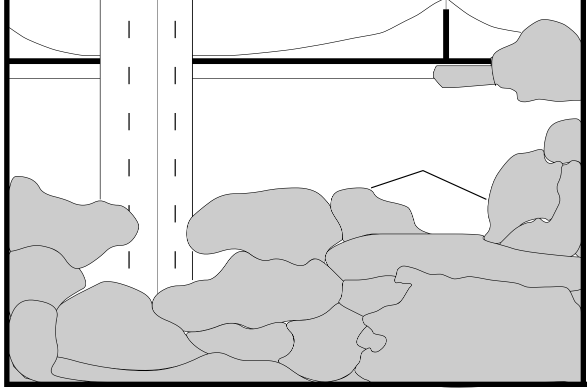
Data Sharing Analysis for Titanium

Ben Liblit
`liblit@cs.berkeley.edu`

Alex Aiken
`aiken@cs.berkeley.edu`

Katherine Yelick
`yelick@cs.berkeley.edu`



Report No. UCB//CSD-01-1165

November 2001

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Data Sharing Analysis for Titanium [*]

Ben Liblit
liblit@cs.berkeley.edu

Alex Aiken
aiken@cs.berkeley.edu

Katherine Yelick
yelick@cs.berkeley.edu

November 2001

Abstract

Parallel programs share data in ways that may not be obvious at the source level. Understanding a program's data sharing behavior is critical to understanding the program as a whole, and is also a necessary component for numerous program analysis, optimization, and run time clients. We report on the design of and experience with an implementation of a data sharing analysis for the Titanium programming language.

1 Introduction

Sharing analysis describes the ways in which data is (or is not) shared among components of a parallel computing system. Liblit et al. [9] develop a family of data sharing models based on type qualifiers, and show how type inference can automatically identify shared and private data in a small pointer- and tuple-manipulation language. That work also enumerates a variety of analysis clients which can potentially benefit from static knowledge of data sharing patterns. In this paper we present a case study of sharing analysis in the real world.

Section 2 briefly introduces the Titanium scientific programming language which we have extended to reflect data sharing. In Section 3 we examine specific Titanium features in greater depth to show how the sharing models and inference strategy used by Liblit et al. [9] map onto a complete, realistic programming language. We consider several analysis and optimization clients in Section 4, and report on the effectiveness of sharing analysis in supporting their implementation and deployment. Section 5 concludes.

*This research was supported in part by NASA Grant No. NAG2-1210, NSF Infrastructure Grant No. EIA-9802069, NSF Grant No. CCR-0085949, NSF Infrastructure Agreement No. ACI-9619020, DOE Prime Contract No. W-7405-ENG-48 through Memorandum Agreement No. B504962 with LLNL, and an NDSEG fellowship. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

2 Titanium: A Language for Distributed Scientific Computing

Titanium is an experimental language for high performance parallel computing. Titanium has the syntax and semantics of Java, although it compiles to native machine code rather than virtual machine bytecodes. Titanium extends Java with a distributed global address space, where processes can address, read, and write each other's data across physical machine boundaries [5].

Titanium does not use Java threads; rather, Titanium programs use a rigidly structured Single-Program Multiple-Data (*SPMD*) model of parallelism: each processor in a distributed computing cluster runs the same program on different data. A common mode of operation for such programs would be to partition a large grid, with each processor computing the local behavior of some simulation on its portion of that grid. Processors communicate to exchange boundary information, but for peak performance one tries to design SPMD algorithms in which most computation is local with only minimal cross-processor coordination.

Any reference in a Titanium program may be declared `local`. Unqualified references are assumed to be `global`, but may be changed to `local` by type inference. Earlier research has shown that automatic inference of `local` qualifiers is quite effective for real Titanium programs [8].

We have added `shared`/`private`/`mixed` qualifiers in the same spirit.[1] Unqualified references are assumed to be `shared`; programmers may declare references as `private` or `mixed` subject to validation by the type checker. Stronger (more private) qualifiers are added automatically using type inference with either late or early enforcement. Our inference engine is based on the

[1]In Titanium source code, "`private`" is spelled "`nonshared`" to avoid a naming conflict with Java's existing but unrelated `private` access qualifier, while "`mixed`" is spelled "`polyshared`" to suggest polymorphic sharing. These changes are merely syntactic, and we will continue to use `shared`/`private`/`mixed` in this paper except where giving literal examples of Titanium source code.

`cqual` qualifier inference engine [4].

Qualifiers appear after the qualified type. Thus, "`Object`" is a global reference to a shared object, while "`Object local nonshared`" is a local reference to a private object. Arrays are references as well, and each level within a multidimensional array carries its own type qualifiers:

$$\underbrace{\texttt{Object local polyshared}}_{\text{innermost element type}}\underbrace{\texttt{[] local}}_{\text{outer array}}\underbrace{\texttt{[] nonshared}}_{\text{inner array}}$$

The innermost array element type appears first, and subsequent left-to-right array qualifiers associate with outermost-to-innermost levels of the array. Filling in implicit global and shared qualifiers where necessary, then, this is the type of a local pointer to a shared array of global pointers to private arrays of local pointers to mixed objects.

Clearly, type syntax can grow cumbersome, particularly for the multidimensional arrays common in scientific programming. An explicit sharing qualification syntax supports the use of types as documentation, but may be cumbersome to maintain over the long term. Maintenance problems become especially acute when dealing with legacy code. Titanium incorporates sixteen thousand lines of Java source code for standard packages such as `java.io`, `java.lang`, and `java.util`. None of this source code contains explicit sharing qualifiers, but it would be impractical to annotate so much code by hand and to maintain those annotations over the long term as Java and Titanium development continue. Thus we wish to provide automated qualifier inference.

3 Accommodating Titanium Features

While the type systems presented by Liblit et al. [9] are a useful formalism for describing distributed data sharing, they also raise certain practical questions about how such approaches work in the real world. Titanium contains many features not present in the tiny data manipulation language used in that work. However, the core issues (such as global pointers to private data) can be extended to accommodate real languages. We briefly describe the highlights.

Titanium is object-oriented, with methods, inheritance, and class- and interface-based polymorphism. A method's actual arguments must match its formals; thus, if a method is observed to receive a shared argument in any context, the corresponding formal parameter is constrained to be shared or mixed within the method body. Method calls, then, are treated as a set of assignments: actuals are assigned to formals, and the returned value is assigned back to the caller as the result of the call.

Native methods, which are implemented by external C code, are treated conservatively. Because the compiler has no access to the implementation, it is never safe to change either the formal parameter types or the return type of a native method. This conservative approach can be taken in any situation where only partial information is available. For example, the analysis accommodates separate compilation by forcing conservative analysis at module boundaries.

Inheritance simply induces additional constraints between parent and child classes. A subclass is constrained to use identical types for any fields inherited from its parent. Interfaces and overridden methods are handled in the same manner.

The small data manipulation language of Liblit et al. [9] forbids access to global private data, where access is defined as dereferencing or assignment. For Titanium, "access" includes such operations as reading or writing a field, calling a non-static method, synchronizing, using the `instanceof` operator, or performing a checked cast. Some of these are design decisions: one might decide to treat an object's monitor lock or dynamic type information as being distinct from the object itself, and thereby allow corresponding operations on global private objects. The underlying models are flexible enough to describe any desired policy.

3.1 Arrays

All elements of an array must share a single type. Thus, an array can be treated as a referenced object containing a single field, where the type of that field is the type of the array elements. We consider each use of an array type to be distinct: if the source program declares two arrays of shared objects, inference may change one to an array of private objects while leaving the other unchanged. There are no implied constraints between the two arrays simply because they had the same type in the source program. This flexibility is not extended to named classes, where we require global agreement upon the types of all fields.

A particularly tricky issue is handling type casts involving arrays. When an array is implicitly cast to `Object`, we forbid changes to any "forgotten" qualifiers below the topmost level of the array type. When an `Object` is dynamically cast back to an array type, we also forbid changes to any "remembered" qualifiers below the topmost level. By holding the qualifiers fixed in both cases, we ensure that any dynamic casts will behave identically in the original and optimized programs. Otherwise, if qualifiers were changed in the array declaration but not the explicit cast, or vice versa, dynamic cast failures would occur where none existed in the original program. This appears to be a general consequence of the Java type system's treatment of arrays and array casting; local qualifier inference deals with an equivalent issue in the same manner [8].

3.2 Conditional Expressions

In general, a conditional expression (`?:`) is sound provided that the two alternatives have some common supertype. However, Java is more restrictive, requiring that one of the alternatives be a supertype of the other. Thus, the following expression is not valid Java, because neither `String` nor `Vector` is a supertype of the other:

```
test ? new String() : new Vector()
```

In order to remain consistent with the spirit of Java's type system, we extend this restriction to include sharing qualifiers. If α and β are the sharing qualifier variables for the two alternative branches, we add the following conditional constraints:

$$\begin{aligned} \alpha \leq \texttt{shared} &\implies \texttt{shared} \leq \beta \\ \beta \leq \texttt{shared} &\implies \texttt{shared} \leq \alpha \\ \alpha \leq \texttt{private} &\implies \texttt{private} \leq \beta \\ \beta \leq \texttt{private} &\implies \texttt{private} \leq \alpha \end{aligned}$$

This forbids solutions that bind one of the qualifiers to `shared` and the other to `private`, but allows all other solutions where the qualifiers match or where at least one of the qualifiers is `mixed`. If inference would otherwise have chosen mismatched qualifiers, these constraints will force one branch or the other to be `mixed` instead. The resolution strategy given by Liblit et al. [9] is biased in favor of `private` qualifiers, so the `private` branch will be retained and the `shared` branch will instead be forced to `mixed`.

This makes the analysis more conservative than strictly required for soundness, and one can construct artificial examples that reflect this. In practice, we have found only a single realistic benchmark in which these restrictions have any affect at all, and even in that case the overall impact on the analysis results is negligible.

3.3 Data Declarations and Early Enforcement

The data manipulation language of Liblit et al. [9] received all data declarations from a predefined environment; in Titanium, data declarations are given by the programmer with explicit types (as in "`Object foo`"). Explicit types can appear when declaring local variables, fields, method formal parameters, method return types, and `catch` clause arguments. The early enforcement type system of Liblit et al. [9] required that a well-formed environment contain no global or shared pointers to private or mixed data. The Titanium type checker enforces a similar restrictions on data declarations: when early enforcement is being used, global pointers must also be shared and the referent of a shared pointer must contain only shared fields.

When performing inference, the first of these restrictions induces strict equality constraints: given a declaration `Object foo`, because `foo` is global we would require $\delta_{\texttt{foo}} = \texttt{shared}$ where $\delta_{\texttt{foo}}$ represents the inferred sharing qualifier for this declaration.

The restriction on contained fields induces conditional constraints, best illustrated by example. Suppose we have the following declarations:

```
class Cell {
    Object local δ1 up;
    Object local δ2 down;
    Object local δ3 left;
    Object local δ4 right;
}

Cell local γ1 red;
Cell local γ2 green;
Cell local γ3 blue;
```

where δ_j and γ_i represent constraint variables for the corresponding field and variable declarations. A naïve implementation of the contained fields restriction would use one conditional constraint relating each data declaration to each contained field:

$$\gamma_i = \texttt{shared} \implies \delta_j = \texttt{shared}$$

This would introduce a number of conditional constraints equal to the product of all data declarations and fields within those declarations.

For better scaling, we introduce an additional qualifier, κ, representing the `Cell` class as a whole. We require that κ reflect the least upper bound of all uses of `Cell` within declarations: $\gamma_i \leq \kappa$. We also require that each field of `Cell` be shared if κ is shared:

$$\kappa = \texttt{shared} \implies \delta_j = \texttt{shared}$$

To extend this into any fields that `Cell` inherits from a superclass, we also add a constraint $\kappa \leq \kappa'$ where κ' is the constraint variable associated with `Cell`'s superclass.

Thus, if any of `red`, `green`, or `blue` is found to be shared, then κ will be bound to `shared`, which will in turn force all fields of `Cell` to be shared as well, which is precisely the desired effect. The introduced κ variable serves to summarize all uses of `Cell`, allowing us to introduce only 3+1 inequality constraints plus 4 conditional constraints, rather than 3×4 conditional constraints under the naïve approach.

3.4 Polymorphism and the Role of Mixed Sharing

As suggested by Liblit et al. [9], the default `Object()` constructor is assumed to receive a mixed `this` argument, so that it can be called to build either shared or private objects. More generally, a compiler-introduced default constructor in any class is always assumed to be mixed, as this offers the greatest flexibility for any subclasses. As in Java, such constructors are only introduced if the class has no explicit constructors.

Field initialization expressions can be thought of as additional code inserted at the start of each constructor. When type checking such expressions, we have three options for how to treat `this`:

1. Assume that `this` is mixed. If an initialization expression type checks under this condition, then it would work in any constructor.

2. Assume that `this` is shared if all constructors are shared, private if all constructors are private, and mixed otherwise. If an initialization expression type checks under this condition, then it would work in any constructor. However, it makes program comprehension harder, as one cannot understand any initializer without considering all constructors as well.

3. Type check each initialization expression up to three times, with `this` as each of shared, private, and mixed. Omit any type checks that do not correspond to at least one constructor. This provides maximal flexibility, but makes program comprehension harder and also increases the cost of type checking.

We have selected the first approach, which is by far the simplest. Mixed data admits few optimizations, but field initialization expressions are neither complex nor performance critical in typical programs. Mixed data cannot be accessed remotely, but `this` is always local during field initialization. Most field initializers do not even refer to `this` at all. Thus, we find the mixed-`this` assumption to be quite reasonable in practice.

4 Experimental Findings

We have used qualifier inference to study the data sharing behavior of several benchmark programs ranging in size from small algorithm kernels to complete applications. All benchmarks are designed for distributed execution, and reflect the scientific focus of SPMD programming. In order of size, the benchmark programs are:

`pi`: 56 lines. A simplistic Monte Carlo simulation, intended as an illustrative micro-benchmark rather than as a full application. We estimate π using 5,000,000 random tosses on one processor.

`sample-sort`: 321 lines. Sample sort, a distributed sorting algorithm. We sort 2^{18} thirty two bit integer keys, with 64 keys per sample, on four processors.

`lu-fact`: 420 lines. LU factorization for dense matrices. We factor a 1024×1024 element random matrix, partitioned into sixty four 128×128 element blocks, on four processors. No pivoting is used.

`cannon`: 518 lines. Cannon's algorithm for dense matrix multiplication. We multiply a pair of random 200×200 matrices on four processors.

`3d-fft`: 614 lines. Fast Fourier transform. We perform a 3D FFT on 64^3 random floating point values on eight processors.

`n-body`: 826 lines. A simple n-body simulation based on n^2 operations for n bodies. We simulate fifty particles on four processors.

`gsrb`: 1090 lines. The Gauss-Seidel Red Black algorithm for solving elliptic partial differential equations. We solve a 2048×128 element problem, partitioned into four 512×128 element patches across 100 full iterations, on four processors.

`particle-grid`: 1095 lines. An n-body simulation with limited range of interaction: only particles in neighboring processors affect each other. We simulate fifty particles on four processors.

`pps`: 3673 lines. A parallel solver for elliptic equations with infinite domain boundary conditions, using a two-level domain decomposition approach [2]. We solve a 512×512 element problem partitioned into four 128×128 element patches, with a refinement ratio of 16 between coarse and fine grids, on four processors.

`ib`: 3777 lines. A heart simulation using the immersed boundary method [10]. Involves a particle/grid method and spectral fluid solver. We model a contractile torus for eight iterations of the immersed boundary method on two processors.

`amr`: 5206 lines. A parallel solver for the Poisson equation using adaptive mesh refinement [11]. We operate on a four-level grid hierarchy for twenty iterations on four processors.

`gas`: 8841 lines. An implementation of the Berger-Colella algorithm for solving hyperbolic equations using adaptive mesh refinement [3]. We model a Mach 10 shock wave hitting a solid surface at an oblique angle on four processors.

4.1 Static Sharing Metrics

Late enforcement permits global private pointers, provided that they are never actually used. While Liblit et al. [9] speculate on why such opaque pointers might be useful, it is not obvious that this corresponds to the behavior of real programs. If opaque global private pointers have no practical value, then we would expect that early and late enforcement would yield substantially equivalent results. Since early enforcement is also usable by more clients, it is reasonable to ask whether late enforcement has any real benefit.

For each benchmark program, we identify all syntactic locations where a sharing qualifier could possibly appear. This includes declarations of local variables, fields, method formal parameters, method return types, and `catch` clause arguments. It also includes the types used in casts and the `instanceof` operator, and an additional qualifier that characterizes the implicit `this` parameter to each non-static method. We exclude declarations of primitive types, and distinguish each level of indirection in multidimensional arrays. Thus, a local variable declared as "`int value`" is not a candidate site, whereas "`Object[][] table`" has three potential sites for sharing qualification.

Table 1(a) shows the number of candidate sites in each benchmark, and the count of sites inferred shared, mixed, and private under late and early enforcement. Although whole-program inference was used, we present here only those sites appearing in the benchmark application code (not in libraries). Table 1(b) presents equivalent counts scaled as a percentage of the total number of candidate sites.

4.1.1 Late versus early

Benchmarks are ordered by size in source code lines. For small- and medium-sized benchmarks, late and early enforcement are indistinguishable: the thousand-line `particle-grid` program shows only a single qualifier change, and smaller benchmarks show no changes whatsoever. For the larger benchmarks, we do see that early enforcement forces several qualifiers to be `shared` or `mixed` where late enforcement allowed `private`. Even in these cases, though, the number of qualifiers changed is relatively small. Only one benchmark, `pps`, shows a pronounced difference: the proportion of `private` qualifiers drops from 54% under late enforcement to 50% under early enforcement. It appears that the late/early distinction is not significant for most programs.

Regardless of which system is used, we do consistently identify large amounts of private data in all benchmarks of all sizes. The largest benchmark, `gas`, is found to use private data at half of all declaration sites. Other benchmarks range from 16% to 75%, and overall 46% of all sites in all benchmarks are inferred `private`. This is encouraging news for analysis clients which may want to exploit such information. It also reinforces the need for inference: it is unlikely that any human programmer could correctly place all such qualifiers by hand, or maintain such qualifiers over time.

4.1.2 Mixed sharing

We observe that a few `mixed` qualifiers appear in nearly every benchmark. In many cases, mixed data is found in utility code shared by distinct parts of the application. For example, the one `mixed` qualifier found in `3d-fft` applies to the constructor for `Imaginary`, an implementation of imaginary numbers. If `mixed` were unavailable, the constructor could only be `shared`, which would in turn force all imaginary numbers to be treated as shared data even when used by only a single process.

In other benchmarks, we see code of the following general form:

```
if (...)  data = myPrivateData;
else      data = mySharedData;
...use data ...
```

When a single piece of code may manipulate either private or shared data based on some runtime condition, that data will be inferred `mixed`. Again, if `mixed` were unavailable, we would have no choice but to declare `data` as `shared`, which in turn would force `myPrivateData` to `shared`. One might consider a limited form of polymorphism available only at methods, but even that would require some form of automated code factoring to isolate the `data`-manipulating code into its own method. The `mixed` qualifier, then, may be more important to the overall system than its small numbers would suggest.

4.2 Analysis Performance

Liblit et al. [9] give an efficient algorithm for computing the best (maximally private) solution to a set of sharing constraints. Table 2 presents the wall clock running

benchmark	sites	late shared	late mixed	late private	early shared	early mixed	early private	difference shared	difference mixed	difference private
`pi`	12	3	0	9	3	0	9			
`sample-sort`	73	28	1	44	28	1	44			
`lu-fact`	150	81	4	65	81	4	65			
`cannon`	162	59	4	99	59	4	99			
`3d-fft`	191	70	1	120	70	1	120			
`n-body`	113	86	1	26	86	1	26			
`gsrb`	281	135	2	144	135	2	144			
`particle-grid`	201	167	1	33	168	0	33	+1	-1	
`pps`	551	224	27	300	244	32	275	+20	+5	-25
`ib`	1094	616	7	471	637	7	450	+21		-21
`amr`	1353	776	12	565	794	12	547	+18		-18
`gas`	1699	851	3	845	867	2	830	+16	-1	-15

(a) Absolute counts of inferred qualifiers

benchmark	late shared	late mixed	late private	early shared	early mixed	early private	difference shared	difference mixed	difference private
`pi`	25%	0%	75%	25%	0%	75%			
`sample-sort`	38%	1%	60%	38%	1%	60%			
`lu-fact`	54%	3%	43%	54%	3%	43%			
`cannon`	36%	2%	61%	36%	2%	61%			
`3d-fft`	37%	1%	63%	37%	1%	63%			
`n-body`	76%	1%	23%	76%	1%	23%			
`gsrb`	48%	1%	51%	48%	1%	51%			
`particle-grid`	83%	$< 1\%$	16%	84%	0%	16%	$< 1\%$	$< 1\%$	
`pps`	41%	5%	54%	44%	6%	50%	+4%	+1%	-4%
`ib`	56%	1%	43%	58%	1%	41%	+2%		-1%
`amr`	57%	1%	42%	59%	1%	40%	+1%		$< 1\%$
`gas`	50%	$< 1\%$	50%	51%	$< 1\%$	49%	+1%	$< 1\%$	$< 1\%$

(b) Relative counts of inferred qualifiers

Table 1: Inference results for static candidate sites

benchmark	late (msec)	early (msec)
pi	546.7	717.1
sample-sort	558.4	597.9
lu-fact	580.8	620.8
cannon	597.8	635.0
3d-fft	613.5	668.1
n-body	567.4	689.6
gsrb	658.1	768.0
particle-grid	597.4	695.0
pps	795.8	851.4
ib	813.5	873.3
amr	842.3	914.7
gas	1028.2	1094.8

Table 2: Time required for sharing inference

time of the sharing inference phase of compilation for each benchmark. The table is ordered by benchmark size, and all times are given in milliseconds. Measurements were taken on a 1.3 GHz Pentium 4 Linux workstation.

Overall, performance is quite good: the largest benchmark takes barely more than one second. Early enforcement calls for additional constraints, and therefore is slower than late enforcement, but not radically so. It is difficult to directly translate these times into per-line or per-site scaling metrics, as our line and site counts exclude the common class library whose size dwarfs that of the benchmark applications. In broad terms, though, it appears that sharing inference could be applied to programs many times larger than those used here without disproportionately slowing compilation.

4.3 Optimizations

There are numerous potential ways to leverage knowledge of the sharing behavior of a program; Liblit et al. [9] lists several. We have implemented a subset of these optimizations, and report our findings here.

4.3.1 Data location management

Shared memory may be a scarce or costly resource on some systems. We have instrumented each benchmark to tally the number of shared and private allocations over the course of an entire run. Table 3 gives these totals, in bytes, for each of late and early enforcement. Observe that we see slight differences between the two enforcement schemes even on small benchmarks which reported identical results in Table 1. This is because that earlier table examined only application code and excluded libraries, whereas these allocation counts apply to the entire program. Slight differences in inference results for library code are visible here as slight differences in allocation counts for late versus early enforcement.

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

Most byte counts to not vary appreciably between late and early enforcement, though `amr` sees an 11% shift. The most dramatic shift is found in `pps`: late enforcement allows 74% private allocation, while early enforcement drops that to merely 19%. In Section 4.1.1 we observed that `pps` showed a relatively large difference in static private declaration counts as well. Clearly those differences encompass data structures which account for a preponderance of `pps`'s runtime memory consumption. When running on machines with costly shared memory, `pps` stands to benefit greatly from data location management guided by sharing inference.

4.3.2 Synchronization elimination

Private data can never come under lock contention, so synchronization operations on private data can be reduced to simple nullity checks. We have added this simple optimization to the Titanium compiler and examined its effect on our benchmarks. For `ib`, 31 out of 107 static synchronization sites can be optimized. For `particle-grid`, we optimize 33 out of 108. For every other benchmark, we eliminate 33 out of 107 synchronizations. These statistics cover all synchronizations in the entire program, including those in standard library code.

Reducing a third of all synchronization sites to nullity checks is good, but the uniformity of the results is cause for skepticism. In all likelihood, there is a stable core of 107 sites appearing in library code which all benchmarks import, such as from the synchronization-intensive `java.io` package. Of these, the same 31–33 are consistently found to be private. Within the benchmarks' source code, only a handful of synchronization sites are found: 4 in each of `particle-grid` and `n-body`, and none anywhere else. Monitor locking is simply not a widely used coordination mechanism in SPMD programs.

As expected, no measurable performance improvement results from eliminating these 33 synchronization sites. However, the fact that one third of sites are eliminated is encouraging, even if they do stem from library code. The intensive use of barrier coordination is a

benchmark	**late**				**early**			
	shared		**private**		**shared**		**private**	
`pi`	76283	(75%)	26072	(25%)	76283	(75%)	26072	(25%)
`sample-sort`	3385510	(5%)	68047664	(95%)	3427766	(5%)	69470768	(95%)
`cannon`	8981006	(60%)	5906248	(40%)	8981006	(60%)	5906248	(40%)
`3d-fft`	4869008	(52%)	4432352	(48%)	4869086	(52%)	4432352	(48%)
`n-body`	376623	(< 1%)	104141288	(100%)	376623	(< 1%)	104141288	(100%)
`particle-grid`	9739460	(99%)	125552	(1%)	9741804	(99%)	125552	(1%)
`pps`	19926363	(26%)	56688947	(74%)	61970108	(81%)	14645175	(19%)
`amr`	37330271	(88%)	4956819	(12%)	41973771	(99%)	313295	(1%)
`gas`	2649713367	(55%)	2209303072	(45%)	2649974879	(55%)	2209041536	(45%)

Table 3: Bytes allocated in shared or private memory. We omit `gsrb` and `ib` due to unrelated Titanium bugs which prevent them from running to completion.

peculiar feature of SPMD programming, and relatively shallow dependence upon the standard Java libraries is peculiar to Titanium's user base of scientific/numerical coders. A wider sample of the distributed programming spectrum may reveal programs for which synchronization speed is performance critical, and which would then accrue sizable benefit from sharing-inference-directed synchronization elimination.

4.3.3 Consistency model relaxation

Titanium uses a fairly weak consistency model [5]. A stronger model would be a more attractive programming target if it did not unacceptably harm performance. We can use sharing inference to selectively weaken a strong consistency model in places where only private data is being manipulated. We have implemented such an optimization for a sequentially consistent variant of Titanium.

Implementation strategy Because the Titanium compiler is implemented as a Titanium-to-C translator, sequential consistency requires cooperation both from the Titanium compiler itself as well as the C compiler which ultimately produces native code. In the current Titanium compiler, the only reordering transformation which would violate sequential consistency is the lifting of invariant expressions out of `foreach` loops. For sequentially consistent Titanium, we suppress lifting of expressions which read or write shared data. (Lifting takes place after compound expression rewriting, so a complex source expression which accesses some shared data and some private data can still have its private portions lifted, so long as the shared portions stay in place.)

Extracting sequential consistency from the C compiler requires a bit of subterfuge. Our general approach is to restrict reordering by placing a read fence before each shared read and a write fence after each shared write. These fences must suppress reordering within the C optimizer and also take any necessary steps to prevent reordering by the host architecture at run time.

Our benchmarks were run on a four-way Pentium III multiprocessor. This platform uses a "write ordered with store-buffer forwarding" memory ordering model [7]. For our purposes, this means that we need only be concerned with read reordering at the architectural level; write reordering cannot occur, even across processors.

Using `gcc` 3.0.1 as our native compiler, we can enforce sequential consistency as follows. Before each shared read, we inject the following C code:

```
asm volatile ("lock; addl $0,0(%%esp)"
              :  :  :  "memory")
```

This inline assembly directive inserts a locked add instruction into the generated code. The add instruction itself has no effect, but the `lock` prefix forces the processor to wait until all preceding instructions have completed and all buffered writes have been drained to memory. The `mfence` instruction would offer a closer match to the required functionality. However, this is part of the SSE2 extensions to the IA-32 architecture, and is not available on the Pentium III [6].

The `volatile` keyword informs `gcc` that the assembly code should not be removed even though it appears to compute no useful result, while the `"memory"` clobber specification declares that the code may have arbitrary effects on memory. Together, these create an optimization barrier that prevents `gcc` from deploying most optimizations across such an instruction; `gcc` will not, for example, retain values in registers or combine or reorder instructions across this barrier.

After each shared write, we need not be concerned with architectural reordering and therefore require only the optimization barrier:

```
asm volatile ("" :  :  :  "memory")
```

Because gcc has such rich inline assembly facilities, other architectures can be accommodated without difficulty. One need merely determine the appropriate instructions to suppress architectural reordering, and embed them within the optimization barrier described above. A SPARC v9 multiprocessor, for example, requires "" before shared reads and "fence #StoreLoad" after shared writes [12].

Benchmark results Table 4 reports wall clock execution times of our benchmarks using the data sets outlined at the start of this section. Benchmarks were run on a Linux 2.4.1 SMP with four 550 MHz Pentium III CPU's and 4GB of physical RAM. We present running times using each of four configurations:

weak The weak consistency model used in standard Titanium.

late Sequential consistency enforced except where late enforcement can identify private data.

late Sequential consistency enforced except where early enforcement can identify private data.

naïve Sequential consistency enforced everywhere.

For ease of comparison we also present the late, early, and naïve times as slowdown factors relative to the weak time. This is computed as $\frac{time}{weak}$ so that, for example, a slowdown of 2.25 for naïve pi indicates that it ran 2.25 times slower than the weakly consistent version.

The large speedup for weak pi confirms that sequential consistency is costly if bluntly applied. Sharing inference is able to identify enough private data, though, to erase that penalty in the late and early variants. Hand inspection shows that sharing inference for pi is perfect: all data in the main computational loop is inferred private and no restrictions are needed on optimizations to enforce sequential consistency. The late and early versions yield machine code identical to that under the weak model. The apparent early pi slowdown (1.15) and late pi speedup (0.99) are measurement noise.

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

Among the larger benchmarks, cannon, 3d-fft, and amr have the most room for benefits. In 3d-fft, sharing inference (either late or early) is able to nearly match the weak model. Modest benefits are seen in cannon, where the larger slowdown is only partly offset by inference. Late and early enforcement yield identical results for cannon; the difference between the late and early slowdown factors is measurement noise.

The results for amr are interesting. None of the key performance-critical data structures can be inferred private using our current system. Like many SPMD programs, amr has an alternating-phase structure: all processors exchange boundary information, then each processor updates its own local portion of the shared grid, then all processors communicate again, and so on. Data is shared widely during amr's communication phase, but we would like to treat that same data as private during local computation phases. These phases are delimited by global barrier operations, so no processor looks at another processors' data while the local computations are taking place. For sharing inference to be effective here, it would need to allow for a limited form of flow sensitivity keyed to these phases. Because the structure of barriers is quite regular in practice [1], we believe such an extension of our techniques should be feasible.

We also observe that two benchmarks, sample-sort and n-body, exhibit unexpected speedups under sequential consistency. Because the direct penalty of sequential consistency here is so small, measurement noise due to secondary effects (such as cache alignment and the layout of the machine code) is noticeable.

5 Conclusions

We have presented a case study of data sharing analysis as applied to the Titanium scientific programming language. With careful design, sharing qualifiers can accommodate all of the expected features of a complete, real-world language. The approach is flexible enough to encompass several formal sharing models in a variety of sound language designs. Efficient type qualifier inference can be added to a compiler at little cost, making static sharing information available for use by a variety of subsequent analyses and optimizations.

Static assessment reveals that benchmark programs do exhibit a variety of sharing patterns, with roughly half of all heap reference declaration sites corresponding to private data. Small differences between late and early enforcement are seen; these may become more significant for larger applications. Optimization effectiveness varies widely. Synchronization elimination is easily

benchmark	execution time (sec)				slowdown vs. weak		
	weak	**late**	**early**	**naïve**	**late**	**early**	**naïve**
`pi`	14.69	14.55	16.96	33.01	0.99	1.15	2.25
`sample-sort`	15.52	11.47	12.05	16.65	0.74	0.78	1.07
`cannon`	11.21	21.08	19.78	22.95	1.88	1.76	2.05
`3d-fft`	8.06	8.15	8.14	9.82	1.01	1.01	1.22
`n-body`	143.48	119.25	114.27	112.66	0.83	0.80	0.79
`particle-grid`	42.21	44.35	44.62	43.41	1.05	1.06	1.03
`pps`	301.25	314.35	323.28	324.19	1.04	1.07	1.08
`ib`	188.73	198.94	199.83	205.72	1.05	1.06	1.09
`amr`	83.48	104.32	104.09	103.96	1.25	1.25	1.25
`gas`	383.97	395.57	395.00	419.25	1.03	1.03	1.09

Table 4: Performance cost of sequential consistency. We omit `gsrb` due to unrelated Titanium bugs which prevent it from running to completion.

implemented, but not useful for our SPMD benchmark suite. Most benchmarks would benefit from data location management, an important issue when shared memory is a costly resource. Consistency model relaxation is difficult to assess, as many of the programs tested do not actually slow down when confined to sequential consistency. Where a cost is seen, sharing inference can sometimes reduce or eliminate it. In other cases, sharing patterns change over time, and a flow- or phase-sensitive analysis would be required to properly capture programs' sharing behavior.

References

[1] A. Aiken and D. Gay. Barrier inference. In *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 342–354, San Diego, California, January 19–21, 1998.

[2] 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.

[3] 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.

[4] J. Foster. `cqual`. Available at `http://bane.cs.berkeley.edu/cqual`, Nov. 2001.

[5] P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, and K. Yelick. Titanium language reference manual. Technical Report CSD-01-1163, University of California, Berkeley, Nov. 2001.

[6] Intel Corp. *Basic Architecture*, volume 1 of *IA-32 Intel Architecture Software Developer's Manual.* Intel Corp., Mt. Prospect, Illinois, 2001.

[7] Intel Corp. *System Programming Guide*, volume 3 of *IA-32 Intel Architecture Software Developer's Manual.* Intel Corp., Mt. Prospect, Illinois, 2001.

[8] B. Liblit and A. Aiken. Type systems for distributed data structures. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 199–213, Boston, Massachusetts, January 19–21, 2000.

[9] B. Liblit, A. Aiken, and K. Yelick. Type systems for distributed data sharing. In preparation, Nov. 2001.

[10] C. S. Peskin and D. M. McQueen. A three-dimensional computational method for blood flow in the heart. I. Immersed elastic fibers in a viscous incompressible fluid. *Journal of Computational Physics*, 81(2):372–405, Apr. 1989.

[11] G. Pike, L. Semenzato, P. Colella, and P. N. Hilfinger. Parallel 3D adaptive mesh refinement in Titanium. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, Mar. 1999.

[12] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual, version 9.* PTR Prentice Hall, Englewood Cliffs, New Jersey, 2000.