible to write code that only runs on a single thread. A block of code guarded by a conditional of the form `Ti.thisProc() ==` *expr*, where *expr* is single-valued, only runs on the thread whose ID is *expr*. At first glance, it appears that accesses in such a block cannot occur concurrently. However, consider the following example:

```
int single i;
for (i = 0; i < 10; i++) {
  if (Ti.thisProc() == i) {
    // (1) some accesses
  }
}
```

The code is legal according to Titanium semantics, since `i` has the same value on each thread in each iteration of the loop. However, the accesses in (1) can run concurrently, since different threads may be in different iterations of the loop.

In order to determine whether or not accesses in such "single-threaded" code can run concurrently, we also split code segments at the entry and exit points of such code so that each piece of "single-threaded" code has its own code segment. We then use the following to determine which code segments it can run concurrently with:

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

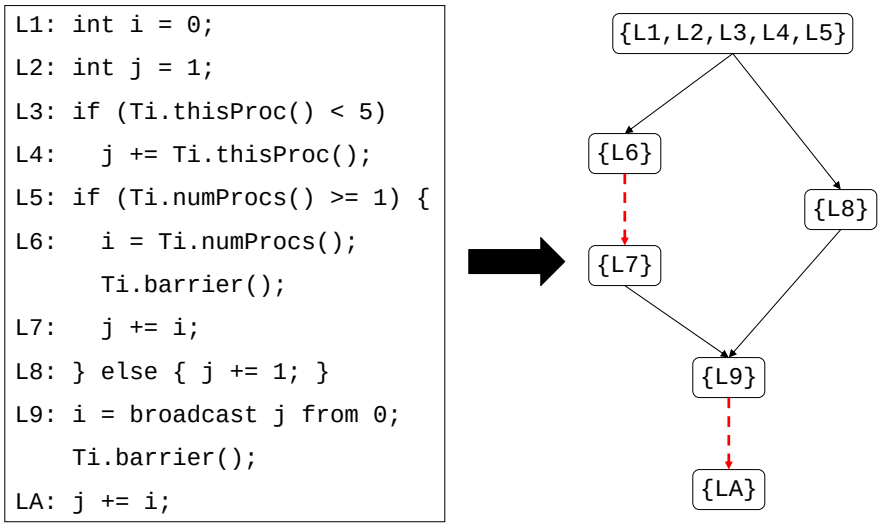


Figure 2: The graph of code segments for an example program. Each node in the graph corresponds to a segment, and the dashed edges correspond to barriers.

Theorem 3: *A "single-threaded" code segment* A *can run concurrently with a "single-threaded" segment* B *only if there is a non-trivial path (a path with at least one edge) from* A *to* B *or from* B *to* A *that does not pass through a barrier.*

Proof: The proof is omitted for brevity.

In the example above, (1) is reachable from itself by a non-trivial path through the multi-threaded increment `i++` and test `i < 10`.

3.4 Broadcasts

The expression

`broadcast` *expr*$_1$ `from` *expr*$_2$

results in a transmission of the value of *expr*$_1$ from thread *expr*$_2$ to all other threads. The other threads must wait at the `broadcast` until the source thread has computed the value to be transmitted, so the expression acts as a sort of weak barrier. It does not, however, prevent different threads from simultaneously executing code preceding and following the `broadcast`, so our analysis cannot treat it as a real barrier. In the following code, however, (1) and (2) cannot occur concurrently, since all other threads must wait for thread 0 to reach the `broadcast` before proceeding:

```
if (Ti.thisProc() == 0)
  a.f = b; // (1)
c = broadcast a from 0;
d = c.f; // (2)
```

In order to handle this case, we split code segments at broadcasts as well. We then use the following:

Theorem 4: *A "single-threaded" code segment* A *that executes only on thread* t *cannot run concurrently with a code segment* B *if* A *is not reachable from* B *without hitting a barrier, and* B *is only reachable from* A *along paths that contain a barrier or a broadcast from* t.

Proof: The proof is omitted for brevity.

4 Other Analyses

The concurrency analysis described in §3 only computes the set of accesses that can occur concurrently, an over-approximation of the set of actual conflicts. The analysis can be augmented by other analyses in order to increase the precision of its results. In addition, only a subset of the actual conflicts violate sequential consistency, which can be determined before generating fences.

4.1 Sharing Inference

In parallel programs, objects can be private to a thread or shared among multiple threads. A private object can never be involved in a conflict, since all operations on it must occur in the thread that owns it. Sequential consistency can be enforced by surrounding all accesses on shared objects with fences. In addition, our concurrency analysis can take advantage of sharing information by removing all conflict pairs in which at least one of the members is private, since they cannot represent a real conflict.

We use the approach (and implementation) of Liblit, Aiken, and Yelick [21] to infer which pointers must reference private data in a Titanium program. It introduces a type system over the qualifiers `private`, `shared`, and `mixed`. Constraints are generated for each variable from the expressions in a program, and the maximally-private solution to this set of constraints is found.

4.2 Alias Analysis

Two memory accesses can only conflict if they reference the same memory location. Aliasing information allows us to remove conflicts in which the accesses cannot actually alias the same location. We implemented different levels of alias analysis to improve our results:

- **Type-based alias analysis**: The accesses in a conflict pair must be on locations of comparable types.
- **Sequential alias analysis**: The accesses in a conflict pair must be on locations that may be aliased, ignoring threads.
- **Thread-aware alias analysis**: The accesses in a conflict pair must be on locations that may be aliased across threads.

In all cases above, the accesses in a conflict pair must be on the same field for non-array types.

4.2.1 Type-Based Alias Analysis

Our type-based alias analysis assumes that any two memory locations of comparable types can be aliased. Two types are *comparable* if one can be cast to the other. Thus any variable of type `Object` is assumed to alias any variable of type `Vector`, but a variable of type `String` cannot alias a variable of type `Vector`.

4.2.2 Sequential Alias Analysis

Our sequential alias analysis is a Java version of Andersen's points-to analysis for C [1] and is thus both flow-insensitive and context-insensitive[3]. Each allocation site in a program corresponds to an *abstract location*, and each variable has a *points-to set*, the abstract locations that it may reference. In addition, abstract locations have points-to sets for each of their fields. The points-to sets are computed by repeatedly updating the points-to set of the target of each explicit and implicit assignment in a program until a fixed point is reached. Two locations may be aliased if the intersection of their points-to sets is non-empty.

[3]We experimented with a partially context-sensitive points-to analysis but found that it did not increase the precision of our concurrency analysis.

The analysis is *sequential* in that it does not distinguish allocations on different threads. Inter-thread transmission of an abstract location thus produces the original location.

We also use alias analysis to limit the potential targets of a dynamic dispatch. Without alias analysis, all overriders of the statically determined target must also be considered as possible targets. Using alias analysis, we can restrict the possible targets of a dynamic dispatch to the methods that can actually be called, according to the types of the abstract locations in the points-to set of the target location.

4.2.3 Thread-Aware Alias Analysis

In addition to the sequential alias analysis, we implemented a *thread-aware* version of the analysis that takes advantage of the SPMD nature of Titanium. An abstract location can be *local* or *remote* and is local by default. Inter-thread transmission of an abstract location a produces the remote analog a_r of the original location in addition to the original itself. The points-to set of the remote location contains the remote version of the original points-to set, as illustrated in figure 3. Two locations x and y may be *aliased across threads* if there is a local abstract location in one points-to set and the remote analog in the other ($(\exists l \in pointsTo(x) \,.\, l_r \in pointsTo(y)) \vee (\exists l \in pointsTo(y) \,.\, l_r \in pointsTo(x))$). In the left side of figure 3, $b.y$ and $c.y$ do not alias each other across threads since d_r is in neither $pointsTo(b.y)$ nor in $pointsTo(c.y)$, but $e.z$ and $f.z$ do since $g \in pointsTo(e.z)$ and $g_r \in pointsTo(f.z)$.

4.3 Cycle Detection

For a program edge between two memory accesses a and b, where a and b both have conflict edges, it is not always necessary to insert a fence between them to enforce sequential consistency. A fence is only needed if a critical cycle can be formed. Cycle detection [29] determines whether or not this is the case.

For the most part, we approximate cycle detection by placing fences around any memory access that has a conflict edge. This guarantees that a fence occurs in any critical cycle containing the access. We also implemented the full cycle detection algorithm proposed by Krishnamurthy and Yelick [16] and only place fences around an access if it is an endpoint of a detected cycle.

The full cycle detection algorithm we use is still not as precise as possible, since it individually places fences in each critical cycle instead of minimizing the total number of fences required for all cycles. Lee and Padua have shown that the minimization problem is **NP**-complete in general [20]. However, we believe that automatic barrier placement algorithms [7] can be adapted to place fences better than the naïve method we use.

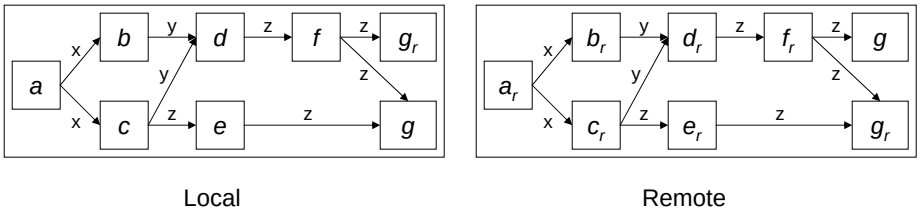


Figure 3: The points-to relations for the local and remote versions of abstract location a. Nodes represent abstract locations, and an edge $a \rightarrow_x b$ denotes that $b \in pointsTo(a.x)$.

5 Implementation

Our actual implementation[4] of concurrency analysis differs somewhat from the algorithm presented in §3. Instead of dividing a program into code segments, we perform the analysis over the control flow graph of the program. Each statement or expression in the program has a node, and edges between nodes represent control flow between the corresponding program constructs. In the Titanium compiler, control flow graphs are generated for each method in the input program. We modify these graphs as follows:

- **Method calls**: Method calls are split into two nodes, representing transfer of control from caller to callee and from callee to caller, respectively. The former is linked by an edge to the entry of the callee's graph, and the latter by an edge from the callee's exit. If multiple callees exist, as for dynamic dispatch, the nodes are linked in this manner to each callee.

- **Cross edges**: The branches of a non-single conditional are not reachable from each other in a control flow graph. They may run concurrently, however, so they require the addition of *cross edges* to represent this fact. We add a cross edge from the end of one branch to the beginning of the other. Method calls to multiple potential targets also require and are given cross edges.

On this resulting control flow graph, we perform a depth first search from each heap access, stopping at barriers as in §3. A conflict pair is generated when another access is reached in which the target can alias the root access's target, as described in §4. When both accesses are reads, however, they don't conflict, so we don't generate a conflict pair.

5.1 Feasible Paths

The search described above can result in paths being followed that cannot actually occur in practice. For example, the path in figure 4 is impossible since control flow from a callee returns to a different location than where the call occurred. In order to prevent the search from following such paths, we label each set of call and return edges with different matching parentheses[5], as shown in figure 4. We then perform context-free language reachability on the resulting graph using a grammar of balanced parentheses as described by Reps [27], though properly modified to account for the fact that the search can start at an arbitrary program point.

5.2 Performance Improvements

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

- **Private data**: As discussed in §4, accesses on private data cannot be in a conflict pair. Our implementation ignores such accesses.

- **Graph compaction**: The control flow graph contains many nodes that are irrelevant to our analysis. Since many searches are conducted over the graph, it is beneficial to remove these nodes before the analysis. Our experiments show that on average, about 95% of nodes can be removed.

5.3 Complexity

The complexity of our implementation is $O(P+H^2+\alpha HM)$, where P is the size of the input program, H is the number of (non-private) heap accesses, M is the number of method calls, and α is the average number of targets, as determined by alias analysis, for a dynamic method dispatch ($\alpha \approx 1$ in most Titanium programs). We are exploring an alternative algorithm with sub-quadratic average case running time. Note that any conflict detection algorithm must have $\Omega(H^2)$ running time in the worst case, since every heap access can potentially conflict with every other access.

[4] The algorithm used in our implementation will be described in detail in a forthcoming paper [15].

[5] Cross edges between method calls are also labeled as necessary.

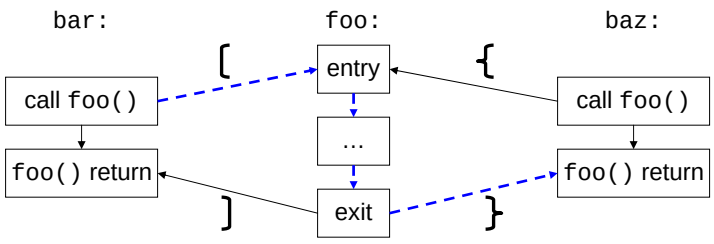


Figure 4: Inter-procedural control flow graph for two calls to the same function. The dashed path is infeasible, corresponding to the unbalanced string "[}".

6 Evaluation

As discussed previously, enforcing sequential consistency can result in a large cost to performance. We evaluate the effectiveness of our algorithm against existing analyses [21] and relaxed consistency by measuring the number of fences generated and executed and the performance of parallel programs.

6.1 Benchmarks

We use the following benchmarks to evaluate our analysis:

- **`3d-fft`** (614 lines): Fourier transform.
- **`gas`** [4] (8841 lines): Hyperbolic solver for a gas dynamics problem in computational fluid dynamics. It uses a hierarchical adaptive mesh refinement approach, which results in a hierarchy of block structured meshes. The adaptive mesh is distributed across threads with each block residing on a single thread, and each thread must find "neighboring" blocks within the hierarchy to perform finite difference operations. This code was written by Peter McQuorquodale and Phillip Collela.
- **`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.
- **`gsrb`*** (1099 lines): A slightly modified version of `gsrb` in which threads cache pointers to their local pieces of the global mesh.
- **`lu-fact`** (420 lines): Dense linear algebra.
- **`pi`** (56 lines): Monte Carlo integration.
- **`pps`** [3] (3673 lines): Parallel Poisson equation solver using the domain decomposition method in an unbounded domain.
- **`sample-sort`** (321 lines): Parallel sorting.
- **`demv`** (122 lines): Dense matrix-vector multiply. The matrix is partitioned by rows. The source and destination vectors are distributed arrays. Each thread has its own section of the source and destination array.
- **`spmv`** (1493 lines): Sparse matrix-vector multiply. The matrix is partitioned by rows, with each thread getting a contiguous block of complete rows. The source and result vectors are dense. Each thread holds the corresponding piece of the source and result vector.

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.

6.2 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 [20]. 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.

Figures 5 and 7 show the number of fences generated for each program using different levels of analysis:

- **naïve**: fences are inserted around every heap access
- **sharing**: fences are inserted around every access on shared data, as computed by sharing inference (§4.1)
- **concur**: fences are inserted around every shared access that is a member of a conflict pair, as computed by concurrency analysis (§3) and type-based alias analysis (§4.2.1)

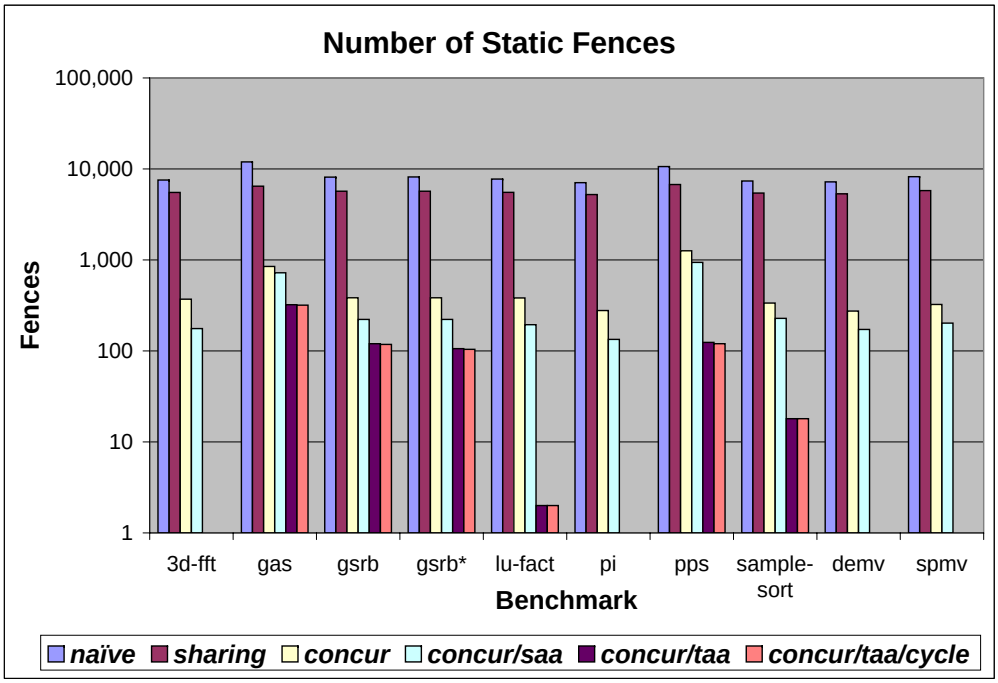


Figure 5: Number of fences statically generated for each level of analysis.

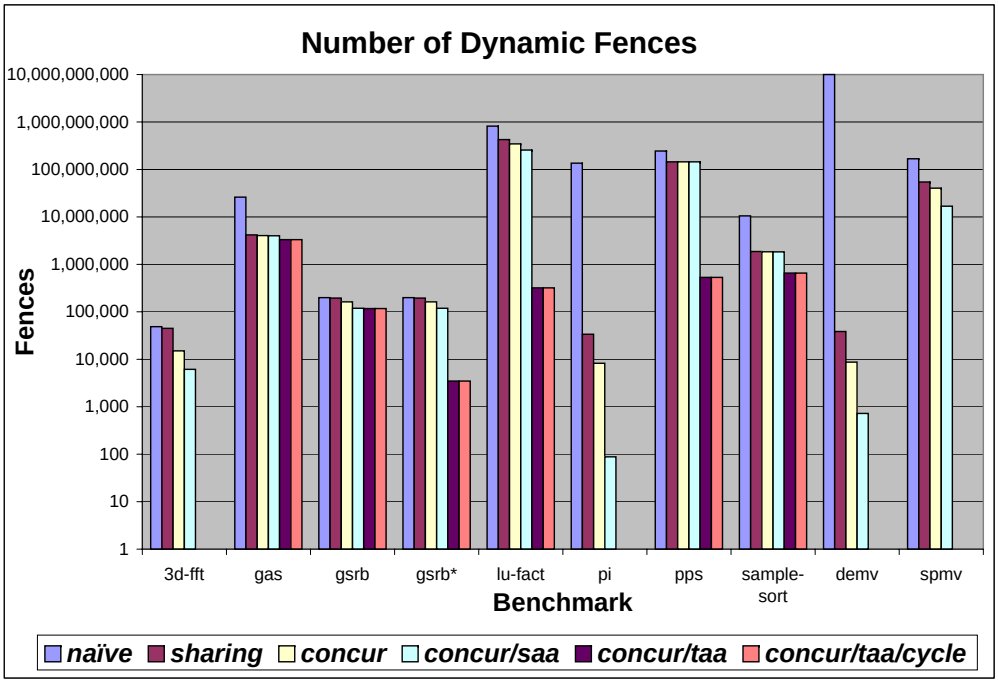


Figure 6: Number of fences hit at runtime for each level of analysis.

Number of Fences for **naïve**		
Benchmark	Static	Dynamic
`3d-fft`	7750	49K
`gas`	11938	26M
`gsrb`	8124	199K
`gsrb`*	8174	199K
`lu-fact`	7734	819M
`pi`	7064	136M
`pps`	10612	245M
`sample-sort`	7378	10.5M
`demv`	7212	10B
`spmv`	8226	168M

Table 1: Number of fences statically generated and hit at runtime using the **naïve** level of analysis.

- **concur/saa**: fences are inserted around every conflict pair, using concurrency analysis augmented by sequential alias analysis (§4.2.2)
- **concur/taa**: fences are inserted around every conflict pair, using concurrency analysis augmented by thread-aware alias analysis (§4.2.3)
- **concur/taa/cycle**: full cycle detection is used to further reduce the number of fences on top of the previous level. Fences are only inserted around memory accesses that can be part of a critical cycle (§4.3).

Figures 6 and 8 show the resulting dynamic counts at runtime. For reference, the static and dynamic counts for the **naïve** level of analysis are shown in table 1.

The results show that our analysis, at its highest precision, is very effective in reducing the numbers of both static and dynamic fences. In most of the benchmarks, our analysis eliminates at least 97% of the static and 87% of the dynamic fences compared to the **naïve** version. The only benchmark that the analysis has trouble with is `gsrb`. The code for this program stores both local and global pointers in the same array, and our analysis currently cannot distinguish between array indices (§7.1). With very little effort, we were able to modify `gsrb` to keep a separate store of local pointers (`gsrb`*), resulting in the removal of most of the remaining fences.

It is interesting to note the effects of the auxiliary analyses in §4. As shown in figures 5 and 6, alias analysis is a crucial component of our concurrency analysis, reducing the number of fences by a large margin. On the other hand, cycle detection has almost no effect, reducing the number of fences only by a marginal amount.

6.3 Optimizations

In measuring the effects of sequential consistency on the performance of parallel programs, we focus on optimizations for communication between threads. In a distributed environment, the communication time is a major component of the running time of a parallel program, so such optimizations potentially provide much larger payoffs than sequential optimizations. We evaluate our analysis using two optimizations: converting blocking array copies to non-blocking copies, and the inspector executor transformation on irregular array accesses.

6.3.1 Non-Blocking Array Copies

An array copy is a bulk communication operation in Titanium between two arrays. The contents of the source array are copied to the destination array where the domains of the two arrays intersect. This operation is *blocking*, which means that the thread executing the array copy waits until the operation completes before moving on to the next instruction. If either the source array or the destination array is remote, then communication over the network is required.

A compiler optimization that automatically converts blocking array copies into non-blocking operations has been developed in the Titanium compiler. The goal of this optimization is to push the synchronization that ensures the completion of the array copy as far down the instruction stream as possible. The time between the issue of the non-blocking operation and the synchronization for that operation can be used for other communication and computation.

The use of sequential consistency as a memory model can reduce the effectiveness of this optimization. An array copy includes multiple reads and writes, so it can appear in critical cycles. When fences are needed around an array copy, the optimization cannot be performed.

6.3.2 Inspector Executor

The Titanium compiler has support for the *inspector executor* framework [30] to optimize irregular remote accesses of the form `a[b[i]]` that appear in a loop. The array access pattern is computed in an initial *inspector* loop. All the required elements are prefetched into a local buffer. The *executor* loop then uses the prefetched elements for the actual computation.

The inspector executor transformation must respect local dependencies. Using a sequentially consistent memory model places further constraints on this optimization: the compiler cannot apply it if their is another memory access in the original loop requiring a fence between it and the indirect array access. The prefetching performed by the optimization aggregates the indirect array reads for all iterations of the loop, changing the order of memory accesses in this case.

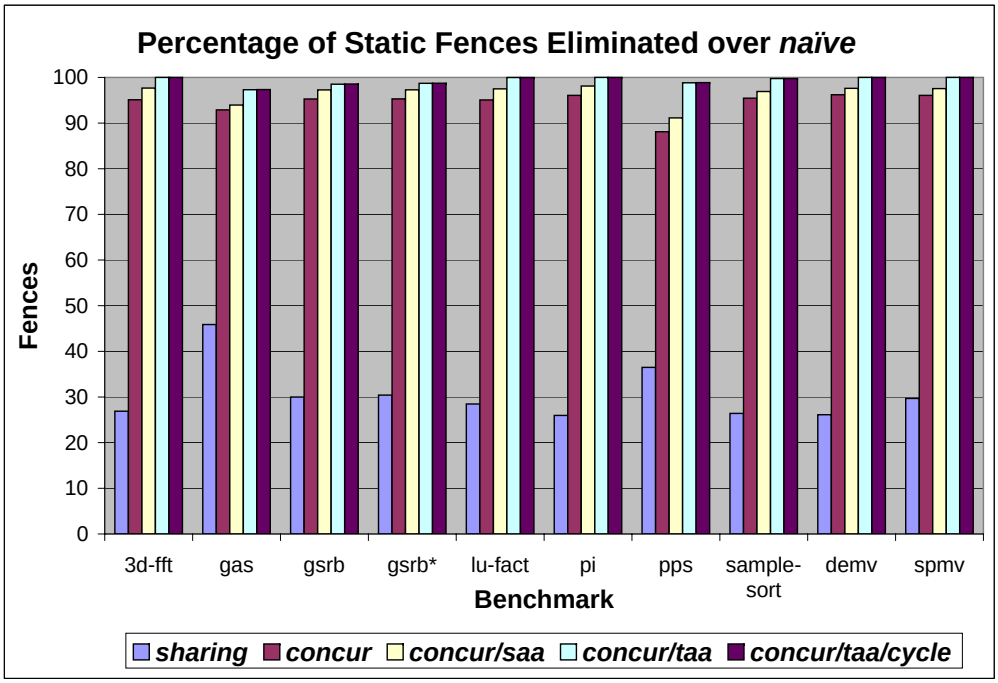


Figure 7: Reduction in statically generated fences relative to the **naïve** level of analysis.

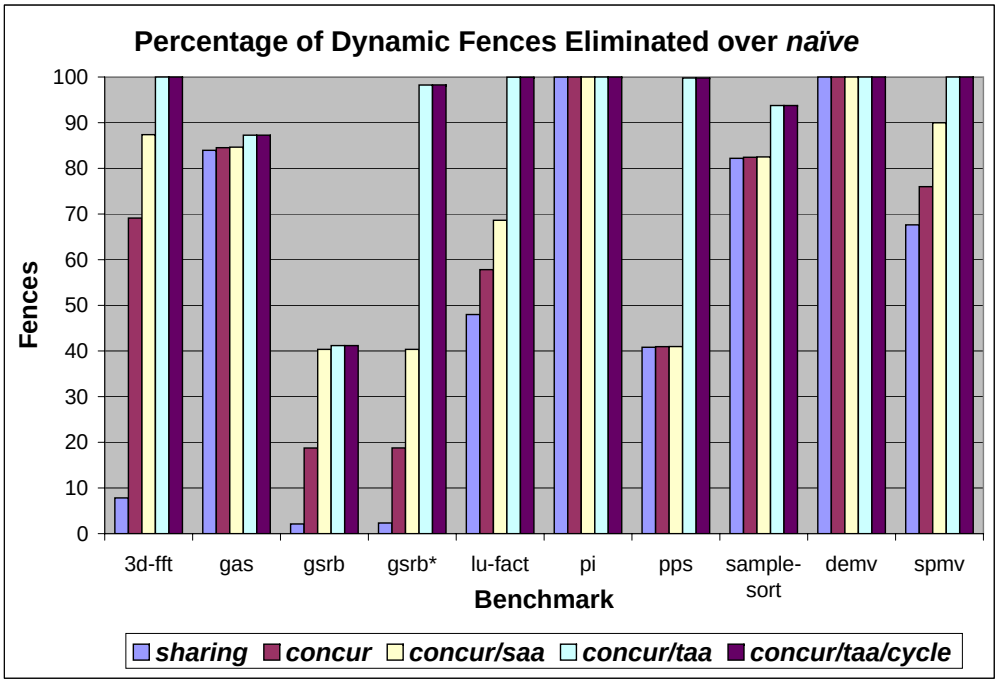


Figure 8: Reduction in fences hit at runtime relative to the **naïve** level of analysis.

6.4 Performance

We measured the performance of two benchmarks in which communication optimizations prove effective[6], comparing our analysis against both the relaxed model and prior work to enforce sequential consistency. We performed our experiments on the Itanium/Myrinet cluster at Rice University, using one processor on each node.

6.4.1 Dense Matrix Vector Multiply

In the `demv` benchmark, each thread uses array copies in order to obtain the contents of the entire distributed source vector. While retrieving each section of the source vector, a thread can do local computation with parts of the source vector that it has already retrieved. Figure 9 compares the performance of `demv` using four versions of the compiler. The **relaxed** version uses the Titanium memory consistency model, which is weaker than sequential consistency. The other three versions provide sequential consistency and differ only in the analysis we use for enforcing it, as described in §6.2. The **naïve** version puts fences around the array copies, which prevents it from being transformed into a non-blocking operation. The **sharing** version infers that the source vector is shared, so it also places fences around the array copies. Our analysis, on the other hand, infers that the array copy destinations are thread-local and thus do not require fences. The compiler can therefore convert the copies into non-blocking operations, resulting in a speedup of 1.45 over the **naïve** and **sharing** versions. Since our analysis determines that no fences at all are required, it performs just as well as under the relaxed model.

6.4.2 Sparse Matrix Vector Multiply

In `spmv`, each thread uses indirect array accesses to read from the distributed source vector. The inspector executor transformation can combine these individual reads into a bulk fetch. Figure 9 illustrates the performance of `spmv` using four versions of the compiler. In the **naïve** version, the compiler puts fences around the reads on the source vector, preventing the inspector executor optimization. The **sharing** version infers that the source vector is shared, so it also puts fences around the reads. Our analysis, on the other hand, finds that the reads have no conflict edges, since there is a barrier between the code that updates the source vector and the read-only multiply phase in which the accesses occur. Since our analysis determines that no fences are required, it generates code identical to that of the relaxed model, resulting in a speedup of more than 80 over the **naïve** and **sharing** versions. Note that the code under the relaxed model and using our analysis is itself an average of 21% faster than the same operation written using Aztec, a popular sparse solver library written in C and MPI [30].

[6] The rest of the benchmarks do not benefit from communication optimizations, and prior work has shown that their performance is similar under the relaxed and sequential consistency models even without our analyses [21].

7 Discussion

In this section, we discuss some of the limitations of and extensions to our analysis, its relation to race detection, and some future work.

7.1 False Positives

A handful of program constructs cannot currently be handled precisely by our analyses and therefore may result in false conflicts:

- **Array indices**: Our alias analysis does not maintain separate points-to sets for array indices and cannot distinguish accesses to different indices, so concurrent accesses to different indices can result in false conflicts.
- **Granularity of thread-aware alias analysis**: Our thread-aware alias analysis only distinguishes between local and remote abstract locations: it does take into account actual thread ownership of locations. In the code below, the locations referenced by `a` are actually those referenced by `b` on thread 0, so the writes to `b` on other threads do not conflict with the reads on `a`.

  ```
  a = broadcast b from 0;
  if (Ti.thisProc() != 0)
    b.f = a.f;
  ```

  Our analysis can only determine that `a` can refer to the locations referenced by `b` on any thread, so it assumes that the writes to `b` and reads on `a` conflict.

With some difficulty, our analyses can be extended to handle the cases above.

7.2 Race Detection

In §3.1, we showed that in order to provide sequential consistency, it is sufficient to only consider conflicting accesses that can run concurrently, which is the definition of a *race condition* [23]. Sequential consistency and race conditions are intimately related: under a relaxed memory model such as release consistency [11], a program with no races cannot violate sequential consistency since the synchronization mechanisms enforce order, and in a program with races, sequential consistency can only be violated with respect to those memory accesses that can be in a race condition. Providing sequential

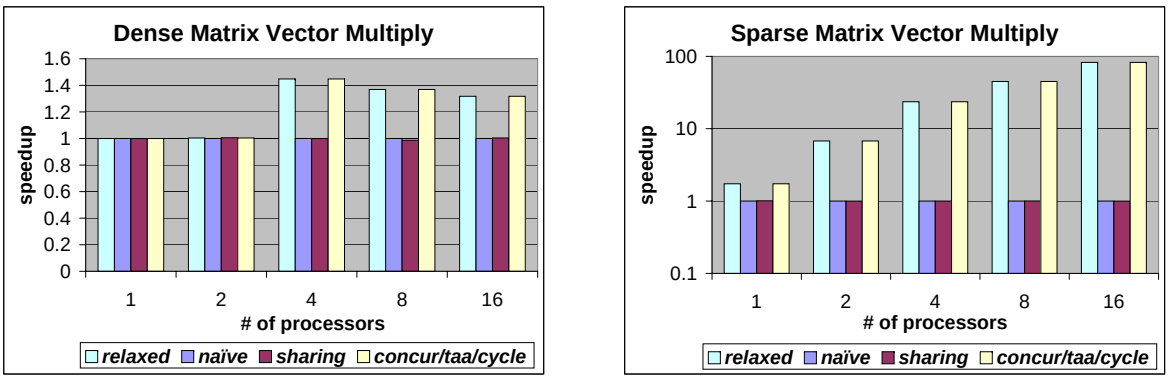


Figure 9: Performance comparisons for the dense and sparse matrix vector multiply benchmarks. Speedups are computed against the **naïve** version.

consistency thus reduces to static race detection. However, an imprecise analysis can only affect performance in the former, while in the latter, it can render the detection useless.

Our analysis can be directly used to statically discover potential races. In `sample-sort`, for example, it finds three concurrent conflicts, one of which corresponds to a real race condition. In order to be useful as a race detector, however, our concurrency analysis needs to be modified to produce a possible execution path on which each race can occur. Currently, it can produce the partial path between the first and second access in a conflict but does not find the path from program entry to the first access.

7.3 Future Work

In the future, we plan to explore the cost of sequential consistency on larger and more varied applications such as adaptive mesh refinement [33] and an implementation of the NAS parallel benchmarks [2] in Titanium [8]. We suspect that in order to achieve comparable performance to a relaxed model in these applications, a precise array index analysis is required.

8 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 [6, 35, 24, 25].

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* [29]. 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. [19]. As part of the Pensieve project, Lee and Padua exploited properties of fences and synchronization to reduce the number of delays in cycle detection [20]. The project also includes a Java compiler that takes a memory model as input [32]. More recently, Sura et al. have shown that cooperating escape, thread structure, and delay set analyses can be used to provide sequential consistency cheaply in Java [31]. 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 [16]. That was done in a simplified subset of the Split-C language and introduced a polynomial time algorithm for cycle detection in SPMD programs. They also used synchronization analysis to reduce the number of fences, but their source language did not have the restriction that barriers must match textually and they did not take advantage of single conditionals. At compile time, they generated two versions of the code, one assuming the barriers line up and the other one not. At runtime, they switched between the two versions depending on how the barriers were executed. Our approach does not suffer the same runtime overhead and code bloat that ex-

ists in theirs. In addition, their compiler used only a simple type-based alias analysis.

There has also been work done in the area of reducing the number of fences required to enforce sequential consistency. Liblit, Aiken, and Yelick developed a type system to identify shared data accesses in Titanium programs [21], and for sequential consistency, they only insert a fence at each shared data access identified. Based on our experimental results in §6, our technique is a significant improvement over theirs in terms of static fence count, dynamic fence count, and running time of the generated programs.

Several other parallel analyses have been developed that do not directly address memory consistency issues. Jeremiassen and Eggers developed a static analysis for barrier synchronization [14] for non-textual barriers. With textual barriers, our analysis is more precise in finding memory accesses that cannot run concurrently. Duesterwald and Soffa used data flow analysis to compute the happened-before and happened-after relation for program statements. The information is used in detecting data races [9]. Masticola and Ryder developed non-concurrency analysis to identify pairs of statements in a parallel program that cannot run concurrently. The results are used for debugging and optimization [22]. Rugina and Rinard developed a thread-aware alias analysis for the Cilk multithreaded programming language [28] that is both flow-sensitive and context-sensitive. Others such as Zhu and Hendren [36] and Hicks [12] have developed flow-insensitive versions for multithreaded languages.

9 Conclusion

Memory consistency models have been a controversial issue in the design of shared memory hardware and languages for roughly two decades. While most experts agree that sequential consistency is the most natural semantics, few agree on what memory model can be used to obtain reasonable performance. This is true for languages designed to run on cache-coherent shared memory architectures as well as on distributed memory multiprocessors and clusters. The question is difficult to answer in the abstract because the penalty for sequential consistency depends on the quality of the optimizations that might be usable in a more relaxed semantics but not under sequential consistency. This paper provides evidence that, even for partitioned global address space languages like Titanium that run on a cluster network, sequential consistency may be a practical model.

The contributions of this paper include a set of compiler analyses that minimize the cost of guaranteeing sequential consistency. Some of the analyses take advantage of the unique synchronization and parallel control constructs in the Titanium language. In particular, they use the fact that threads must always reach the same textual instance of a barrier and that control expressions guarded by single-valued expressions will execute identically on all threads. We presented an experimental evaluation of several different levels of compiler analyses, all of which ensure sequential consistency but with increasing accuracy. The accuracy allows memory fences to be eliminated and other optimizations to be applicable. We experimented with several benchmark programs and showed that our most aggressive analysis was able to eliminate over 97% of the static memory fence instances that were needed by a naïve implementation. At runtime, these accesses accounted for 87 to 100% of the dynamically encountered memory fences in all but one benchmark, which required only slight modification to eliminate most of the remaining fences. We then combined the analysis with two communication optimizations, overlapping remote array copy operations with local computation and optimizing irregular accesses on remote arrays, and applied them to two linear algebra kernels. Our results show that even when combined with these high level communication optimizations designed for distributed memory environments, our most aggressive analysis for sequential consistency was able to obtain the same performance as a relaxed model. While additional work on parallel optimizations and analyses is needed, we believe these results provide important evidence on the viability of using a simple memory consistency model for global address space languages without sacrificing performance.

References

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

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks—summary and preliminary results. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165, New York, NY, USA, 1991. ACM Press.

[3] 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.

[4] 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.

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

[6] 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.

[7] A. Darte and R. Schreiber. A linear-time algorithm for optimal barrier placement. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN*

symposium on Principles and practice of parallel programming, pages 26–35, New York, NY, USA, 2005. ACM Press.

[8] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study, 2005. Submitted.

[9] 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.

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

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 15–26, New York, NY, USA, 1990. ACM Press.

[12] J. Hicks. Experiences with compiler-directed storage reclamation. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 95–105, New York, NY, USA, 1993. ACM Press.

[13] 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.

[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 and K. Yelick. Concurrency analysis for parallel programs with textually aligned barriers, 2005. Submitted.

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

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

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

[19] 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.

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

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

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

[23] 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.

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

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

[26] 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.

[27] 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.

[28] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 77–90, New York, NY, USA, 1999. ACM Press.

[29] 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.

[30] J. Su and K. Yelick. Automatic support for irregular computations in a high-level language. In *19th International Parallel and Distributed Processing Symposium*, April 2005.

[31] 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.

[32] 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.

[33] T. Wen and P. Colella. Adaptive mesh refinement in Titanium. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 89.1, Washington, DC, USA, 2005. IEEE Computer Society.

[34] 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.

[35] 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.

[36] Y. Zhu and L. J. Hendren. Communication optimizations for parallel C programs. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 199–211, New York, NY, USA, 1998. ACM Press.