

# Data Caching, Garbage Collection, and the Java Memory Model

Wolfgang Puffitsch  
Institute of Computer Engineering  
Vienna University of Technology, Austria  
wpuffits@mail.tuwien.ac.at

## ABSTRACT

Multiprocessors often feature weaker memory models than sequential consistency. Relaxed memory models can be implemented more efficiently and allow more optimizations. The Java memory model is a relaxed memory model that is a natural choice for a Java processor. In this paper, we show that a data cache that obeys the restrictions of the Java memory model can be implemented efficiently. The proposed implementation requires only local actions to ensure data consistency and is therefore timing analysis friendly.

Furthermore, we investigate the impact of the proposed data cache design on the correctness of real-time garbage collection. Relaxed memory models must be taken into account when initializing and moving objects during heap compaction. Also, write barriers and the start of a garbage collection cycle require careful design under the Java memory model.

## Keywords

Garbage Collection, Data Cache, Java Memory Model

## 1. INTRODUCTION

Sequential consistency constrains the possible optimizations and requires complex cache coherence protocols. Therefore, multiprocessors often feature weaker memory models. The Java memory model (JMM) [18, 12] is a memory model that is relatively loose and can be implemented efficiently. Many garbage collection (GC) algorithms are formulated with uniprocessors in mind and do not take into account weak memory models. Some multiprocessor GC algorithms do take into account the requirements of the underlying memory model (e.g., [17, 3, 5]), but these papers provide ad-hoc solutions, tailored to the needs of the algorithm and the underlying hardware.

Common processors implement their own memory model, which may provide stronger guarantees than the JMM. For a Java processor, there is no apparent need to provide a stronger memory model than the JMM. However, the JMM relies on programs being properly synchronized for inter-thread communication; GC acts on a different level, which the application is not aware of. Therefore we investigate the implications of the JMM on GC algorithms.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

JTRES'09 September 23-25, 2009, Madrid, Spain  
Copyright 2009 ACM 978-1-60558-732-5/09/09 ...\$10.00.

Furthermore, we propose an implementation of the JMM in the context of the Java Optimized Processor (JOP) [23, 19]. We show that the JMM can be implemented efficiently while retaining temporal predictability for data caches. This implementation is therefore a good match for hard real-time systems, where worst-case execution time (WCET) analysis is required. The proposed data cache is a write-through cache; it is invalidated upon monitor enter and reads from volatile variables.

This paper is organized as follows: The next section provides an overview about related work. Section 3 provides a short introduction to the JMM. Section 4 describes the cache design and explains why the proposed implementation of the JMM is advantageous for hard real-time systems. The interactions between the JMM and GC algorithms are described in Section 5. Section 6 concludes the paper and provides an outlook on future work.

## 2. RELATED WORK

There are numerous memory models, and even more descriptions of these models. A good introduction to various memory models is provided in [1]; starting from the strongest memory model commonly used, sequential consistency, more relaxed memory models are described.

Doug Lea's "Cookbook for Compiler Writers" [16] provides an overview of how the JMM can be implemented on modern platforms such as the Intel IA-32 architecture [14]. Such architectures usually feature *fence* instructions, which act as memory barriers<sup>1</sup> that can be used to enforce the required ordering. They are however not directly related to higher-level operations that are often used to formally describe memory models. A compiler writer has to map the higher-level constructs to these low-level barrier operations. Our approach is different, because we do not fit the implementation onto a given memory model, but rather try to design the processor's memory system such that it obeys the rules of the JMM.

In [17] and [3], the impact of weak memory models is considered for the respective GC algorithms. They provide details about which parts of the algorithms require sequential consistency. Ordering requirements in the write barrier code are enforced by adding fence instructions. The interaction between the write barrier code and the GC algorithm proposed in [3] requires to run tracing in several phases that are repeated until tracing is done. Furthermore, additional fence instructions are required to correctly handle object updates. Levanoni and Petrank [17] exploit the fact that many processors are sequentially consistent for accesses within the same cache line. Their algorithm thereby avoids synchronization in some cases. The correct start of a GC cycle is ensured by additional handshakes between the mutators and the garbage collector. It is remarkable

<sup>1</sup>Memory barriers are unrelated to barriers in the context of GC.

that the authors did not integrate the proposed solutions into their actual implementations.

Barabash et al. [5] describe the modifications to their algorithm that were necessary to correctly run on processors with relaxed memory models. Synchronization for accessing the shared mark stack is reduced by inserting fence instructions for groups of objects rather than each individual object. The second issue they consider is the fact that object tracing could see objects that are not fully initialized. They suggest to insert a fence instruction immediately after the creation of each object; for small objects, execution of the fence is batched. They note that a future revision of the JMM (i.e., JSR-133, which led to the now current specification) would deprecate parts of their solution.

Cheng [7] considers synchronization issues in the context of a replicating garbage collector. In such a collector it must be ensured that writes to the original object and to the replica object are performed in the same order. Otherwise, updates may leave reachable objects invisible to the GC algorithm.

### 3. THE JAVA MEMORY MODEL

In the case of single-threaded programs, a memory model is not important. As long as the illusion is retained that the program is executed as stated by the application code, any optimization is allowed. This is unfortunately not true for multi-threaded programs, especially when running on multiprocessors. Thread-local optimizations can lead to surprising program behavior, and caching may keep updates of one thread invisible to another thread. A memory model has to specify which program behaviors are allowed w.r.t. memory accesses. Depending on the memory model, programmers can rely on different guarantees. This makes it possible to reason about the correctness of multi-threaded applications.

#### 3.1 Terminology

Memory models can be described from several points of view. While some descriptions (e.g., in [1]) describe the allowed reorderings, other memory models like the JMM describe which writes a read may see. In order to allow for sound reasoning, this section presents the definitions used throughout this paper.

**visible** A value is *visible* to a processor if a read operation by that processor returns that value.

**available** A value is *available* to a processor if a read operation may return that value. A value in main memory is available to all processors, but not necessarily visible to all processors.

**happens-before** A partial order (“ $x$  happens-before  $y$ ” is denoted by  $x \leq_{hb} y$ ) that describes the allowed orderings of memory accesses. For example, a read operation  $r$  must not return the value of a write operation  $w$  if  $r \leq_{hb} w$ .

The JMM [18, 12] describes which values threads may read from memory that is shared between threads. It does so by specifying which write operations may be seen by a read operation. Memory locations on the heap are simply referred to as variables, in contrast to local variables, which are part of the run-time stack. A central definition of the JMM is the *happens-before* relation, which determines which actions have a defined ordering.

Several types of actions are distinguished by the JMM: volatile read, volatile write, non-volatile read, non-volatile write, lock, unlock, special synchronization actions, external actions, and thread divergence actions. Special synchronization actions include starting and stopping of threads. External actions interact with the environment, e.g., by printing a value. Thread divergence actions occur

if a thread is stuck in an infinite loop in which no memory, synchronization, or external actions are executed. For the scope of this paper, the read/write and lock/unlock actions are most important. The volatile read/write, lock/unlock, and the special synchronization actions are referred to as synchronization actions.

Apart from the happens-before order, the JMM uses a *synchronization order*, which is a total order over all synchronization actions and is denoted by  $x \leq_{so} y$ . This order is used to describe the ordering of synchronization actions in an actual execution.

#### 3.2 Happens-before

The JMM uses the following definition for the happens-before partial order:<sup>2</sup>

- (1) If  $x$  and  $y$  are actions of the same thread and  $x$  comes before  $y$  in program order, then  $x \leq_{hb} y$ .
- (2) An unlock action  $u$  on monitor  $m$  happens-before all subsequent lock actions  $l$  on  $m$ , i.e.,  $u(m) \leq_{so} l(m) \Rightarrow u(m) \leq_{hb} l(m)$ .
- (3) A write  $w$  to a volatile variable  $v$  happens-before all subsequent reads  $r$  of  $v$ , i.e.,  $w(v) \leq_{so} r(v) \Rightarrow w(v) \leq_{hb} r(v)$ .
- (4) The happens-before order is transitively closed, i.e., if  $x \leq_{hb} y$  and  $y \leq_{hb} z$  then  $x \leq_{hb} z$ .
- (5) The write of the initial value (zero, false or null) to each variable happens-before the first action of every thread.
- (6) An action that starts a thread happens-before the first action of the started thread.
- (7) The final action in a thread happens-before any action in another thread that detects that the thread has terminated.
- (8) If a thread interrupts a different thread, the interrupt happens-before any point where any other thread detects that the thread was interrupted.
- (9) The end of a constructor happens-before the invocation of the finalizer for an object.

The basic rules for a programmer to understand the JMM are the first four rules in the definition above. The first rule entails that the program order is obeyed. The second and third rule establish an ordering between threads. The fourth rule causes the happens-before order to span across threads in case they synchronize.

The JMM poses two basic restrictions onto the visibility of values. Let  $r$  be a read operation that returns the value written by a write  $w$ . Then it must not be the case that  $r \leq_{hb} w$ . Additionally, there must be no write  $w'$  that writes to the same variable as  $w$  with  $w \leq_{hb} w' \leq_{hb} r$ . These definitions allow that the write a read sees is ambiguous for programs that are not properly synchronized. Proper synchronization however removes such ambiguities.

The synchronization order also implies visibility constraints. Let  $r$  be a volatile read that returns the value written by a write  $w$ . It then must not be the case that  $r \leq_{so} w$  and there must be no write  $w'$  to the same variable with  $w \leq_{so} w' \leq_{so} r$ .

Consider the small Example 1. The initial writes of the default values to  $x$  and  $b$  happen-before the first actions of each thread. The synchronization order entails that the volatile read  $!b$  in T2 is totally ordered with the write  $b=true$  in T1, so either  $!b \leq_{so} b=true$  or  $b=true \leq_{so} !b$ . In the first case, the visibility rules entail that T2 can only see the initial value for  $b$  and is hence stuck in the loop `while(!b) {}`. When the second case becomes true, T2 must not see

<sup>2</sup>The definition is presented in a slightly simplified form here.

the initial write to `b` anymore, because  $b=false \leq_{hb} b=true \leq_{hb} !b$ . Transitivity of the happens-before relation also entails that  $x=0 \leq_{hb} x=1 \leq_{hb} r1=x$ . Therefore, the final read in T2 cannot see the initial write to `x` and `r1` must be 1 at the end of the execution.

```
int x = 0; volatile boolean b = false;
```

Thread T1	Thread T2
<code>x = 1;</code> <code>b = true;</code>	<code>while(!b) {}</code> <code>int r1 = x;</code>

Example 1: Local variable `r1` is guaranteed to see `x` with value 1.<sup>3</sup>

### 3.3 Causality

Using only the happens-before relation to define the JMM is unfortunately not sufficient. Most notably, it allows values to appear out of thin air through causal loops. Consider Example 2: The reads of `x` and `y` in threads T1 and T2 are not ordered with the writes of the other thread through a happens-before relation. Even though a “first cause” is missing, each thread may see the write of the other thread, leading to  $r1 == r2 == 1$ . As this behavior is highly counter-intuitive, the JMM forbids such violations of causality.

```
int x = 0; int y = 0;
```

Thread T1	Thread T2
<code>int r1 = x;</code> <code>if (r1 != 0)</code> <code>y = 1;</code>	<code>int r2 = y;</code> <code>if (r2 != 0)</code> <code>x = 1;</code>

Example 2: Happens-before model allows  $r1 == r2 == 1$ .<sup>3</sup>

The JMM defines causality requirements to validate executions that basically follow a happens-before memory model. They prohibit optimizations that could break causality such as speculative writes while still allowing optimizations such as reordering. Explaining the details of these requirements here would go beyond the scope of this paper. They are defined in terms of committing actions from an execution. If all actions can be committed when following a number of rules, the execution is valid. To show that an implementation follows the rules of the JMM, it has to be shown that the possible executions are valid under the causality requirements.

Despite the causality requirements, the JMM allows some surprising behaviors, as Example 3 shows. The read and write within each thread may not be reordered. However, the write to `x` in T2 is not ordered w. r. t. the read in T1 and vice versa. Therefore it is legal that the threads see each others writes, but none of them sees the initial write.

```
int x = 0;
```

Thread T1	Thread T2
<code>int r1 = x;</code> <code>x = 1;</code>	<code>int r2 = x;</code> <code>x = 2;</code>

Example 3: Java memory model allows  $r1 == 2, r2 == 1$ .<sup>3</sup>

There is evidence [6, 25] that the causality requirements do not fully achieve what is claimed by the JMM. For example, there is a counterexample for the claim that the JMM allows the reordering of independent statements. However, it is unlikely that corrections to repair the causality requirements will impact our cache design.

<sup>3</sup>Example taken from [18].

## 4. DATA CACHE DESIGN

Memory bandwidth is clearly the bottleneck for chip multiprocessors (CMPs). Memory access times are relatively long, especially when considering worst-case behavior. Therefore, caches are necessary to achieve acceptable performance. The Java Optimized Processor (JOP) [23] and its CMP version JopCMP [19] are designed to be timing-analysis friendly. A cache for these processors has to be timing-analysis friendly as well, otherwise one of their major design goals would be voided. Furthermore, these processors are designed to require only little hardware resources, so they can be implemented in low-cost field-programmable gate arrays (FPGAs). As the JMM is a natural memory model for a Java processor, we investigated how the JMM can be implemented efficiently while still being timing-analysis friendly.

### 4.1 Coherence and Consistency

In a multiprocessor system, two cores do not necessarily share the same view of the memory; memory accesses may appear in a different order to different processors. It is necessary to either enforce a consistent ordering in hardware, or to provide means to do so in software.

Reordering of memory accesses can appear at different levels. One level is reordering of accesses by an optimizing compiler. These reorderings do not change the intra-thread semantics, but other threads obviously observe a different ordering than specified by the program code. A second level of reordering appears in many modern processors where memory accesses can bypass each other. For example, while a read operation waits for the result from main memory, values can be written to the cache. A third, more subtle form of reordering can occur in the presence of caches without a coherence protocol. If a core issues two write operations, they can appear to a different core in reversed order if one of the values is present in the cache and becomes visible only after it has been evicted from the cache, while the other value is read from main memory.

Example 4 shows how a cache can cause writes to appear in reversed order. On the right-hand side of the column for each thread the values for the fields `A.f` and `B.f` are displayed. The two leftmost columns show which values are available for `A.f` and `B.f`, i.e., which values are in main memory. Thread T2 first reads `A.f`, which is consequently cached. Thread T1 writes to `A.f` and `B.f`. Thread T2 now reads `B.f` and `A.f`. While it reads 1 for `B.f`, `A.f` still appears to have a value of 0. Only after T2 has invalidated its cache (or the respective cache line has been evicted) it sees the value 1 for `A.f`. The writes to `A.f` and `B.f` therefore appear to T2 in reversed order.

The definition of the *happens-before* relationship between memory accesses in the JMM is very similar to the definitions for *lazy release consistency* [15]. This memory model has some properties that make it a good match for a time-predictable multiprocessor. In the following, we will explain the concept of lazy release consistency and why our design is consistent with the JMM.

### 4.2 Lazy Release Consistency

Lazy release consistency [15] divides memory accesses into ordinary accesses (*read*, *write*) and special accesses, where especially *acquire* and *release* are important. These accesses relate to acquiring and releasing a lock. In a JVM, they appear as *monitorenter* and *monitorexit* and when accessing volatile variables. Memory updates from other processors are guaranteed to become visible to a processor when it performs an *acquire*. It is guaranteed that all memory updates until the last *release* on the same location (lock) become visible upon that *acquire*.

Step	Thread T1	A.f	B.f	Thread T2	A.f	B.f	global A.f	global B.f
1:		0	0	r1 = A.f;	0	0	0	0
2:	A.f = 1;	1	0		0	0	1	0
3:	B.f = 1;	1	1		0	1	1	1
4:		1	1	r2 = B.f;	0	1	1	1
5:		1	1	r3 = A.f;	0	1	1	1
5:		1	1	invalidateCache();	1	1	1	1
5:		1	1	r4 = A.f;	1	1	1	1

Example 4: The writes to A.f and B.f appear in reversed order to Thread T2.

The primary work on lazy release consistency considered it for software distributed shared memory, which has different trade-offs than shared memory for CMPs. The original definitions for lazy release consistency include considerations for paging and a protocol to propagate updates. Our implementation can be considered as coarse-grain implementation of the original lazy release consistency protocol. Most notably, our implementation does not determine which writes actually happened and always assumes that the whole memory has been updated.

### 4.3 Consistency Implementation

We consider a write-through cache; the processor stalls until the data is actually written to main memory. Furthermore, reads stall the processor until a value is returned. Reads and writes of a processor cannot overlap. Memory arbitration guarantees that all writes to main memory are ordered. A processor may see some stale value, but it cannot “go back in time”. Once it sees a memory update, it can never see a memory update on the same location that happened earlier. According to the definitions in [11], the processor is *cache coherent*.<sup>4</sup> However, there is no global store order, which would imply that writes to different memory locations are seen consistently by all processors. Example 4 shows how caching can violate a global store order although retaining the order for writes to an individual field.

The write-through cache trivially ensures that all writes are available (though maybe not yet visible) to other processors after a *release*. Therefore, all memory updates become visible to a processor when invalidating its own cache upon an *acquire*. Invalidating the cache upon an *acquire* is the only action that needs to be taken to implement lazy release consistency in the proposed infrastructure. The invalidation is part of the implementation of the monitor-enter bytecode. For read accesses to volatile variables, we chose to insert a special invalidate bytecode, instead of introducing special bytecodes for each possible read access to a volatile variable. This simplifies the implementation without impairing the principal concept. Potential hazards from thread preemption between invalidation and the actual read can be avoided by invalidating the cache before resuming a thread. Cache invalidation is always a local action, which does not depend on the behavior of other cores.

The actions discussed above are not the only sources of happens-before relations. However, synchronization actions such as starting a thread are not directly visible to the hardware. Typically, the implementation of these actions implicitly establishes the required ordering. Where this is not the case, the invalidate bytecode can be used to enforce the appropriate ordering.

Lazy release consistency uses a happens-before relation that is consistent with the happens-before relation of the JMM. As the proposed consistency implementation is compliant to lazy release

<sup>4</sup>In informal contexts the term “cache coherence” is often used to describe whether inconsistent views of the memory are allowed. This usage differs considerably from some definitions of this term in the memory model literature.

consistency, it is also consistent with the JMM. The JMM causality requirements prohibit optimizations that could violate causality. Our hardware executes memory accesses strictly in-order and does not speculatively execute any accesses. Therefore, the proposed implementation cannot violate causality.

Invalidating the whole cache upon monitor-enter and volatile reads is costly, compared to other consistency implementations. As volatile reads usually vastly outnumber writes, a scheme that enforces consistency upon writes could be more efficient. However, such schemes provide only poor temporal predictability. The solution presented in this section enables temporal predictability, while still providing the benefits of data caching.

A potential optimization to the proposed consistency implementation is to check whether a cache invalidation is actually necessary. If no other processor performed a release since a processor performed its most recent acquire, the cache invalidation can be omitted. The hardware to enable such checks would require relatively few resources. However, this is an optimization for the average case rather than for the worst-case performance. As we are mainly interested in worst-case performance, we did not implement this optimization.

### 4.4 Analysis Friendliness

For WCET analysis, write-through caches are considerably easier to handle than write-back caches. Write-back caches require the analysis also to model the write buffer of the cache. This in turn introduces a whole new dimension into the analysis, because the analysis has to consider which words already have been written back. While this new dimension requires the analysis to be aware of the timing, the analysis for a write-through cache only requires knowledge about the possible memory access traces. The timing information can be delegated to the actual WCET computation.

The consistency implementation only requires local actions. Therefore, WCET analysis is aware of these actions without making assumptions about the behavior of other threads. This helps to minimize the pessimism for WCET bounds. Furthermore, no hardware cache coherence protocol is required, which would increase the indeterminism for memory access times. To solve the issue of unpredictable memory arbitration, a time-predictable memory arbiter [19] can be used.

As shown in [24], the ideal heap data cache for analysis should be fully associative. This is due to the fact that an access to an unknown address affects all cache sets in the analysis. Especially for a data cache, it is not always predictable which addresses are accessed — the addresses of heap allocated data are unknown until the data has been allocated.

Furthermore, the ideal replacement strategy for an analyzable cache is *least recently used* (LRU) [13, 21]. For all other replacement strategies, only a fraction of the cache can be analyzed. The downside of this is that LRU is complex to implement for highly associative caches, which is one of the reasons why it is hardly used in mainstream processors for general caches.

One way to avoid the burden of a large, fully associative cache is to split data caches for different memory areas [24]. While a small, fully associative cache with LRU replacement is used for heap-allocated data, other memory areas may use a simpler cache, e.g., a direct-mapped cache. Split caches also provide the advantage that only caches that contain shared data need to be invalidated. Caches that contain read-only data, such as the constant pool, do not need to be invalidated, because they are implicitly cache coherent.

A split data cache following the considerations in this section has been prototyped in an FPGA and integrated into JOP and JopCMP. Evaluating the cache with respect to its resource consumption, performance impact, and worst-case behavior is ongoing work.

## 5. GARBAGE COLLECTION

While the previous section considered whether the memory model complies to the requirements on the Java language level, the question remains what it implies for the design of the underlying JVM. At the Java language level, synchronization needs to be symmetric: unless a program is properly synchronized, only few guarantees can be made. A processor that implements only what is necessary to comply with the JMM can trouble garbage collectors. Application threads interact with the garbage collector through barriers. For efficiency reasons, GC algorithms are designed to require as little interaction as possible. State-of-the-art real-time garbage collectors [4, 26, 22, 2] require interaction at writes to reference fields and often some small overhead for other memory accesses, e.g., to follow forwarding pointers. Reads from objects typically do not require any synchronization. The synchronization therefore becomes asymmetric and the garbage collector actively has to ensure that its actions are seen by the mutator threads.

For a runtime-system in general, a second issue arises: the runtime-system acts on a lower level than the application code, but needs to be consistent with the JMM. Starting a thread, for example, infers a happens-before edge in the JMM. The runtime code has to take this into account and synchronization at the runtime- and application-level must be coordinated.

For our considerations, we use the cache design described in Section 4 as a basis. The cache is a write-through cache, which is invalidated upon monitor enter and reads from volatile variables. Furthermore, memory accesses of a processor cannot overlap. Where appropriate we also describe alternative solutions and issues for other implementations.

### 5.1 GC Algorithm

In this section, we present a rather generic GC algorithm, which we assume for further reasoning. While details for specific implementations may of course vary, the listings represent the basic approach of common real-time GC algorithms.

We use Dijkstra’s tricolor abstraction [8] to classify the state of objects during garbage collection. *Black* objects have been visited by the garbage collector and do not need to be visited again. Objects are *gray* if the garbage collector is aware that they need to be visited during object graph tracing. *White* objects are unvisited and considered garbage after object graph tracing has finished.

Listing 1 shows the basic top-level algorithm for a garbage collector. In the first step, `startCycle()`, a new GC cycle is initiated. Usually, this includes exchanging the meaning of black and white, so all objects that were marked in the previous cycle become unmarked. Some algorithms also include a handshake with the mutators such that they are aware of the start of the GC cycle. In `gatherRoots()`, the references in local and static variables are collected. The garbage collector may scan the stacks of the mutators by itself, or it may delegate this to the mutators. The latter solution requires

```
void runGC() {
    // initiate new GC cycle
    startCycle();
    // retrieve roots
    gatherRoots();
    // trace the object graph
    traceObjectGraph();
    // clear objects that are still white
    sweepUnusedObjects();
    // optional memory defragmentation
    defragment();
}
```

Listing 1: GC cycle

```
void traceObjectGraph() {
    // while there are still gray objects
    while (!grayObjects.isEmpty()) {
        // get a gray object
        Object obj = grayObjects.removeFirst();
        // iterate over all reference fields
        for (Field f in getRefFields(obj)) {
            Object fieldVal = getField(obj, f);
            // mark referenced objects as needed
            if (color(fieldVal) == white) {
                markGray(fieldVal);
            }
        }
        // mark object black
        markBlack(obj);
    }
}
```

Listing 2: Object Graph tracing

```
void putFieldRef(Object obj, Field f, Object newVal) {
    // snapshot-at-beginning barrier
    Object oldVal = getField(obj, f);
    if (color(oldVal) == white) {
        markGray(oldVal);
    }
    // write new value to field
    putField(obj, f, newVal);
}
```

Listing 3: Snapshot-at-beginning write barrier

a handshake to ensure the completeness of the roots. After the roots have been gathered, the object graph is traced and all reachable objects are marked. In `sweepUnusedObjects()`, the objects that have not been marked are recycled. The final `defragment()` step is optional and not needed for copying collectors.

A generic tracing algorithm is shown in Listing 2. While there are any gray objects left, an object is taken from the set of gray objects. The algorithm then iterates through all reference fields of this object and marks all white objects gray. The object itself is marked black afterwards. For a copying garbage collector, `markBlack()` would also include copying the object.

Listing 3 shows pseudo-code for a snapshot-at-beginning write barrier [27]. First, the old value of the field to be overwritten is read. If the reference to be overwritten is white, it is marked gray and added to the set of gray objects via `markGray()`. Afterwards, the actual update takes place.

We assume that `markGray()`, `markBlack()`, `color()` and all operations regarding `grayObjects` are properly synchronized, albeit not necessarily following the JMM. For more detailed consideration regarding the internal correctness of GC algorithms, see Section 5.7.

The `getField()` and `putField()` methods follow the semantics of the respective bytecodes.

## 5.2 Tracing the Object Graph

Threads may have different views of the heap. A garbage collector must take this into account, so it can correctly trace the object graph without leaving any reachable objects unmarked.

**DEFINITION 1.** Object graph roots: *References in local and static variables (which are directly accessible to mutator threads) are called roots.*

**DEFINITION 2.** Correctness of tracing: *A garbage collector correctly traces the object graph if at the end of the tracing no less than all reachable objects are marked black, assuming that all roots were marked gray at the start of tracing.*

**DEFINITION 3.** Consensus: *A consensus is a state of the object graph where each processor core and each thread has the same view of the object graph.*

When starting from a consensus and inhibiting modifications of the object graph by stopping all mutator threads, a tracing algorithm as given in Listing 2 is trivially correct. In such a case, the algorithm visits exactly those objects that are reachable from the root set. The situation however becomes more complex if mutator threads are allowed to modify the object graph during tracing.

**THEOREM 1.** *The object graph can be traced correctly if*

- (a) *a snapshot-at-beginning write barrier is used, and*
- (b) *new objects are allocated non-white, and*
- (c) *a consensus is established at the beginning of tracing*

**PROOF.** When starting from a consensus, any differences in the object graph views of threads (including GC threads) must stem from modifications of the object graph. Any such update triggers the write barrier. Within the write barrier, the current value is read, marked grey, and afterwards replaced by the new value. This ordering within the write barrier is enforced by the JMM (executions must obey program order).

However, the JMM allows the situation that two concurrent field updates see the updated values of each other, but none of them sees the original value (see Example 3). In the proposed cache design, memory accesses of a processor cannot overlap. The update is not issued before the read of the old reference has returned a value. Therefore, the proposed cache guarantees that one of the threads has to see the value in the consensus. Consequently, objects that are reachable in the consensus remain visible to the garbage collector.

When storing a reference, the referenced object must be either already reachable and hence exist in the snapshot of the consensus, or it must have been newly allocated. Newly allocated objects are non-white and are thus not garbage collected in the ongoing GC cycle. All reachable objects are therefore visible to the garbage collector. □

The preconditions for Theorem 1 are not unreasonable to achieve. Most garbage collectors that are written for stronger memory models will perform this GC phase correctly also on weaker memory models than sequential consistency, as long as situations as shown in Example 3 are precluded.

```
void putFieldRef(Object obj, Field f, Object newVal) {
    // get meaning of "white" once
    Color w = white;
    // mark old value
    Object oldVal = getField(obj, f);
    if (color(oldVal) == w) {
        markGray(oldVal);
    }
    // mark new value
    if (color(newVal) == w) {
        markGray(newVal);
    }
    // write new value to field
    putField(obj, f, newVal);
}
```

Listing 4: Double barrier

## 5.3 Sliding Consensus

While conditions (a) and (b) of Theorem 1 can be easily fulfilled, establishing a consensus can provide a challenge. A straightforward way to create a consensus is by invalidating all caches at once. Unfortunately, this creates timely indeterminism for the application threads. However, such an invalidation is only necessary once per GC cycle, and the indeterminism can be taken into account without introducing severe pessimism. Still, using an algorithm that is similar to *sliding view* root scanning [10, 17, 20], would remove the need for the hardware overhead to invalidate the caches all at once.

Sliding view root scanning is a technique where each thread scans its own stack, instead of being stopped while the garbage collector scans its stack. By doing so, root scanning can be scheduled to take place when there is little data to scan and no blocking of threads is necessary. We propose that each core invalidates its own cache before scanning its local root set. By doing so, a *sliding consensus* is created.

To ensure the integrity of GC during the phase where some stacks are scanned while others stacks are not, a *double barrier* is used. Such a double barrier marks both the reference that is overwritten and the new reference. It has been shown in [20], that it is also possible to only use a snapshot-at-beginning barrier, if objects are allocated grey and the write barrier is atomic. In the following, we consider a double barrier, because it simplifies reasoning for the scope of this paper.

Listing 4 show pseudo-code for a double barrier. We assume here that the start of a GC cycle flips the meaning of black and white instead of re-coloring individual objects. If white is read immediately before the start of a GC cycle (when all reachable objects are black), neither the old nor the new value are shaded. If white is read immediately after the start of a GC cycle, both values are marked gray. More precisely, we say that a field update started *before* the start of a GC cycle, if it sees the old notion of white, and that it started *after* the start of a GC cycle if it sees the updated notion.

Even without inconsistent views of the heap, it is possible to leave reachable objects invisible to the garbage collector in some cases. This is demonstrated by Example 5; a similar example was presented by Doligez and Gonthier [9]. Thread T1 writes Y to field A.f. As at the end of the GC cycle all objects are black, no object is marked. Thread T2 then writes Z to A.f; after it has checked whether anything should be marked, a new GC cycle is initiated. Thread T1 then overwrites A.f and marks Y. Before T1 actually writes to A.f, T2 executes the update of A.f, which started before the new GC cycle was initiated. Thread T3, which already has scanned its root set, reads Z from A.f. Thread T1 now actually updates A.f. Thread

Step	Thread T1	A.f	Thread T2	A.f	Thread T3	A.f	global A.f
1:	A.f = Y;	X		X		X	X
2:	// A.f == white?;	X		X		X	X
3:	// A.f ← Y;	Y	A.f = Z;	Y		Y	Y
4:		Y	// A.f == white?;	Y		Y	Y
start GC cycle							
5:	A.f = null;	Y		Y	scanRoots();	Y	Y
6:	// A.f == white?;	Y		Y		Y	Y
7:	// mark(Y);	Y		Y		Y	Y
8:		Z	// A.f ← Z;	Z		Z	Z
9:		Z		Z	r1 = A.f;	Z	Z
10:	// A.f ← null;	null		null	// r1 == Z;	null	null

Example 5: Overlap of update and write barrier leaves object invisible.

Step	Thread T1	A.f	Thread T2	A.f	global A.f
1:		X	// r1 == Y	X	X
2:		X	A.f = r1;	Y	Y
3:		X	// A.f ← Y;	Y	Y
4:		X	r1 = null;	Y	Y
start GC cycle					
5:		X	scanRoots();	Y	Y
6:	A.f = Z;	X	r1 = A.f;	Y	Y
7:	// mark(X);	X	// r1 == Y	Y	Y
8:	// mark(Z);	X		Y	Y
9:	// A.f ← Z;	Z		Y	Z
10:	scanRoots();	Z	invalidateCache();	Z	Z

Example 6: Object Y is reachable from T2 but remains invisible to the GC.

Step	Thread T1	A.f	Thread T2	A.f	global A.f
1:	// r1 == X		// r2 == Y		
2:	A.f = r1;	X	A.f = r2;	Y	
3:	// A.f ← X;	X	// A.f ← Y;	Y	X
4:	r1 = null;	Y	r2 = null;	X	X
start GC cycle					
5:		Y	scanRoots();	X	X
6:	A.f = Z;	Y	r2 = A.f;	X	X
7:	// mark(Y);	Y	// r2 == X	X	X
8:	// mark(Z);	Y		X	X
9:	// A.f ← Z;	Z		X	Z
10:	scanRoots();	Z	invalidateCache();	Z	Z

Example 7: Object X is reachable from T2 but remains invisible to the garbage collector.

T3 has a local variable pointing to Z, which is however not visible to the garbage collector. Obviously, such behavior is not acceptable.

We identified the following requirements for the correctness of the GC algorithm:

LEMMA 1. *The correctness of object graph tracing can be ensured if*

- (a) *Field updates that started before the start of a GC cycle must be visible to write barriers for the same field that start after the start of this GC cycle, unless they already have been marked by such a barrier.*
- (b) *All field updates that started before the start of a GC cycle must be visible for local root scanning, unless such an update already has been marked by a write barrier that started after the start of this GC cycle.*

- (c) *Field updates that started before the start of a GC cycle must be perceived consistently by write barriers and local root scanning.*

Example 6 shows that the respective write must be visible as stated by Condition (a) of Lemma 1. Thread T2 writes Y to field A.f; although the write is made available to all threads, it is not visible to Thread T1. Thread T2 clears the local variable pointing to Y. A new GC cycle is initiated, and thread T2 scans its local root set. Thread T1 overwrites A.f with Z; as it sees A.f to point to X, it marks both X and Z, but not Y. Before Z is actually written, thread T2 reads Y from A.f. Finally, thread T1 scans its root set, and T2 (not necessarily on purpose) invalidates its cache. Y is now reachable from the local root set of T2, but not visible to the GC. Lemma 1 is violated, because the write of Y by T2 is not visible to T1 when its update of A.f begins.

Lemma 1 also requires that updates that started before the start of a GC cycle are perceived consistently by write barriers and lo-

cal root scanning. Example 7 shows a case where the updates are visible to another processor, but not seen consistently. Thread T1 writes X to field A.f, Thread T2 writes Y to the same field. As these writes are not ordered, one of the writes “wins” the race to main memory, but a cache coherence protocol may make the write of the other thread visible. A new GC cycle is initiated, and thread T2 scans its local root set. Thread T1 overwrites A.f with Z; as it sees A.f to point to Y, it marks both Y and Z, but not X. Before Z is actually written, thread T2 reads X from A.f. Finally, thread T1 scans its root set, and T2 (not necessarily on purpose) invalidates its cache. X is now reachable from the local root set of T2, but not visible to the GC. As there is no ordering between the writes of X and Y to A.f, such behavior is legal under the JMM, but of course it voids the correctness of GC. Although such behavior cannot occur in the cache design as described in Section 4, future optimizations could enable this. A general GC algorithm therefore has to take into account such behavior.

We justify (though not strictly prove) the correctness of Lemma 1 with the following considerations:

- If a processor adds a reference to its local root set before root scanning, one of the following must be true:
  - The reference is still contained in the local root set at the time of root scanning. In that case, root scanning obviously makes that reference visible to the garbage collector.
  - The reference is stored to an object field and removed from the local root set afterwards. In that case, a write barrier is executed, which marks both the old and the new value. If the old value was written by an update that started before the start of the GC cycle, the value must be seen consistently. Otherwise, the overwritten value must have been written by an update that executed a write barrier and is thus visible to the garbage collector.
  - The reference is not written to any object and not contained in the local root set any more. In that case, the reference is unreachable and hence irrelevant.
- If a processor adds a reference to its local root set after it has scanned its root set, the reference must already be reachable through the local root set or newly allocated and hence visible to the garbage collector. As root scanning invalidates the cache, any divergence between available and visible values must stem from an update, which in turn has made the reference visible to the garbage collector through the write barrier.

One consequence of Lemma 1 is that threads should not be preempted while executing a write barrier. Otherwise, preempted threads would have to be resumed before garbage collection can continue. Depending on the scheduler implementation, preemption during execution of the write barrier may be avoided by making the write barrier non-interruptible or by introducing “safe points”. Threads are only be preempted at such safe points; correct placement of these safe points ensures that threads are only preempted at appropriate points of the execution.

Ensuring that updates that started before the start of the GC cycle are visible is only necessary upon root scanning and field updates. For root scanning, the cost is small, because it is executed only once per GC cycle. Visibility in the write barrier can be achieved by bypassing or invalidating the cache. The overhead for this is greater, because it appears for each write barrier. Depending on the

general overhead for the write barrier code, this overhead may or may not be acceptable. For the current write barrier implementation in JOP, the overhead seems to be small enough not to notably affect performance.

## 5.4 Moving Objects

If a garbage collector moves an object, it has to take measures to ensure that the new location of the object becomes visible to the application threads. Otherwise, the application may never become aware of the change.

In both object layouts with handles and with forwarding pointers, threads must be forced to see the new location at some point. Handle-based layouts have a slight advantage, because only one location per object has to be updated, the reference to the actual object in the handle area. The interference of the garbage collector and the mutator threads is therefore limited. For object layouts with forwarding pointers, the stack and fields that reference a moved object must be patched at some point. It is therefore necessary to keep a far greater number of locations consistent.

A straightforward extension of the consistency implementation described in Section 4 is to allow cores to (atomically) invalidate the caches of all cores. Obviously, such a feature allows the garbage collector to make the new location of objects visible to all cores. The downside of this is that the cache behavior is not fully local anymore. WCET analysis has to take into account the effects of these invalidation. If  $N$  objects are moved in the worst case,  $N$  cache invalidations and corresponding cache fills have considered as worst-case overhead. More sophisticated cache architectures may lower the overhead — such architectures however effectively result in cache coherence protocols, which are known to be complex and introduce an unwanted amount of indeterminism to the execution time of memory accesses.

One way to keep the cache behavior fully local is to avoid moving objects. A fixed-block memory layout like in the JamaicaVM’s garbage collector [26] not only eliminates moving objects, but also bounds the fragmentation. Future work will have to investigate whether it is more efficient in terms of worst-case performance to avoid moving objects or to invalidate the caches of other processors.

## 5.5 Object Creation

The JMM assumes that objects are initialized with default values before they are ever used. When the default values are written upon object allocation, it is trivial to ensure that the thread that allocates the object sees the correct values. For other threads, this is not so trivial — object fields may (in theory) still be cached by other cores. Therefore, it would be (in theory) necessary to explicitly enforce visibility initialization for all potential uses. However, this can be relaxed in some cases.

Consider that the default values are written in the context of the thread that allocates the object. The addresses where the object is allocated cannot have been accessed since the last GC cycle (otherwise, the garbage collector could not have reclaimed the space). If caches are invalidated during root scanning, no thread can have the respective memory area consciously cached. However, there is a potential pitfall: if a newly allocated object happens to be in the same cache line as a live object, fields of the new object may be cached despite not having been explicitly accessed. In such a case the allocating thread would have to invalidate or update the caches of other processors. Such situations can be avoided if objects are aligned to cache line boundaries.

## 5.6 Final Variables

Not directly related to GC, but somewhat similar to the initialization during allocation is the issue of final values. If an object reference is obtained through correct publication of the reference, threads are guaranteed to see the proper final values. As the constructor is invoked immediately after the allocation of an object, the same considerations apply as for initial field values.

If a reference is published incorrectly (i.e., before the constructor exits), threads may see the initial or the proper final field values. The implementation must guarantee that reordering does not cause references to be incorrectly published. Writes to final fields must be made available before the reference to the respective object may be used.

## 5.7 Internal Data Structures

One issue that has been left out so far are internal data structures. One example are flags to indicate phases of the GC algorithm or the color of an object. Such structures are necessary to exchange information between GC and mutator threads. Of course, it is possible to use the same mechanisms for consistency and synchronization as for the mutator threads. However, this merges the happens-before relations of these two layers and induces stronger orderings than required by the application code.

Consider a flag that is used to indicate the color of an object. This flag is read in each write to a reference field to check whether the write barrier must shade the object. Using volatile accesses for each such read not only induces an ordering on accesses to the flag, but effectively a happens-before relation between all writes to reference fields, that transitively extends to the application code. Depending on the architecture, it may be cheaper to simply bypass the cache for accessing this flag.

The GC implementor must decide how to establish the required consistency. On some architectures, it may be cheaper to bypass the cache than to wait until the cache coherence protocol has settled and a fence instruction is completed.

## 6. CONCLUSION AND OUTLOOK

In this paper, we presented the design of a timing analysis friendly data cache. We showed that this cache is consistent with the JMM while avoiding complex cache coherence protocols. The cache avoids non-local modifications of the cache state, which usually lead to unpredictable cache states.

We also investigated the impact of such a cache on GC algorithms. Object creation and tracing of the object graph are not affected by the cache design. However, other phases of GC cannot be implemented in a straight-forward manner. When moving objects, the garbage collector has to ensure that the new location of an object is visible to all threads. This can be achieved by either allowing global invalidation/updates of caches or by avoiding to move objects at all.

A more complex challenge occurred for the start of a GC cycle. We found that reaching a consensus by invalidating the local caches in a way that is similar to sliding view root scanning requires some overhead in the write barriers. As this overhead seems to be small enough for our current write barrier implementation, a global cache invalidation is not necessary.

Future work will on the one hand concentrate to implement an analysis for the proposed cache to classify accesses as hits or misses. This will enable a decision whether the analysis-friendliness actually pays off in terms of WCET performance. Furthermore, it will be investigated whether the consistency implementation can be optimized w. r. t. worst-case performance. On the

other hand, future work will investigate the trade-off between invalidating caches to enable object moving and avoiding to move objects. Conclusive evidence on this topic will have to include performance as well as memory consumption metrics.

## 7. ACKNOWLEDGEMENT

The author would like to thank Martin Schöberl, who provided feedback on early drafts, and Doug Lea, who helped to prepare the final version of this paper.

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOP-ARD).

## 8. REFERENCES

- [1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.
- [2] J. Auerbach, D. F. Bacon, P. Cheng, D. Grove, B. Biron, C. Gracie, B. McCloskey, A. Micic, and R. Sciampanone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *Proceedings of the 7th ACM International Conference on Embedded software*, pages 245–254, Atlanta, GA, 2008.
- [3] H. Azatchi, Y. Levanoni, H. Paz, and E. Petrank. An on-the-fly mark and sweep garbage collector based on sliding view. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, USA, Nov. 2003.
- [4] D. F. Bacon, P. Cheng, and V. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices 38(1), New Orleans, LA, USA, Jan. 2003.
- [5] K. Barabash, O. Ben-Yitzhak, I. Gofit, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages and Systems*, 27(6):1097–1146, Nov. 2005.
- [6] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. *Lecture Notes in Computer Science*, 4421:331, 2007.
- [7] P. Cheng. *Scalable Real-Time Parallel Garbage Collection for Symmetric Multiprocessors*. PhD thesis, Carnegie Mellon University, 2001.
- [8] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, Nov. 1978.
- [9] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, OR, USA, Jan. 1994. ACM Press.
- [10] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 113–123, Charleston, SC, USA, Jan. 1993. ACM Press.
- [11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event

- ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(3a):15–26, 1990.
- [12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley Professional, Boston, Mass., 2005.
- [13] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.
- [14] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A: System Programming Guide, Part 1*, March 2009.
- [15] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 13–21, New York, NY, USA, 1992. ACM.
- [16] D. Lea. The JSR-133 cookbook for compiler writers. Available at <http://gee.cs.oswego.edu/dl/jmm/cookbook.html>. Accessed June 24, 2009.
- [17] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices 36(11), Tampa, FL, USA, Nov. 2001.
- [18] J. Manson. *The Java memory model*. PhD thesis, University of Maryland, College Park, MD, USA, 2004.
- [19] C. Pitter. *Time-Predictable Java Chip-Multiprocessor*. PhD thesis, Vienna University of Technology, Austria, 2009.
- [20] W. Puffitsch and M. Schoeberl. Non-blocking root scanning for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, September 2008.
- [21] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Journal of Real-Time Systems*, 37(2):99–122, Nov. 2007.
- [22] M. Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, Apr. 2006.
- [23] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.
- [24] M. Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.
- [25] J. Sevcik and D. Aspinall. On validity of program transformations in the Java memory model. In *Ecoop 2008-Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008, Proceedings*, page 27. Springer, 2008.
- [26] F. Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. aicas Books, 2002.
- [27] T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.