

# Compiling Multi-Threaded Object-Oriented Programs

(Preliminary Version)

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

## Abstract

*A compiler must take special care when translating and transforming a multi-threaded program: The presence of synchronization operations and the potential of access conflicts have implications on register allocation and instruction scheduling. Moreover, a compiler might want to inhibit hardware assisted instruction reordering through memory fences, to enable a strong memory model on a platform with weakly ordered memory semantics.*

*Unless the language or user asserts that an input program is free from 'access conflicts', the compiler must be aware of the concurrency and the data sharing in a program, to guarantee that the output program complies with a certain (well-defined) program semantics.*

*This paper reports on an approach to compile multi-threaded object-oriented programs sharing data on a global heap. We developed a whole-program analysis that determines compile-time abstractions for threads and objects. Beyond escape information, the analysis determines the use and update actions of threads to objects and their approximated happens-before ordering.*

*This information is used to implement two variants of sparse instrumentation, addressing the problem of access conflict detection in concurrent programs: (1) Object Race Detection checks if accesses to objects follow a locking discipline at runtime. (2) Object Consistency verifies if threads behave such that accesses to objects happen in a serializable order. Overlapping object accesses, which entail the possibility of data corruption, are detected.*

---

Supported, in part, by a grant from Intel Corp., Microprocessor Research Lab (MRL).

## 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 to a uniprocessor memory system is simple and direct. However, transferring this abstraction to parallel programs with a shared memory space 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.

The definition of a memory abstraction at the programming level is difficult for two reasons: (1) First, common hardware architectures optimize (cache and reorder) memory access in a way that cannot be anticipated by the compiler and may be incompatible with the programming level memory model. Such processor-level optimization can be disabled with specific fence instructions, however at a high cost. (2) Second, common compiler optimizations are designed to preserve the memory semantics of serial, not parallel programs [20]. In the presence of a specific programming-level memory model, an optimizer is inhibited in transforming regions of a program where parallelism and shared data accesses may occur at runtime.

Java has been among the first languages that defined a memory model at the language level [18]. The initial definition has been ill-fated [21, 24] and demonstrates that the efficient implementation of a programming-level memory model in the context of (1) and (2) presents new challenges for compiler- and runtime systems.

A compiler that accounts for memory model issues needs information about the sharing of objects and the ordering of accesses to these objects. Section 3.1 briefly describes a static whole program analysis for this purpose.

The system that is discussed in this paper exploits information about object accesses and checks if the execution of a conventional Java program conforms to a specific memory model. The memory model is not enforced by the runtime-system or hardware, but potential violations are detected and reported. The reporting is safe, because executions without reports comply with the demanded memory model. The checker is implemented as a program instrumentation and extension of the runtime system. The specific memory model that is the topic of this paper is *Object Consistency* (Section 2).

## 2 Object Consistency

With sequential consistency, no thread ever sees an inconsistent state of a variable, and object consistency extends this model to objects.

### 2.1 Programming model

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* (OC) [2, 7] 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. Hence this model guarantees *atomicity* for all object accesses to the programmer. Accesses to objects are *sequentially consistent*, i.e., the execution behaves as if there was a global order of accesses compatible with the program order of individual threads.

At runtime, object access can lead to a violation of OC due to insufficient inter-thread synchronization. The goal of the OC checker is to detect and report such *overlapping* object accesses while guaranteeing OC for the remaining accesses that appear to be ordered. Hence the checker is charged with verifying an *object access discipline*. In this model, there is no notion of consistency for arrays and hence array access is not subject to a specific access discipline.

A simple approach to achieve OC is to require that object accesses are serialized and naturally leave the object always in a consistent state. This discipline is unnecessarily restrictive, because according to the definition of OC, only those accesses for which different execution orders leave the object in different final states [7] must be serialized.

We pursue a pragmatic approach and let the programmer explicitly declare members that tolerate concurrent unordered access from multiple threads at runtime (i.e.,

*volatile*) and then check that mutual exclusion is fulfilled for the other members that read or write the internal state of the object.

With this access discipline, all actual violations of OC are detected; in addition, some object accesses with benign overlap also violate the access discipline. Hence there is no *under-reporting*, but there might be some *over-reporting*. Over-reporting is moderate and has not been a problem in our initial experience (Section 4).

To simplify the presentation, we use the terms "violation of OC" instead of "violation of the object access discipline" if the distinction between both is apparent or irrelevant.

### 2.2 Compile-time abstractions

Our goal is to provide object consistency through a combination of compile-time and runtime techniques. The compiler analyzes the program but can direct the runtime system to insert tests wherever the compiler cannot guarantee exclusive access.

#### 2.2.1 Abstract objects and threads

We use the terms *abstract object* and *abstract thread* to refer to the compiler's entities that conservatively approximate runtime entities but do not necessarily correspond to unique runtime instances (see also [26]). The terms *runtime object* and *runtime thread* refer to actual objects and threads that exist when the program is executed. When there is no risk of confusion, we just talk about *objects* and *threads*. Classes and static variables in Java are treated like objects and fields.

#### 2.2.2 Dynamic consistency domains

Along the execution of a parallel program, a conceptual partitioning of the shared memory space into disjoint *consistency domains* is maintained. There is one domain for each thread. This model aids the compiler to abstract from the set of situations that can exist at runtime. Each domain covers the data that a 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 objects: All ordinary variables<sup>1</sup> of an object belong to the same consistency domain.

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

---

<sup>1</sup>"Ordinary variables" refers to fields that are not *volatile* or *final*.

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 OC model; large consistency domains are however desirable for memory access optimizations.

Dynamic consistency domains suggest to distinguish accesses to locations inside and outside the consistency domain. Section 2.2.3 describes the concept of *isolation*, which categorizes object accesses by determining the affiliation of abstract objects to consistency domains.

### 2.2.3 Isolation

An object access in a program is *isolated* if there is a structural guarantee that accesses from other threads will not lead to a violation of the object access discipline. If such a guarantee cannot be given, an access is *not isolated*.

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

*Thread locality*: The visibility of objects that can only be reached through references on the stack is limited to a single thread. Moreover objects that are reachable through global variables but are not accessed from multiple threads are thread-local.

*Immutability*: Access to immutable objects (no writes after initialization/constructor, `this` reference does not escape constructor) cannot violate OC.

*Lock protection*: Accesses to objects that are consistently protected by a common lock cannot violate OC.

*Object consistency*: The programming model (Section 2.1) allows to assume the absence of concurrent access to ordinary instance variables of the current object (access through the `this` reference). Violations are detected.

*Object locality*: If an object  $A$  is solely reachable in the execution scope of methods of some other object  $B$ , then accesses to variables of  $A$  are isolated.

*Control-flow*: An access site  $\mathcal{A}$  to an object  $A$  can be *protected in the control-flow of a thread* if (1) the reference to  $A$  has not escaped along the control-flow up to  $\mathcal{A}$  or (2) other threads cannot run concurrently to the

```
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 1. Dining Philosopher application (Philo).**

execution of  $\mathcal{A}$  (either no other thread has been started or all those that have been started have been joined in the control-flow up to  $\mathcal{A}$ ).

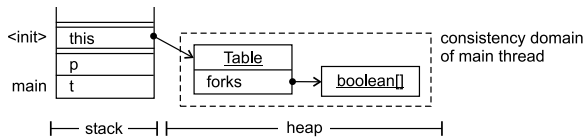
The static whole-program analysis that determines *isolation* for object accesses in concurrent Java programs is briefly described in Section 3.1.

### 2.2.4 Example

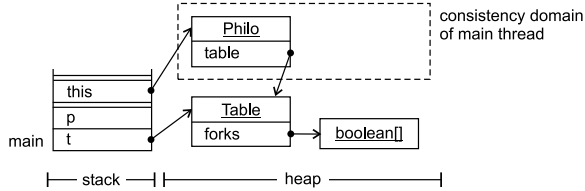
We use the Dining Philosopher example in Figure 1 to illustrate the notion of dynamic consistency domains and isolation.

Figures 2 to 5 illustrate the heap shape graph and the extent of the consistency domains at different stages during program execution. The figures show only reference variables on the stack and object instances on the heap.

In Figure 2, the new `Table` object and the boolean array `forks_` are in the consistency domain of the `main` thread, the only thread in the system at that moment. The `Table`



**Figure 2. Memory shape and consistency domain with `main` thread at position (1).**



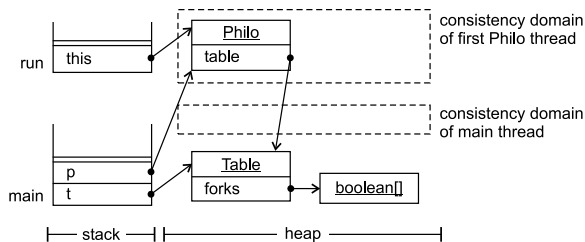
**Figure 3. Scenario with `main` thread at position (2).**

object is in the domain because the constructor has not finished. 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.

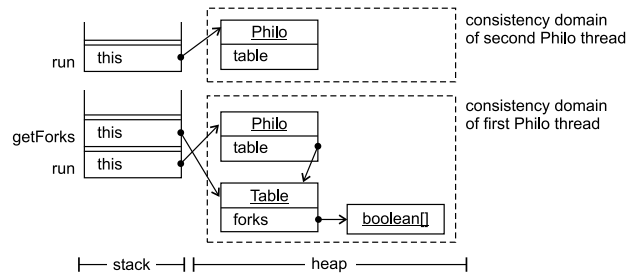
In Figure 3, execution has processed till the end of the constructor of the first `Philo` object. The `Table` object and its attached array are withdrawn from the domain after the `Table` constructor returns, because both objects become accessible to other threads. The new `Philo` object is similarly handled after the constructor returns, because it must be consistently viewed by the new thread constituted by itself.

In Figure 4, 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 5 shows the partitioning of the heap after the



**Figure 4. Scenario with `main` thread at position (3a) and first `Philo` thread at position (3b).**



**Figure 5. Scenario with `Philo` threads at position (4a) and (4b).**

`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 membership in 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 dynamic expansions and contractions of the runtime stack.

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

### 3 Implementation

We have implemented the static analysis and the OC checker in a Java-IA32 way-ahead compilation environment. Our runtime system is based on GNU libgcj 2.96 [12].

#### 3.1 Determining isolation

This section summarizes briefly the steps of the analysis that determines isolation properties of individual access sites. A detailed description of the analysis and the generation of method specializations is given in [26].

1. Determine *abstract threads* and their *call graphs* (see [22]).
2. Determine the *heap shape graph*: Nodes represent abstract objects, edges represent "field-references" relations. The graph contains all abstract objects that are reachable from global variables or thread root objects. The graph approximates the shape of the heap-allocated objects at any time during program execution (see [22]).

3. Determine abstract objects that are *object-local*.
4. Determine statements that can delay the execution with respect to the progress of another thread. A *delaying statement* is one of the following: (a) A monitor operation (b) an access to a volatile variable, (c) a call of `Thread::join` or `Object::wait` or (d) a call to a method that executes a delaying statement.
5. Perform a *symbolic execution of the the abstract threads* and determine for all abstract objects an approximated happens-before ordering of access events<sup>2</sup> (read/write/call) from different threads. Besides access events, also lock protection is recorded. Details about the symbolic execution and the data structure that captures the events for individual abstract objects is described in [26].
6. Determine *access conflicts* among the events that have been recorded during the symbolic execution. Two events are conflicting, if (a) there is no ordering between the events (or the events are in a loop or recursion), (b) the events stem from different abstract threads, (c) at least one event is a write and (d) there is no common unique lock that ensures some ordering at runtime.

A field or method access site is not isolated if the following holds: (a) The variable through which the object is accessed is not `this` nor an access to an object-local object, and (b) the access participates in a conflict.

### 3.2 Instrumentation

The key idea of the OC violation checking is to mark an object upon entering a consistency domain and release the mark as soon as the object leaves the domain. When marking and unmarking an object, the view on this object has to be consistent for all threads. As each thread sets a specific mark, overlapping accesses from different threads — and hence potential violations of OC — are detected.

A non-isolated access is embraced by the following prologue and epilogue sequence that marks and unmarks the target object *o*.

```
// prologue
tmp_before = o.mark;
o.mark = {this_thread, write};
mfence;

<accesses to o>
```

<sup>2</sup>In this context, the term *event* refers to a compile-time abstraction modeling one or several runtime events. Similar to abstract objects and threads, it would be appropriate to say *abstract event*.

```
// epilogue
lfence;
if (write)
    sfence;
tmp_after = o.mark;
o.mark = tmp_before;
if (tmp_after != {this_thread, <any>})
    check(write, tmp_after);
if (tmp_before != null &&
    tmp_before != {this_thread, <any>})
    check(write, tmp_before);
```

A mark consists of the id of the accessing thread, and the read/write behavior of the non-isolated accesses being enclosed. Initially an object has the mark *null*. The variables *tmp\_before* and *tmp\_after* serve to temporarily store marks. If an access overlap is detected, method *check* is called to determine if the overlap is critical (at least one write) and reports accordingly.

The instrumentation is optimized and aligned with the register allocation, such that the typical cost of the prologue is 3 memory accesses, and for the epilogue 3 memory accesses, 2 branches, and 3 arithmetic operations. In addition to handling the mark, prologue and epilogue must take care of memory consistency. In the presence of hardware-based access reordering, critical accesses must not “escape” from their containment between prologue/epilogue. Hence the prologue is followed by a *memory fence*, because the write to *o.mark* must precede subsequent accesses to *o* (reads and writes must be ordered wrt. a write). Similarly, accesses from inside the protected region must not sink below the epilogue. Reads must have performed before the mark *o.mark* is read, hence a *load fence* is placed before the prologue. Writes must have been performed before the update of *o.mark*, otherwise inconsistent state of *o* could become visible to other threads, and the mark/unmark protocol would fail to detect it. If there is no write in the protected region, the *store fence* can be omitted.

### 3.3 Optimization

If the isolated access is only a simple field read- or write, then the runtime overhead of the instrumentation is very high. In many cases, the critical access corresponds to a method invocation, and then the instrumentation at the call site allows the compiler to omit the instrumentation of accesses through the `this` reference inside the called method.

A simple intra-procedural optimization of the instrumentation is to cover more than a single access with one prologue/epilogue pair if accesses are not separated by a delaying statement (see Section 3.1). We use several data flow passes to hoist prologue and sink epilogue statements inside methods that define multiple isolated accesses to the same abstract object.

	philo	elevator	mtrt	sor	tsp	hedc	moldyn	raytracer	montecarlo
<i>program characteristics</i>									
appl loc	81	528	11298	300	706	28299	1402	1972	3674
appl classes	2	5	37	7	4	89	11	19	20
methods in callgraph	135	235	591	143	224	808	165	204	353
abstract threads	2	2	2	3	2	5	2	2	2

**Table 1. Compile-time characteristics.**

## 4 Experience

We use several multi-threaded benchmark programs [25] to evaluate the cost and precision of our program analysis and the overhead of the OC checker. The focus of this paper is on the OC checker and the runtime overhead of the instrumentation. In [26], we give a detailed evaluation of the program analysis. The programs *moldyn*, *raytracer*, *montecarlo* are multi-threaded numeric applications from the Java Grande benchmarks [15]. The programs have been compiled with moderate optimization (method inlining and copy propagation).

Table 1 describes the benchmarks. The lines of code *appl loc* and classes *appl classes* account only for the application, not for the Java library. The number of methods is given in *methods in callgraph* including native and abstract ones.

Row *access characteristic* in Table 2 lists the number of array and field accesses. Some benchmarks exhibit characteristics of scientific codes, and a large fraction of their accesses are accesses to arrays (resp. array elements). The static analysis in combination with the context-sensitive specialization of methods allows the compiler to reduce the number of accesses with instrumentation significantly for all benchmarks (row *instrumented accesses*), especially for those benchmarks that have an object-oriented flavor (*philo*, *elevator*, *mtrt* and *hedc*). Array accesses have not been instrumented.

The optimization of the instrumentation is successful for the benchmarks *moldyn*, *raytracer*, and *montecarlo*. For other benchmarks, the static analysis enabled already significant savings in the number of instrumented accesses. In addition, the typical method size of object-oriented benchmarks like (*philo*, *elevator* and *hedc*) is smaller (even with inlining), such that the intra-procedural optimizer has not enough leeway to combine checks for access sequences.

Table 2 row *OC violations* reports the number of runtime objects where the object access discipline is violated (Section 2.1). The numbers have been collected from a typical run with a non-optimized instrumentation. The reports of different runs vary slightly due to the scheduling dependence of the checker. All incidents can be attributed to benign object access overlap, e.g., in several benchmarks (*sor*, *moldyn*, *raytracer*) multiple threads execute concurrently

on a barrier object.

Table 3 reports the execution times of the benchmarks on an Intel Pentium III 933MHz uniprocessor. *philo*, *elevator* and *hedc* are not included since they are not CPU-bound. The runtime overhead is mainly introduced by the memory fences. In addition, the resolution of polymorphic calls in the specialized methods creates overhead: Instead of a vtable lookup, our implementation chooses the appropriate variant at a polymorphic call site through an *instanceof-cascade*. This is the reason for the slowdown of *mtrt*, despite only 24 runtime checks.

## 5 Limitations

The compile-time model for threads does not account for the execution of code in the dynamic scope of a class initializer. Depending on the scheduling of threads, class initialization may not be attributed to the same thread in all executions. Similarly, finalizer methods are not considered by the static analysis either. Hence OC violations that involve accesses from class initializers or finalizers are not recognized.

We have experienced limitations of the static analysis in the precision of alias and type information, especially in the presence of recursion. This imprecision causes conservatism in the classification of objects and object accesses. The problem could be mitigated if the analysis took hints from the programmer in the form of program annotations into account.

The program analysis (Section 3.1) requires whole-program knowledge and hence Java features like reflection and dynamic class loading are not accommodated.

## 6 Related work

This section discusses fundamental approaches at different layers of a system to compensate and hide the programmer-visible effects of low-level memory access optimizations. Hardware-centric SC systems [10, 13] 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.

	philo	elevator	mtrt	sor	tsp ( $\times 10^6$ )	hedc	moldyn ( $\times 10^6$ )	raytracer ( $\times 10^6$ )	montecarlo
<i>access characteristic</i>									
field accesses	6180	17830	$5.5 \times 10^6$	$150.6 \times 10^6$	686.8	216057	1330.6	3444.2	$91.4 \times 10^6$
array accesses	2700	5649	$1.0 \times 10^6$	$251.0 \times 10^6$	299.0	179319	363.9	107.1	$90.0 \times 10^6$
<i>instrumented accesses</i>									
not optimized	1395	2472	28	100075720	68.4	6288	332.0	3118.9	60031
optimized	232	72	30	518	75.7	6288	105.4	762.0	30029
<i>OC violations</i>									
runtime objects	0	0	1	1	1	1	1	2	0

**Table 2. Counts for accesses to arrays and fields and number of instrumented accesses (no arrays).**

	mtrt	sor	tsp	moldyn	raytracer	montecarlo
<i>no instrumentation</i>						
orig	21.5	3.4	9.5	43.0	48.2	24.1
<i>object consistency checking</i>						
not optimized	23.2 (107%)	11.5 (338%)	16.4 (173%)	51.9 (120%)	323.5 (671%)	32.7 (136%)
optimized	23.5 (109%)	3.4 (100%)	16.4 (173%)	38.9 (90%)	142.3 (295%)	32.5 (135%)

**Table 3. Execution times in seconds and relative overhead of the instrumentation.**

### Weak memory systems

Weak memory systems [1, 11, 4, 14, 8] constitute a *programmer-centric* [9] 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 are 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 [11], 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. The approach presented here differs from Scope Consistency [14] and Entry Consistency [4] 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 [5], Orca [3], SAM [23], and Concert [6]) 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-based memory consistency

Lee et. al. [17] investigate constraints on basic compiler optimizations in the presence of access conflicts. In [16, 19] they describe a compile-time algorithm for the sparse placement of memory fence instructions to guarantee a specific memory behavior in explicitly parallel programs with access conflicts. The work proposes the design of an optimizing compiler that exploits information about variable sharing and memory model of the underlying hardware. [19] does not quantify the cost of the analysis and the runtime impact of the program instrumentation.

Our work is complementary to the study of Lee et. al.. We focus on the identification of shared data and structural protection guarantees. Our goal is to provide a pragmatic memory model with access checking (OC) for Java while limiting the impact of instrumentation on execution performance.

## 7 Concluding remarks

Multi-threaded Java programs pose many interesting challenges for a compiler. One of the interesting aspects is that the language includes a model of parallel programming based on threads and therefore provides the compiler an op-

portunity to get involved in the management of threads. We have presented here one way how the compiler can improve the translation of multi-threaded programs.

In the absence of generally accepted programming model that hides parallelism, semi-explicit parallel programming with the thread model is likely going to be more important in the future. Object consistency is a programming model that extends the well-known model of sequential consistency to the realm of object oriented programs. But whereas sequential consistency is expensive to implement, object consistency can be supported by a combination of compile-time and runtime techniques. The approach presented here exploits object consistency to provide the user with more detailed reports on violations of the programming model. If the scheduler of the host machine – for whatever reason – selects a schedule that exposes an access conflict, then the runtime instrumentation detects this violation of object consistency and alerts the user. Although our approach includes the possibility of overreporting, such overreporting is rare and manageable in the executions of programs investigated so far.

The development of parallel programs is difficult. Better tools are important if multi-threaded parallel programs are to become common. The approach presented here provides a simple approach that is effective and efficient.

## 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] H. Bal, F. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.
- [4] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, Feb. 1993.
- [5] 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.
- [6] A. Chien and J. Dolby. The Illinois Concert System: A problem-solving environment for irregular applications. In *Proc. of DAGS'94, The Symposium on Parallel Computing and Problem Solving Environments*, 1994.
- [7] 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.
- [8] 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.
- [9] K. Gharachorloo. Retrospective: Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years of the International Symposium on Computer Architecture (ISCA), Selected Papers*, pages 67–70, 1998.
- [10] K. Gharachorloo and P. Gibbons. Detecting violations of sequential consistency. In *Symp. on Parallel Algorithms and Architectures, SPAA'91*, pages 316–326, July 1991.
- [11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the Annual Int'l Symp. on Computer Architecture (ISCA'90)*, *Computer Architecture News*, pages 15–26, June 1990.
- [12] GNU Software. gcj - The GNU compiler for the Java programming language. <http://gcc.gnu.org/java>, 2000.
- [13] M. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, 31(8):28–34, Aug. 1998.
- [14] 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.
- [15] Java Grande Forum. Multi-threaded benchmark suite. <http://www.epcc.ed.ac.uk/javagrande/>, 1999.
- [16] 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.
- [17] J. Lee, D. Padua, and S. Midkiff. Basic compiler algorithms for parallel programs. In *Proc. of the*

- [18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1999.
- [19] S. Midkiff, J. Lee, and D. Padua. A compiler for multiple memory models. In *Proc. of the 9th Workshop on Compilers for Parallel Computers (CPC 2001)*, June 2001.
- [20] S. Midkiff and D. Padua. Issues in the optimization of parallel programs. In D. Padua, editor, *Proc. of the Int'l Conference on Parallel Processing*, pages 105–113, Aug. 1990.
- [21] W. Pugh. Fixing the Java memory model. In *Proc. of the Java Grande Conference*, pages 89–98, June 1999.
- [22] E. Ruf. Effective synchronization removal for Java. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI'00)*, pages 208–218, June 2000.
- [23] 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.
- [24] The Java Memory Model. Mailing list and web page. <http://www.cs.umd.edu/~pugh/java/memoryModel>, 2000.
- [25] C. von Praun and T. Gross. Object race detection. In *OOPSLA 2001*, pages 70–82, Oct. 2001.
- [26] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. Technical report, ETH Zurich, Department of Computer Science, Mar. 2002.