There is Nothing Wrong with Out-of-Thin-Air: Compiler Optimization and Memory Models

Clark Verbrugge
School of Computer Science
McGill University
Montréal, Canada
clump@cs.mcgill.ca

Allan Kielstra
IBM Toronto Lab
Markham, Canada
kielstra@ca.ibm.com

Yi Zhang
School of Computer Science
McGill University
Montréal, Canada
yi.zhang6@mail.mcgill.ca

Abstract

Memory models are used in concurrent systems to specify visibility properties of shared data. A practical memory model, however, must permit code optimization as well as provide a useful semantics for programmers. Here we extend recent observations that the current Java memory model imposes significant restrictions on the ability to optimize code. Beyond the known and potentially correctable proof concerns illustrated by others we show that major constraints on code generation and optimization can in fact be derived from fundamental properties and guarantees provided by the memory model. To address this and accommodate a better balance between programmability and optimization we present ideas for a simple concurrency semantics for Java that avoids basic problems at a cost of backward compatibility.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.3.3 [*Language Constructs and Features*]: Concurrent, distributed, and parallel languages; D.3.4 [*Processors*]: Compilers

General Terms Languages, Design, Performance

Keywords memory consistency, compiler optimization

1. Introduction

The recently revised Java memory model (JMM) provides a precise memory consistency model for a Java context [1]. The JMM tries to satisfy perhaps three main goals: allowing compiler optimizations, giving precise and useful formal guarantees to programmers writing "correct" concurrent programs, and giving a precise semantics to concurrent programs even when "incorrect," or containing data races.

A precise semantics and guarantees of sequential consistency (SC) for data-race-free (DRF) programs are extremely useful to programmers and for helping define the allowable behaviours of a Java virtual machine; Saraswat et al. consider the connection between DRF and SC as the *Fundamental Property* [2]. The importance of allowing compiler optimizations is also critical; Java virtual machine implementations achieve impressive speeds due to a wide array of compiler optimizations [3], and it is practically necessary to allow current and potential future optimizations to co-exist with any memory consistency model.

The JMM does not allow *all* compiler optimizations; certainly some forms of code motion are forbidden, as described in the original specification [1]. Manson *et al.* provide a proof that some basic optimizations are allowed, but the actual scope of allowed versus forbidden optimizations is far from clear. Other authors have since demonstrated that through examination of the model subtleties, other simple reordering-optimizations are also disallowed [4, 5]. Some of these concerns have been recently addressed, and repairs to the model have been described [6]. Here we show that even with improvements to the justification process to permit these optimizations, other very basic problems exist. In particular, the often-mentioned "out-of-thin-air" guarantee, as well as the necessity of preserving sequential consistency for race-free programs result in significant constraints on optimization, limiting important existing techniques and imposing additional analysis costs.

As a possible solution we explore a conceptually simple modification to Java semantics that guarantees lack of race-conditions by construction. This design uses a syntactically trivial change to the type system, enforcing behaviours akin to OpenMP [7], UPC [8] or C# [9] data-sharing directives to ensure shared data are easily distinguished by the compiler. The design we propose has similar goals to the JMM, but takes the stance that program comprehension by both developers and compiler writers must take priority. Our approach represents a basic shift in perspective—rather than starting from a model which *requires* expensive and difficult-to-understand conflict/race detection for basic safety, we start from an inherently safe model, allowing compiler optimizations to more easily preserve safety during transformation.

We make the following specific contributions:

- "Out-of-thin-air" values are a known consequence of code optimization and generation [2, 10]. We extend these observations and show that disallowing such behaviour impacts large classes of useful optimization that use or reuse space to improve performance.
- We show that the most useful, basic guarantee of the JMM, ensuring sequential consistency for race-free programs, imposes major and arguably unacceptable constraints on program optimization.
- We propose a syntactically trivial change to the Java language that enables practical preservation of race-free, sequentially consistent execution. Our design simplifies preservation of safety properties in optimization while still enabling advanced optimizations, and could also be easily incorporated into different concurrent language environments, such as C++ [10].

2. Related Work

Our work is intended to provide a simpler and more practical memory model that accommodates both programmers and compiler writers. We build quite specifically on previous work on the Java memory model, as well as on more generic consistency properties.

The current JMM [1] is a significant achievement, addressing basic flaws found by Pugh [11] in the original specification, and in particular relaxing the need for compilers to ensure a consistency property slightly stronger than coherence [12]. The need to provide a semantics for racey programs, however, make the justification part of this specification quite complex. Close analysis by several authors subsequently has resulted in identification of subtle flaws. Cenciarelli *et al.* point out that some independent statements cannot be reordered under the justification model in the JMM [4], and Aspinall and Ševčík show a number of examples of disallowed code changes, illustrating a variety of surprising consequences [5]. The latter two authors have extended their work, demonstrating both specific proof errors as well as offering modifications that eliminate most of the problems [6, 13].

Much of the complexity in the JMM is unnecessary in the new C++ model for multithreading, which provides no guarantees at all for programs with race conditions [10]. Programs must be correctly synchronized to have defined semantics, and if so, sequential consistency is guaranteed. This is in keeping with the overall emphasis on complete programmer control (and responsibility) in C/C++ programming, and provides a dramatically simpler conceptual model.

The JMM and C++ consistency models, as well as many others, focus on the data-race-free-0 (DRF0) property as core and desireable behaviour [14]. Programs without races can be shown to result in only sequentially consistent execution, provided the compiler treats synchronization points as kinds of code-motion barriers. As we will argue in the next section, however, the dynamic nature of race-freedom means even this basic guarantee is quite strong, with significant implications for compilers.

The complexity and variety of memory models [15] has inspired a number of efforts to unify designs, both for theoretical and practical exploration. Midkiff *et al.* also make the observation that memory models are too complex for programmers [16] and propose a framework for exploring consistency models that separates programs and consistency specifications. The aim of their design is to make prototyping and practical examination of consistency rules easier. Arvind and Maessen describe a more abstract framework for representing relaxed models based on different reordering rules [17]. *Store atomicity*, is central to their model; associating individual loads and stores within a partial order of program events as a means of proving serializability. Relaxation of this allows them to represent weaker models as well. Ferreira *et al.* raise the abstraction level further in their general semantics, able to represent a variety of weak memory models [18]. Their approach is parameterized by a relation describing program equivalences, and they use a particular instantiation of the relation to show DRF properties can be preserved under several basic compiler optimizations. Interestingly, they also make the point that out-of-thin-air is not necessarily problematic, provided type-safety is preserved. Saraswat *et al.* describe another, more concrete operational framework for representing memory models, based on atomic actions and decomposition rules for composite actions [2]. They also raise the issue of complexity in ensuring race-freedom, as well as the impact of forbidding out-of-thin-air values on compiler optimization, and point out that out-of-thin-air values can easily arise from incremental long word constructions. In Java avoiding out-of-thin-air values can be seen as part of a security guarantee, ensuring data cannot leak to unintended contexts. Our approach in this work is to provide simple (non-)observability guarantees, greatly reducing the cost and complexity of determining whether out-of-thin-air data has been exposed.

Our work is partly motivated by the complexity that determining data conflicts, and more specifically data races adds to analysis. Static and dynamic approaches to this problem have been tried, with the former having the advantage of avoiding runtime overhead, and several static, type-based approaches have been explored [19, 20]. These designs use the (sets of) locks that guard shared data as types, ensuring that any potential runtime access respects the type system by holding the required locks. Although this approach has been shown to be quite effective, the resulting type systems require non-trivial language extension, significant annotation, and still require some form of escape mechanism to allow the programmer to perform (statically) type-unsafe activities. For more precise data-race analysis, dynamic approaches that track actual control flow and thread behaviour can be more accurate. Overhead is a concern, however, and recent results have shown the performance loss of performing precise, runtime verification can be reduced to an impressive, but still quite significant 8-fold slowdown [21].

Designation of private versus shared data is a of course a common concept in parallel and multi-process programming. Our model essentially provides a Java embedding of a subset of UPC [8], Titanium [22], or OpenMP [7] data sharing behaviours, with a similar model to UPC in using a default *private* scope for data and requiring explicit specification of *shared* memory. More complex sharing models, such as the different OpenMP data initialization modes (*firstprivate* or *copyin*) can also be supported. Other designs for full implementations of OpenMP such as JOMP [23] and the more recent JaMP [24] have concentrated on supporting the parallel primitives more than the sharing directives, and are based on parsing structured comment blocks containing standard OpenMP directives. This approach is in fact the basis for a complete Java OpenMP proposal [25]. This design avoids language modifications, although it also necessarily involves co-existence with the existing Java behaviours, and thus any benefit provided through the higher-level, OpenMP model is dependent on programmers exercising appropriate restraint. By enforcing a simple, race-free model we lose backward-compatibility, but can guarantee sequential consistency and avoid the more complex memory model concerns present in Java, as well as the memory consistency concerns in full OpenMP implementations [26] which also complicate analysis [27].

3. The Java Memory Model and Optimization

The Java memory model defines behaviour for race-free (correctly synchronized) programs and for programs containing races. In the former case sequential consistency is guaranteed, and in the latter the behaviour can be determined through a defined process consisting of two main components, *happens-before* consistency, and a system for justifying *well-behaved* traces to guarantee causality in the resulting execution. Unfortunately, these semantics, both in terms of correctly synchronized and incorrectly synchronized programs impose strong constraints on optimization. Below we describe both concerns, beginning with limits due to potential races, and following with limitations due to being race-free.

3.1 Out-of-thin-Air Guarantees

The semantics of programs containing data races is intended to provide significant lattitude in optimization, while still ensuring behaviour is bounded in some reasonable sense. Although the basic happens-before consistency is a simple, easily understood model, a conundrum introduced by the use of happens-before consistency is the potential for *causal cycles* to exist, and in particular to allow the validation of “out-of-thin-air” values. Manson et-al provide an example of such an undesirable situation (Figure 1).

```
Thread 1        Thread 2
r1 = x;         r2 = y;
y = r1;         x = r2;
```

Figure 1. Example of an undesirable out-of-thin-air situation; any value can be justified, e.g., `r1==r2==42` [1].

Unfortunately, causal cycles can also be produced by compiler optimizations, so disallowing all cycles is not acceptable. A complex system for justifying execution traces based on observed writes in previous traces is instead used to eliminate the out-of-thin-air problem, and ensures values read from a variable have a well-defined causal write.

This approach, however, has a major impact on the ability of a compiler to efficiently (re-)use space—-whenever data is stored in a variable, it must comport with out-of-thin-air guarantees, and this affects instances of storage reuse, storage in speculative execution, as well as advanced, higher-level algorithmic optimizations. The code at the top of Figure 2, for example, counts boolean true values in an array. If a previous analyis has shown that most of the values of `x.a[]` are true then an effective optimization would be to invert the calculation, counting down rather than up and reducing the total number of arithmetic operations. The optimized code is shown at the bottom of Figure 2. Note that if n booleans in `a` are true, the observable set of values of `x.f` in the original program are $0 \ldots n$. In the optimized code the observed set of values are `UPPER`$\ldots n$. Both versions converge to the correct value; from an observer's perspective, however, the optimized code contains out-of-thin-air values.

```
                 Original Code
x.f = 0;
for (int i = 0; i < UPPER; ++i) {
   if (x.a[i]) ++x.f;
}
                 Optimized Code
x.f = UPPER;
for (int i = 0; i < UPPER; ++i) {
   if (!x.a[i]) --x.f;
}
```

Figure 2. Reducing arithmetic operations.

The fundamental problem here is that in general compiler optimizations ensure only *functional equivalence* of optimized code. The only guarantee of execution behaviour provided by standard, single-threaded code optimization is that the optimized code will have the same external side effects (output) as the unoptimized code, given the same input. For concurrent programs this is not sufficient, since the behaviour of a single thread of code may depend on the behaviour of other threads.

For racey programs, this interaction becomes even more intricate. The allowance of race conditions in the Java memory model implies that a concurrent thread may at any point perform observations of the state of another thread's computation. The ability of an external thread to observe the internals of a computation means any optimizing transformations must obey the requirements of the observational semantics. Any useful constraint on the external observations of a function computation will necessarily restrict the kinds of functions that can be expressed, and thus the kinds of functional equivalences that may be exploited by an optimization strategy.

3.2 Sequential Consistency and Divergence

An attractive property of the Java memory model is the DRF guarantee, ensuring sequential consistency for race-free, "correctly synchronized" programs. This property is core to many memory models, providing a simple programming model on the condition that the program uses appropriate synchronization to avoid race-conditions.

The difficulty with this approach, however, is that being race-free is not a static property of the code. It is a dynamic one, dependent on the behaviour of a sequentially consistent execution of the same code. Code which under conservative, static analysis could result in a runtime race is not necessarily incorrect, as long as the race could never occur in any sequentially consistent execution. This introduces serious problems for optimization, making the application of code transformations dependent on proving the precise runtime behaviour (a complex and generally unsolvable problem), or being forced to resort to simpler but less effective conservative approximations when such information is not available.

An example is in fact provided by Manson *et al.* to illustrate a known reduction in optimization potential, the assumption that execution continues forward to program exit. In a sequential context the code used by each of the threads in the top part of Figure 3 would be optimized separately. Since the write to x (respectively y) is not control or data dependent on earlier instructions, in each case it could be transformed during optimization into the code shown on the bottom of Figure 3.

```
Thread 1                Thread 2
do {                    do {
   r1 = x;                 r2 = y;
} while(r1==0);         } while(r2==0);
y = 42;                 x = 42;
```

```
Thread 1                Thread 2
y = 42;                 x = 42;
do {                    do {
   r1 = x;                 r2 = y;
} while(r1==0);         } while(r2==0);
```

Figure 3. Above is a program correctly synchronized through divergence. Below, after optimization the program is now incorrectly synchronized [1].

The original version of the program did not contain any data races since neither thread would ever exit its loop. *Divergence* within each thread is used to avoid execution of the actual race condition at runtime, and thus the program is required to retain its sequentially consistent semantics. The transformed version introduces data races and new behaviour (termination), and so is not a valid transformation under the Java memory model.

Manson *et al.* present this as a necessary restriction on Java optimization. The extent of the restriction is, however, quite large. In particular, powerful, well-established optimizations like Partial Redundancy Elimination, trace or other global instruction scheduling, as well as many advanced speculative optimizations, may move expressions through control flow. Under the Java memory model to able to safely apply these optimizations a compiler needs to ensure that data moved out of or into any one control flow is either certainly not shared or if so is not being exposed prematurely.

Accurately detecting all statically conflicting data accesses is difficult, dynamic conflicts even more so, and poses a particular problem for heap-intensive programs where points-to analysis is typically less precise. The problem is magnified for optimization, which now needs to determine potential races based on the intended code motion, and of course maintain or regenerate this information after each transformation.

An essential complexity emerges from this, in that whether for correctly or incorrectly synchronized programs basic optimizations are now dependent on conflict detection. This adds significant additional cost and development complexity, and constrains optimization effectiveness to the accuracy of the underlying conflict detection. Although conflict and race-detection problems are being aggressively examined in the research community [21, 28], and recent work has also begun to develop efficient SC-preservation techniques [29], it is also possible to avoid these problems linguistically, modifying the language so the compiler does not need to rely on conservative approximations in order to determine what is potentially shared or not. In the next section we explore a simple linguistic change to Java that has a large impact in terms of programming paradigm and the availability of information to an optimizing compiler.

4. Allowing Optimization

In a general sense, offering any useful semantics in the presence of (potential) race conditions adds significant complexity to optimization. If only fixed forms of behaviour are allowed then a compiler must provide appropriate analysis to ensure it limits its activities to those that preserve the required behaviour. Through shared variables (races) a thread is allowed uncontrolled observation of another thread's activities, and so to impose a semantics of observability is to impose a restriction on the observed thread's potential (optimized) execution. This is inherently unsatisfying from an optimization perspective, adding extra cost and another source of conservativeness. Having no semantics for racey programs, however, is not an acceptable solution either. Although it greatly relaxes constraints on optimization, it makes race conditions errors that should not occur with correct input, and so care must still be taken in optimization to avoid introducing races through optimizing transformations. In both cases, on top of the complexity relegated to the programmer of ensuring a base race-freedom there is additional, non-trivial compiler cost.

The alternative we explore here is to change the Java semantics in order to remove all possibility of race conditions, as well as clearly identify shared data for compiler consideration. Below we describe a modified semantics for concurrent Java programs based on explicitly declaring all shared variables, and making use of a trivial syntactic extension to the language (more accurately, to the type/modifier system). This prototype design has limitations with respect to backward compatibility for legacy code (containing races), but has attractive features of a simple programming model, a feasible implementation design, and permitting sequential-based compiler optimizations without necessitating expensive conflict analysis as a safety requirement. Additional costs are incurred as well, although here we have the advantage that optimizations can easily start from a certainly correct execution. Nevertheless, note that our presentation here is intended to support a feasibility argument; significant work would still be required to fully, and more formally address the complete range of Java concurrency concerns, as well as investigate optimal implementation designs.

4.1 A Race-Free Execution Model

Our model is intended to supply two main features: a trivial mechanism for a programmer to ensure race-freedom, and an equally trivial but effective method for an optimizing compiler to identify shared as opposed to thread-private data. In Java, C++ and many other multithreading language all heap and global variables are thread-shared by default. For obvious reasons this programming convenience is also a primary source of difficulty in ensuring race-freedom as well as in identifying actual shared variable accesses during optimization.

Our solution begins with a simple change to syntactically identify shared data. Unfortunately, although this improves the ability of a compiler to prove the existence or non-existence of races, it is not necessarily by itself sufficient. Purely static or ahead-of-time analysis may still not be accurate enough, and furthermore this still leaves the problem of defining the semantics of code that contains race conditions. Our proposal is not only to identify shared variables, but to enforce a separation of memory spaces between threads so races are impossible. This is done by dynamically allocating data either globally or in thread-local spaces, and using the type system to manage movement to and from global space. Below we describe first the overall design in detail, followed by discussion of more subtle concerns.

Basic Model

Our changes involve only a few basic principles, and relatively trivial changes to the language, and can be summarized as follows:

- All static and heap variables have thread-specific values by default. Two threads may thus access the same field, storing and loading different values without conflict.
- All shared fields are explicitly tagged with the keyword `volatile`. Such data behaves as current Java volatiles, with all implied visibility, atomicity, and ordering properties.

As an additional, syntactic convenience an object type can be declared volatile, meaning its content (fields) are all volatile by default. We do not define atomicity over whole object access, however, leaving coarser atomicity to existing, lock-based control.

Our proposed model represents a significant internal, semantic change in how data is stored and accessed. Java's *volatile* keyword is a natural choice for specifying shared variables, and so no deep linguistic or syntactic changes are necessary. To maintain the clear identification of shared and unshared data it is also important to control how data can be moved from local to shared memory and/or back, particularly in the case of references (pointers). Several models are available; trivially, assignment from a non-volatile variable to a volatile can be disallowed unless the assigned type is an atomic primitive type (extension to immutable constants such as String or Integer would also be possible, as would copy-out semantics). By this we permit publishing of local data, but only through variables which can be value-copied with atomic operations.

The resulting execution model provides very useful guarantees. Since shared data cannot be accessed except through volatile variables, the program is necessarily race-free. All non-volatile data accesses only thread-specific (local) versions, and shared data is properly protected as volatile. Since access to shared data is statically identifiable through the volatile modifier, compilers can trivially identify the subset of data for which order access and/or potential out-of-thin-air visibility requires more expensive conflict analysis. Of course there are also significant overhead concerns in this basic scheme. Volatile access has additional overhead, and ubiquitous thread-local storage can be expensive as well. These are, however, semantic properties of the code, open to optimization, and Section 4.2 discusses potentially efficient implementation designs. Below we describe first an example to further illustrate the memory separation, followed by a brief discussion of thread initialization and copyin/out concerns.

Example

A small code example and resulting behaviour under our model are shown in Figures 4 and 5 respectively. Interpreted with existing Java semantics (erasing or even propagating the volatile class declaration to its fields), this program has several race conditions and is thus incorrectly synchronized. Using our model of explicitly shared storage, however, the program is race-free and has clear specification of shared versus unshared data.

Two classes are declared, one with a volatile modifier and one without. Three variables are then declared; `v` is a volatile variable of a volatile object, `w` is a normal variable referencing a volatile object, and `a` is a normal variable referencing a normal object. We discuss the case of a a volatile pointer to a normal object following this example.

```
class Q {                          volatile class P {
   volatile Object x;                 Object x;
}                                  }

                       volatile P v;
                       P w;
                       Q a;
Thread 1                           Thread 2
v = new P();                       v = new P();
w = new P();                       w = new P();
a = new Q();                       a = new Q();
v.x = w;                           v.x = w;
w.x = v;                           w.x = v;
a.x = w;                           a.x = w;
```

Figure 4. Example using explicitly shared variables.

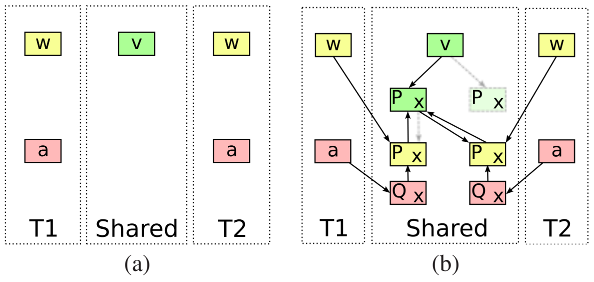

Figure 5. Memory layout prior to the start of execution (a) and one potential state after execution (b) of the code in Figure 4. The surrounding dotted boxes show local and shared memories, and the colours in the solid boxes indicate to which object each memory location logically belongs. Arrows show stored values, and grey/dashed lines show overridden assignments and data allocation.

The program begins with the memory layout shown on the left side of Figure 5. If we (arbitrarily) assume a runtime synchronization order in which Thread 1's atomic write to `v` overwrites Thread 2's, and symmetrically Thread 2's assignment to `v.x` overwrites Thread 1's, then the execution terminates in the state shown on the right of Figure 5. Note that the declaration of `v` as volatile implies the reference itself lives in shared space, while the lack of such a modifier for the declaration of `w` (even though the object is volatile) means each thread has a local copy, with each having field content in the shared space.

In our context the resulting execution is necessarily sequentially consistent. The existence of duplicate, thread-specific data arguably stretches the concept, but each shared variable is declared volatile and although synchronization order allows for several execution variations, ordering the writes of `v` and `v.x`, no race-conditions exist. Importantly, compiler optimizations can also easily recognize shared and unshared data. This is of course a conservative approximation, and not all programs will use all shared data in all threads. With safety as a baseline, however, it is much easier for optimization to guarantee correctness, applying further analysis effort only as required to refine behaviours where it is most profitable.

Shared to Local

The advantage of this design is in the easy separation of local versus shared data. As described above, each thread has access to and may indirectly reference shared data, but without the ability to link shared to local it may not directly access data stored in other threads. Data communication must then be by explicit publishing of data, copying local data into and out of shared structures. This represents a design choice meant to enforce clear and simple thread communication, and of course prevents race-conditions. Syntactic sugar could easily be defined for programmer specified transfer of larger structures, and although deep-copying can be expensive the design is amenable to optimizations similar to those applied to strict pass-by-value or copy-in/copy-out semantic contexts.

The design as described roughly corresponds (memory model considerations aside) to a Java embedding of the OpenMP *private* directive as default, with volatile serving as a *shared* specifier. Copyin/out would allow expression of *firstprivate, lastprivate,* and *threadprivate* models [7]. An alternative approach is to allow shared reference to local data, but ensure that any local access reaches only a thread's own (thread-specific) version of the local data. This could be accomplished through a thread local storage mechanism like the existing Java `ThreadLocal` design (essentially a hash table mapping thread ids to data values), or with greater complexity but more efficiency through the use of segmented memory designs. We discuss implementation concerns in more detail in the next subsection.

Synchronization

Although our design has a large conceptual impact, it is mostly orthogonal to the existing synchronization mechanisms in Java. Locks and condition variables behave as before; since shared data is always volatile, however, synchronized blocks function primarily as atomicity specifications rather than combined with visibility/publishing properties.

Use of volatiles within synchronized blocks represents unnecessary overhead, since in most cases it is likely that locks are used to correctly enforce mutual exclusion. In our design this therefore induces an optimization problem in removing synchronization order requirements on lock-based mutually-exclusion of volatile accesses. Synchronized blocks *may* allow the compiler to package up the effects of multiple volatile accesses in a block, negating the impact of adding memory fences to every volatile access. Again, however, the optimization process can begin from an assumption of race-free behaviour, directly applying optimization resources to the improvement rather than to establishing the basic safety requirement.

4.2 Implementation Concerns

As a trivial transformation, our design would represent significant overhead to existing concurrent programs. Naïvely converting all non-volatile variables to `ThreadLocal` types would be prohibitively expensive, if only for the extra memory management costs. As the default behaviour, however, significant improvements are possible. Conceptually, each non-volatile variable access can be thought of as predicated with the current thread identifier, a context indirection already used in many execution models. Copy-on-write approaches [30] can also be used to support *firstprivate*-like initialization. Further work is of course needed to determine the practical impact on GC and data locality.

Significant optimization is also possible. In many cases unshared variables may not be accessed by multiple threads, negating the need to maintain thread-specific values or mappings. Identification of such situations represents an analysis cost, although once again one that can be applied as part of a performance improving optimization rather than as a safety baseline. For example, C. Lin *et al.* find only 8% of the fences used to guarantee sequential consistency are really needed [31]. This implies significant potential in reducing the cost of volatile access, and thus eliminating much of the associated cost in practice.

5. Conclusions & Future Work

It is well-accepted that concurrent code should avoid data races and use synchronization primitives to ensure correct, understandable behaviour. Enforcing this, however, both for programmers and compilers is non-trivial, and in many cases the additional correctness burden is as or more important than the performance improvement. In this work we show how basic requirements to prevent out-of-thin-air data, and to ensure sequentially consistent semantics pose significant concerns for optimization.

Simple, safe models facilitate easy reasoning by both programmers and compilers. We describe ideas for an approach that guarantees race-free by design, as well as easy identification of shared data, and offers an easily understood conceptual model. Semantic changes are transparent to single-threaded programs, and correctly-synchronized, race-free concurrent programs map directly (modulo conceptually-simple modifications to tag data in synchronized blocks as volatile). As an initial exploration, performance and analysis requirements trade-offs exist of course; an important property of our model, however, is that correct synchronization is guaranteed by construction, readily apparent to both compiler and programmer.

Future work for our design mostly centers around developing a prototype implementation to demonstrate feasibility, examine programmability, and establish optimization requirements. As mentioned in Section 4.2, the use of offset or segmented memory descriptors has potential to simplify thread-specific access costs. At the language level many improvements are possible. Of particular concern is the simplistic requirement for a duplicate (volatile) object type-hierarchy. This may be mitigated by further language modifications or compiler support to allow volatile object constructions without corresponding static declarations.

Acknowledgments

This research was supported by the Natural Science and Engineering Research Council of Canada, and IBM Canada Ltd.

References

[1] Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM (January 2005) 378–391

[2] Saraswat, V.A., Jagadeesan, R., Michael, M., von Praun, C.: A theory of memory models. In: PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM (2007) 161–172

[3] Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., Nakatani, T.: Design and evaluation of dynamic optimizations for a Java just-in-time compiler. ACM Trans. Program. Lang. Syst. **27**(4) (2005) 732–785

[4] Cenciarelli, P., Knapp, A., Sibilio, E.: The Java memory model: Operationally, denotationally, axiomatically. In Nicola, R.D., ed.: 16th European Symposium on Programming, ESOP'07. Volume 4421 of Lecture Notes in Computer Science., Springer (2007) 331–346

[5] Aspinall, D., Ševčík, J.: Java memory model examples: Good, bad and ugly. In: VAMP 2007. (Sep 2007)

[6] Ševčík, J., Aspinall, D.: On validity of program transformations in the Java memory model. In: ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming, Berlin, Heidelberg, Springer-Verlag (2008) 27–51

[7] OpenMP Architecture Review Board: OpenMP application program interface. `http://www.openmp.org/mp-documents/spec30.pdf` (May 2008) Version 3.0.

[8] UPC Consortium: UPC language specifications v1.2. `http://www.gwu.edu/~upc/publications/LBNL-59208.pdf` (May 2005)

[9] Burckhardt, S., Baldassin, A., Leijen, D.: Concurrent programming with revisions and isolation types. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. OOPSLA '10, ACM (2010) 691–707

[10] Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, ACM (2008) 68–78

[11] Pugh, W.: Fixing the Java memory model. In: JAVA '99: Proceedings of the ACM 1999 conference on Java Grande, ACM (1999) 89–98

[12] Gontmakher, A., Schuster, A.: Java consistency: Nonoperational characterizations for Java memory behavior. ACM Trans. Comput. Syst. **18**(4) (2000) 333–386

[13] Aspinall, D., Ševčík, J.: Formalising Java's data race free guarantee. In: TPHOLs'07: Proceedings of the 20th international conference on Theorem proving in higher order logics, Berlin, Heidelberg, Springer-Verlag (2007) 22–37

[14] Adve, S.V., Hill, M.D.: Weak ordering—a new definition. In: ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture, ACM (1990) 2–14

[15] Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer **29** (1996) 66–76

[16] Midkiff, S.P., Lee, J., Padua, D.A.: A compiler for multiple memory models. Concurrency and Computation: Practice & Experience **16**(2-3) (2004) 197–220

[17] Arvind, A., Maessen, J.W.: Memory model = instruction reordering + store atomicity. In: ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture, Washington, DC, USA, IEEE Computer Society (2006) 29–40

[18] Ferreira, R., Feng, X., Shao, Z.: Parameterized memory models and concurrent separation logic. In: ESOP 2010: Proceedings of the 19th European Symposium on Programming. (2010) 267–286

[19] Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: OOPSLA '02, ACM (2002) 211–230

[20] Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for java. ACM Trans. Program. Lang. Syst. **28**(2) (2006) 207–255

[21] Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, ACM (2009) 121–133

[22] Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: a high-performance Java dialect. Concurrency: Practice and Experience **10**(11–13) (September 1998) 825–836 Special Issue: Java for High-performance Network Computing.

[23] Bull, J.M., Kambites, M.E.: JOMP—an OpenMP-like interface for Java. In: JAVA '00: Proceedings of the ACM 2000 conference on Java Grande, ACM (2000) 44–53

[24] Klemm, M., Bezold, M., Veldema, R., Philippsen, M.: JaMP: an implementation of OpenMP for a Java DSM. Concurrency and Computation: Practice & Experience **19**(18) (2007) 2333–2352

[25] Klemm, M., Veldema, R., Bezold, M., Philippsen, M.: A proposal for OpenMP for Java. OpenMP Shared Memory Parallel Programming (2008) 409–421

[26] Bronevetsky, G., de Supinski, B.R.: Complete formal specification of the OpenMP memory model. Int. J. Parallel Program. **35**(4) (2007) 335–392

[27] Basumallik, A., Eigenmann, R.: Incorporation of OpenMP memory consistency into conventional dataflow analysis. In: IWOMP'08: Proceedings of the 4th international conference on OpenMP in a new era of parallelism, Berlin, Heidelberg, Springer-Verlag (2008) 71–82

[28] Flanagan, C., Freund, S.N.: Adversarial memory for detecting destructive races. In: PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, ACM (2010) 244–254

[29] Marino, D., Singh, A., Millstein, T., Musuvathi, M., Narayanasamy, S.: A case for an SC-preserving compiler. In: PLDI '11: Proceedings of the 2011 ACM SIGPLAN conference on Programming language design and implementation. (2011) *to appear*.

[30] Tozawa, A., Tatsubori, M., Onodera, T., Minamide, Y.: Copy-on-write in the PHP language. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (2009) 200–212

[31] Lin, C., Nagarajan, V., Gupta, R.: Efficient sequential consistency using conditional fences. In: Proceedings of the 19th international conference on Parallel architectures and compilation techniques. PACT '10, ACM (2010) 295–306