

Compiler-Based Object Consistency

(Preliminary Version)

Christoph von Praun and Thomas R. Gross
Laboratory for Software Technology
Department Informatik
ETH Zürich
8092 Zürich, Switzerland

Abstract

An application developer rarely interacts with the memory model provided by the bare machine; instead, the model that is created by layering compiler and runtime system over the hardware determines the abstraction for the programmer. Sequential Consistency (SC) is a fundamental memory model for multi-threaded environments. In an object-oriented environment, the notion of consistency extends beyond individual memory locations: Object Consistency (OC) mandates that no intermediary state created during method execution becomes visible, i.e., operations on objects are locally serializable.

In existing systems, optimizations of memory access operations by the compiler and the hardware implementation (like reordering or out-of-order execution) wreak havoc with SC and thus also with OC. This paper describes a solution that provides SC/OC to the programmer while allowing traditional compiler transformations when executing on realistic machines.

The key idea is to separate the aspects of control-flow and memory synchronizations shifting the responsibility to maintain memory and Object Consistency from the programmer to the compiler and (language) runtime system. The integrated system maintains dynamic consistency domains for each thread and synchronizes memory whenever the domain is enlarged or reduced. In addition, accesses violating OC are detected. Since executions always experience a SC memory view, such parallel programs are platform-independent.

The paper contains a preliminary evaluation based on Java, demonstrating that compile-time analysis for typical programs unveils sufficient information about concurrency; this information helps to significantly reduce the amount of memory synchronization and access checking necessary for providing OC.

1. Introduction

The memory abstraction for serial programs is intuitive and simple: A load operation returns the most recently stored value. Mapping the program's view of the data space (references and assignments) to a uniprocessor memory system is simple and direct. However, transferring this abstraction to parallel programs with shared memory raises the issues of access *ordering* and *atomicity* that are defined in a *memory consistency model*. Such a model defines the semantics of the memory abstraction and is the foundation for reasoning about the behavior of a parallel program.

A simple memory model for parallel programs is *Sequential Consistency* (SC): Memory accesses from all threads of a program appear serialized according to a global order. For the resulting serialization, the simple memory abstraction of serial programs applies. Extending the notion of SC to objects means that method executions appear without interruption, i.e., intermediate states of objects are not visible during method execution. This principle is known from Actor-based languages [2, 12] as *Object Consistency* (OC). SC/OC is widely accepted as an intuitive model for the programmer but is hard or expensive to implement in a high-performance system.

Optimizations of memory accesses as performed by the compiler or the hardware implementation influence ordering and atomicity of memory operations. Common compiler optimizations are designed to preserve the memory semantics of serial, not parallel programs: *Reordering* may change how different threads perceive the effects of memory operations. *Caching* can violate atomicity such that not all threads see memory updates at the same time. While these optimizations support execution performance, they complicate the semantics of the memory abstraction presented to the user. There have been two fundamental directions to handle this problem.

First, *weak memory models* provide conditional SC; this means that executions are SC as long as the programmer follows certain conventions regarding the application of control-flow synchronization and access to shared variables. The correctness of programs that follow this approach cannot be verified in general and depends on the adherence to usage conventions. This model places few restrictions on hardware and compiler optimizations.

Another approach evolved in the context of concurrent

object-oriented programming (COOP). Several languages provide abstractions that support [14, 12] or guarantee [5] SC/OC. While this approach is safe for the programmer, the effective overhead relative to sequential programs can be considerable [24]. In addition, approaches that are based on monitors and object-level locking introduce the hazard of deadlock.

Our approach is different from previous approaches: We do not define a new memory model nor a new language. We adhere to SC/OC as programming model and propose an efficient implementation in the context of a standard programming language with support for concurrency. The core idea is to combine compiler analysis and runtime system checks to determine *when* memory must be made consistent and whether object access is admissible. Compiler and hardware optimizations like reordering remain possible for many operations; optimizations that could lead to a violation of SC/OC at the programming level are either omitted by the compiler or are selectively inhibited at the processor level.

2. Related work

This section discusses fundamental approaches at different layers to compensating and hiding the effects of low-level memory access optimizations to the programmer. Hardware-centric SC systems are not included since, due to the combined effects of compiler and runtime system, hardware-based solutions can simplify but not entirely resolve the aspect of memory consistency at the programming level.

Weak memory systems

Weak memory systems [1, 17, 7, 19, 15] constitute a *programmer-centric* [16] approach to hiding the effects of buffering and reordering of memory accesses from the user. Weak memory models are defined from the hardware perspective and thus not specific about programming language aspects. For a certain class of programs where synchronization is present, e.g., according to the conventions imposed by Release Consistency [17], the memory abstraction for the user is SC. Restricting the scope of programs for which guarantees are made is however a limitation of weak memory systems: The property of compliance to the programming model is undecidable and cannot be efficiently determined from program executions either. As a consequence, the correctness of optimizations depends on the program being optimized. Our approach differs from Scope Consistency [19] and Entry Consistency [7], in that programming conventions for these memory models define memory scopes based on critical regions (i.e., program segments), whereas our approach uses a partitioning based on the object space.

Concurrent object systems

Various systems (including Amber [10], Orca [6], SAM [26], and Concert [11]) have exploited the idea of user-defined objects as the basis for sharing. These systems have been successful in mapping parallel programs onto memory systems that are far more complicated than the shared memory systems that are the target of our work and have even targeted distributed memory machines (where explicit copy or transfer operations are necessary to maintain a shared object space). These systems define tailored programming

languages and require system specific annotations by the user, e.g., about available parallelism and how data will be accessed.

Compiler analysis

Lee, Padua and Midkiff [22] investigate constraints on basic compiler optimizations in the presence of access conflicts. In [21], they describe a compile-time algorithm for the sparse placement of memory fence instructions to guarantee SC in explicitly parallel programs with access conflicts. Unlike our approach, they assume complete knowledge about the sharing of variables and the structure of parallelism.

3. Programming model

3.1 Object Consistency

The high-level objective of a model for concurrent programming is to prevent data structures from corruption or inconsistency through concurrent access. We employ a programming model called *Object Consistency* to achieve protection against corruption of objects through concurrent usage. OC can be understood as a simple extension of SC from individual memory cells to objects. The central idea is that the intermediate object state during method execution must not become visible to concurrent threads. Accesses through fields and methods are subject to this model and must be *locally serializable* [12].

3.2 Example

We could use any appropriate language as a base, but for practicality we take sequential Java as foundation. Figure 1 shows the implementation of a deque (adopted from Lea's util.concurrent framework [20]) that tolerates concurrent access.

This implementation assumes that the top of the structure is accessed only from a single owner thread (`push` and `pop`). Hence these methods are not protected against concurrency, whereas access to the base of the structure (`put` and `take`) are explicitly declared mutually exclusive (`synchronized`). If the structure becomes empty, a conflict between `pop` and `take` is resolved through the special case `confirmPop`.

The central idea is to require that a programmer specifies in the declaration of code and data how concurrent overlapping accesses should be treated. The conventions used in our model follow an *object access discipline* that is described in Section 3.3.

3.3 Object access discipline

A simple approach to achieve OC is to define objects as monitors, hence object accesses are serialized and naturally leave the object always in a consistent state. This access policy is however unnecessary restrictive. Instead, only those accesses for which different execution orders leave the object in different final states [12] must be subject to mutual exclusion. Access members of an object can be grouped according to their interference through a conservative data-flow analysis.

Ordinary variables must not be accessed concurrently. This means that if two methods access the same variable in a conflicting way (at least one write), their execution must

```

class Deque {
  static final int CAPACITY = 20;
  private volatile int base_ = 0;
  private volatile int vtop_ = 0;
  private int top_ = 0;
  private final Object[] deq_ = new Object[CAPACITY];

  synchronized Object take() {
    int b = base_++;
    if (b < vtop_)
      return deq_[b];
    else {
      base_ = b;
      return null;
    }
  }
  synchronized void put(Object r) {
    int b = base_ - 1;
    deq_[b] = r;
    base_ = b;
  }
  void push(Object r) {
    deq_[top_] = r;
    vtop_ = ++top_;
  }
  Object pop() {
    Object ret;
    if (base_ + 1 < --top_)
      ret = deq_[top_];
    else
      ret = confirmPop();
    vtop_ = top_;
    return ret;
  }
  private synchronized Object confirmPop() {
    if (base_ <= top_)
      return deq_[top_];
    else {
      top_ = vtop_ = base_ = 0;
      return null;
    }
  }
}

```

Figure 1: Example of a deque that tolerates concurrent access at both ends. The implementation is simplified and omits checking for over- and underflow.

not overlap in a multi-threaded context (field access through references other than `this` are treated like through an accessor method). Programs that do not enforce this policy (e.g., through control-flow synchronization among threads) will be rejected by the runtime system.

Variables eligible for concurrent access must be explicitly declared: For *volatile* fields, the programmer takes the responsibility to orchestrate access, and the runtime system ensures SC; for *final* fields, the compiler takes the responsibility that the first access is a write and subsequent accesses are reads (hence not conflicting). Thus, *final* and *volatile* variables are not considered as intermediary state when determining conflicting members of an object.

Considering the example in Figure 1, this discipline means that methods from the groups `{push,pop}` and `{put,take}` can execute concurrently because the only commonly used variable `deq_` is accessed read-only, and variables `base_` and `vtop_` are declared *volatile* (and thus are not considered as

intermediary state). The situation is different for methods from the same group: Simultaneous invocations of `push` and `pop` are detected as an error because both methods read and modify the intermediary state in variable `top_`. For `put` and `take`, mutual exclusion is ensured by the runtime system and threads are delayed on a conflict.

In the presence of memory access optimizations, control flow synchronization (i.e., delaying a conflicting thread) is however not enough; the visibility of memory updates must also be guaranteed to the delayed thread, and this is done by *memory synchronization*. In Figure 1, the execution of `push` and `pop`, e.g., must ensure that the value of `vtop_` and the updated contents of array `deq_` are globally visible.

Previous programming models have combined the issues of control-flow and memory synchronization, e.g., Java [18]. In such systems, the actual consistency of memory during an execution depends on the skill of the programmer to apply appropriate synchronization. A weak memory model can have severe effects on portability and safety aspects of a programming language, if programs are not properly synchronized.

Thus, we pursue a different approach that *separates the issues of synchronizing control-flow and memory view*: Compiler and runtime automatically ensure access ordering and synchronization of memory, while the user is responsible for the program logic including control-flow synchronization.

4. Compile-time abstractions

4.1 Dynamic consistency domains

Along the execution of a parallel program, we maintain a conceptual partitioning of the shared memory space into disjoint *consistency domains*, one domain for each thread. This is a model which aids the compiler as an abstraction of the situation at runtime. Each domain covers data that the thread is privileged to access (i.e., a thread can access data inside its domain without effecting immediate global visibility of the updates). The data inside a domain are not necessarily up-to-date from the viewpoint of all threads, but are certainly up-to-date for the thread operating on the domain. The partitioning of data into consistency domains is done at the level of fields, not objects: Fields of the same object can belong to distinct consistency domains, if the object access discipline allows for concurrent method execution on an instance (example in Figure 1). Fields that do not belong to any domain are consistent for all threads.

Consistency domains are dynamic, i.e., their coverage changes at runtime. The dynamics stem from the threads' accesses to objects. The object access discipline allows the compiler to identify positions in the program where (1) consistency requirements can be relaxed (e.g., upon entering a method, the consistency domain of the accessing thread is enlarged by the ordinary fields being accessed through 'this'; accesses to these fields can be cached.) or (2) consistency must be explicitly established (e.g., upon method termination, when ordinary fields must be made visible to other threads, i.e., the consistency domain is down-sized). Consistency domains can be down-sized at will without violating the SC/OC model; large consistency domains are however desirable for memory access optimizations.

Dynamic consistency domains suggests to distinguish accesses to locations inside and outside the consistency domain. Section 4.2 describes the concept of *isolation*, which categorizes object accesses by determining the affiliation of objects to consistency domains.

4.2 Isolation

A data access in a program is *isolated* if there is a structural guarantee that accesses from other threads will not lead to a violation of SC. If such guarantee cannot be given, an access is *non-isolated*.

The following language aspects allow us to deduce *structural protection guarantees* of data at compile-time; these guarantees are based on the accessibility, visibility and lifetime of data:

Object consistency: The programming model (Section 3) ensures the absence of concurrent accesses to ordinary instance variables of the current object (access through the `this` reference).

Object locality: If an object *A* is solely reachable through one specific other object *B*, then accesses to variables of *A* are isolated and can be treated like accesses through `this`.

Thread locality: The visibility of stack data and data that can solely be reached through references on the stack is limited to a single thread.

Accesses to variables that are protected by one of the aforementioned structural guarantees are isolated. Section 5.1 briefly describes compile-time analyses that conservatively determine thread- and object locality. Section 5.2 and 5.3 discusses the consequences and optimization opportunities that this compile-time classification of accesses into isolated and non-isolated has on the implementation.

In a simple model, arrays are treated like objects and an array access is treated like an access to volatile variables. This view might lead to inefficiencies for data parallel applications with concurrent access to disjoint regions of a shared array, because all accesses are non-isolated according to the classification schema.

4.3 Example

We use the Dining Philosopher example in Figure 2 to illustrate the notion of dynamic consistency domains and isolation.

For both classes of the example program, members fall into a single group of interfering object accessors. Thus all accesses to objects of class `Philo` and `Table` must follow a monitor-like access discipline. This facilitates the illustration of consistency domains, because entire objects belong to consistency domains.

Figures 3 to 6 illustrate the memory shape and the extent of the consistency domains at different stages during program execution. The figures show only reference variables and object instances on the stack and heap.

```

class Philo extends Thread {
    static final int PHILS = 2;
    static final int ITERS = 10;
    private final int id_;
    private final Table table_;

    public static void main(String args[]) {
        Table t = new Table();
        for (int i=0; i < PHILS; ++i) {
            Philo p = new Philo(i, t);
            p.start();
        }
    }
    Philo(int i, Table t) {
        id_ = i; table_ = t;
    }
    public void run() {
        for (int i=1; i < ITERS; ++i) {
            table_.getForks(id_, (id_ + 1) % PHILS);
            Thread.sleep(Math.random() * 500);
            table_.putForks(id_, (id_ + 1) % PHILS);
            Thread.sleep(Math.random() * 500);
        }
    }
}

class Table {
    private final boolean forks[];

    Table() {
        forks_ = new boolean[Philo.PHILS];
    }
    synchronized void getForks(int i, int j) {
        while(forks_[i] || forks_[j]) wait();
        forks_[i] = forks_[j] = true;
    }
    synchronized void putForks(int i, int j) {
        forks_[i] = forks_[j] = false;
        notify();
    }
}

```

Figure 2: Dining Philosopher application (Philo).

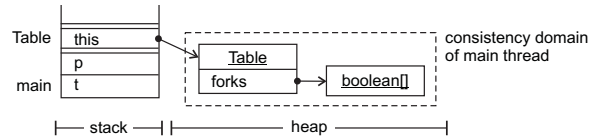


Figure 3: Memory shape and consistency domain with `main` thread at position (1).

In Figure 3, the new `Table` object and the `boolean` array are in the consistency domain of the `main` thread, the only thread in the system at that moment. The `Table` object is in the domain because the constructor has not finished and thus the *monitor access discipline* guarantees that no other thread accesses ordinary fields of the object being constructed. The array object is in the domain, because it is only reachable through the `Table` object, i.e., it is object-local and thus considered as part of the `Table` object.

In Figure 4, execution has processed till the end of the constructor of the first `Philo` object. The `Table` object and its attached array are withdrawn from the domain after the `Table` constructor returns, because both objects become ac-

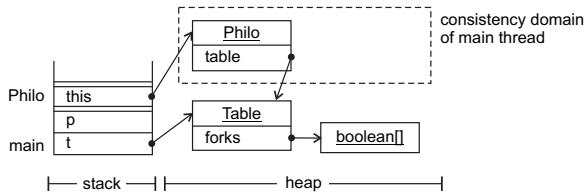


Figure 4: Scenario with main thread at position (2).

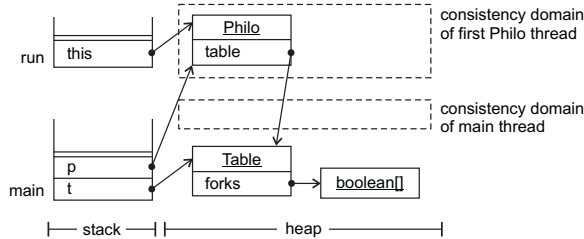


Figure 5: Scenario with main thread at position (3a) and first Philo thread at position (3b).

cessible to other threads. The new `Philo` object is handled similarly after the constructor returns, because it must be consistently viewed by the new thread constituted by itself.

In Figure 5, the newly established thread starts executing the `run` method, thus the `Philo` object is in the consistency domain of the new thread. The domain of the `main` thread remains empty until the constructor of the second `Philo` object is called.

Figure 6 shows the partitioning of the heap after the `main` thread finished and both `Philo` threads are active. During the course of the program, the `Table` object is passed between the consistency domains according to the control flow of the threads. Between activations of the synchronized methods (`getForks`, `putForks`), the `Table` object is outside the consistency domains of both threads.

The example demonstrates that an object's affiliation to a consistency domain is independent from its allocating thread and the overall spread of references to it. The size of a consistency domain correlates with the pulsation of the runtime stack.

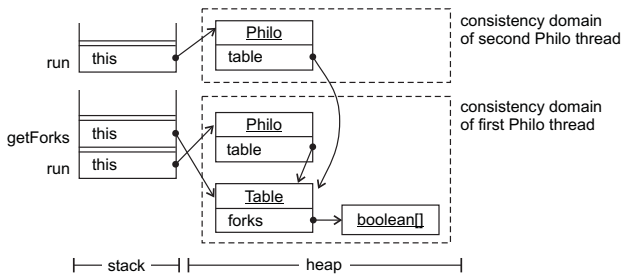


Figure 6: Scenario with `Philo` threads at position (4a) and (4b).

The example does not show objects that are thread-local and thus permanently inside the domain of the allocating thread. Volatility and final variables are not covered by the example either.

5. Implementation

5.1 Determining isolation

Escape information

A fundamental information for the classification of accesses through references is the *escape property* of reference variables. A reference variable is *escaping*, if any statement inside its declaration scope discloses the value, i.e., copies the value to a location that is accessible to other threads. Various procedures have been presented that determine escape information for Java [9, 28, 13]. The fact that a reference variable is non-escaping is a guarantee that all uses of that variable *preserve isolation*. This does not mean that all values this variable takes on at runtime refer to objects that are actually visible to only one thread.

Thread-locality

Reference variables that are guaranteed to refer in any runtime context only to objects visible to one thread are termed *thread-local*. Thread-locality is determined from escape information through a whole program data-flow analysis. Thereby, the data-flow of references is traced from the corresponding object allocation sites. Thread-locality is lost as soon as a reference flows through an escaping variable.

A conservative approximation of thread-locality is obtained for stack variables that are not escaping and do not alias with method parameters, return values, or objects reachable through those. Ruf [25] extended this approach by not only determining confinement of references to the dynamic scope of a method; his analysis also tracks the threads' accesses to escaping objects. Although this approach significantly improves the classification of references as thread-local (for programs that do not create threads, all references are recognized as thread local), we have not implemented this analysis for our evaluation (Section 6).

Object-locality

Similar to thread-locality for stack variables, *object-locality* is a property associated with fields. A field is object-local if all accesses to it do not escape its value. For private fields, e.g., a data-flow analysis on the defining class is sufficient: Fields that only flow through thread-local variables are object-local.

5.2 Memory accesses

Memory access is implemented differently for isolated and non-isolated accesses.

For non-isolated accesses, the target data is conceptually added (before the access) and retracted (after the access) from the consistency domain of the accessing thread. As isolation is a context-dependent property of accesses and not of the data itself, the compiler augments access sites with appropriate memory fence and synchronization instructions: Memory synchronization after the access ensures the global visibility of changes made inside the consistency domain. Depending on the processor architecture, reading of

stale data is prevented automatically through cache coherence; some architectures may however necessitate to reconcile memory before every first access to data inside a consistency domain.

For isolated accesses, no instrumentation is necessary and explicit freedom is granted to the compiler and hardware to apply memory access optimization (caching, reordering).

5.3 Runtime checks

The purpose of runtime checks is to guarantee adherence of an execution to the object access discipline. If a user knows that a certain program meets this discipline (e.g., because the program has been generated by a program generator), maintenance of access information in objects and runtime checks can be omitted.

Runtime checks are inserted by the compiler at non-isolated access sites only. The check requires that an object has additional information associated with the current activity of threads inside its methods. Access checks need to determine the id of the current thread and manage activity information associated with objects through atomic compare and swap operations. For such an implementation, the runtime overhead is comparable to *thin locks* [4]. Bacon et. al. report an overhead of a factor of 1.5 and 3 for locked vs. unlocked object access (field and method access).

6. Assessment

The runtime overhead of programs that follow the SC/OC model stems from memory synchronization and check operations. We give an estimation on the frequency of operations for typical Java benchmarks [27] and the Dining Philosopher example (Figure 2). Most of the applications are single-threaded; this constraint does not limit their significance for our study, because our programming model is compatible with a very general form of control-flow synchronization, and eligibility for sharing is declared independently from the usage context of objects. Thus, the effectiveness of our concepts and compiler analysis is independent of whether objects are actually shared at runtime. Despite the fact that these applications have not been specially designed for the SC/OC programming model, a significant number of heap accesses can be classified as isolated (50-90%), and only a moderate number of accesses is afflicted with runtime checks (3-35%).

6.1 Compile-time aspects

In our implementation, escape information and thread-locality are determined through a compile-time analysis designed by Bogda and Hölzle [9]. The analysis determines conservative escape information for each reference variable based on a compositional data-flow analysis ignoring control-flow. All numbers refer to application code and do not account for code in Java library procedures. The numbers we present result from a preliminary study. More accurate analysis procedures for determining thread- and object-locality could improve the results.

Table 1 lists the number of access sites of fields, arrays and methods. Accesses are classified according to declaration properties of the accessed variable and properties of the reference variables through which the access is performed. At

first, access to final and volatile variables is reported. Access to ordinary variables and methods are categorized into 'static' (class variables and methods), 'this', 'thread-local', 'object-local', 'synchronized', and 'other'. Every access site is reported once, namely in the first category matching from the top.

The effectiveness of the escape analysis, i.e., the total number of thread-local accesses (last row in Table 1), is expressed relative to the overall number of accesses to members that are not 'static' or 'final'. For the mpeg application, Java's jagged array implementation results in many non-isolated accesses to arrays. The problem is partly due to a Java feature (it is impossible to express 'final' array contents), and partly due to a restriction of our escape analysis, which cannot identify thread-locality of objects accessed through more than two levels of indirection.

6.2 Runtime aspects

Table 2 enumerates runtime accesses along the same dimensions that are used to classify access sites at compile-time in Table 1. The frequency of explicit memory synchronization corresponds to the number of method accesses through references of category 'other'. Memory synchronization could be omitted for read-only methods [8] irrespective of the kind of reference through which the method is accessed; we do not consider this optimization here.

Candidates for optimization and reordering are isolated accesses of fields and arrays. The total number of isolated accesses includes 'final' variables and accesses through 'this', 'thread-local' and 'object-local' references. The percentage numbers are reported relative to the total number of array and field accesses.

Runtime checks are inserted before accesses to fields and methods through references of category 'other' (Section 5.3). Static accesses and array accesses of category 'other' are not included, assuming SC treatment of the corresponding variables. The last row of Table 2 lists the total number of runtime checks relative to the overall number of accesses.

All applications demonstrate that field access is predominantly done through 'this' which justifies our model of Object Consistency. Moreover applications like mpeg, raytrace and jess also invoke a significant number of methods through 'this'.

Information about thread-locality also contributes significantly to determining isolation and the reduction of runtime checks. The applications compress and mpeg are exceptions: In compress, data access and manipulation are mainly done through field variables accessed through 'this'; mpeg makes extensive use of arrays, but our analysis procedure in combination with Java language properties fails to determine thread-locality / isolation for those accesses.

The raytrace and db applications particularly benefit from object-locality of small objects and arrays. For db, arrays with index information are typically encapsulated inside objects, yielding object-locality for accesses to those arrays.

variable	reference	compress	jess	raytrace	db	mpeg	Philo
<i>field accesses</i>							
final		3	36	0	0	0	15
volatile		0	0	0	0	0	0
ordinary	static	257	337	247	280	557	0
ordinary	this	377	1209	664	225	1189	0
ordinary	thread-local	0	271	0	7	0	0
ordinary	object-local	0	0	0	0	0	0
ordinary	other	4	317	7	14	179	0
<i>array accesses</i>							
ordinary	thread-local	4	131	84	9	4	0
ordinary	object-local	4	9	45	15	195	4
ordinary	other	70	173	167	11	6150	2
<i>method accesses</i>							
	static	55	461	89	58	91	4
	this	732	1087	865	732	866	4
	thread-local	203	1644	658	312	298	3
	object-local	35	213	29	22	92	0
	synchronized	24	103	26	36	12	2
	other	126	1131	615	164	464	0
<i>total thread-local</i>		207	1915	742	328	302	3
		13.1%	30.5%	23.5%	21.2%	3.2%	20.0%

Table 1: Compile-time classification of object access sites.

variable	reference	compress	jess	raytrace	db	mpeg	Philo
<i>field accesses</i>							
final	any	33.4	6.4	-	-	155.6	253
volatile	any	-	-	-	-	-	0
ordinary	static	-	2.3	-	0.2	0.1	0
ordinary	this	2331.1	154.2	332.4	96.6	824.2	0
ordinary	thread-local	-	2.2	-	0.1	-	0
ordinary	object-local	-	-	-	-	-	0
ordinary	other	-	81.9	-	47.9	119.5	0
<i>array accesses</i>							
ordinary	thread-local	-	1.3	7.5	2.8	-	0
ordinary	object-local	92.3	5.9	33.8	93.4	366.3	40
ordinary	other	558.1	76.6	39.2	-	1264.5	88
<i>total isolated</i>		2456.8	170.0	373.7	192.9	1346.1	293
		81.5%	51.4%	90.5%	80.0%	49.3%	76.9%
<i>method accesses</i>							
	static	-	5.4	0.2	0.1	1.3	80
	this	19.7	9.2	29.2	0.1	41.2	31
	thread-local	-	22.0	71.2	39.2	-	5
	object-local	430.7	2.6	1.5	-	25.1	0
	synchronized	-	2.1	-	47.9	-	40
	other	113.9	65.9	179.5	2.9	42.1	0
<i>total runtime checks</i>		113.9	147.8	179.5	50.8	161.6	0
		3.2%	33.7%	25.8%	15.3%	5.7%	0%

Table 2: Runtime statistics for heap accesses reported as multiples of $\times 10^6$ or '-' if negligible. Numbers for the Philo application are reported as integers.

```

class Safe {
    static Safe x;
    char[] s;
    Safe(String t) { s = t.toCharArray(); }
}

```

Thread 1	Thread 2
x = new Safe("Hello"); println(x.s);	if (x != null) println(x.s);

Figure 7: Safe initialization: No thread will print null or a partly initialized string.

6.3 Initialization safety

An important safety aspect of object initialization is that accesses to fields outside the dynamic scope of a constructor yield the value assigned in the constructor — regardless of the thread which performs the access. In the scenario of Figure 7, we assume that two threads access an object of class `Safe` through a global variable `x`. Both threads read field `s` of that object and should both obtain a reference to the char array “Hello”.

The intuitive behavior of the program is however not self-evident in the presence of access optimizations: The assignment to `x`, `s` and the elements of the character array could be done or perceived out of order, and such a step can cause access to un- or partly initialized data in Thread 2. A detailed discussion of this problem in the context of Java and its memory model can be found in [3, 23].

An execution according to SC/OC guarantees the intuitive semantics and behaves as follows: Variable `x` is initialized in Thread 1 after the constructor executed. Sequential semantics prescribe that Thread 1 will always print the value assigned to `s` inside the constructor. `x` is a global variable and accesses to it are categorized as ‘other’. Thus accesses cannot be isolated, which means that Thread 1 must (1) not reorder accesses to `x` with respect to other accesses and (2) issue an explicit memory synchronization operation after constructor execution. This regimen ensures the visibility of the initialization to Thread 2.

The situation is more subtle if objects escape their constructor. Concurrent access to an object whose constructor has not finished could lead to a violation of initialization safety. The runtime system we propose detects such accesses as a violation of the object access discipline.

7. Conclusion

There are many memory consistency models. Rather than defining another one, we show how a compiler and a cooperating runtime system can provide SC/OC to a program if the programmer is willing to follow a few simple rules: Data that is eligible for concurrent access (and thus must be visible consistently across all threads) must be specially declared (`volatile` or `final`). For ordinary variables, the programmer must ensure that some form of control-flow synchronization among threads (e.g., access in the dynamic scope of `synchronized`) prevents concurrent access. The interesting property of this approach is that violations of these rules are detected: If during an execution, two methods that

entail the possibility of a conflicting access are invoked on a object, a runtime exception is raised. A consequence of this combined compile- and runtime checking is that a sub-optimal description of access concurrency leads to performance loss; an incorrect description leads to runtime exception (but never to platform-dependent non-SC behavior).

Programs that follow SC/OC may require more runtime checking than, e.g., the same program based on the current Java concurrency system. However, the current Java definition is host to subtle flaws (e.g., it allows references to uninitialized data) and does not provide SC. To reduce the number of runtime checks, a compiler can use analyses like escape analysis to unleash memory accesses and prune checks and memory synchronization. Our preliminary evaluation indicates that Java is well-amenable for those analyses and hence the runtime overhead for providing Object Consistency is moderate. In light of the benefits (a simple memory model and the capability to detect conflicting object accesses!) the price seems acceptable.

As multi-processors become more prevalent, multi-threaded programs will gain in importance. A simple model that can form the basis for portability is necessary if we want to see proliferation of sharable portable programs. The approach described here aims to provide a balance between simplicity, performance, and portability.

References

- [1] S. Adve and M. Hill. Weak ordering — A new definition. In *Proc. of the Annual Int’l Symp. on Computer Architecture (ISCA’90), Computer Architecture News*, pages 2–14, June 1990.
- [2] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 22(9):125–141, Sept. 1990.
- [3] D. Bacon, J. Bloch, J. Bogda, C. Click, P. Haahr, D. Lea, T. May, J.-W. Maessen, J. Mitchell, K. Nielsen, and B. Pugh. The “Double-Checked Locking is Broken” Declaration. <http://www.cs.umd.edu/~pugh/java/memoryModel>, 2000.
- [4] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *Proc. of the ACM Conf. on Programming Language Design and Implementation (PLDI’98)*, pages 258–268, Montreal, Canada, June 1998.
- [5] D. Bacon, R. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*, pages 382–400, Oct. 2000.
- [6] H. Bal, A. Tanenbaum, and F. Kaashoek. Orca: A language for distributed programming. *SIGPLAN Notices*, 25(5):17–24, May 1990.
- [7] B. Bershad and M. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, Sept. 1991.

- [8] J. Bogda. Detecting read-only methods in Java. In *Proc. of the ACM Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers (LCR 2000)*, pages 64–69, May 2000.
- [9] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 35–46, Nov. 1999.
- [10] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The amber system: Parallel programming on a network of multiprocessors. In *Proc. of the 12th ACM Symp. on Operating Systems Principles (SOSP)*, pages 147–158, 1989.
- [11] A. Chien and J. Dolby. The Illionis Concert System: A problem-solving environment for irregular applications. In *Proc. of DAGS'94, The Symposium on Parallel Computing and Problem Solving Environments*, 1994.
- [12] A. Chien, U. Reddy, J. Plevyak, and J. Dolby. ICC++ — A C++ dialect for high performance parallel computing. In *2nd Intl. Symp. on Object Technologies for Advanced Software (ISOTAS)*, pages 190–205, Mar. 1996.
- [13] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 1–19. ACM Press, Nov. 1999.
- [14] I. Foster and C. Kesselman. Language constructs and runtime systems for compositional parallel programming. In *Proc. of COMPAR'94*, pages 5–14, 1994.
- [15] G. Gao and V. Sarkar. Location consistency — A new memory model and cache consistency protocol. CAPSL Technical Memo 16, University of Delaware, Department of Electrical and Computer Engineering, Feb. 1998.
- [16] K. Gharachorloo. Retrospective: Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years of the International Symposia on Computer Architecture (ISCA), Selected Papers*, pages 67–70, 1998.
- [17] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the Annual Int'l Symp. on Computer Architecture (ISCA'90), Computer Architecture News*, pages 15–26, June 1990.
- [18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 2nd Edition*. Addison-Wesley, 2000.
- [19] L. Iftode, J. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, June 1996.
- [20] D. Lea. A Java fork/join framework. <http://gee.cs.oswego.edu/dl/papers/fj.pdf>, 2000.
- [21] J. Lee and D. Padua. Hiding relaxed memory consistency with compilers. In *Proc. of The IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 111–122, Oct. 2000.
- [22] J. Lee, D. Padua, and S. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the 1999 ACM Sigplan Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 1–12, May 1999.
- [23] J.-W. Maessen, Arvind, and X. Shen. Improving the Java memory model using CRF. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*, pages 1–12, Oct. 2000.
- [24] J. Plevyak, X. Zhang, and A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proc. of 22nd Symp. on Principles of Programming Languages (POPL'95)*, pages 311–321, 1995.
- [25] E. Ruf. Effective synchronization removal for Java. In *Proc. of the ACM Conf. on Programming Language Design and Implementation (PLDI'00)*, pages 208–218, June 2000.
- [26] D. Scales and M. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proc. of the First Symp. on Operating Systems Design and Implementation (OSDI'94)*, pages 101–114, 1994.
- [27] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. <http://www.spec.org/osg/jvm98>, 1996.
- [28] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 187–206, Nov. 1999.