DiSquawk: 512 cores, 512 memories, 1 JVM

Foivos S. Zakkak
FORTH-ICS and University of Crete
zakkak@ics.forth.gr

Polyvios Pratikakis
FORTH-ICS
polyvios@ics.forth.gr

Technical Report FORTH-ICS/TR-470, June 2016

Abstract

Trying to cope with the constantly growing number of cores per processor, hardware architects are experimenting with modular non cache coherent architectures. Such architectures delegate the memory coherency to the software. On the contrary, high productivity languages, like Java, are designed to abstract away the hardware details and allow developers to focus on the implementation of their algorithm. Such programming languages rely on a process virtual machine to perform the necessary operations to implement the corresponding memory model. Arguing, however, about the correctness of such implementations is not trivial.

In this work we present our implementation of the Java Memory Model in a Java Virtual Machine targeting a 512-core non cache coherent memory architecture. We shortly discuss design decisions and present early evaluation results, which demonstrate that our implementation scales with the number of cores up to 512 cores. We model our implementation as the operational semantics of a Java Core Calculus that we extend with synchronization actions, and prove its adherence to the Java Memory Model.

Keywords: Java Virtual Machine; Java Memory Model; Operational Semantics; Non Cache Coherent Memory; Software Cache

1 Introduction

Current multicore processors rely on hardware cache coherence to implement shared memory abstractions. However, recent literature largely agrees that existing coherence implementations do not scale well with the number of processor cores, incur large energy and area costs, increase on-chip traffic, or limit the number of cores per chip [9, 35, 7], despite several attempts to design less costly or more scalable coherence protocols [24, 26].

To address that issue, recent work on hardware design proposes modular many-core architectures. Such examples are the Intel® Runnemede [7] architecture, the Formic prototype [20], and the EUROSERVER architecture [11]. These architectures are designed in a way that allows scaling up by plugging in more modules. Each module is self-contained and able to interface with other modules. Connecting multiple such modules builds a larger system that can be seen as a single many-core processor. In such architectures the trend is to use multiple mid-range cores with local scratchpads interconnected using efficient communication channels.

The lack of cache coherence renders the software responsible for performing the necessary data transfers to ensure data coherency in parallel programs. However, in high productivity languages,

such as Java, the memory hierarchy is abstracted away by the process virtual machines rendering the latter responsible for the data transfers. Process virtual machines provide the same language guarantees to the developers as in cache coherent shared-memory architectures. Those guarantees are formally defined in the language's memory model. The efficient implementation of a language's memory model on non cache coherent architectures is not trivial though. Furthermore, arguing about the implementation's correctness is even more difficult.

In this work we present an implementation of the Java Memory Model (JMM) [23] in DiSquawk, a Java Virtual Machine targeting the Formic-cube, a 512-core non cache coherent prototype based on the Formic architecture [20, 1]. We shortly discuss design decisions and present evaluation results, which demonstrate that our implementation scales with the number of cores. To prove our implementation's adherence to the Java Memory Model, we model it as the operational semantics of Distributed Java Calculus (DJC), a Java Core Calculus that we define for that purpose.

Specifically, this work makes the following contributions:

- We present a Java Memory Model (JMM) implementation for non cache coherent architectures that scales up to 512 cores, and we shortly discuss our design decisions.
- We present Distributed Java Calculus (DJC), a Java core calculus with support for Java synchronization actions and explicit cache operations.
- We model our JMM implementation as the operational semantics of DJC.
- We prove that the operational semantics of DJC adheres to JMM and present the proof sketch.

The remainder of this paper is organized as follows. §2 shortly presents JDMM, a JMM extension for non cache coherent memory architectures, and the motivation for this work; §3 presents our implementation of JDMM and shortly discusses the design decisions; §4 presents DJC, its operational semantics, and a proof sketch of its adherence to JDMM; §5 discusses related work; and §6 concludes.

2 Background and Motivation

In order to reduce network traffic and execution time, Java Virtual Machines (JVMs) on non cache coherent architectures usually implement some kind of software caching [25, 4] or software distributed shared memory [36, 34, 38, 12]. Both approaches rely on similar operations; to access a remote object they *fetch* a local copy; to make dirty copies globally visible they write them back (*write-back*); and to free space in the cache or force an update on the next access they *invalidate* local copies. Since JMM [23] is agnostic about such operations, we base our work on the Java Distributed Memory Model (JDMM) [37].

The JDMM is a redefinition of JMM for distributed or non cache coherent memory architectures. It extends the JMM with cache related operations and formally defines when such operations need to be executed to preserve JMM's properties. The JDMM is designed to be as relaxed as the JMM. Following a similar approach to that of Owens *et al.* [27] in the x86 Total Store Order (x86-TSO) definition, the JDMM first defines an abstract machine model and then defines the memory model based on it.

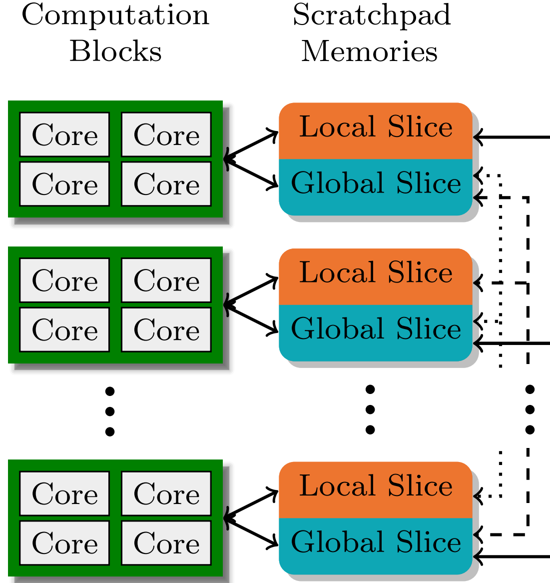

Figure 1: The memory abstraction.

Figure 1 presents an instance of the abstract machine as presented in the JDMM paper. On the left side there are several computation blocks with four cores in each of them. Each computation block connects directly to its local scratchpad memory. The scratchpad memory is split in a local and a global slice. In this model, each local slice connects with every other global slice in the system, but not with any local slice. The connections are bi-directional: a core can copy data from a remote global slice to the local cache to improve performance; after finishing the job it can transfer back the new data.

The local slice of the scratchpad is used for the local data (*i.e.*, Java stacks) and for caching remote data. The global slices are partitions of a total *virtual* Java Heap, similarly to Partitioned Global Address Space (PGAS) models. The state of the memory can only be altered by the computation blocks or by committing a fetch, a write-back, or an invalidate instruction.

In this abstract machine memory model the software needs to explicitly transfer data in such a way that JMM guaranties are preserved. At a high level, JMM guarantees that data-race-free (DRF) programs are sequentially consistent, and that variables cannot get *out-of-thin-air* values under any circumstances. To define our core calculus and couple it with the JDMM, we use a subset of the notation used in the JDMM paper, which we present here along with the JDMM short presentation. The JDMM describes program executions as tuples consisting of:

1) a set of instructions,

2) a set of actions, some of which are characterized as synchronization actions.

   The JDMM uses the following abbreviations to describe all possible kinds of actions:

   - R for read, W for write, and In for initialization of a heap-based variable,
   - Vr for read and Vw for write of a volatile variable,
   - L for the lock and U for the unlock of a monitor,

- S for the start and Fi for the end of a thread,
- Ir for the interruption of a thread and Ird for detecting such an interruption by another thread,
- Sp for spawning (`Thread.start()`) and J for joining a thread or detecting that it terminated,
- E for external actions, *i.e.*, I/O operations,
- F for fetch from heap-based variables,
- B for write-backs of heap-based variables,
- I for invalidations of cached variables.

Note that actions with kind In, Ir, Ird, Vr, Vw, L, U, S, Fi, Sp, or J are characterized as *synchronization actions* and form the only communication mechanism between threads.

3) the program order, which defines the order of actions within each thread,
4) the synchronization order, which defines a total ordering among the synchronization actions,
5) the synchronizes-with order, which defines the pairs of synchronization actions —release and acquire pairs,
6) the happens-before order that defines a partial order among all actions and is the transitive closure of the program order and the synchronizes-with order, and
7) some helper functions that we do not use in this paper.

The JDMM explicitly defines the conditions that a Java program execution needs to satisfy on a non cache coherent architecture, to be a well-formed execution. These conditions are introduced in [37, §3 and §4.2]; we briefly present them here. Note that **WF-1**–**WF-9** were first introduced in [23].

WF-1 Each read of a variable sees a write to it.

WF-2 All reads and writes of volatile variables are volatile actions.

WF-3 The number of synchronization actions preceding another synchronization action is finite.

WF-4 Synchronization order is consistent with program order.

WF-5 Lock operations are consistent with mutual exclusion.

WF-6 The execution obeys intra-thread consistency.

WF-7 The execution obeys synchronization order consistency.

WF-8 The execution obeys happens-before consistency.

WF-9 Every thread's start action happens-before its other actions except for initialization actions.

WF-10 Every read is preceded by a write or fetch action, acting on the same variable as the read.

WF-11 There is no invalidation, update, or overwrite of a variable's cached value between the action that cached it and the read that sees it.

WF-12 Fetch actions are preceded by at least one write-back of the corresponding variable.

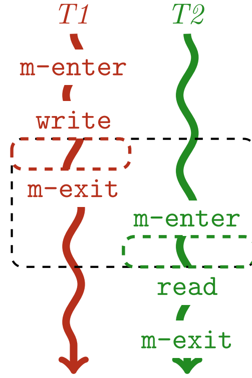

Figure 2: Time window example.

WF-13 Write-back actions are preceded by at least one write to the corresponding variable.

WF-14 There are no other writes to the same variable between a write and its write-back.

WF-15 Only cached variables can be invalidated. Invalid cached data cannot be invalidated.

WF-16 Reads that see writes performed by other threads are preceded by a fetch action that fetches the write-back of the corresponding write and there is no other write-back of the corresponding variable happening between the write-back and the fetch.

WF-17 Volatile writes are immediately written back.

WF-18 A fetch of the corresponding variable happens immediately before each volatile read.

WF-19 Initializations are immediately written-back; their write-backs complete before the start of any thread.

WF-20 The happens-before order between two writes is consistent with the happens-before order of their write-backs.

Two additional conditions must hold for executions containing thread migration actions. Intuitively:

WFE-1 There is a corresponding fetch action between a thread migration and every read action.

WFE-2 Additionally, to make sure the fetched value is the latest according to the happens-before order, any dirty data on the *old* core need to be written-back.

Note that, in the core JDMM, context switching without thread migration is examined only as an extension. As a result, we hereto use a slightly modified version of **WF-16** to allow DJC to be more relaxed in the case of context switches and still comply with the JDMM. The modified rule enables different threads running on the same core to share the contents of a single cache, without breaking the adherence to JMM, as shown in [37, §5.2]. That is:

WF-16 Reads that see writes performed by another *core* are preceded by a fetch action that fetches the write-back of the corresponding write and there is no other write-back of the corresponding variable happening between the write-back and the fetch.

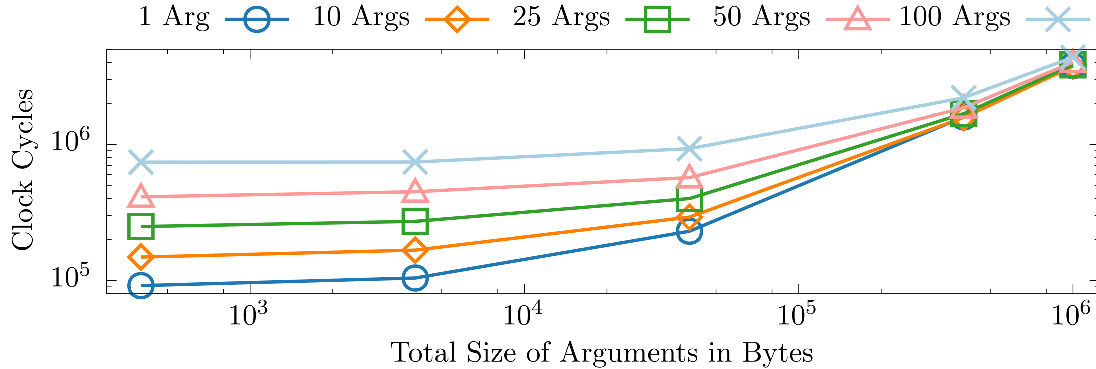

Figure 3: Performance impact of arguments size

The JDMM intuitively states that a write-back and its corresponding fetch may be executed any time in the time window between a write and the corresponding read, given that the write happens-before[1] this read. For instance, in Figure 2 the thread *T1* performs a `write` that happens-before the corresponding `read` in thread *T2*. The happens-before relationship is a result of the monitor release, `m-exit`, by *T1* and the subsequent monitor acquisition, `m-enter`, by *T2*. The time window that the JDMM allows a write-back and its corresponding fetch to be performed is marked with the big black dashed rectangle.

This flexibility on when these operations can be executed, allows for great optimization in theory. However, in practice it is very difficult to even estimate this time window. The JVM needs to keep extra information for every field in the program and constantly update it. It needs to know the sequence of lock acquisition, who was the last writer, if their write has been written-back, and whether the cached value (if any) is consistent with the main memory or not. Implementing these over software caching seems prohibitive, as the cost of the bookkeeping and the extra communication is expected to be much higher than the expected benefits regarding energy, space, and performance.

An intuitive implementation is to issue all the write-backs at release actions. However, this may result in long blocking release actions for critical sections that perform writes on large memory segments. To demonstrate the overhead of such operations we perform a simple experiment, where one core transfers a given data set from another core's scratchpad to its own. Figure 3 shows the impact of the arguments' size and number on the data transfer time. On the y-axes we plot the clock cycles consumed to transfer all the data from one core's to another core's scratchpad. On the x-axes we plot the total size of the data in Bytes. Each line in the plot represents a different partitioning of the data, in 1, 10, 25, 50, and 100 arguments respectively. We observe that apart from the total data size the partitioning of the data impacts the transfer time as well. This is a result of performing multiple data transfers instead of a single bulk transfer. As a result, keeping a lot of dirty data cached until a release operation is expected to perform badly, as it most probably will need to perform multiple data transfers to write-back non contiguous dirty data.

Hera-JVM [25] —the only, to the best of our knowledge, JVM for a non cache coherent architecture that claims adherence to the JMM— issues a write-back for every write and then waits for all pending write-backs to complete at release actions. This approach significantly reduces the blocking time at release actions, but results in multiple redundant write-backs in cases where a vari-

[1]as defined in [18]

able is written multiple times in a critical section. Such redundant memory operations are usually overlapped with computation, keeping their performance overhead low. However, the additional energy consumption they impose might still be significant in energy-critical systems. Additionally, in the case of writing to array elements, their approach results in one memory transfer per element when a bulk transfer can be used to improve performance and energy efficiency.

In this work we propose an alternative policy regarding write backs, that aims to mitigate such cases by caching dirty data up to a certain threshold. Additionally, since the Formic architecture is more relaxed than the Cell B.E. [29] architecture that Hera-JVM is targeting, we also present novel mechanisms to handle synchronization.

3 Implementation

We implement our memory and cache management policy in DiSquawk, a JVM we developed for the Formic-cube 512-core prototype. Formic-cube is based on the Formic architecture [20], which is modular and allows building larger systems by connecting multiple smaller modules. The basic module in the Formic architecture is the Formic-board. Each board consists of 8 MicroBlaze™-based, non cache coherent cores and is equipped with 128MB of scratchpad memory. Each core also features a private software-managed, non-coherent, two-level cache hierarchy; a hardware queue (mailbox) that supports concurrent en-queuing, and de-queuing only by the owner core; and a DMA engine. All of Formic's scratchpads are addressable using a global address space, and data are transferred through DMA transfers and mailbox messages to and from remote memory addresses.

3.1 Software Cache Management

As the Formic-cube does not provide hardware cache coherence, we build our JVM based on software caching. Each core is assigned a part of the local scratchpad, which it uses as its private software cache. This software cache is entirely managed by the JVM, transparently to the programmer.

To limit the amount of cached dirty data up to a given threshold we split the software cache in two parts. The first part, called *object cache*, is used for caching objects and is append-only —writes on this cache are not permitted. The second part, called *write buffer*, is dedicated to caching dirty data. When the write buffer becomes full, we write back all its data and update the corresponding fields in the object cache, if the corresponding object is still cached. Note that the combination of the write-buffer and the object cache form a memory-hierarchy, where the write-buffer is below the object cache. That is, read accesses first go through the write-buffer and only if they miss they go to the object cache. If they miss again, the JVM proceeds to fetch the corresponding object. This way, we *a*) set an upper limit on the release operations' blocking time; *b*) allow for overlapping write-backs with computation when the threshold is met; *c*) allow for bulk transfer of contiguous data, *e.g.*, written elements of an array; and *d*) allow for multiple writes to the same variable without the need to write back every time. At acquisition operations, we write back all the dirty data, if any, and invalidate both the object cache and the write buffer, in order to force a re-fetch of the data if they get accessed in the future. The write-back of the dirty data at acquisition operations is necessary since we invalidate all the cached data. Consider an example where a monitor is entered (acquire operation) then a write is performed, and a different monitor is now entered (acquire operation). In this case simply invalidating all cached data, would result in the loss of the write.

This approach is safe and sound, as we later show, but shrinks the aforementioned time window thus limiting the optimization space. A visualization of the shrunk time window is presented in Figure 2. The small red dashed rectangle on the upper left corner of the big rectangle is the time window in which the write-back can be executed. Respectively the small green dashed rectangle on the lower right corner is the time window in which the corresponding fetch can be executed. Note that although pre-fetching data, even in the shrunk time window, allows for significant performance optimizations we do not implement it in this work. Alternatively, we only fetch data at cache misses. Pre-fetching depends on program analysis to infer which data are going to be accessed in the future. Such analyses are not specific to non cache coherent architectures or the Java Memory Model, thus they our out of the scope of this work.

Despite the aforementioned reduction of flexibility regarding when a data transfer can happen, and the lack of support for pre-fetching, we are still able to achieve good performance and scale with the number of cores due to the efficient on-chip communication channels. To demonstrate this we use the Crypt, SOR, and Series benchmarks from the Java Grande [33] suite and the Black-Scholes benchmark from the PARSEC suite [5], ported to Java. Due to the lack of garbage collection and the upper limit of 4 GB heap we are unable to run reasonable workloads with the rest of the Java Grande benchmarks. These benchmarks require larger than 4 GB datasets to produce meaningful results on a large number of cores and some of them also create objects with short lifespans, relying on garbage collection to reclaim their memory. Series and Black-Scholes are embarrassingly parallel benchmarks. Each thread operates on a different subset of data from an input set and creates a new set with the corresponding results. The results are then accessed by the main thread for validation. Crypt comprises of two embarrassingly phases. In the first phase each thread encrypts a subset of the input data and then waits on a barrier. When all threads reach the barrier they proceed to decrypt each a subset of the encrypted data. The results are then compared to the original input for validation. SOR performs a number of iterations where each thread acts on a different block of an array accessing the previous and next neighboring blocks as well. As a result, each iteration depends on the neighboring blocks. To ensure that the neighboring blocks are ready, SOR uses a volatile counter for each thread. This counter reflects the iteration the corresponding thread is on. Each thread updates the counter at the end of each iteration and accesses the two counters of the neighboring threads.

Figure 4 presents the speedup of the four benchmarks on both DiSquawk, running on the formic-cube, and HotSpot running on a 4-chip NUMA machine with 16 cores per chip, totalling 64 cores. Since formic-cube is a prototype clocked at 10MHz, a comparison of the throughput or the execution time is not possible, thus we chose to compare the applications' scaling on both architectures. The presented speedups are over the performance of the application running on a single core on each architecture respectively. Since DiSquawk does not support JIT compilation, we also disable it in HotSpot (using the `-Xint` flag); this allows us to better understand the applications' behavior on both architectures. The number of Java threads, one per core, is placed on the x-axis, and the speedup is placed on the y-axis. Both axes are in logarithmic scale of base 2. We observe that all benchmarks manage to scale with the number of cores in both architectures. Black-Scholes and Series scale better on DiSquawk than HotSpot when using 32 or more cores, while Crypt performs better on HotSpot than DiSquawk when using up to 32 cores.

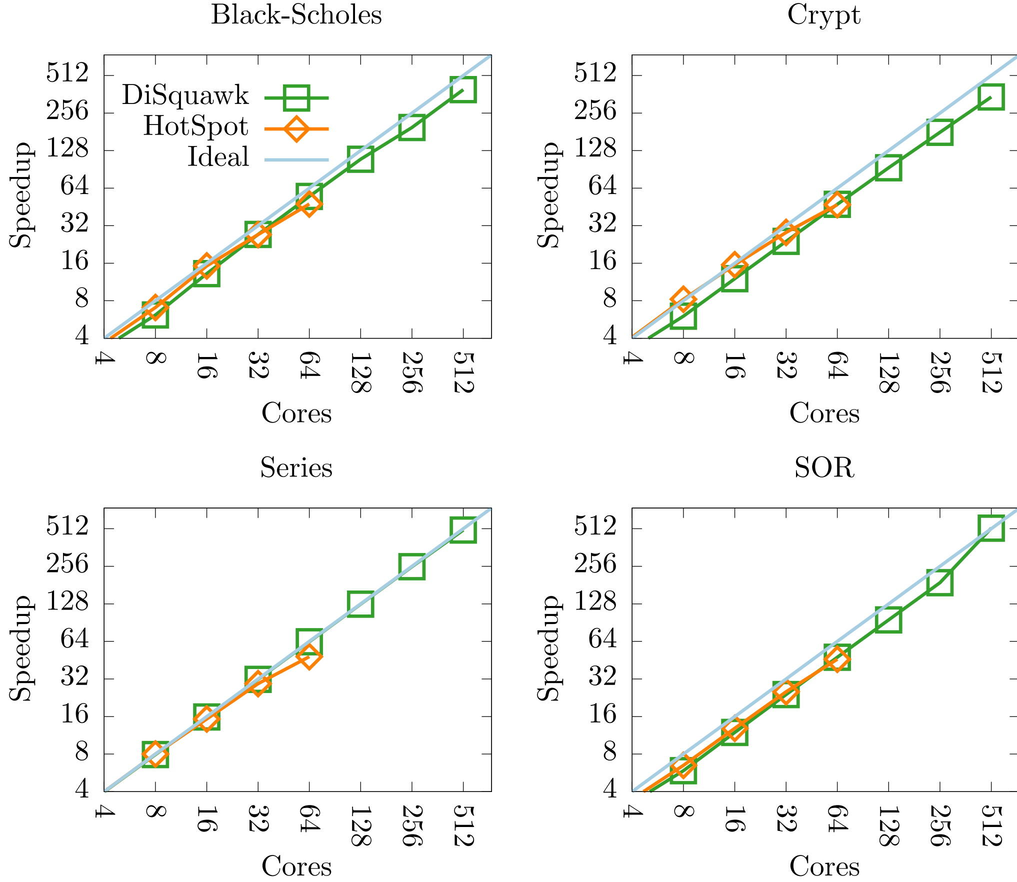

Figure 4: Speedup Results

3.2 Java Monitors

Apart from the data movement, JDMM also dictates the operation of Java monitors. Java monitors are essentially re-entrant locks associated with Java objects. In Java, each object is implicitly associated with a monitor and can be used in a `synchronized` block as the synchronization point. Java monitors are usually implemented using atomic operations, such as *compare and swap*, in shared-memory cache coherent architectures, relying on the hardware to synchronize multiple threads trying to obtain the monitor. Such atomic operations are not standard in non cache coherent architectures, though [14, 20].

To implement the Java monitors on such architectures we propose a synchronization manager: a server running on a dedicated core, handling monitor enter/exit requests. To keep contention at low levels we use multiple synchronization managers according to the number of available cores on the system. Each synchronization manager is responsible for a number of objects in the system, and each object can be associated with its synchronization manager using a hash function. When a thread executes a monitor-enter the JVM communicates with the corresponding synchronization manager and requests ownership of the monitor. This way all requests regarding a single monitor end up in the corresponding synchronization manager's hardware message queue, from where they

are handled by the synchronization manager one by one, in the order they arrived. We essentially delegate the synchronization of the requests to the architecture's network on chip, and provide mutual exclusion through the synchronization managers.

To reduce the synchronization managers' load, the network's traffic and contention, and to keep energy consumption low we take advantage of the blocking nature of monitors. Instead of sending back negative responses, when a monitor is already acquired by some other thread, we queue the monitor-enter requests in the synchronization manager, and assign the monitor to the oldest requester when it becomes available. This way we ensure fairness in the order that the requests are handled. Although this is not required by the Java Language Specification [13], we consider it better than arbitrarily choosing one of the waiting threads, since it avoids the starvation of threads. Additionally, when a thread is waiting for a monitor it yields to free up resources for other threads. Instead of periodically rescheduling such waiting threads —as we do with other yielded threads— we use a mechanism that reschedules them only when the monitor they requested has been assigned to them. That is, the synchronization manager has send an acknowledgement message to the core executing the waiting thread.

Using a synthetic micro-benchmark which constantly issues requests to a single monitor manager from X cores in the system, where $0 < X < 512$, we find that, on our system, at least one synchronization manager per 243 cores is required to avoid scenarios where the synchronization manager becomes a bottleneck.

3.3 Volatile Variables

Another challenging part is the support of volatile variables. Volatile variables are special, because accessing them is a form of synchronization. Specifically, volatile reads act as acquire operations, while volatile writes act as release operations. That said, after a volatile read any data visible to the last writer of the corresponding volatile variable must become visible to the reader. Volatile accesses are usually implemented using memory fences provided by the underlying architecture in shared-memory cache coherent systems [19].

Since non cache coherent architectures do not provide memory fences, in our implementation we rely on synchronization managers to ensure a total ordering between the various accesses to a volatile variable. Essentially we treat volatile accesses as synchronized blocks protected by a special monitor, unique per volatile variable. Therefore, we write back and invalidate any cached data before volatile accesses, and write back the dirty data immediately after volatile writes. This approach comes at the cost of unnecessary cache invalidations in the case of volatile writes, which should not be often since volatile variables are usually employed as a completion, interruption or status flag [28, §3.1.4] —meaning that they are being mostly read during their life-cycle.

A side-effect of this implementation is the provision of mutual exclusion to concurrent accesses on the same volatile variable. Since Formic provides no guarantees about the atomicity of memory accesses, we rely on this side-effect to ensure a volatile read will never return an *out-of-thin-air* value due to a partial update.

3.4 Wait/Notify Mechanism

Java also offers the wait/notify mechanism, which allows a thread to block its execution and wait for another thread to unblock it. Since `wait()` and `notify()` require the monitor of the corresponding object to be held by the executing thread, we use the synchronization manager to keep track of

such operations as well. The synchronization managers are holding a list of waiters for each object they are responsible for. Note that to keep the space overhead low we only allocate records when the first request for an object arrives. Initially, the synchronization managers hold no data for the objects they are responsible for. Whenever a thread invokes `wait()` a special message is send to the synchronization manager that adds the corresponding thread to the waiters queue and releases the monitor. As a result, before sending such messages we write back any dirty data. To support `wait()` invocations with a timeout we also support messages to the synchronization manager that request the removal of a thread from the waiters list. When `notify()` is invoked it sends a message to the synchronization manager, which notifies and removes the longest waiting thread (if any). In the case of `notifyAll()`, all threads in the waiters queue get notified and removed.

3.5 Liveness Detection

For the detection of thread termination and checking of liveness we rely on volatile variables. Each thread is described using a JVM internal object, which holds a volatile variable with the state of the thread. The supported states are, *spawned*, *alive*, *dead*. We implement `isAlive()` as a simple read to that state, if it is equal to *alive* then we return `true`. On the other hand, for the `join()` method we avoid spinning on the state variable in an effort to reduce energy consumption and free up resources for other threads in the system. We base our `join()` implementation on the `wait()/notify()` mechanism. Since a thread invoking `join()` will have to wait until the completion of the thread it joins, we yield it by invoking wait on the JVM internal object, describing the thread. When the corresponding thread reaches completion it invokes `notifyAll()` on that internal object and wakes up any joiners.

DiSquawk currently does not support interruptions. We consider their implementation regarding synchronization to be straightforward. Before sending an interrupt, all dirty data of the sending thread need to be written back, and upon interruption the receiving thread needs to write back any dirty data if present and invalidate its object cache.

4 The Calculus

To argue about the correctness of our implementation, we model it using a Java core calculus and its operational semantics. We base our calculus on the Java core calculus introduced by Johnsen *et al.* [16], which omits inheritance, subtyping, and type casts, and adds concurrency and explicit lock support. We extend that calculus by replacing the explicit lock support with synchronization operations and adding support for cache operations. We define the operational semantics of the resulting Distributed Java Calculus (DJC) and use it to argue about the correctness of the cache and monitor management techniques used in DiSquawk.

4.1 Syntax

The syntax of DJC is presented in Figure 5. A Java program J consists of a sequence $\vec{D}$ of class definitions. A class is defined as $\textsf{class}\ C(\overrightarrow{f:\tau})\{e\}\{\vec{M}\}$ where C is the class name; $\overrightarrow{f:\tau}$ is the list of field declarations, where each f_i is unique; e is the body of the class constructor; and $\vec{M}$ is a sequence of method definitions. The calculus types are class names C, boolean scalar types *Bool*, scalar natural numbers *Nat*, and *Unit* for the unit value (). A method is defined as $m(\overrightarrow{x:\tau})\{\textsf{return}\ e;\}:\tau$ where m is the method's name; $\overrightarrow{x:\tau}$ is the set of formal arguments; e is the

$$
\begin{array}{llcl}
\text{Program} & J & ::= & \vec{D} \\
\text{Class Def.} & D & ::= & \mathsf{class}\ C(\overrightarrow{f:\tau})\{e\}\{\vec{M}\} \\
\text{Types} & \tau & ::= & C \mid \mathit{Bool} \mid \mathit{Nat} \mid \mathit{Unit} \\
\text{Methods} & M & ::= & m(\overrightarrow{x:\tau})\{\mathsf{return}\ e;\} : \tau \\
\text{Expressions} & e & ::= & x \mid \mathsf{new}\ C(\vec{e}) \mid e.f \mid e.f := e \\
 & & \mid & \mathsf{let}\ x:\tau = e\ \mathsf{in}\ e \\
 & & \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid e.m(\vec{e}) \\
 & & \mid & e.\mathsf{acquire} \mid e.\mathsf{release} \\
 & & \mid & e.\mathsf{monitorenter} \mid e.\mathsf{monitorexit} \\
\text{Values} & v & ::= & r \mid () \mid \mathsf{true} \mid \mathsf{false} \mid n \\
\text{Contexts} & E(\bullet) & ::= & \mathsf{new}\ C(v,\ldots,\bullet,\ldots,e) \mid \bullet.f \\
 & & \mid & e.f := \bullet \mid \bullet.f := v \\
 & & \mid & \mathsf{let}\ x:\tau = \bullet\ \mathsf{in}\ e \\
 & & \mid & \mathsf{if}\ \bullet\ \mathsf{then}\ e\ \mathsf{else}\ e \\
 & & \mid & e.m(v,\ldots,\bullet,\ldots,e) \\
 & & \mid & \bullet.\mathsf{monitorenter} \mid \bullet.\mathsf{monitorexit} \\
\text{Threads} & T & ::= & c\langle r, \mathsf{start}\rangle \mid c\langle r, e\rangle \mid (T \parallel T) \mid \mathbf{0} \\
\text{Object} & o & \doteq & C(\overrightarrow{f \mapsto v}) \mid C(\overrightarrow{f \mapsto v}, \mathsf{started}) \\
 & & \mid & C(\overrightarrow{f \mapsto v}, \mathsf{spawned}) \\
 & & \mid & C(\overrightarrow{f \mapsto v}, \mathsf{finished}) \\
 & & \mid & C(\overrightarrow{f \mapsto v}, \mathsf{interrupted}) \\
\text{Heap} & \mathcal{H} & \doteq & \overrightarrow{r \mapsto (o,l)} \\
\text{Object Cache} & \mathcal{C} & \doteq & \overrightarrow{r \mapsto o} \\
\text{Write Buffer} & \mathcal{D} & \doteq & \overrightarrow{r.f \mapsto v} \\
\text{Cache per Core} & \vec{\mathcal{C}} & \doteq & \overrightarrow{c \mapsto \mathcal{C}} \\
\text{Buffer per Core} & \vec{\mathcal{D}} & \doteq & \overrightarrow{c \mapsto \mathcal{D}} \\
\text{Lock State} & l & ::= & 0 \mid r(n)
\end{array}
$$

Figure 5: Abstract syntax of DJC

method body; and τ is the return type. To keep the calculus simple we do not support method overloading.

The syntax includes variables x; creation of class instances as new $C(\vec{e})$; field accesses as $e.f$, where f is a unique field identifier; field updates as $r.f := e$; and sequential composition using the let-construct as let $x : \tau = e$ in e. Note that the evaluation of e may have side-effects. Conditional expressions are expressed as if e then e else e; and method calls as $e.m(\vec{e})$, where m is the method name.

The syntax also includes monitor enter and exit actions as expressions e.monitorenter and e.monitorexit, respectively. Note that volatile accesses do not have separate bytecodes in Java; they appear as normal memory accesses and the JVM checks at runtime whether they are volatile or not. Thus, we do not provide special syntax for them.

Values v are references to objects r, the unit value $()$, boolean constants true and false and scalar numerical constants n, abstracting over all other Java scalar types. Contexts are used to show the evaluation sequence of the expressions. In each expression in $E(\bullet)$ the $\bullet$ is evaluated first.

To argue about threads at runtime we extend DJC's syntax with run-time threads. A thread is defined as $c\langle r, \mathsf{start}\rangle$ or $c\langle r, e\rangle$, where c is the *unique* identification of the core that executes it; r is the corresponding instance of the `Thread` class; start is the thread start action, that signals

Notation	Definition
r	Reference value
m	Method identifier
f	Field identifier
c	Core identifier
$dom\,(X)$	Returns the keys of the map X
$rng\,(X)$	Returns the values of the map X
$\vec{X}[X'_i/X_i]$	Replaces X_i with X'_i in X
$\vec{X} \downarrow \vec{x}$	The subset of map bindings in X with keys in $\vec{x}$
$volatile\,(r.f)$	Returns true if $r.f$ is `volatile`
$C\left(\overrightarrow{f \mapsto v}\right)$	A Java object that is an instance of class C with mappings of field names to values $\overrightarrow{f \mapsto v}$

Figure 6: Definition of Notation

the start of its execution and is not to be confused with the `start()` method of the `Thread` class; and e is the thread's body. Threads can be composed in parallel pairs using the associative and commutative binary operator $\|$. The empty thread is marked with $\mathbf{0}$ and is the neutral element of $\|$.

We represent an object in the runtime syntax as $C(\overrightarrow{f \mapsto v})$ or $C(\overrightarrow{f \mapsto v}, state)$. The first form is used for every object in the memory, while the second is only used for thread objects whose `start()` method has been invoked, and *state* can be one of spawned, started, finished, and interrupted. Each object contains the name of its class and a map of field names f to values v. A thread whose `start()` method has been invoked is *spawned.* A thread whose run() method has been invoked is *started.* A thread that has reached completion is *finished.* A thread whose `interrupt()` method has been invoked is *interrupted.*

The memory of the system is split into the Heap $\mathcal{H}$, the object cache $\mathcal{C}$, the write buffer $\mathcal{D}$, the object cache per core $\vec{\mathcal{C}}$, and the write buffer per core $\vec{\mathcal{D}}$. The heap is a map from references r to objects o and their monitor l. The object cache is a map from references r to objects o. The write buffer is a map from object fields $r.f$ to values v. The object cache per core is a map from core ids c to object caches $\mathcal{C}$. Similarly, the write buffer per core is a map from core ids c to write buffers $\mathcal{D}$.

To model mutual exclusion we also add a lock state to the runtime syntax. A lock l may be free, i.e., 0, or acquired by some thread r, n times.

4.2 Operational Semantics

The operational semantics of DJC are based on those introduced by Johnsen *et al.* [16]. In this work we introduce new rules for *fetch*, *write-back*, *invalidate*, *volatile-read*, *volatile-write*, *start*, *finish*, *join*, *interrupt*, *interrupt detection*, and *migrate* operations. Note that we do not model `java.util.concurrent`, a Java library providing more synchronization mechanisms, in our formalization, since its interference with JMM is not yet fully defined.

Figure 6 presents a summary of the notations we use in the operational semantics of DJC, along with their definitions. We discuss these definitions in detail below, together with the operational semantics. To improve readability, we split the operational semantics in four categories: *core* semantics regarding the core language; *synchronization* semantics regarding volatile accesses,

$$\boxed{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e\rangle \xrightarrow{\alpha} \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle t_r, e\rangle}$$

$$[\textsc{CtxStep}]\ \frac{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e\rangle \xrightarrow{\alpha} \mathcal{H}';\mathcal{C}';\mathcal{D}' \vdash c\langle r_t, e'\rangle}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, E(e)\rangle \xrightarrow{\alpha} \mathcal{H}';\mathcal{C}';\mathcal{D}' \vdash c\langle r_t, E(e')\rangle}$$

$$\begin{array}{l}[\textsc{IfTrue}]\\ \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2\rangle \rightarrow \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e_1\rangle\end{array}$$

$$\begin{array}{l}[\textsc{IfFalse}]\\ \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2\rangle \rightarrow \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e_2\rangle\end{array}$$

$$\begin{array}{l}[\textsc{Let}]\\ \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, \mathsf{let}\ x : \tau = v\ \mathsf{in}\ e\rangle \rightarrow \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e[v/x]\rangle\end{array}$$

$$\begin{array}{l}[\textsc{Call}]\\ \dfrac{\mathcal{H}(r) = C(\overrightarrow{f \mapsto v'}) \qquad m(\overrightarrow{x:\tau})\{\mathsf{return}\ e;\} \in C}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.m(\vec{v})\rangle \rightarrow \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e[\vec{v}/\vec{x}][r/\mathsf{this}]\rangle}\end{array}$$

$$[\textsc{Field}]\ \frac{\begin{array}{cc} r \in \mathit{dom}\,(\mathcal{H}) & \neg \mathit{volatile}\,(v.f) \\ \mathcal{C}(r.f) = v & r.f \notin \mathit{dom}\,(\mathcal{D})\end{array}}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.f\rangle \xrightarrow{R} \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, v\rangle}$$

$$[\textsc{FieldDirty}]\ \frac{\begin{array}{cc} r \in \mathit{dom}\,(\mathcal{H}) & \neg \mathit{volatile}\,(v.f) \\ \multicolumn{2}{c}{\mathcal{D}(r.f) = v}\end{array}}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.f\rangle \xrightarrow{R} \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, v\rangle}$$

$$[\textsc{Assign}]\ \frac{v \in \mathit{dom}\,(\mathcal{H}) \qquad \neg \mathit{volatile}\,(v.f) \qquad \mathcal{D}' = \mathcal{D}[r.f \mapsto v]}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.f := v\rangle \xrightarrow{W} \mathcal{H};\mathcal{C};\mathcal{D}' \vdash c\langle r_t, v\rangle}$$

$$[\textsc{New}]\ \frac{r - \mathit{fresh} \qquad \mathcal{H}(r) = C(\overrightarrow{f \mapsto 0}) \qquad \mathsf{class}\ C(\overrightarrow{f:\tau})\{e\}\{\vec{M}\} \in J}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, \mathsf{new}\ C(\vec{v})\rangle \rightarrow \qquad \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, \mathsf{let}\ _ : \mathit{Unit} = e[\vec{v}/\vec{f}][r/\mathsf{this}]\ \mathsf{in}\ r\rangle}$$

Figure 7: Semantics of Local Operations

monitor handling, join, and interrupts; semantics for *implicit operations* performed by the JVM; and *global* semantics regarding parallel execution.

4.2.1 Core Semantics

Figure 7 presents the *core* semantics of DJC. Following the notation of Johnsen *et al.*, local configurations are of the form $\mathcal{H};\mathcal{C};\mathcal{D} \vdash e$. Note that in the conclusions of some semantic rules we annotate the $\rightarrow$ binary operator with an action kind from JDMM or α, *e.g.*, we use $\xrightarrow{R}$ to show that Field performs a read action R. In the proof presented in Appendix B, we present all action kinds along with their abbreviations used in the annotations, and use this information to argue about the adherence of the operational semantics to JDMM. Note that c and r_t in $c\langle r_t, e\rangle$, although present in every rule, are not involved in any of the rules in Figure 9. We use them to argue about the

global semantics, shown in Figure 10. This syntax allows us to argue about which core is executing a thread and what is the corresponding object of this thread.

The CtxStep rule describes the evaluation of an expression in a context. The IfTrue and IfFalse rules handle conditional expressions in the standard manner. Rule Let handles substitution in the standard manner. Rule Call handles method calls. We use $r.m(\vec{v})$ for invocations with arguments $\vec{v}$ of the method with name m of the object referenced by r. To determine the body of the method we use $m(\overrightarrow{x:\tau})\{\mathsf{return}\ e;\}$, where $\overrightarrow{x:\tau}$ are the formal arguments of the method and e is the method body. We evaluate method calls by substituting the formal arguments with the given ones and `this` with r in the method body.

In our VM, all memory accesses first go through the write buffer; if they miss they proceed to the object cache. Thus, to access a field we need it to be present either in the write buffer or the object cache. To reason about such accesses we define two structural rules, Field and FieldDirty. Rule Field handles non-volatile field accesses, when the field is cached in the object cache, and FieldDirty handles non-volatile field accesses, when the field is cached in the write buffer.

In Field, the first premise requires that the object containing the field being accessed is in the heap (has been allocated and initialized). The second premise requires the access to not refer to a volatile field. To achieve this we use the function $\mathit{volatile}\,(r.f)$ which returns true if the field f is `volatile` in the object referenced by r and false otherwise. This function models the distinction, performed internally by the JVM, of volatile fields from normal fields. The third premise requires that the core performing the read has a local copy of the field in its object cache, and the cached value is v. The last premise requires that the field is not cached in the write buffer. Considering $\mathcal{H}$, $\mathcal{C}$, and $\mathcal{D}$ as maps X, we use $X(k)$ to get the value of the cached object or field with key k. We also use $\mathcal{C}(r.f) = v$ as a shorter notation of $\mathcal{C}(r) = C(f'_1 \mapsto v'_1, \ldots, f \mapsto v, \ldots, f'_n \mapsto v'_n)$ to show that f maps to v in the object returned by $\mathcal{C}(r)$. Additionally, we use $\mathit{dom}\,(X)$ to get all the map keys, i.e., references in the case of $\mathcal{H}$ and $\mathcal{C}$ or field names in the case of $\mathcal{D}$.

Similarly, FieldDirty handles field accesses of fields that are cached in the write buffer. The only difference from Field is that we require f to be cached in the write buffer and get its value from there instead of the object cache.

Rule Assign handles non-volatile field writes, which also go through the write buffer. As a result, writes change the contents of the write buffer instead of the heap, as required by the last two premises. Given a map X, $X' = X \setminus k$ is used to show that X' contains the same mappings as X except a mapping for key k, thus $k \notin \mathit{dom}\,(X')$ and $X' \subseteq X$. Note that we use $\subseteq$ instead of $\subset$, since k might not be in the map in the first place.

Rule New invokes the constructor of the corresponding $\mathsf{class}\ C(\overrightarrow{f:\tau})\{e\}\{\vec{M}\}$ in a similar manner to Call. Rule CtxStep ensures that the constructor will be evaluated before the reference r will be assigned to any variable. This ensures that final fields are initialized before *publishing* the new object. Similarly to Johnsen *et al.*, we use $C\,(\vec{v})$ for instances of class C with field values $\vec{v}$, i.e., field f_i contains the value v_i. Note that according to the JMM "*conceptually every object is created at the start of the program*" [23, §4.3]. That said, in DJC we assume that the object is already present in the memory, with its fields initialized to the default value, and that New just invokes the constructor and returns a reference to the object. We use $r - \mathit{fresh}$ to show that there is no other reference to that object already.

$$\boxed{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e\rangle \rightarrow \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e\rangle}$$

$$[\textsc{Fetch}]\ \frac{\mathcal{H}(r) = C(\overrightarrow{f \mapsto v}) \qquad \mathcal{C}' = \mathcal{C}[r \mapsto \mathcal{H}(r)]}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e\rangle \xrightarrow{F} \mathcal{H};\mathcal{C}';\mathcal{D} \vdash c\langle r_t, e\rangle}$$

[WriteBack]

$$\frac{\neg volatile\,(r.f) \qquad r.f \in dom\,(\mathcal{D}) \qquad \begin{array}{c} r \in dom\,(\mathcal{H}) \qquad r \in dom\,(\mathcal{C}) \\ \mathcal{H}' = \mathcal{H}[r.f \mapsto \mathcal{D}(r.f)] \qquad \mathcal{C}' = \mathcal{C}[r.f \mapsto \mathcal{D}(r.f)] \end{array} \qquad \mathcal{D}' = \mathcal{D} \setminus r.f}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e\rangle \xrightarrow{B} \mathcal{H}';\mathcal{C}';\mathcal{D}' \vdash c\langle r_t, e\rangle}$$

$$[\textsc{Invalidate}]\ \frac{r \in dom\,(\mathcal{C}) \qquad \mathcal{C}' = \mathcal{C} \setminus r}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e\rangle \xrightarrow{I} \mathcal{H};\mathcal{C}';\mathcal{D} \vdash c\langle r_t, e\rangle}$$

$$[\textsc{Start}]\ \frac{\mathcal{C} = \emptyset \qquad \mathcal{D} = \emptyset \qquad \mathcal{H}(r_t) = C(\overrightarrow{f \mapsto v}, \mathsf{spawned}) \qquad \mathcal{H}'(r_t) = C(\overrightarrow{f \mapsto v}, \mathsf{started})}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, \mathsf{start}\rangle \xrightarrow{S} \mathcal{H}';\mathcal{C};\mathcal{D} \vdash c\langle r_t, r_t.run()\rangle}$$

$$[\textsc{Finish}]\ \frac{\mathcal{D} = \emptyset \qquad \mathcal{H}(r_t) = C(\overrightarrow{f \mapsto v}, \mathsf{started}) \qquad \mathcal{H}'(r_t) = C(\overrightarrow{f \mapsto v}, \mathsf{finished})}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, ()\rangle \xrightarrow{Fi} \mathcal{H}';\mathcal{C};\mathcal{D} \vdash c\langle r_t, ()\rangle}$$

Figure 8: Operational Semantics for Implicit Operations

4.2.2 Semantics of Implicit Operations

Figure 8 presents the operational semantics for implicit operations. These are operations performed implicitly by the virtual machine and do not map to language expressions. Rules Fetch, WriteBack, and Invalidate handle fetching, write-back, and invalidation of a cached object, respectively. Fetching an object requires that it exists in the heap (first and second premise). A fetch results in the addition of the object referenced by r in the object cache $\mathcal{C}$. Writing back a field $r.f$ requires that the object referenced by r is present in the heap $\mathcal{H}$ and the object cache $\mathcal{C}$, $r.f$ is not volatile, and there is a dirty copy of it in the write buffer $\mathcal{D}$. Writing-back a field results in the update of its value both in the heap $\mathcal{H}$ and the object cache $\mathcal{C}$. Invalidating an object's cached copy requires that it is cached. Note that this does not force that object's fields to not be cached in the write buffer. An invalidation results in the removal of the object referenced by r from the object cache, $\mathcal{C}$, of the core executing the invalidation. Rule Start enforces the evaluation of the thread start action before any other action in the thread and —treating thread start as an acquire action— requires the object cache and the write buffer to be empty on the running core.

Rule Finish handles the completion of a thread. Note that a thread reaches completion when its thread body is equal to the unit value (). As a release action requires the write buffer to be empty, and changes the state of the thread to allow joiners to proceed.

4.2.3 Semantics of Synchornization Operations

Figure 9 presents the *synchronization* operational semantics. That is, rules about volatile accesses, monitor handling, join, and interrupts.

$$\boxed{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e\rangle \rightarrow \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e\rangle}$$

[VolatileReadL]
$$\frac{r \in dom\,(\mathcal{H}) \qquad volatile\,(r.f) \qquad \mathcal{H}(r.f.l) = 0 \qquad \mathcal{H}' = \mathcal{H}[r.f.l \mapsto r_t]}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.f\rangle \rightarrow \mathcal{H}';\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.f\rangle}$$

[VolatileRead]
$$\frac{r \in dom\,(\mathcal{H}) \qquad \mathcal{H}(r.f.l) = r_t \qquad \mathcal{C} = \emptyset \qquad \mathcal{D} = \emptyset \qquad \mathcal{H}' = \mathcal{H}[r.f.l \mapsto 0] \qquad \mathcal{H}(r.f) = v}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.f\rangle \xrightarrow{Vr} \mathcal{H}';\mathcal{C};\mathcal{D} \vdash c\langle r_t, v\rangle}$$

[VolatileWriteL]
$$\frac{r \in dom\,(\mathcal{H}) \qquad volatile\,(r.f) \qquad \mathcal{H}(r.f.l) = 0 \qquad \mathcal{H}' = \mathcal{H}[r.f.l \mapsto r_t]}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.f \coloneqq v\rangle \rightarrow \mathcal{H}';\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.f \coloneqq v\rangle}$$

[VolatileWrite]
$$\frac{r \in dom\,(\mathcal{H}) \qquad \mathcal{H}(r.f.l) = r_t \qquad \mathcal{D} = \emptyset \qquad \mathcal{H}' = \mathcal{H}[r.f \mapsto v][r.f.l \mapsto 0]}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.f \coloneqq v\rangle \xrightarrow{Vw} \mathcal{H}';\mathcal{C};\mathcal{D} \vdash c\langle r_t, v\rangle}$$

[MonitorEnter]
$$\frac{r \in dom\,(\mathcal{H}) \qquad \mathcal{C} = \emptyset \qquad \mathcal{D} = \emptyset \qquad \mathcal{H}(r) = (o, 0) \qquad \mathcal{H}' = \mathcal{H}[r \mapsto (o, r_t(1))]}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.\textsf{monitorenter}\rangle \xrightarrow{L} \mathcal{H}';\mathcal{C};\mathcal{D} \vdash c\langle r_t, ()\rangle}$$

[NestedMonitorEnter]
$$\frac{r \in dom\,(\mathcal{H}) \qquad \mathcal{H}(r) = (o, r_t(n)) \qquad \mathcal{H}' = \mathcal{H}[r \mapsto (o, r_t(n+1))]}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.\textsf{monitorenter}\rangle \xrightarrow{L} \mathcal{H}';\mathcal{C};\mathcal{D} \vdash c\langle r_t, ()\rangle}$$

[MonitorExit]
$$\frac{r \in dom\,(\mathcal{H}) \qquad \mathcal{D} = \emptyset \qquad \mathcal{H}(r) = (o, r_t(1)) \qquad \mathcal{H}' = \mathcal{H}[r \mapsto (o, 0)]}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.\textsf{monitorexit}\rangle \xrightarrow{U} \mathcal{H}';\mathcal{C};\mathcal{D} \vdash c\langle r_t, ()\rangle}$$

[NestedMonitorExit]
$$\frac{r \in dom\,(\mathcal{H}) \qquad \mathcal{H}(r) = (o, r_t(n+2)) \qquad \mathcal{H}' = \mathcal{H}[r \mapsto (o, r_t(n+1))]}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r.\textsf{monitorexit}\rangle \xrightarrow{U} \mathcal{H}';\mathcal{C};\mathcal{D} \vdash c\langle r_t, ()\rangle}$$

$$[\textsc{Join}]\ \frac{\mathcal{C} = \emptyset \qquad \mathcal{D} = \emptyset \qquad \mathcal{H}(r'_t) = C(\overrightarrow{f \mapsto v}, \textsf{finished})}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r'_t.\textsf{join}()\rangle \xrightarrow{J} \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, ()\rangle}$$

[Interrupt]
$$\frac{\mathcal{D} = \emptyset \qquad \mathcal{H}(r'_t) = C(\overrightarrow{f \mapsto v}, \textsf{started}) \qquad \mathcal{H}'(r'_t) = C(\overrightarrow{f \mapsto v}, \textsf{interrupted})}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r'_t.\textsf{interrupt}()\rangle \xrightarrow{Ir} \mathcal{H}';\mathcal{C};\mathcal{D} \vdash c\langle r_t, ()\rangle}$$

[InterruptedT]
$$\frac{\mathcal{C} = \emptyset \qquad \mathcal{D} = \emptyset \qquad \mathcal{H}(r'_t) = C(\overrightarrow{f \mapsto v}, \textsf{interrupted})}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r'_t.\textsf{interrupted}()\rangle \xrightarrow{Ird} \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, ()\rangle}$$

[InterruptedF]
$$\frac{state \neq \textsf{interrupted} \qquad \mathcal{H}(r'_t) = C(\overrightarrow{f \mapsto v}, state)}{\mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, r'_t.\textsf{interrupted}()\rangle \rightarrow \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, ()\rangle}$$

Figure 9: Semantics of Synchronization Operations

Rules VOLATILEREADL and VOLATILEREAD handle reads of volatiles. Rules VOLATILEWRITEL and VOLATILEWRITE handle volatile writes. The combination of VOLATILEREADL and VOLATILEREAD results in a single *volatile-read*. The same holds for VOLATILEWRITEL, VOLATILEWRITE and the *volatile-write* action. Specifically, for each volatile field $r.f$ we assume a synthetic lock $r.f.l$. This lock is used to force a total ordering on the accesses to this variable and guarantee atomicity to the corresponding hardware memory accesses, as described in §3.3. When $r.f.l$ is 0, it means the volatile variable $r.f$ is not being accessed by another thread. Assigning the thread r_t to $r.f.l$ we essentially block other threads from accessing this volatile variable. Additionally, volatile accesses are exceptions to the rule that all accesses go through the cache. Since volatile reads are *acquire* actions and volatile writes are *release* actions, before volatile writes, any dirty data in the corresponding core's cache must be written-back and before volatile reads, the corresponding core's cache must be invalidated. We use $\emptyset$ for empty maps.

Rules MONITORENTER and NESTEDMONITORENTER handle monitor acquisition; similarly, rules MONITOREXIT and NESTEDMONITOREXIT handle monitor release. These rules use $r.l$ — not to be confused with the synthetic lock $r.f.l$ of volatile variables— to represent the implicit monitor associated with the object with identity r. Our monitor handling is similar to the lock handling introduced in [16]. The notation $H(r.l) = 0$ dictates that the corresponding monitor is not acquired by any thread in the system. $H(r.l) = r_t(n)$ dictates that the corresponding monitor has been acquired n times by the thread r_t. Rule MONITORENTER requires that a monitor must be free before its acquisition. Rule NESTEDMONITORENTER requires that a monitor is already owned by some thread before it gets re-entered by that same thread. Rules MONITOREXIT and NESTEDMONITOREXIT ensure that a monitor is released only by its owner and the same number of times it was previously acquired.

In the case of nested monitor acquisition we can avoid invalidating the object caches and writing-back data at nesting monitor release. By definition, nested acquisition of monitors requires that the monitor is owned by the same thread at any nesting level. Under that assumption, any concurrent actions that operate on the cached data used in the critical section would be the result of a data-race, meaning that the program is not DRF. In that case, it is not necessary for any of the corresponding dirty data to become visible, to the threads performing the racy accesses, at nested monitor releases. Note that racy accesses are not guaranteed to see the latest write if the thread executing them did not *synchronize-with* an action that *happens-after* that write. Similarly, since the monitor is already owned by the current thread, there is no need to invalidate its core's cache in order to get the latest values, since those values are the results of some data-race. As a result, rules NESTEDMONITORENTER and NESTEDMONITOREXIT do not need any special premises regarding object caches and write buffers.

Rule JOIN handles invocations to the `join()` method of a thread. Its first two premises require that the object cache and the write buffer are empty, since join is an acquire action. The third premise requires the state of the thread object to be finished, modeling the way a join blocks on the state of a thread in the JVM implementation.

Rule INTERRUPT handles invocations to the `interrupt()` method of a thread. Its first premise requires that the write buffer is empty, since interrupt is a release action. The second and third premises require the state of the thread object to be started before the interrupt and started after it, modeling the way interrupts are implemented by changing the thread's state in the JVM implementation or setting a hardware register in the case of using hardware interrupts.

Rules INTERRUPTEDT and INTERRUPTEDF handle invocations to the `interrupted()` method

$$\boxed{\mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash T \xrightarrow[\vec{c}]{\vec{\alpha}} \mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash T}$$

$$[\textsc{Lift}]\ \frac{\begin{array}{c}\mathcal{C}_c = \vec{\mathcal{C}}(c) \qquad \mathcal{D}_c = \vec{\mathcal{D}}(c) \\ \mathcal{C}'_c = \vec{\mathcal{C}}'(c) \qquad \mathcal{D}'_c = \vec{\mathcal{D}}'(c) \\ \mathcal{H};\mathcal{C}_c;\mathcal{D}_c \vdash c\langle r_t, e\rangle \xrightarrow{\alpha} \mathcal{H}';\mathcal{C}'_c;\mathcal{D}'_c \vdash c\langle r_t, e'\rangle \qquad \vec{\mathcal{C}}' = \vec{\mathcal{C}}[c \mapsto \mathcal{C}'_c] \qquad \vec{\mathcal{D}}' = \vec{\mathcal{D}}[c \mapsto \mathcal{D}'_c]\end{array}}{\mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash c\langle r_t, e\rangle \xrightarrow[\{c\}]{\{\alpha\}} \mathcal{H}';\vec{\mathcal{C}}';\vec{\mathcal{D}}' \vdash c\langle r_t, e'\rangle}$$

$$[\textsc{Spawn}]\ \frac{\begin{array}{c}\mathcal{H}(r_{t'}) = C(\overrightarrow{f \mapsto v}) \\ \mathcal{H}'(r_{t'}) = C(\overrightarrow{f \mapsto v}, \mathsf{spawned}) \qquad \mathsf{run}()\{\mathsf{return}\ e;\} \in C \qquad \vec{\mathcal{D}}(c) = \emptyset \qquad c' \in \textsc{Cids}\end{array}}{\mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash c\langle r_t, r_{t'}.\mathsf{start}()\rangle \xrightarrow[\{c\}]{\{Sp\}} \mathcal{H}';\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash c\langle r_t, ()\rangle \parallel c'\langle r_{t'}, \mathsf{start}\rangle}$$

$$[\textsc{Migrate}]\ \frac{\begin{array}{c}c' \in \textsc{Cids} \qquad c \neq c' \\ \mathcal{D}(c) = \emptyset \qquad \mathcal{D}(c') = \emptyset \qquad \mathcal{C}(c') = \emptyset\end{array}}{\mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash c\langle r_t, e\rangle \xrightarrow[\{c\}]{\{M\}} \mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash c'\langle r_t, e\rangle}$$

$$[\textsc{Blocked}]\ \frac{}{\mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash T_1 \xrightarrow[\emptyset]{\emptyset} \mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash T_1}$$

$$[\textsc{ParG}]\ \frac{\begin{array}{c}\vec{c_1} \cap \vec{c_2} = \emptyset \qquad \vec{\mathcal{C}_1} = \vec{\mathcal{C}} \downarrow \vec{c_1} \qquad \vec{\mathcal{C}_2} = \vec{\mathcal{C}} \downarrow \vec{c_2} \qquad \vec{\mathcal{C}_3} = \vec{\mathcal{C}} \setminus (\vec{\mathcal{C}_1} \cup \vec{\mathcal{C}_2}) \\ \vec{\mathcal{D}}_1 = \vec{\mathcal{D}} \downarrow \vec{c_1} \qquad \vec{\mathcal{D}}_2 = \vec{\mathcal{D}} \downarrow \vec{c_2} \qquad \vec{\mathcal{D}}_3 = \vec{\mathcal{D}} \setminus (\vec{\mathcal{D}}_1 \cup \vec{\mathcal{D}}_2) \\ \mathcal{H};\vec{\mathcal{C}_1};\vec{\mathcal{D}} \vdash T_1 \xrightarrow[\vec{c_1}]{\vec{\alpha_1}} \mathcal{H}';\vec{\mathcal{C}}'_1;\vec{\mathcal{D}}'_1 \vdash T'_1 \\ \mathcal{H};\vec{\mathcal{C}_2};\vec{\mathcal{D}} \vdash T_2 \xrightarrow[\vec{c_2}]{\vec{\alpha_2}} \mathcal{H};\vec{\mathcal{C}}'_2;\vec{\mathcal{D}}'_2 \vdash T'_2 \qquad \vec{\mathcal{C}}' = \vec{\mathcal{C}}'_1 \cup \vec{\mathcal{C}}'_2 \cup \vec{\mathcal{C}_3} \qquad \vec{\mathcal{D}}' = \vec{\mathcal{D}}'_1 \cup \vec{\mathcal{D}}'_2 \cup \vec{\mathcal{D}}_3\end{array}}{\mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash T_1 \parallel T_2 \xrightarrow[\vec{c_1} \cup \vec{c_2}]{\vec{\alpha_1} \cup \vec{\alpha_2}} \mathcal{H}';\vec{\mathcal{C}}';\vec{\mathcal{D}}' \vdash T'_1 \parallel T'_2}$$

Figure 10: Global Operational Semantics

of a thread. Rule InterruptedT handles cases where the thread is interrupted. Its first two premises require that the object cache and write buffer are empty, since interrupt detection is an acquire action. The third premise requires the state of the thread object to be interrupted.

Rule InterruptedF handles cases where the thread is not interrupted. Its premises require the state of the thread object to not be interrupted, in such cases the invocation is not a synchronization action so there is no need for flushing the object cache or the write buffer.

4.2.4 Semantics of Global Operations

In Figure 10 we present the global operational semantics of DJC. Similarly to the local configurations, the global configurations are of the form $\mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash e$, where $\vec{\mathcal{C}}$ and $\vec{\mathcal{D}}$ are all the system's object caches and write buffers respectively, while $\vec{\mathcal{C}}(c)$ and $\vec{\mathcal{D}}(c)$ are the object cache and write buffer of core c, respectively. Note that the heap is the same in global and local configurations since it is shared among all cores.

Rule Lift lifts local reduction steps to the global level. We use $\vec{\mathcal{C}}[c \mapsto \mathcal{C}'_c]$ and $\vec{\mathcal{D}}[c \mapsto \mathcal{D}'_c]$ to

show that the state of $\vec{\mathcal{C}}(c)$ and $\vec{\mathcal{D}}(c)$ in the system is replaced by $\mathcal{C}'_c$ and $\mathcal{D}'_c$, respectively.

Rule Spawn handles thread spawns (i.e., `Thread.start()` calls). For every spawn —which is also a release action— we require that all dirty data are written-back. Then the JVM picks one of the available cores, marked as c' and schedules thread v' to it. We represent this by introducing $c'\langle r'_t, \mathsf{start}\rangle$ in parallel to the previously running $c\langle r_t, r'_t.\mathsf{start}()\rangle$. Note that Spawn changes the state of the thread to started to mark that this thread has started and forbid any re-spawns.

Rule Migrate handles the Java thread migration to another core by the scheduler. It picks one of the available cores, marked as c' and replaces c with it, representing that thread r will continue its execution on core c instead of c'.

Rule Blocked is essentially a no-op that allows threads to block and not step in every transition in an execution trace, as *e.g.*, a finished but not joined thread.

In DJC, two (or more) Java threads can step concurrently through the ParG rule. Each thread may change its core's object cache and write buffer state and thus affect $\vec{\mathcal{C}}$ and $\vec{\mathcal{D}}$. Since the object caches and write buffers are disjoint for each core, the resulting global state of object caches and write buffers after a concurrent step is the union of the changed object buffers and write buffers by each set of cores that step in the parallel transition and those that where left unchanged by both. To get the object caches and write buffers that a set of cores $\vec{c}$ changes we use $\vec{\mathcal{C}} \downarrow \vec{c}$ (projection). Note that the first premise of ParG required the two sets of cores that perform a step in the parallel transition to be disjoint. This is to model that each core is running a single thread and performs a single step each time. Additionally, inspecting its eighth and ninth premise it only allows a single set of threads to modify the heap. This limitation partially models the hardware memory bus and how it orders memory transfers. We allow only one write per step to the heap, this way we allow parallelism but not concurrent writes to the heap. To improve this, one can slice the heap, then different synchronization managers may handle different slices of the heap and increase parallelism.

4.3 Proof Sketch

This section briefly describes the proof of DJC's adherence to the JDMM. Appendix B presents a detailed proof of adherence. Intuitively, the correctness property can be expressed as:

Theorem 1. *DJC's operational semantics generates only well-formed execution traces.*

To prove Theorem 2, we show by induction that DJC's operational semantics satisfies every well-formedness rule. That is, given any well formed execution trace:

$$\mathcal{H}; \vec{\mathcal{C}}; \vec{\mathcal{D}} \vdash T_1 \parallel T_2 \rightarrow^* \mathcal{H}'; \vec{\mathcal{C}'}; \vec{\mathcal{D}}' \vdash T'_1 \parallel T'_2$$

we show that the trace after taking one more step:

$$\mathcal{H}; \vec{\mathcal{C}}; \vec{\mathcal{D}} \vdash T_1 \parallel T_2 \rightarrow^* \mathcal{H}'; \vec{\mathcal{C}'}; \vec{\mathcal{D}}' \vdash T'_1 \parallel T'_2 \rightarrow \mathcal{H}''; \vec{\mathcal{C}''}; \vec{\mathcal{D}}'' \vdash T''_1 \parallel T''_2$$

is well-formed as well.

This amounts to essentially a preservation proof for each rule, many of which are straightforward. It is trivial to show that structural rules with conclusions that do not affect the memory state and do not regard synchronization actions preserve the well-formedness of the execution. For the rest, we argue about their effects on the execution state. Since DJC's operational semantics is tailored after JDMM's well-formedness rules, for most inference rules, inspecting their premises and conclusions is enough to show that a well-formedness rule is preserved.

As DJC models DiSquawk executions, we claim that DiSquawk executions adhere to the JDMM, and consequently to the JMM.

5 Related Work

To the best of our knowledge, the only other JVM implementing the Java memory model on a non cache coherent architecture is Hera-JVM [25]. Hera-JVM also employs caches which it handles in a similar manner to our implementation, with the difference that it starts a write-back at every write, as we discuss in §3. Regarding the synchronization mechanisms, Hera-JVM relies on the Cell B.E.'s `GETLLAR` and `PUTLLC` instructions to build an atomic compare-and-swap operation. However, such instructions are not available on the architectures at hand [14, 20]. Additionally, Hera-JVM did not aim to formally prove its adherence to the JMM.

Contrary to the implementation, language operational semantics are often used to formalize memory models. Previous work describes the memory semantics for shared memory multicore processor architectures, such as Power [21], x86 [27, 32], and ARM [3] processors, without focusing on a specific language semantics or memory model. Sarkar *et al.* [31] first combined the semantics of an architecture with the memory model definition of the C++ language, focusing on its execution on shared-memory Power processors. Pratikakis *et al.* [30] similarly present operational semantics for a specialized task-parallel programming model designed to target distributed-memory architectures. Our work differs from the aforementioned in that it is targeting distributed or non cache coherent memory architectures.

Boudol and Petri [6] define a relaxed memory model using an operational semantics for the Core ML language. Their work takes into account write buffers that must become empty before a lock release. Although the handling of write buffers is similar to handling caches regarding the write backs, the fetching and invalidation handling part is not covered in that work. Additionally, the authors only consider lock releases as synchronization points, while in the Java language there are multiple synchronization points according to JMM. Joshi and Prasad [17] extend the above work and define an operational semantics that accounts for caches, namely update and invalidation cache operations not previously supported. The authors use a simple imperative language, claiming it has greater applicability. Unfortunately, this approach further abstracts away details regarding the correct implementation of a specific programming language's memory model. In our work we focus on the Java language and provide all the needed details for the implementation of its memory model. Furthermore, both of the above papers define operational semantics for generic relaxed memory models. We believe that defining the operational semantics for a specific memory model, in this case the JMM, is a different task that focuses on the issues specific to the Java language.

Demange *et al.* [10] present the operational semantics of BMM, a redefinition of JMM for the TSO memory model. BMM is similar to this work in that it aims to bring the Java Memory Model definition closer to the hardware details. BMM, however, focuses on buffers instead of caches and assumes the TSO memory model, which is stricter than the memory model of the non cache coherent architectures at hand.

Jagadeesan *et al.* [15] also describe an operational semantics for the Java Memory Model. Their work, however, does not account for caches or buffers. It abstracts away the hardware details and considers reads and writes to become actions that float into the evaluation context. This approach does not explicitly define when and where writes should be eventually committed to satisfy the JMM. In our approach, we explicitly define where data get stored after any evaluation step.

We thus consider our approach to be closer to the implementation. Cenciarelli *et al.* [8] use a combination of operational, denotational, and axiomatic semantics to define the JMM. In that work, the authors show that all the generated executions adhere to the JMM, but as in [15] they do not account for the memory hierarchy.

6 Conclusions

This paper presents DiSquawk, a Java VM implementation of the Java Memory Model that targets a 512-core non cache coherent architecture, and a proof sketch that it adheres to JMM. We discuss design decisions and present evaluation results from the execution of a set of benchmarks from the Java Grande suite [33]. To prove the correctness of our implementation, we model all key points of the design using a core calculus DJC and its operational semantics. DJC is a concurrent java calculus aware of software caches and their mechanisms. DiSquawk has been developed as part of the GreenVM project [2] and is available for download at `https://github.com/CARV-ICS-FORTH/disquawk`.

References

[1] The Formic Architecture. `http://www.formic-board.com`, 2014.

[2] The GreenVM project. `http://www.ics.forth.gr/carv/greenvm/`, 2015.

[3] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The Semantics of Power and ARM Multiprocessor Machine Code. In *DAMP '09*, pages 13–24, 2008.

[4] G. Antoniu, L. Bougé, P. J. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, 27(10):1279–1297, 2001.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT' 08*, 2008.

[6] G. Boudol and G. Petri. Relaxed memory models: An operational approach. In *POPL '09*, pages 392–403, 2009.

[7] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. B. Fryman, I. Ganev, R. A. Golliver, R. C. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu. Runnemede: An architecture for Ubiquitous High-Performance Computing. In *HPCA*, pages 198–209, 2013.

[8] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java Memory Model: Operationally, Denotationally, Axiomatically. In *ESOP '07*, pages 331–346, 2007.

[9] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *PACT '11*, pages 155–166, 2011.

[10] D. Demange, V. Laporte, L. Zhao, S. Jagannathan, D. Pichardie, and J. Vitek. Plan B: A Buffered Memory Model for Java. In *POPL '13*, pages 329–342, 2013.

[11] Y. Durand, P. Carpenter, S. Adami, A. Bilas, D. Dutoit, A. Farcy, G. Gaydadjiev, J. Goodacre, M. Katevenis, M. Marazakis, E. Matus, I. Mavroidis, and J. Thomson. Euroserver: Energy efficient node for european micro-servers. In *DSD '14*, pages 206–213, 2014.

[12] M. Factor, A. Schuster, and K. Shagin. JavaSplit: a runtime for execution of monolithic Java programs on heterogenous collections of commodity workstations. In *CLUSTER '03*, pages 110–117, 2003.

[13] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java(TM) Language Specification, Java SE 8 Edition*. 2015.

[14] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC '10*, pages 108–109, 2010.

[15] R. Jagadeesan, C. Pitcher, and J. Riely. Generative Operational Semantics for Relaxed Memory Models. In *ESOP'10*, pages 307–326, 2010.

[16] E. B. Johnsen, T. M. T. Tran, O. Owe, and M. Steffen. Safe locking for multi-threaded java with exceptions. *The Journal of Logic and Algebraic Programming*, 81(3):257 – 283, 2012.

[17] S. Joshi and S. Prasad. An Operational Model for Multiprocessors with Caches. In C. Calude and V. Sassone, editors, *Theoretical Computer Science*, volume 323 of *IFIP Advances in Information and Communication Technology*, pages 371–385. 2010.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[19] D. Lea. The jsr-133 cookbook for compiler writers, 2008.

[20] S. Lyberis, G. Kalokerinos, M. Lygerakis, V. Papaefstathiou, D. Tsaliagkos, M. Katevenis, D. Pnevmatikatos, and D. Nikolopoulos. Formic: Cost-efficient and scalable prototyping of manycore architectures. In *FCCM '12*, pages 61–64, 2012.

[21] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An Axiomatic Memory Model for POWER Multiprocessors. In *CAV'12*, pages 495–512, 2012.

[22] J. Manson. *The Java Memory Model*. PhD thesis, Department of Computer Science, University of Maryland, 2004.

[23] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL '05*, pages 378–391, 2005.

[24] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why On-chip Cache Coherence is Here to Stay. *Communications of the ACM*, 55(7):78–89, July 2012.

[25] R. McIlroy and J. Sventek. Hera-JVM: A Runtime System for Heterogeneous Multi-core Architectures. In *OOPSLA '10*, pages 205–222, 2010.

[26] L. G. Menezo, V. Puente, and J. A. Gregorio. The Case for a Scalable Coherence Protocol for Complex On-chip Cache Hierarchies in Many Core Systems. In *PACT '13*, pages 279–288, Piscataway, NJ, USA, 2013.

[27] S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In *TPHOLs '09*, 2009.

[28] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java concurrency in practice*. 2006.

[29] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. In *ISSCC '05*, pages 184–592 Vol. 1, Feb 2005.

[30] P. Pratikakis, H. Vandierendonck, S. Lyberis, and D. S. Nikolopoulos. A Programming Model for Deterministic Task Parallelism. In *MSPC '11*, pages 7–12, 2011.

[31] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *PLDI '12*, pages 311–322, 2012.

[32] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The Semantics of x86-CC Multiprocessor Machine Code. In *POPL '09*, pages 379–391, 2009.

[33] L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *SC '01*, 2001.

[34] R. Veldema, R. Bhoedjang, and H. Bal. Distributed Shared Memory Management for Java. In *ASCI '99*, pages 256–264, 1999.

[35] Q. Yang, J. Fu, R. Poss, and C. Jesshope. On-chip Traffic Regulation to Reduce Coherence Protocol Cost on a Microthreaded Many-core Architecture with Distributed Caches. *ACM TECS*, 13(3s), 2014.

[36] W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9:1213–1224, 1997.

[37] F. S. Zakkak and P. Pratikakis. JDMM: A Java Memory Model for Non-cache-coherent Memory Architectures. In *ISMM '14*, pages 83–92, 2014.

[38] W. Zhu, C.-L. Wang, and F. C. M. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *CLUSTER '02*, pages 381–388, 2002.

A JDMM Formal Definitions

This appendix presents the JDMM's formal definitions and their corresponding formalism in DJC, where appropriate.

Distributed Execution: A distributed execution E_D is a tuple:

$$E_D = \langle P, A_D, \leq^d_{po}, \leq^d_{so}, W, V, Cs, Bf, Ab, Ai, \leq^d_{sw}, \leq^d_{hb} \rangle$$

where:

- The program P is a set of instructions, in DJC this is the program J.
- A_D is a set of *actions*.

  Actions: The JMM abstracts thread operations as actions [22, §5.1]. An action is a tuple $\langle r_t, k, r.f, u \rangle$, where t is the thread performing the action; k is the kind of action; v is the (runtime) variable, monitor, or thread, involved in the action; and u is a unique, among the actions, identifier.

  JDMM uses the following abbreviations to describe all possible kinds of actions:

  – R for read, W for write, and In for initialization of a heap-based variable
  – Vr for read and Vw for write of a volatile variable
  – L for the lock and U for the unlock of a monitor
  – S for the start and Fi for the end of a thread
  – Ir for the interruption of a thread and Ird for detecting such an interruption by another thread
  – Sp for spawning (`Thread.start()`) and J for joining a thread or detecting that it terminated
  – E for external actions, i.e., I/O operations
  – F for fetch from heap-based variables,
  – B for write-backs of heap-based variables,
  – I for invalidations of cached variables.

  In DJC we use $\Sigma \xrightarrow[\vec{c}]{(c' \mapsto \langle r_t, k, r.f, u \rangle) \in \vec{\alpha}} \Sigma'$ to denote a transition from state Σ to state Σ', where $\vec{c}$ is the set of cores involved in the transition and c' is the core performing the JDMM action $\langle r_t, k, r.f, u \rangle$ in this transition.

  To get the set of actions A_D, from a program's DJC execution trace:

  $$\Sigma_0 \xrightarrow[\vec{c_1}]{\vec{\alpha_1}} \Sigma_1 \xrightarrow[\vec{c_2}]{\vec{\alpha_2}} \Sigma_2 \ldots \Sigma_n \xrightarrow[\vec{c_n}]{\vec{\alpha_n}} \Sigma_{n+1}$$

  we take the union of the ranges $rng\,(\vec{\alpha})$, where $\vec{\alpha}$ is a set of mappings from cores to JDMM actions, i.e.:

  $$\overrightarrow{(c \mapsto \langle r_t, k, r.f, u \rangle)}$$

  Formally: $$A_D = rng\,(\vec{\alpha_1}) \cup rng\,(\vec{\alpha_2}) \cup \ldots \cup rng\,(\vec{\alpha_n})$$

- The program order $\leq^d_{po}$ is a relation on A_D defining the order of actions regarding a single thread t in A_D. JDMM uses $x \leq_{po} y$ to show that x comes before y according to the program order within a thread. Every pair of actions executed by a single thread t are ordered by the program order:

$$\big((x \neq y) \wedge (x.t = y.t)\big) \Leftrightarrow \big((x \leq^d_{po} y) \vee (y \leq^d_{po} x)\big)$$

- The synchronization order $\leq^d_{so}$ is a relation on A_D defining a global ordering among all *synchronization actions* in A_D

  Synchronization Actions: Any actions with kind *In*, *Ir*, *Ird*, *Vr*, *Vw*, *L*, *U*, *S*, *Fi*, *Sp*, or *J* are *synchronization actions*, which form the only communication mechanism between threads. JDMM uses $x \in \text{SA}(A_D)$ to show that x is a synchronization action in A_D:

  $\text{SA}(A_D) = \{x \in A : x.k \in \{In, Ir,\ Ird,\ Vr,\ Vw, L, U, S, Fi,\ Sp, J\}, F, B\}$

  JDMM uses $x \leq^d_{so} y$ to show that x comes before y according to the synchronization order. Every pair of synchronization actions are ordered by synchronization order:

$$x.k, y.k \in \text{SA}(A_D) \Leftrightarrow \big((x \leq^d_{so} y) \vee (y \leq^d_{so} x)\big)$$

  In DJC we group syncrhonization actions of the kinds *Ird*, and *J* in the acquire actions family, denoted by *Acq*. We also group syncrhonization actions of the kinds *Fi*, *Ir*, and *In* in the release actions family, denoted by *Rel*.

  As a result, in DJC:

  $\text{SA}(A_D) = \{x \in A : x.k \in \{Vr,\ Vw, L, U, S,\ Sp, F, B, Acq, Rel\}\}$

- The *write-seen* function W for every read action r returns the write action seen by r, in A_d. As a result, $W(r).v = r.v$.

- The *value-written* function V returns the value written by every write action w, in A_D. As a result, every read r, in A_D, reads the value $V\big(W(r)\big)$.

- The *cache-action-seen* function *Cs* returns the fetch or write action seen by any read r, in A_D. Note that: $Cs(r) \leq^d_{po} r$ and $Cs(r).k \in \{W, F\}$.

- The *write-back-fetched* function *Bf* returns the write-back action whose data each fetch action fetches, in A_D.

- The *action-written-back* function Ab returns the write action whose data each write-back writes-back, in A_D. Note that:
  $Ab(b) \leq^d_{po} b$ and $Ab(b).k \in \{In, W, Vw\}$.

  In DJC, $Ab(\langle r_t, B, r.f, u' \rangle)$ returns the initialization or write action $\langle r_t, In$ or $W, r.f, u\rangle$ whose data $\langle r_t, B, r.f, u' \rangle$ writes-back, according to the execution trace. Note, that in DJC we exclude volatile writes from the possible kind of actions returned by Ab, since volatile writes are never written-back by a separate write-back action, they are immediately written to the heap.

- The *action-invalidated* function Ai, returns the write or fetch action that cached the data invalidated by each invalidation action, in A_D. Note that: $Ai(i) \leq^d_{po} p$ and $Ai(i).k \in \{W, F\}$.

  In DJC, $Ad(\langle r_t, I, r.f, u' \rangle)$ returns the write-back or fetch action $\langle r_t, W$ or $F, r.f, u \rangle$ writing or fetching a value v_w that $\langle r_t, I, r.f, u' \rangle$ invalidates, according to the execution trace. Note that in DJC instead of write actions the function returns write-back actions, since write actions update the write buffer, which cannot be invalidated, and write-back actions update the values in the object cache, removing the corresponding entries from the write buffer.

- The distributed synchronizes-with order $\leq^d_{sw}$ is a relation on A_D defining which actions in A_D synchronize with each other.

  JDMM uses $x \leq^d_{sw} y$ to show that x synchronizes-with y. Note that $x \leq^d_{sw} y \Rightarrow x \leq_{so} y$. An action x synchronizes-with an action y, written $x \leq^d_{sw} y$, when:

  – x is the initialization of variable v and y is the first action of any thread:
  $$\big((x.k = In) \wedge (y.k = S)\big)$$
  – y is a subsequent read of the volatile variable written by x:
  $$(x.k = Vw) \wedge (y.k = Vr) \wedge (x \leq_{so} y)$$
  – y is a subsequent lock of the monitor that x unlocked:
  $$(x.k = U) \wedge (y.k = L) \wedge (x.v = y.v) \wedge (x \leq_{so} y)$$
  – y is the start action of thread t and x is the spawn of t:
  $$(x.k = Sp) \wedge (y.k = S) \wedge (x.v = y.t)$$
  – y is a call to `Thread.join()` or `Thread.isAlive()` and x is the finish action of this thread:
  $$(x.k = Fi) \wedge (y.k = J) \wedge (x.t = y.v)$$
  – y is an action detecting if a thread has been interrupted and x is an interrupt to that thread:
  $$(x.k = Ir) \wedge (y.k = Ird) \wedge (x.v = y.v)$$
  – y is the implicit read of a reference to the object being finalized and x is the end of the constructor of this object.

  In the synchronizes-with examples above, when comparing the variable v of one action with the thread t of the other (i.e., $x.t = y.v$) means that y acts on thread $x.t$. The x action is a *release* action and y is an *acquire* action. A *release* action must make all writes, visible to the executing thread, visible to the actions following (according to any of the orders defined till now) the *acquire* action.

  In DJC, given any execution trace:
  $$\ldots \Sigma_1 \xrightarrow[_]{\vec{\alpha_1}:\langle _, k, r.f, u \rangle \in \vec{\alpha_1}} \Sigma_2 \ldots \Sigma_{n-1} \xrightarrow[_]{\vec{\alpha_n}:\langle _, k', r.f, u' \rangle \in \vec{\alpha_n}} \Sigma_n \ldots$$
  where $k, k' \in \mathrm{SA}(A_D)$, *if and only if* k and k' can form a synchronization pair and there is no other transition:

$$\Sigma_x \xrightarrow[_]{\vec{\alpha_x}:\langle _, k \text{ or } k', r.f, u'' \rangle \in \vec{\alpha_x}} \Sigma_y$$

between the transitions that contain the actions with id u and u' then:

$$\langle _, k, r.f, u\rangle \leq^d_{sw} \langle _, k', r.f, u'\rangle$$

- The happens-before order $\leq^d_{hb}$ is a relation on A_D that defines a partial order among actions in A_D.

  The happens-before notion is the one introduced by Lamport in [18]. In the context of the JMM this is the transitive closure of the program order and the synchronizes-with order. JDMM uses $x \leq^d_{hb} y$ to show that x happens-before y.

  In DJC, given any execution trace:

  $\ldots \Sigma_1 \xrightarrow[_]{\vec{\alpha_1}:\langle _,_,_,u\rangle \in \vec{\alpha_1}} \Sigma_2 \ldots \Sigma_{n-1} \xrightarrow[_]{\vec{\alpha_n}:\langle _,_,_,u'\rangle \in \vec{\alpha_n}} \Sigma_n \ldots$

  if any of the following holds:

  - $\langle _, _, _, u\rangle \leq^d_{po} \langle _, _._, u'\rangle$
  - $\langle _, _, _, u\rangle \leq^d_{sw} \langle _, _._, u'\rangle$
  - there exists a transition $\Sigma_x \xrightarrow[_]{\vec{\alpha_x}:\langle _,_._,u''\rangle \in \vec{\alpha_x}} \Sigma_y$ that appears between the transitions that contain the actions with ids u and u', in the execution trace, and

    $$\langle _, _, _, u\rangle \leq^d_{hb} \langle _, _._, u''\rangle \leq^d_{hb} \langle _, _._, u'\rangle$$

    (transitivity)

  then $\langle _, _, _, u\rangle \leq^d_{hb} \langle _, _._, u'\rangle$.

Conflicting Accesses: If one of two accesses to the same variable is a write then these two accesses are *conflicting.*

Data-Race: A data-race occurs when two conflicting accesses may happen in parallel. That is, they are not ordered by happens-before.

Correctly Synchronized or Data-Race-Free Program:
A program is correctly synchronized or DRF if and only if all sequentially consistent executions are free of data-races.

Well-Formed Distributed Execution:
JDMM defines well-formed executions similarly to the JMM. Specifically, in JDMM, a distributed execution E_D is well-formed when:

WF-1 Each read of a variable v sees a write to v:

$$\forall r \in A_D : \exists y \in A_D : \big(W(r) = y\big)$$

Note that the original formal definition in JDMM [37, §3] is:

$$\forall x \in A_D : (x.k = R) \Rightarrow \exists y \in A_D : \big(W(x) = y\big)$$

where volatile reads are not considered. However, JMM [23, §4.4] states that "*For all reads $r \in A$, we have $W(r) \in A$ and $W(r).v = r.v$. The variable $r.v$ is volatile if and only if r is a volatile read, and the variable $w.v$ is volatile if and only if w is a volatile write.* ", where to our understanding w refers to $W(r)$, and r refers to both volatile and non-volatile reads. As a result, in this work, we chose to take volatile reads into account as well.

In DJC, this means that given the execution trace of E_D, for every transition containing a read action:

$$\Sigma \xrightarrow[_]{\vec{\alpha}:\langle _, R \text{ or } Vr, r.f, _\rangle \in rng(\vec{\alpha})} \Sigma'$$

in that trace, there is at least one transition containing a write or initialization action:

$$\Sigma_x \xrightarrow[_]{\vec{\alpha'}:\langle _, In \text{ or } W \text{ or } Vw, r.f, _\rangle \in rng(\vec{\alpha'})} \Sigma_y$$

which writes in $r.f$ the value that this read action sees.

WF-2 All reads and writes of volatile variables are volatile actions:

$\forall x \in A_D : x.k \in \{Vw, Vr\} \Rightarrow \nexists y \in A_D : (y.k \in \{R, W\}) \wedge (x.v = y.v)$

In DJC, this means that given the execution trace of E_D, in every transition $\Sigma \xrightarrow[_]{\vec{\alpha}} \Sigma'$ for every action

$$\langle _, k, r.f, _\rangle \in rng\,(\vec{\alpha})$$

k is either Vr or Vw, *if and only if* $r.f$ is a volatile variable.

WF-3 The number of synchronization actions preceding another synchronization action y is finite:

$$\forall y \in \mathrm{SA}(A_D) : \#\{x \in \mathrm{SA}(A_D) : x \leq^d_{so} y\} < \infty$$

WF-4 Synchronization order is consistent with program order:

$\forall x, y, z \in A_D : \big((x.t = z.t) \wedge (x \leq^d_{so} y \leq^d_{so} z)\big) \Rightarrow (x \leq^d_{po} z)$

In DJC this means that given the execution trace of E_D, if it contains a trace:

$$\ldots \Sigma_1 \xrightarrow[_]{\vec{\alpha_1}:\langle r_t, k_1, _, u_1\rangle \in rng(\vec{\alpha_1})} \Sigma_2 \xrightarrow[_]{\vec{\alpha_2}:\langle r'_t, k_2, _, u_2\rangle \in rng(\vec{\alpha_2})} \Sigma_3 \ldots \Sigma_n \xrightarrow[_]{\vec{\alpha_n}:\langle r_t, k_n, _, u_n\rangle \in rng(\vec{\alpha_n})} \Sigma_{n+1} \ldots$$

where $k_1, k_2, k_n \in \mathrm{SA}(A_D)$ and consequently

$$\langle r_t, k_1, _, u_1\rangle \leq^d_{so} \langle r'_t, k_2, _, u_2\rangle \leq^d_{so} \langle r_t, k_n, _, u_n\rangle$$

then it cannot also contain the trace:

$$\ldots \Sigma_n \xrightarrow[_]{\vec{\alpha_n}:\langle r_t, k_n, _, u_n\rangle \in rng(\vec{\alpha_n})} \Sigma_{n+1} \ldots \Sigma_1 \xrightarrow[_]{\vec{\alpha_1}:\langle r_t, k_1, _, u_1\rangle \in rng(\vec{\alpha_1})} \Sigma_2 \ldots$$

where $\langle r_t, k_n, _, u_n\rangle \leq^d_{po} \langle r_t, k_1, _, u_1\rangle$.

WF-5 Lock operations are consistent with mutual exclusion.

The number of lock actions performed on the monitor m by any thread t' before, according to the synchronization order, the lock action l performed by thread t on the monitor m must be equal to the number of unlock actions performed by thread t' before l on the monitor m:

$$\forall x \in A_D : \forall t \in T : (x.k = L) \wedge (x.t \neq t) \Rightarrow$$
$$\#\{y \in A_D : (y.t = t) \wedge (y.k = L) \wedge (y.v = x.v) \wedge (y \leq^d_{so} x)\} =$$
$$\#\{z \in A_D : (z.t = t) \wedge (z.k = U) \wedge (z.v = x.v) \wedge (y \leq^d_{so} x)\}$$

where T is the set of all the execution threads:

$$T = \{r_t : (\exists x \in A_D : t = x.t)\}$$

In DJC, this means that given the execution trace of E_D, if a transition containing a lock acquisition action for a monitor $r.l$:

$$\Sigma_x \xrightarrow[-]{\vec{\alpha}:\langle r_t,L,r.l,u\rangle \in \vec{\alpha}} \Sigma_y$$

exists in the trace, then for every thread r'_t, where $r'_t \neq r_t$ the number of transitions containing a lock acquisition action for $r.l$:

$$\Sigma_L \xrightarrow[-]{\vec{\alpha'}:\langle r'_t,L,r.l,u'\rangle \in \vec{\alpha'}} \Sigma'_L$$

which appear earlier in the trace:

$$\langle r'_t, L, r.l, u'\rangle \leq^d_{so} \langle r_t, L, r.l, u\rangle$$

is equal to the number of transitions containing a lock release action for $r.l$:

$$\Sigma_U \xrightarrow[-]{\vec{\alpha''}:\langle r'_t,U,r.l,u''\rangle \in \vec{\alpha''}} \Sigma'_U$$

that also appear earlier in the trace:

$$\langle r'_t, U, r.l, u''\rangle \leq^d_{so} \langle r_t, L, r.l, u\rangle$$

WF-6 The execution obeys intra-thread consistency.

In DJC this means that given the execution trace of E_D, for every trace:

$$\ldots \Sigma_1 \xrightarrow[-]{\vec{\alpha}:\langle r_t,In \text{ or } W \text{ or } Vw,r.f,u\rangle \in rng(\vec{\alpha})} \Sigma_2 \ldots \Sigma_n \xrightarrow[-]{\vec{\alpha'}:\langle r_t,R \text{ or } Vr,r.f,u'\rangle \in rng(\vec{\alpha'})} \Sigma_{n+1} \ldots$$

in it, the read action with id u' may return the value written by the action with id u, *if and only if* between the two transitions, performed by thread r_t, there is no other transition, performed by thread r_t, that includes a write action that acts on the same variable $r.f$

$$\Sigma_x \xrightarrow[-]{\vec{\alpha''}:\langle r_t,In \text{ or } W \text{ or } Vw,r.f,u''\rangle \in rng(\vec{\alpha''})} \Sigma_y$$

WF-7 The execution obeys synchronization order consistency.

JMM states that "*Synchronization order consistency says that (i) synchronization order is consistent with program order and (ii) each read r of a volatile variable v sees the last write*

to v to come before it in the synchronization order" [23, §3.2]. The first condition is satisfied *if and only if* **WF-4** is satisfied, so JDMM examines only the second condition in **WF-7**.

$$\forall r \in A_D : (r.k = Vr) \Rightarrow$$

$$\Big(\neg\big(r \leq^d_{so} W(r)\big) \wedge \nexists w' \in A_D : (w'.k = Vw) \wedge$$

$$(w'.v = r.v) \wedge \big(W(r) \leq^d_{so} w' \leq^d_{so} r\big)\Big)$$

In DJC this means that given the execution trace of E_D, for every trace:

$$\ldots \Sigma_1 \xrightarrow[\text{-}]{\vec{\alpha}:\langle _, Vw, r.f, u\rangle \in rng(\vec{\alpha})} \Sigma_2 \ldots \Sigma_n \xrightarrow[\text{-}]{\vec{\alpha'}:\langle _, Vr, r.f, u'\rangle \in rng\left(\vec{\alpha'}\right)} \Sigma_{n+1} \ldots$$

in it, the volatile read action with id u' returns the value written by the volatile write action with id u, *if and only if* between the two transitions there is no other transition that includes a volatile write action that acts on the same variable $r.f$

$$\Sigma_x \xrightarrow[\text{-}]{\vec{\alpha''}:\langle _, Vw, r.f, u''\rangle \in rng\left(\vec{\alpha''}\right)} \Sigma_y,$$

WF-8 The execution obeys happens-before consistency:

$$\forall r \in A_D : \Big(\neg\big(r \leq^d_{hb} W(r)\big) \wedge \nexists w' \in A_D : (w'.v = r.v) \wedge \big(W(r) \leq^d_{hb} w' \leq^d_{hb} r\big)\Big)$$

In DJC this means that given the execution trace of E_D, for every trace:

$$\ldots \Sigma_1 \xrightarrow[\text{-}]{\vec{\alpha}:\langle _, In \text{ or } W \text{ or } Vw, r.f, u\rangle \in rng(\vec{\alpha})} \Sigma_2 \ldots \Sigma_n \xrightarrow[\text{-}]{\vec{\alpha'}:\langle _, R \text{ or } Vr, r.f, u'\rangle \in rng\left(\vec{\alpha'}\right)} \Sigma_{n+1} \ldots$$

in it, where

$$\langle _, In \text{ or } W \text{ or } Vw, r.f, u\rangle \leq^d_{hb} \langle _, R \text{ or } Vr, r.f, u'\rangle$$

the read action with id u' may return the value written by the action with id u, *if and only if* there is no other transition, between the two transitions, that is ordered with them by happens-before and includes a write action that acts on the same variable $r.f$:

$$\Sigma_x \xrightarrow[\text{-}]{\vec{\alpha''}:\langle r_t, In \text{ or } W \text{ or } Vw, r.f, u''\rangle \in rng\left(\vec{\alpha''}\right)} \Sigma_y$$

where $\langle _, In \text{ or } W \text{ or } Vw, r.f, u\rangle \leq^d_{hb} \langle _, In \text{ or } W \text{ or } Vw, r.f, u''\rangle \leq^d_{hb} \langle _, R \text{ or } Vr, r.f, u'\rangle$

WF-9 Every thread's start action happens-before its other actions except for initialization actions:

$$\forall x, y, z \in A_D : \big((z.k \notin \{S, In\}) \wedge (x.k = In) \wedge (y.k = S)\big) \Rightarrow (x \leq^d_{hb} y \leq^d_{hb} z$$

JMM states that "*The write of the default value (zero, false or null) to each variable synchronizes-with to the first action in every thread. Although it may seem a little strange to write a default value to a variable before the object containing the variable is allocated, conceptually every object is created at the start of the program with its default initialized values. Consequently, the default initialization of any object happens-before any other actions (other than default writes) of a program.*" [23, §4.3]

As a result, in DJC we assume that in the starting state of a program's execution trace all the variables used in that trace are already initialized and written back to the main memory, i.e, all of them fit in the memory and are initialized to zero. Since in this work we do not examine allocation techniques and garbage collection, this assumption does not interfere with our implementation's proof of adherence to JDMM. We essentially model a JVM that initializes the heap at boot and does not perform any garbage collections during the execution, which is actually how our JVM works when garbage collection is turned off. To be consistent with the JDMM requirements about the ordering of initialization actions we define the beginning of every execution trace in DJC to be $\Sigma_{init} \rightarrow^{*} \Sigma'_{init}$, where $\rightarrow^{*}$ contains only transitions performing the initialization actions and their write-backs, for every variable in the execution trace, and $\Sigma_{init} \rightarrow^{*} \Sigma'_{init}$ is well-formed —each initialization happens-before its write-back.

WF-10 Every read is preceded by a write or fetch action, acting on the same variable as the read.

In JDMM all reads of heap-based variables see cached values. Formally:
$\forall r \in A_D : \Big(\big(W(r) \leq^{d}_{po} r\big) \vee \, \exists f \in A_D : \big((f.v = r.v) \wedge (f \leq^{d}_{po} r) \big) \Big).$

Note that JDMM does not consider simultaneous multithreading and context switching in the core model, thus it does not support cache sharing in its formal rules [37, §4.2]. As a result it requires for the read action that sees a value written or fetched by another action to be ordered with the latter according to program order. Cache sharing, however, is examined in [37, §5.2] and is shown to be safe under JDMM and not break the execution's well-formedness if enabled.

In DJC, which supports simultaneous multithreading with shared caches, this means that given the execution trace of E_D, for every transition $\Sigma_R \xrightarrow[_]{\vec{\alpha}:(c \mapsto \langle _, R, r.f, u \rangle) \in \vec{\alpha}} \Sigma'_R$, there is at least one transition $\Sigma \xrightarrow[_]{\vec{\alpha}:(c \mapsto \langle _, W \text{ or } F, r.f, u' \rangle) \in \vec{\alpha}} \Sigma'$ earlier in that trace as well, which essentially means that every read performed by a core c is preceded by a write or fetch action, also performed by c, acting on the same variable as the read.

Note that in the DJC definition of **WF-10** we do not include volatile accesses. This is justified by the fact that in DJC volatile reads access the heap directly, which can be seen as fetching, reading, and invalidating the variable in a single step. As a result, in DJC there is no other action before a volatile read that caches the variable. However, we still comply to the JDMM since we conceptually pack a fetch in the volatile read itself, meaning that every volatile read is indeed preceded by a (conceptual) fetch.

WF-11 There is no invalidation, update, or overwrite of a variable's cached value between the action that cached it and the read that sees it. Formally:
$\forall r \in A_D : \nexists x \in A_D : \Big((x.k \in \{I, F, W\}) \wedge \big(Cs(r) \leq^{d}_{po} x \leq^{d}_{po} r \big) \Big)$

In DJC, this means that given the execution trace of E_D, for every trace:
$\ldots \Sigma_1 \xrightarrow[_]{\vec{\alpha_1}:(c \mapsto \langle _, W \text{ or } F, r.f, u \rangle) \in \vec{\alpha_1}} \Sigma_2 \ldots \Sigma_n \xrightarrow[_]{\vec{\alpha_n}:(c \mapsto \langle _, R, r.f, u' \rangle) \in \vec{\alpha_n}} \Sigma_{n+1} \ldots$

in it, if the read action with id u' sees the value written or fetched by the action with id u, then there is no other transition $\Sigma \xrightarrow[-]{\vec{\alpha}:(c\mapsto\langle r_t, I \text{ or } F \text{ or} W, r.f, u''\rangle)\in\vec{\alpha}} \Sigma'$ between the transitions that contain the actions with ids u and u'.

Note that, as we explain for **WF-10**, we do not take in account volatile accesses and do not require a program order between the actions, instead we require that the actions are performed by the same core c.

WF-12 Fetch actions are preceded by at least one write-back of the corresponding variable.

For a value to be fetched, it must first be written to the main memory. The only way to write to the main memory, by definition, is through a write-back. Formally:

$$\forall f \in A_D, \exists b \in A_D : \big(b = Bf(f)\big)$$

WF-13 Write-back actions are preceded by at least one write to the corresponding variable.

For a variable to be written-back, it must be dirty in some cache; a cached copy becomes dirty only when written. Formally:

$$\forall b \in A_D, \exists w \in A_D : \big(w = Ab(b)\big)$$

In DJC this means that given the execution trace of E_D, for every transition $\Sigma \xrightarrow[-]{\vec{\alpha}:(c\mapsto\langle _, B, r.f, u\rangle)\in\vec{\alpha}} \Sigma'$, in it, there is a at least one transition $\Sigma_w \xrightarrow[-]{\vec{\alpha'}:(c\mapsto\langle _, W, r.f, u'\rangle)\in\vec{\alpha'}} \Sigma'_w$ earlier in that trace as well.

WF-14 There are no other writes to the same variable between a write and its write-back. Formally:

$\forall b \in A_D : \Big(\nexists w` \in A_D : \big((w'.v. = b.v) \wedge (Ab(b) \leq^d_{po} w' \leq^d_{po} b)\big)\Big)$

In DJC this means that given the execution trace of E_D, for every trace:

$\ldots \Sigma_1 \xrightarrow[-]{\vec{\alpha}:(c\mapsto\langle r_t, W, r.f, u\rangle)\in\vec{\alpha}} \Sigma_2 \ldots \Sigma_n \xrightarrow[-]{\vec{\alpha'}:(c\mapsto\langle r_t, B, r.f, u'\rangle)\in\vec{\alpha'}} \Sigma_{n+1} \ldots$

in it, the write-back action with id u' writes back the value written by the action with id u, *if and only if* there is no other transition containing a write $\Sigma \xrightarrow[-]{\vec{\alpha_w}:(c\mapsto\langle r_t, W, r.f, u''\rangle)\in rng\left(\vec{\alpha'_w}\right)} \Sigma'$ between the transitions that contain the actions with ids u and u'.

Note that, as in **WF-10** and **WF-11**, we do not take in account volatile accesses and do not require a program order between the actions, instead we require that the actions are performed by the same core c.

WF-15 Only cached variables are invalidated.

Invalid cached data cannot be invalidated. Formally:

$$\forall p \in A_D : \nexists p' \in A_D : \Big(\big(Ai(p) = Ai(p')\big) \wedge \big(Ai(p) \leq^d_{po} p' \leq^d_{po} p\big) \Big)$$

In DJC this means that given the execution trace of E_D, transitions containing invalidation actions:

$$\mathcal{H}; \vec{\mathcal{C}}; \vec{\mathcal{D}} \vdash T \xrightarrow[_]{\vec{\alpha}:(c \mapsto \langle r_t, I, r.f, u\rangle \in rng(\vec{\alpha})} \mathcal{H}'; \vec{\mathcal{C}'}; \vec{\mathcal{D}'} \vdash T'$$

appear in the trace only when $r.f \in dom\left(\vec{\mathcal{C}}(c)\right)$.

WF-16 Reads that see writes performed by other threads are preceded by a fetch action that fetches the write-back of the corresponding write and there is no other write-back of the corresponding variable happening between the write-back and the fetch.

Since all writes go through the cache, for a write to be seen by a read on a different thread, there must exist a write-back action and a subsequent fetch action for it. Formally:

$$\forall r \in A_D : \big(W(r).t \neq r.t\big) \Rightarrow \exists b, f \in A_D :$$
$$\Big(\big(Ab(b) = W(r)\big) \wedge \big(Bf(f) = b\big) \wedge \big(W(r) \leq^d_{po} b \leq^d_{sw} f \leq^d_{po} r\big) \wedge$$
$$\big(\nexists b' : (b'.v = b.v) \wedge (b \leq_{hb} b' \leq_{hb} r)\big) \Big)$$

In DJC, which supports simultaneous multithreading with shared caches, **WF-16** essentially translates to "Reads that see writes performed by other *cores* are preceded by a fetch action that fetches the write-back of the corresponding write and there is no other write-back of the corresponding variable happening between the write-back and the fetch"

This means that given the execution trace of E_D, for every trace:

$$\ldots \Sigma_1 \xrightarrow[_]{\vec{\alpha}:(c \mapsto \langle r_t, W, r.f, u\rangle) \in \vec{\alpha}} \Sigma_2 \ldots \Sigma_n \xrightarrow[_]{\vec{\alpha'}:(c' \mapsto \langle r'_t, R, r.f, u'\rangle \in \vec{\alpha'}} \Sigma_{n+1} \ldots$$

in it, where $c \neq c'$, the read action with id u' may see the value written by the action with id u, *if and only if* all of the following hold:

1. There is a transition containing a fetch action:
$$\Sigma_f \xrightarrow[_]{\vec{\alpha_f}:(c' \mapsto \langle r'_t, F, r.f, u_f\rangle) \in \vec{\alpha_f}} \Sigma'_f$$
between the transitions that contain the actions with ids u and u',
2. There is a transition containing a write-back action:
$$\Sigma_b \xrightarrow[_]{\vec{\alpha_b}:(c \mapsto \langle r_t, B, r.f, u_b\rangle) \in \vec{\alpha_b}} \Sigma_b$$
between the transitions that contain the actions with ids u and u_F,
3. There is no other transition containing a write-back action:
$$\Sigma'_b \xrightarrow[_]{\vec{\alpha'_b}:(c \mapsto \langle _, B, r.f, u'_b\rangle) \in \vec{\alpha'_b}} \Sigma''_b$$
between the transitions that contain the actions with ids u_B and u_F.

Note that, as in **WF-10**, **WF-11**, and **WF-14** we do not take in account volatile accesses and do not require a program order between the actions, instead we require that the corresponding actions are performed by the same core c.

WF-17 Volatile writes are immediately written back.

Allowing other actions between a volatile write and its write-back may result in other threads observing these actions as if they were executed before the volatile write. This is similar to moving these actions before the volatile write, which is an invalid reordering according to the JMM. Formally:

$\forall w \in A_D : (w.k = Vw) \Rightarrow \exists b \in A_D : \big((w \leq^d_{po} b) \wedge (w.v = b.v) \wedge \nexists x \in A_D : (w \leq^d_{po} x \leq^d_{po} b)\big)$

In DJC this means that given the execution trace of E_D, transitions containing volatile write actions:

$\mathcal{H}; \vec{\mathcal{C}}; \vec{\mathcal{D}} \vdash T : c\langle r_t, r.f \coloneqq v\rangle \in T \xrightarrow[-]{\vec{\alpha}:\langle r_t, Vw, r.f, u\rangle \in rng(\vec{\alpha})} \mathcal{H}'; \vec{\mathcal{C}'}; \vec{\mathcal{D}'} \vdash T' : c\langle r_t, v\rangle \in T'$

update the value of $r.f$ to v in the heap, i.e.:

$$\Big(r \mapsto C(\overrightarrow{f' : \tau})\Big) \in \mathcal{H}' \wedge\ (f \mapsto v) \in (\overrightarrow{f' : \tau})$$

WF-18 A fetch of the corresponding variable happens immediately before each volatile read.

Allowing other actions between a volatile read and its fetch may result in other threads observing these actions as if they were executed after the volatile read. This is similar to moving these actions after the volatile read, which is an invalid reordering according to the JMM. Formally:

$\forall r \in A_D : (r.k = Vr) \Rightarrow \exists f \in A_D : \Big((f \leq^d_{po} r) \wedge \big(f = Cs(r)\big) \wedge\ \nexists x \in A_D : (f \leq^d_{po} x \leq^d_{po} r)\Big)$

In DJC this means that given the execution trace of E_D, transitions containing volatile read actions:

$\mathcal{H}; \vec{\mathcal{C}}; \vec{\mathcal{D}} \vdash T : c\langle r_t, r.f\rangle \in T \xrightarrow[-]{\vec{\alpha}:\langle r_t, Vr, r.f, u\rangle \in rng(\vec{\alpha})} \mathcal{H}'; \vec{\mathcal{C}'}; \vec{\mathcal{D}'} \vdash T' : c\langle r_t, v\rangle \in T'$

always see the value v of $r.f$ from the heap, i.e.:

$$\Big(r \mapsto C(\overrightarrow{f' : \tau})\Big) \in \mathcal{H} \wedge\ (f \mapsto v) \in (\overrightarrow{f' : \tau})$$

WF-19 Initializations are immediately written-back and their write-backs are completed before the start of any thread.

In DJC this rule is always satisfied, since as we explain in **WF-9** we define the beginning of every execution trace in DJC to be $\Sigma_{init} \rightarrow^* \Sigma'_{init}$ where $\rightarrow^*$ contains only transitions performing the initialization actions and their write-backs, for every variable in the execution trace. As a result, in every execution trace initialization actions are written-back and their write-backs are completed before the start of any thread.

WF-20 The happens-before order between two writes is consistent with the happens-before order of their write-backs.

If, for two write actions w and w', $w \leq_{hb}^{d} w'$, then the corresponding write-back actions, b for w and b' for w', must also be ordered, so that $b \leq_{hb}^{d} b'$ and vice versa. Formally:

$$\forall b, b' \in A_D : \big(Ab(b) \leq_{hb} Ab(b')\big) \Leftrightarrow (b \leq_{hb} b')$$

WFE-1 There is a corresponding fetch action between thread migration and every read action.

$$\forall m, r \in A_D : \big((m.k = M) \wedge (m \leq_{po}^{d} r)\big) \Rightarrow \big(\exists f \in A_D : (m \leq_{po}^{d} f \leq_{po}^{d} r)\big)$$

In DJC, this means that given the execution trace of E_D, for every trace:

$$\ldots \Sigma_1 \xrightarrow[-]{\vec{\alpha}:\langle r_t, M, _, u\rangle \in rng(\vec{\alpha})} \Sigma_2 \ldots \Sigma_n \xrightarrow[-]{\vec{\alpha'}:\langle r_t, R, r.f, u'\rangle \in rng\left(\vec{\alpha'}\right)} \Sigma_{n+1} \ldots$$

there exists at least one transition containing a fetch action:

$$\Sigma \xrightarrow[-]{\vec{\alpha}:\langle r_t, F, r.f, u_f\rangle \in rng(\vec{\alpha})} \Sigma'$$

between the actions with ids u and u',

Note that, as in **WF-10**, **WF-11**, **WF-14**, and **WF-16** we do not take in account volatile accesses.

WFE-2 At migration, there are no dirty data at the *old* core. Formally:

$$\forall m, w \in A : \Big((m.k = M) \wedge \big(w \leq_{po} B(w) \leq_{po} m\big)\Big)$$

In DJC, this means that given the execution trace of E_D, for every trace:

$$\Sigma_1 \xrightarrow[-]{\vec{\alpha_1}:\langle r_t, W, r.f, u\rangle \in rng(\vec{\alpha_1})} \Sigma_2 \ldots \Sigma_n \xrightarrow[-]{\vec{\alpha_n}:\langle r_t, M, _, u'\rangle \in rng(\vec{\alpha_n})} \Sigma_{n+1} \ldots$$

there exists at least one transition containing a write-back action $\Sigma \xrightarrow[-]{\vec{\alpha}:\langle r_t, B, r.f, u_f\rangle \in rng(\vec{\alpha})} \Sigma'$ between the actions with ids u and u',

B Proof of adherence to JDMM

In this section we prove the adherence of DJC to JDMM. To achieve this we show that its operational semantics generates only well-formed, according to JDMM, executions. That is, given any well-formed execution trace, as described in Appendix A, $\Sigma \rightarrow^* \Sigma'$, where the $\rightarrow^*$ binary operator denotes an arbitrary number of transitions, we show that any execution trace $\Sigma \rightarrow^* \Sigma' \rightarrow \Sigma''$ is well-formed as well. In our reasoning we introduce some additional well-formedness rules that we prove true for any DJC execution trace. We mark such rules with **WFH-X**

WFH-1: For every non-volatile variable $r.f$ that appears in the execution trace, *if and only if* it is present in $\mathcal{H}$, then its value in $\mathcal{H}$ is the one written back by the last, according to synchronization order, write-back action, acting on $r.f$, in that execution trace.

WFH-2: For every non-volatile variable $r.f$ that appears in the execution trace, *if and only if* it is present in $\mathcal{C}(c)$, then its value in $\mathcal{C}(c)$ is the one fetched or written back by the last fetch or write-back action in that execution trace, which acts on $r.f$ and is performed by c.

WFH-3: For every non-volatile variable $r.f$ that appears in the execution trace, *if and only if* it is present in $\mathcal{D}(c)$, then its value in $\mathcal{D}(c)$ is the one written by the last write action in that execution trace, which acts on $r.f$ and is performed by c.

WFH-4: For every object r that appears in the execution trace, if $r \in dom(\mathcal{C}(c))$, then there is at least one transition $\Sigma_f \xrightarrow[\vec{c'}:c\in\vec{c'}]{\vec{\alpha}:\langle_,F,r,u_f\rangle\vec{\alpha}} \Sigma'_f$ in the execution trace.

WFH-5: For every variable $r.f$, that appears in the execution trace, if:
$r \in dom(\mathcal{H}) \vee r \in dom(sscache(c)) \vee r.f \in dom(\mathcal{D}(c))$
then the value stored in them is the result of a write to $r.f$.

WFH-6: For every volatile variable $r.f$ in $\mathcal{H}$, its value is the one written by the last, according to synchronization order, volatile write action, acting on it, in that execution trace, or the value written-back by the write-back action of the initialization action, acting on it, if there are no volatile write actions, acting on it, in that execution trace.

WFH-7: Each thread is assigned to a core *if and only if* it is spawned, and is assigned to a single core. Formally,

$$\forall c \in \text{CIDs} : \forall r_t \in dom(\mathcal{H}) : \big(\mathcal{H}(r_t) = C(\overrightarrow{f \mapsto v}, \mathsf{spawned}) \iff \exists T \in \vec{T} : c\langle r_t, _\rangle \in T\big) \wedge \big(\forall T \in \vec{T} : c\langle r_t, _\rangle \in T \Rightarrow \nexists c' \in \text{CIDs} : c' \neq c \wedge c'\langle r_t, _\rangle \in T\big)$$

where $\vec{T}$ are all the sets of threads in the execution trace.

WFH-8: Each thread appears only on a single set of threads in a pair of set of threads. That is, for every pair of set of threads $T_1 \parallel T_2$ in the execution trace:
$\forall r_t \in dom(\mathcal{H}) : (_\langle r_t, _\rangle \in T_1 \Rightarrow _\langle r_t, _\rangle \notin T_2) \wedge (_\langle r_t, _\rangle \in T_2 \Rightarrow _\langle r_t, _\rangle \notin T_1)$

WFH-9: The contents of the object cache and the write buffer of each core are altered only by that core.
$\forall c, c' \in \text{CIDs} : (_; \vec{\mathcal{C}}; \vec{\mathcal{D}} \vdash c\langle _, _\rangle \rightarrow _; \vec{\mathcal{C}}[c' \mapsto \mathcal{C}'_{c'}]; \vec{\mathcal{D}}[c' \mapsto \mathcal{D}'_{c'}] \vdash _) \Rightarrow c = c'$

Lemma 1. *Initialization actions happen-before every thread's start action.*

Proof. Satisfied for every execution trace by the definition of the beginning of every execution trace in DJC to be $\Sigma_{init} \rightarrow^* \Sigma'_{init}$ where $\rightarrow^*$ contains only transitions performing the initialization actions and their write-backs, for every variable in the execution trace. □

Lemma 2 (WF-12)**.** *Fetch actions are preceded by at least one write-back of the corresponding variable.*

Proof. In DJC this rule is always satisfied, since as we explain in **WF-9** we define the beginning of every execution trace in DJC to be $\Sigma_{init} \rightarrow^* \Sigma'_{init}$ where $\rightarrow^*$ contains only transitions performing the initialization actions and their write-backs, for every variable in the execution trace. □

Lemma 3 (WF-17)**.** *Volatile writes are immediately written back.*

Proof. Satisfied by the definition of VolatileWrite that writes the variable directly to the heap. □

Lemma 4 (WF-18)**.** *A fetch of the corresponding variable happens immediately before each volatile read.*

Proof. Satisfied by the definition of VolatileRead that reads the variable directly from the heap. □

Lemma 5 (WF-19)**.** *Initializations are immediately written-back and their write-backs are completed before the start of any thread.*

Proof. In DJC this rule is always satisfied, since as we explain in **WF-9** we define the beginning of every execution trace in DJC to be $\Sigma_{init} \rightarrow^* \Sigma'_{init}$ where $\rightarrow^*$ contains only transitions performing the initialization actions and their write-backs, for every variable in the execution trace. As a result, in every execution trace initialization actions are written-back and their write-backs are completed before the start of any thread. □

Lemma 6. *DJC's local operational semantics generates only well-formed execution traces.*

Proof. We show, by induction on the number of steps, that for each well formed execution trace $\Sigma \rightarrow^* \Sigma'$, $\Sigma \rightarrow^* \Sigma' \rightarrow \Sigma''$, where $\rightarrow^*$ and $\rightarrow$ are reductions of the local operational semantics, is also well-formed.

Rules CtxStep, IfTrue, IfFalse, Let, and Call regard the control flow of the program and are of no interest, since it is trivial to show that they preserve the well-formedness of the execution. Additionally, for each case we omit well-formedness rules that do not correlate with the transition at hand, e.g., we do not argue about **WF-2** if the rule at hand does not act on a volatile variable. Furthermore, we do not argue about **WF-4**, **WF-7** and **WF-8**, since in the local operational semantics the happens-before order is equivalent to the program order, since the creation of new threads is not possible. As a result, **WF-4**, **WF-7** and **WF-8** are also satisfied if **WF-6** is satisfied. Similarly we do not argue about **WF-16** and **WFE-1-WFE-2**, since in the local operational semantics it is not possible to spawn new threads or migrate the main thread, thus all the transitions are performed by a single core.

Base case: Any execution trace
$\Sigma_{init} \rightarrow^* \{(r_t \mapsto \mathsf{VMThread}(\emptyset, \mathsf{spawned}))\}; \emptyset; \emptyset \vdash c\langle r_t, \mathsf{start}\rangle \rightarrow \Sigma'$
, is well-formed.

In DJC the execution starts with a single thread –the main thread– and the beginning of any execution trace is:
$\Sigma_{init} \rightarrow^* \{(r_t \mapsto \mathsf{VMThread}(\emptyset, \mathsf{spawned}))\}; \emptyset; \emptyset \vdash c\langle r_t, \mathsf{start}\rangle$
where $\rightarrow^*$ contains only transitions performing the initialization actions and their write-backs, for every variable in the execution trace, and
$\Sigma_{init} \rightarrow^* \{(r_t \mapsto \mathsf{VMThread}(\emptyset, \mathsf{spawned}))\}; \emptyset; \emptyset \vdash c\langle r_t, \mathsf{start}\rangle$
is well-formed.
as a result,

$$\Sigma = \{(r_t \mapsto \mathsf{VMThread}(\emptyset, \mathsf{spawned}))\}; \emptyset; \emptyset \vdash c\langle r_t, \mathsf{start}\rangle$$

In the local operational semantics,

$$\mathcal{H}; \mathcal{C}; \mathcal{D} \vdash c\langle e\rangle \xrightarrow{\alpha} \mathcal{H}; \mathcal{C}; \mathcal{D} \vdash c\langle e\rangle$$

the only rule that can step is START.

WF-3 is satisfied, since this is the first synchronization action, other than initialization actions, in the execution trace and the number of initialization actions is equal to the number of variables, in a program, which is finite.

WF-9 is satisfied by Lemma 1 and the fact that the action at hand is a start action and is the first action, other than initialization and write-backs, in the program.

WFH-7 is satisfied, since initially there only exists a single thread, the main thread, that starts in a single core.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

As a result, the lemma is true for the first transition of any program.

Inductive step: Given a well-formed execution trace $\Sigma \rightarrow^* \Sigma'$, $\Sigma \rightarrow^* \Sigma' \rightarrow \Sigma''$ is also well-formed.

We examine each case for $\Sigma' \rightarrow \Sigma''$ in the local operational semantics:

$$\mathcal{H}; \mathcal{C}; \mathcal{D} \vdash c\langle r_t, e\rangle \xrightarrow{\alpha} \mathcal{H}; \mathcal{C}; \mathcal{D} \vdash c\langle r_t, e\rangle$$

and show that it satisfies the well-formedness rules.

Case 6.1. FIELD

$$\Sigma \rightarrow^* \Sigma' \xrightarrow{\langle r_t, R, r.f, u\rangle} \mathcal{H}; \mathcal{C}; \mathcal{D} \vdash c\langle r_t, e'\rangle$$

where $r.f \notin \mathit{dom}\,(\mathcal{D})$ and $\Sigma' = \mathcal{H}; \mathcal{C}; \mathcal{D} \vdash c\langle r_t, e\rangle$.

By the premises of FIELD:

$$r \in \mathit{dom}\,(\mathcal{H}) \wedge \neg \mathit{volatile}\,(v.f) \wedge \mathcal{C}(r.f) = v$$

WF-1: Since the value of $r.f$ is read from the object cache and $\Sigma \rightarrow^* \Sigma'$ is well formed, according to **WFH-5** that value will be the result of a write action, acting on $r.f$, that is performed by a transition in the execution trace. As a result, **WF-1** is satisfied.

WF-6: Since $r.f$ is present in the object cache and $\Sigma \rightarrow^* \Sigma'$ is well formed and according to **WFH-2**, it was either fetched or updated through a write-back. In both cases, since $\Sigma \rightarrow^* \Sigma'$ is well formed, according to **WFH-1** and **WFH-2**, respectively, the cached value will be that of the last write-back in the execution trace. Additionally, according to **WF-20** the happens-before order between two writes is consistent with the happens-before order of their write-backs, meaning that the cached value will be that of the last write in the execution trace. That said, **WF-6** is satisfied.

WF-10: Since $\Sigma \rightarrow^* \Sigma'$ is well formed and $\mathcal{C}(r.f) = v$, according to **WFH-4**, there exists a transition $\Sigma_f \xrightarrow{\langle _, F, r, u_f\rangle} \Sigma'_f$ in $\Sigma \rightarrow^* \Sigma'$. As a result, **WF-10** is also satisfied.

WF-11: Since the value of $r.f$ is read from the object cache and $\Sigma \to^* \Sigma'$ is well formed, according to **WFH-2** that value will be the result of the last fetch or write-back action, acting on $r.f$, that is performed by a transition in the execution trace. As a result, there are no updates or overwrites of the cached value between between the value that cached it and the read that sees it. An invalidation of $r.f$ between the last, in the execution trace, fetch or write-back action, that cached $r.f$, and the read, would result in the premises of FIELD not being satisfied, since the object cache would not contain a value for $r.f$. As a result there is also no invalidation of the variable's cached value between the action that cached it and the read that sees it. As a result, **WF-11** is satisfied.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.2. FIELDDIRTY

$$\Sigma \to^* \mathcal{H}; \mathcal{C}; \mathcal{D} \vdash c\langle r_t, e\rangle \xrightarrow{\langle r_t, R, r.f, u\rangle} \mathcal{H}; \mathcal{C}; \mathcal{D} \vdash c\langle r_t, e'\rangle$$

where $r.f \in \mathit{dom}\,(\mathcal{D})$.

By the premises of FIELDDIRTY:

$$r \in \mathit{dom}\,(\mathcal{H}) \wedge \neg \mathit{volatile}\,(v.f) \wedge \mathcal{C}(r.f) = v'$$

wf1: Since the value of $r.f$ is read from the write buffer and $\Sigma \to^* \Sigma'$ is well formed, according to **WFH-5** that value will be the result of a write action, acting on $r.f$, performed by a transition in the execution trace. As a result, **WF-1** is satisfied.

WF-6: Since the value of $r.f$ is read from the write buffer and $\Sigma \to^* \Sigma'$ is well formed, according to **WFH-3** that value will be the result of the last write action, acting on $r.f$, that is performed by a transition in the execution trace. As a result, **WF-6** is satisfied.

WF-11: Since the value is read from the write buffer and $\Sigma \to^* \Sigma'$ is well formed, according to **WFH-3** that value will be the result of the last write action, acting on $r.f$, that is performed by a transition in the execution trace. As a result, there are no updates or overwrites of the cached value between between the value that cached it and the read that sees it. Additionally, an invalidation of $r.f$ (possible through WRITEBACK) between the last, in the execution trace, write action that added $r.f$ to the write buffer and the read would result in the premises of FIELDDIRTY not being satisfied, since the write buffer would not contain a value for $r.f$. As a result there is also no invalidation of the variable's cached value between the action that cached it and the read that sees it. As a result, **WF-11** is satisfied.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.3. ASSIGN

$$\Sigma \to^* \mathcal{H}; \mathcal{C}; \mathcal{D} \vdash c\langle r_t, e\rangle \xrightarrow{\langle r_t, W, r.f, u\rangle} \mathcal{H}; \mathcal{C}; \mathcal{D}' \vdash c\langle r_t, e'\rangle$$

where $\mathcal{D}' = \mathcal{D}[r.f \mapsto v]$.

By the premises of Assign:

$$r \in dom\,(\mathcal{H}) \wedge \neg volatile\,(v.f)$$

WFH-3 and **WFH-5** are satisfied since the new value of $r.f$ in the write buffer is the one written by the write action of the last transition in the execution trace.

WFH-9 is satisfied, since the new value is added to the write buffer of the core performing the action.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.4. New

$$\Sigma \to^* \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e\rangle \xrightarrow{\langle r_t, In, r.f, u\rangle} \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e'\rangle$$

where $r - fresh \wedge\ \mathcal{H}' = \mathcal{H}[r \mapsto C(\overrightarrow{f \mapsto 0})] \wedge\ C(\overrightarrow{f : \tau})\{e\} \in C$

WFH-1, **WFH-5**, and **WFH-6** are satisfied since the values of the new object's variables in the heap are those of the last write-back to these variables, namely the write-back of their initialization.

WFH-2–**WFH-3** are satisfied since they are satisfied in $\Sigma \to^* \Sigma'$ and New does not modify the object cache, or the write buffer.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.5. VolatileReadL

WF-5 is satisfied since it is satisfied in $\Sigma \to^* \Sigma'$ and VolatileReadL requires $r.f.l$ to be free before acquiring it.

WFH-1, **WFH-5**, and **WFH-6** are satisfied since it is satisfied in $\Sigma \to^* \Sigma'$ and VolatileReadL does not modify any variables in the heap, only the synthetic lock of the volatile variable at hand.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.6. VolatileRead

$$\Sigma \to^* \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e\rangle \xrightarrow{\langle r_t, Vr, r.f, u\rangle} \mathcal{H}';\mathcal{C};\mathcal{D} \vdash c\langle r_t, e'\rangle$$

where $r \in dom\,(\mathcal{H}) \wedge\ \mathcal{H}(r.f.l) = r_t \wedge\ \mathcal{C} = \emptyset \wedge\ \mathcal{D} = \emptyset \wedge\ \mathcal{H}' = \mathcal{H}[r.f.l \mapsto 0] \wedge\ \mathcal{H}(r.f) = v$

wf1: Since the value of $r.f$ is read from the heap and $\Sigma \to^* \Sigma'$ is well formed, according to **WFH-6** that value will be the result of the last volatile write action, acting on $r.f$, in that execution trace, or by the initialization action, acting on $r.f$, if there are no volatile write actions, acting on $r.f$, in that execution trace. As a result, **WF-1** and **WF-6** are satisfied.

WF-2 is satisfied since in $\Sigma \rightarrow^* \Sigma'$ all volatile variables where accessed by volatile actions according to **WF-2** and the volatile read at hand is also a volatile action.

WF-3 is satisfied, since $\Sigma \rightarrow^* \Sigma'$ is well formed and according to **WF-3** the number of synchronization actions in it are finite.

WFH-1, **WFH-5**, and **WFH-6** are satisfied since it is satisfied in $\Sigma \rightarrow^* \Sigma'$ and VolatileWriteL does not modify any variables in the heap, only the synthetic lock of the volatile variable at hand.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.7. VolatileWriteL

WF-5 is satisfied since it is satisfied in $\Sigma \rightarrow^* \Sigma'$ and VolatileWriteL requires $r.f.l$ to be free before acquiring it.

WFH-1, **WFH-5**, and **WFH-6** are satisfied since it is satisfied in $\Sigma \rightarrow^* \Sigma'$ and VolatileWriteL does not modify any variables in the heap, only the synthetic lock of the volatile variable at hand.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.8. VolatileWrite

WF-2 is satisfied since in $\Sigma \rightarrow^* \Sigma'$ all volatile variables where accessed by volatile actions according to **WF-2** and the volatile write at hand is also a volatile action.

WF-3 is satisfied, since $\Sigma \rightarrow^* \Sigma'$ is well formed and according to **WF-3** the number of synchronization actions in it are finite.

WF-5 is satisfied since it is satisfied in $\Sigma \rightarrow^* \Sigma'$ and VolatileWrite requires $r.f.l$ to be acquired by the thread performing the action to release it.

WFH-1 is satisfied since it is satisfied in $\Sigma \rightarrow^* \Sigma'$ and VolatileWrite does not modify any non-volatile variables in the heap.

WFH-5 and **WFH-6** are satisfied since the new value of $r.f$ in the heap is the one written by the volatile write action of the last transition in the execution trace.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.9. MonitorEnter

WF-3 is satisfied, since $\Sigma \rightarrow^* \Sigma'$ is well formed and according to **WF-3** the number of synchronization actions in it are finite.

WF-5 is satisfied since it is satisfied in $\Sigma \to^* \Sigma'$ and MonitorEnter requires that the monitor $r.l$ is free before acquiring it.

WFH-1, **WFH-5**, and **WFH-6** are satisfied since it is satisfied in $\Sigma \to^* \Sigma'$ and VolatileWriteL does not modify any variables in the heap, only the monitor of the object at hand.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.10. NestedMonitorEnter

WF-3 is satisfied, since $\Sigma \to^* \Sigma'$ is well formed and according to **WF-3** the number of synchronization actions in it are finite.

WF-5 is satisfied since it is satisfied in $\Sigma \to^* \Sigma'$ and NestedMonitorEnter requires that the monitor $r.l$ is already acquired by the thread performing the action in order to re-acquire it.

WFH-1, **WFH-5**, and **WFH-6** are satisfied since it is satisfied in $\Sigma \to^* \Sigma'$ and VolatileWriteL does not modify any variables in the heap, only the monitor of the object at hand.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.11. MonitorExit

WF-3 is satisfied, since $\Sigma \to^* \Sigma'$ is well formed and according to **WF-3** the number of synchronization actions in it are finite.

WF-5 is satisfied since it is satisfied in $\Sigma \to^* \Sigma'$ and MonitorExit requires that the monitor $r.l$ is already acquired a single time by the thread performing the action in order to release it.

WFH-1, **WFH-5**, and **WFH-6** are satisfied since it is satisfied in $\Sigma \to^* \Sigma'$ and VolatileWriteL does not modify any variables in the heap, only the monitor of the object at hand.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.12. NestedMonitorExit

WF-3 is satisfied, since $\Sigma \to^* \Sigma'$ is well formed and according to **WF-3** the number of synchronization actions in it are finite.

WF-5 is satisfied, since it is satisfied in $\Sigma \to^* \Sigma'$ and NestedMonitorExit requires that the monitor $r.l$ is already acquired more than one times by the thread performing the action in order to decrease by one the acquisitions by that thread.

WFH-1, **WFH-5**, and **WFH-6** are satisfied since it is satisfied in $\Sigma \to^* \Sigma'$ and VolatileWriteL does not modify any variables in the heap, only the monitor of the object at hand.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.13. Acquire

WF-3 is satisfied, since $\Sigma \rightarrow^* \Sigma'$ is well formed and according to **WF-3** the number of synchronization actions in it are finite.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.14. Release

WF-3 is satisfied, since $\Sigma \rightarrow^* \Sigma'$ is well formed and according to **WF-3** the number of synchronization actions in it are finite.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.15. Fetch

WF-1: Since r and its variables are fetched from the heap and $\Sigma \rightarrow^* \Sigma'$ is well formed, according to **WFH-1** for each variable $r.f$ in r its value is the one written back by the last write-back action, acting on $r.f$, in that execution trace. As a result, **WF-12** is satisfied.

WF-3 is satisfied, since $\Sigma \rightarrow^* \Sigma'$ is well formed and according to **WF-3** the number of synchronization actions in it is finite.

WFH-2 and **WFH-4** are satisfied since the value of $r.f$ in the object cache is the one fetched from the last fetch action in the execution trace.

WFH-9 is satisfied, since the fetch value is added to the object cache of the core performing the action.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.16. WriteBack

WF-3 is satisfied, since $\Sigma \rightarrow^* \Sigma'$ is well formed and according to **WF-3** the number of synchronization actions in it is finite.

WF-13 and **WF-14**: Since $r.f$ is written back from the write buffer and $\Sigma \rightarrow^* \Sigma'$ is well formed, according to **WFH-3** its value in the write buffer is the one written by the last write action in that execution trace, which acts on $r.f$ and is performed by c. As a result, **WF-13** and **WF-14** are satisfied.

WF-20: Since $\Sigma \rightarrow^* \Sigma'$ is well-formed **WF-20** is satisfied for any pair of writes and the corresponding pair of their write-backs in it. As a result, we examine the cases where the second write w of the pair is the last write in the trace, which the write-back b at hand writes back. Given any

pair of write and write-back actions w' and b' in $\Sigma \rightarrow^* \Sigma'$ (if there exists one), where $w' \leq^d_{hb} w$, according to **WF-14** the write-back action b' writing back w' can only appear between the two writes $w' \leq^d_{hb} b' \leq^d_{hb} w$. Additionally, we know that $w \leq^d_{po} b$. As a result, $w' \leq^d_{hb} b' \leq^d_{hb} w \leq^d_{hb} b$ which satisfies **WF-20**.

WFH-1 is satisfied since the value of $r.f$ in the heap is the one written back by the last write-back action in the execution trace.

WFH-2 is satisfied since the value of $r.f$ in the object cache is the one written back by the last write-back action in the execution trace.

WFH-3 and **WFH-5** are satisfied since WriteBack just removes $r.f$ from the write buffer and does not introduce or restore another value in its place.

WFH-9 is satisfied, since the value is moved from the write buffer, of the core performing the action, to the object cache of the same core.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.17. Invalidate

WF-3 is satisfied, since $\Sigma \rightarrow^* \Sigma'$ is well formed and according to **WF-3** the number of synchronization actions in it is finite.

WF-15 is satisfied, since the first premise of Invalidate requires that the object being invalidated is present in the object cache. As a result, only cached variables are invalidated.

WFH-2 and **WFH-5** are satisfied since Invalidate just removes a value from the object cache and does not introduce or restore another value in its place.

WFH-9 is satisfied, since the value is removed from the object cache of the core performing the action.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 6.18. Start

WF-3 is satisfied, since $\Sigma \rightarrow^* \Sigma'$ is well formed and according to **WF-3** the number of synchronization actions in it are finite.

WF-9 is satisfied by Lemma 1 and the fact that in the local operational semantics there is no way to step to the start expression. The only start exception in the program is that of the main thread in the initial state.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

DJC's local operational semantics generates only well-formed execution traces. □

Lemma 7. *Lifting a well-formed execution trace from the local operational semantics to the global operational semantics preserves the well-formedness of the execution.*

Proof. In DJC the lifting is performed by LIFT. LIFT does not introduce new modifications to the memory state or new actions in the execution trace, other than those performed by the local operational semantics. As a result, since according to Lemma 6 the local operational semantics only generates well formed executions, lifting it to the global operational semantics preserves its well-formedness. □

Theorem 2. *DJC's operational semantics generates only well-formed execution traces. As a result, all executions performed by DiSquawk adhere to JDMM and consequently to JMM.*

Proof. We show, by induction on the number of steps, that for each well formed execution trace $\Sigma \rightarrow^* \Sigma'$, $\Sigma \rightarrow^* \Sigma' \rightarrow \Sigma''$, where $\rightarrow^*$ and $\rightarrow$ are reductions of the global operational semantics, is also well-formed.

For each case we omit well-formedness rules that do not correlate with the transition at hand, e.g., we do not argue about **WF-2** in the case of SPAWN since it does not act on a volatile variable.

Base case: Any execution trace $\Sigma \rightarrow \Sigma'$, is well-formed.

In DJC the execution starts with a single thread –the main thread– and the beginning of any execution trace is:
$\Sigma_{init} \rightarrow^* \{(r_t \mapsto \mathsf{VMThread}(\emptyset, \mathsf{spawned}))\}; \emptyset; \emptyset \vdash c\langle r_t, \mathsf{start}\rangle$
where $\rightarrow^*$ contains only transitions performing the initialization actions and their write-backs, for every variable in the execution trace, and $\Sigma_{init} \rightarrow^* \{(r_t \mapsto \mathsf{VMThread}(\emptyset, \mathsf{spawned}))\}; \emptyset; \emptyset \vdash c\langle r_t, \mathsf{start}\rangle$ is well-formed.
As a result, $\Sigma = \{(r_t \mapsto \mathsf{VMThread}(\emptyset, \mathsf{spawned}))\}; \emptyset; \emptyset \vdash c\langle r_t, \mathsf{start}\rangle$

In the global operational semantics,

$$\mathcal{H}; \vec{\mathcal{C}}; \vec{\mathcal{D}} \vdash T \xrightarrow[\vec{c}]{\vec{\alpha}} \mathcal{H}; \vec{\mathcal{C}}; \vec{\mathcal{D}} \vdash T$$

the interesting cases are LIFT and MIGRATE. SPAWN cannot step since its premises are not satisfied. BLOCKED does not change the state and for PARG there is no other thread in the context to step.

Case 2.1. LIFT

In the case of LIFT, the well-formedness of the execution is preserved according to Lemma 7.

Case 2.2. MIGRATE

In the case of MIGRATE the main thread is transferred to another core. The memory state remains as before and all well-formedness rules are satisfied.

WFE-2 is satisfied by MIGRATE's premises —there are no data in the write buffer.

WFH-7: Since $\Sigma \rightarrow^* \Sigma'$ is well formed and satisfies **WFH-7**, the thread at hand is spawned. MIGRATE transfers the thread at hand to a new core and resigns it from its previous core complying to **WFH-7**. As a result, **WFH-7** is satisfied.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

As a result, the theorem is true for the first transition of any program.

Inductive step: Given a well-formed execution trace $\Sigma \rightarrow^* \Sigma'$, $\Sigma \rightarrow^* \Sigma' \rightarrow \Sigma''$ is also well-formed.

We examine each case in the global operational semantics:

$$\mathcal{H}; \vec{\mathcal{C}}; \vec{\mathcal{D}} \vdash T \xrightarrow[\vec{c}]{\vec{\alpha}} \mathcal{H}; \vec{\mathcal{C}}; \vec{\mathcal{D}} \vdash T$$

and show that it satisfies the well-formedness rules.

Case 2.3. LIFT

In the case of LIFT, the well-formedness of the execution is preserved according to Lemma 7.

Case 2.4. SPAWN

WF-4: Since $\Sigma \rightarrow^* \Sigma'$ is well formed, according to **WF-4**, synchronization order is consistent with program order. The action at hand is placed after, according to the program order and the synchronization order, any actions in $\Sigma \rightarrow^* \Sigma'$. As a result the synchronization order remains consistent with the program order and **WF-4** is satisfied.

WFH-7 and **WFH-8**: The spawned thread is assigned to a single core and the old thread remains assigned to its core. The spawned thread also gets marked as spawned in order to forbid future re-spawns of the same thread (first and second premise of SPAWN). As a result, **WFH-7** and **WFH-8** are satisfied, since they are also satisfied in $\Sigma \rightarrow^* \Sigma'$.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 2.5. MIGRATE

WFE-2 is satisfied since it is satisfied in $\Sigma \rightarrow^* \Sigma'$ and in the new transition is satisfied by MIGRATE's premises —there are no data in the write buffer.

WFH-7: Since $\Sigma \rightarrow^* \Sigma'$ is well formed and satisfies **WFH-7**, the thread at hand is spawned. MIGRATE transfers the thread at hand to a new core and resigns it from its previous core complying to **WFH-7**. As a result, **WFH-7** is satisfied.

The rest of the rules are omitted since they do not correlate with the transition at hand and thus it is trivial to show that they are satisfied.

Case 2.6. BLOCKED

In the case of BLOCKED all well formed rules are satisfied since they where satisfied in $\Sigma \rightarrow^* \Sigma'$ and BLOCKED does not introduce any state modifications or new actions in the execution trace.

Case 2.7. PARG

WF-1: Since $\Sigma \rightarrow^* \Sigma'$ is well-formed, **WF-1** and **WFH-5** are true for it.

In the case of non-volatile reads the read of a variable $r.f$ sees the value written in the object cache or the write buffer of the core that performs the action (see FIELD and FIELDDIRTY), which according to **WFH-5** is the result of a write to $r.f$. Since the object caches and the write buffers

of different cores are disjoint **WF-1** and **WFH-5** are true for the unions of the object caches and the write buffers as well.

In the case of volatile reads the read of a volatile variable $r.f$ sees the value written in the heap (see VolatileRead), which according to **WFH-5** is the result of a write to $r.f$. By induction on the eighth premise of ParG, only one core may modify the heap. Since VolatileRead modifies it, then there are no writes to the heap executed in parallel with VolatileRead and the latter will see the last write to $r.f$, according to **WFH-6**, since $\Sigma \rightarrow^* \Sigma'$ is well-formed. As a result, **WF-1** is satisfied.

WF-2: By induction on the eighth and ninth premise of ParG, every thread steps through the Lift, Spawn, or Migrate. According to Lemma 7 every step performed by Lift is well formed and thus satisfies **WF-2**. Spawn and Migrate do not act on volatile variables, so they always preserve **WF-2**. As a result **WF-2** is satisfied.

WF-3: Since $\Sigma \rightarrow^* \Sigma'$ is well-formed, **WF-3**, **WFH-7**, and **WFH-8** are true for it, as a consequence, the number of spawned threads in the system is finite, since the spawn action is a synchronization action. Additionally by **WFH-7** each spawned thread is assigned to a single core and by **WFH-8** each thread appears only on a single set of threads. As a result, the number of synchronization actions that can be performed in parallel is bound by the number of the spawned threads in the system. As a result, **WF-3** is satisfied.

WF-4: Since $\Sigma \rightarrow^* \Sigma'$ is well-formed, **WF-4**, **WFH-7**, and **WFH-8** are true for it. By **WFH-7** each spawned thread is assigned to a single core and by **WFH-8** each thread appears only on a single set of threads. As a result, a thread may not step in parallel with itself, and any action is appended to the program order. However, in the case of synchronization actions, F, I, J, and *Ird* may step in parallel with other synchronization actions, so they are not actually ordered with those actions. Nevertheless, any arbitrary ordering of them does not break the consistency of the synchronization order with the program order, since only a single action maybe performed by each thread in every transition. As a result, **WF-4** is satisfied.

WF-5: Since $\Sigma \rightarrow^* \Sigma'$ is well-formed, **WF-5** is true for it. Additionally, only a single lock operation may be performed at any parallel transition, since lock operations modify the heap and according to the eighth and ninth premises of ParG only one set of threads is allowed to modify it. By induction on the eighth premise we conclude that only a single thread may modify the heap, through Lift. Since according to Lemma 7 Lift preserves the well-formedness, **WF-5** is satisfied by ParG as well.

WF-6: Since $\Sigma \rightarrow^* \Sigma'$ is well-formed, **WF-6**, **WFH-7**, and **WFH-8** are true for it. By **WFH-7** each spawned thread is assigned to a single core and by **WFH-8** each thread appears only on a single set of threads. As a result, a thread may not step in parallel with itself, and any action is appended to the program order. By induction on the eighth and ninth premise of ParG, every thread steps through the Lift, Spawn, or Migrate. According to Lemma 7 every step performed by Lift is well formed and thus satisfies **WF-6**. Spawn and Migrate do not perform any reads, so they always satisfy **WF-6**.

WF-7: By induction on the eighth and ninth premise of PARG, every thread steps through the LIFT, SPAWN, or MIGRATE. According to Lemma 7 every step performed by LIFT is well formed and thus satisfies **WF-7**. SPAWN and MIGRATE do not correspond to volatile actions, so they always preserve **WF-7**. Additionally, by induction on the eighth premise of PARG, only one core may modify the heap. Since volatile actions modify it, then there are no other volatile actions executed in parallel with VOLATILEREAD and the latter will see the last write to $r.f$, according to **WFH-6**, since $\Sigma \rightarrow^* \Sigma'$ is well-formed. As a result, **WF-7** is satisfied.

WF-8: The happens-before order is the transitive closure of the synchronizes-with order and the program order.

As we show for **WF-6**, since $\Sigma \rightarrow^* \Sigma'$ is well-formed, **WF-6**, **WFH-7**, and **WFH-8** are true for it. By **WFH-7** each spawned thread is assigned to a single core and by **WFH-8** each thread appears only on a single set of threads. As a result, a thread may not step in parallel with itself, and any action is appended to the program order.

Regarding the synchronizes-with order, we examine each pair and show that both actions of a pair can not step in parallel. Note that we omit the last pair regarding finalization and the constructor of the object, since we do not model finalization in our semantics.

- $In \leq^d_{sw} S$: According to Lemma 1 initialization actions are performed before the start of the program.
- $Vw \leq^d_{sw} Vr$: Since both Vw and Vr modify the heap they cannot step in parallel. By induction on the eighth premise of PARG, only one core may modify the heap.
- $U \leq^d_{sw} L$: Since both U and L modify the heap they cannot step in parallel. By induction on the eighth premise of PARG, only one core may modify the heap.
- $Sp \leq^d_{sw} S$: Since both Sp and S modify the heap they cannot step in parallel. By induction on the eighth premise of PARG, only one core may modify the heap.
- $Fi \leq^d_{sw} J$: Since Fi modifies the heap and J reads it, although they are allowed to step in parallel by PARG, the third premise of JOIN would not be satisfied, as a result they never step in parallel.
- $Ir \leq^d_{sw} Ird$: Since Ir modifies the heap and Ird reads it, although they are allowed to step in parallel by PARG, the third premise of INTERRUPTEDT would not be satisfied, as a result they never step in parallel.

As a result, **WF-8** is satisfied.

WF-9: According to Lemma 1 every initialization action in the execution trace happens-before the start of the program. Additionally, since $\Sigma \rightarrow^* \Sigma'$ is well-formed, **WF-9** is true for it and start actions modify the heap to mark the thread as started. By induction on the eighth premise of PARG, only one core may modify the heap. As a result, there can only be a single start action in a parallel transition and that will be evaluated by LIFT that according to Lemma 7 preserves the well-formedness of the execution. That is, in the execution trace preceding the transition at hand all thread actions where ordered after the start action of the corresponding thread according to the happens-before order. Additionally, the same is true for the local execution trace of the core that

starts the thread. As a result the only case that remains to be examined is that of running a start action in parallel with another action of that thread. Since $\Sigma \rightarrow^* \Sigma'$ is well-formed, **WF-6**, **WFH-7**, and **WFH-8** are true for it. By **WFH-7** each spawned thread is assigned to a single core and by **WFH-8** each thread appears only on a single set of threads. As a result, a thread may not step in parallel with itself, and any action is appended to the program order. As a result **WF-9** is satisfied.

WF-10: By induction on the eighth and ninth premise of ParG, every thread steps through the Lift, Spawn, or Migrate, and specifically reads step through Lift. According to Lemma 7 every step performed by Lift is well formed and thus satisfies **WF-10**. As a result, there is a write or fetch action, acting on the same variable as the read, earlier in the execution trace.

WF-11: By induction on the eighth and ninth premise of ParG, every thread steps through the Lift, Spawn, or Migrate, and specifically reads step through Lift. According to Lemma 7 every step performed by Lift is well formed and thus satisfies **WF-11**. As a result for each non-volatile read there is no invalidation, update, or overwrite of the variable's value between the read and fetch or write that cached it. By **WFH-7** on $\Sigma \rightarrow^* \Sigma'$ each spawned thread is assigned to a single core, by **WFH-8** each thread appears only on a single set of threads, and by **WFH-9** the contents of the object cache and the write buffer of each core are altered only by that core. As a result, since **WF-9** holds by Lift it is also true for the whole transition, since the core performing the read is the only that can alter the object cache and the write buffer, and it cannot perform another action in parallel with itself (first premise of ParG), to invalidate, update, or overwrite the value.

WF-12: See Lemma 2.

WF-13: By induction on the eighth and ninth premise of ParG, every thread steps through the Lift, Spawn, or Migrate, and specifically write-backs step through Lift. According to Lemma 7 every step performed by Lift is well formed and thus satisfies **WF-13**. As a result, there is a write, to the corresponding variable being written-back, earlier in the execution trace.

WF-14: By induction on the eighth and ninth premise of ParG, every thread steps through the Lift, Spawn, or Migrate, and specifically write-backs step through Lift. According to Lemma 7 every step performed by Lift is well formed and thus satisfies **WF-14**. By **WFH-14** on $\Sigma \rightarrow^* \Sigma'$ each spawned thread is assigned to a single core, by **WFH-8** each thread appears only on a single set of threads, and by **WFH-9** the contents of the object cache and the write buffer of each core are altered only by that core. As a result, since **WF-14** holds by Lift it is also true for the whole transition, since the core performing the write-back is the only that can alter the object cache and the write buffer and it cannot perform a write action in parallel with itself (first premise of ParG).

WF-15: By induction on the eighth and ninth premise of ParG, every thread steps through the Lift, Spawn, or Migrate, and specifically invalidations step through Lift. According to Lemma 7 every step performed by Lift is well formed and thus satisfies **WF-15**. By **WFH-15** on $\Sigma \rightarrow^* \Sigma'$ each spawned thread is assigned to a single core, by **WFH-8** each thread appears only on a single set of threads, and by **WFH-9** the contents of the object cache and the write buffer of each core are altered only by that core. As a result, since **WF-15** holds by Lift it is also true for the whole transition, since the core performing the invalidation is the only that can alter the object

cache and the write buffer and it cannot perform an invalidation action in parallel with itself (first premise of ParG).

WF-16: By induction on the eighth and ninth premise of ParG, every thread steps through the Lift, Spawn, or Migrate, and specifically reads step through Lift. According to Lemma 7 every step performed by Lift is well formed and thus satisfies **WF-15**. By **WFH-16** on $\Sigma \rightarrow^* \Sigma'$ each spawned thread is assigned to a single core, by **WFH-8** each thread appears only on a single set of threads, and by **WFH-9** the contents of the object cache and the write buffer of each core are altered only by that core. As a result, since **WF-16** holds by Lift it is also true for the whole transition, since the core performing the read is the only that can alter the object cache and the write buffer and it cannot perform a write-back action in parallel with itself (first premise of ParG).

WF-17: See Lemma 3.

WF-18: See Lemma 4.

WF-19: See Lemma 5.

WF-20: By **WF-20** on $\Sigma \rightarrow^* \Sigma'$ we know that the happens-before order between two writes is consistent with the happens-before order of their write-backs. As a result we only need to examine new write-back actions. By induction on the eighth premise of ParG, only one core may modify the heap. As a result there can only be one write-back in the transition at hand, which cannot break the happens before order consistency. As a result **WF-20** is satisfied.

WFE-1: By induction on the eighth and ninth premise of ParG, every thread steps through the Lift, Spawn, or Migrate, and specifically reads step through Lift. According to Lemma 7 every step performed by Lift is well formed and thus satisfies **WFE-1**. That is, there is a corresponding fetch action between thread migration and every read action performed by the core that the corresponding thread migrated to. As a result, only the parallel evaluation of a migration and a read action could break this rule. However, since those two actions should be performed by the same thread this is not possible. By **WFH-7** on $\Sigma \rightarrow^* \Sigma'$ each spawned thread is assigned to a single core, and by **WFH-8** each thread appears only on a single set of threads, and by **WFH-9** the contents of the object cache and the write buffer of each core are altered only by that core. As a result, since **WFE-1** holds by Lift it is also true for the whole transition, since the core performing the read is the only that can step the thread at hand and it cannot perform another action in parallel with itself (first premise of ParG).

WFE-2: By induction on the eighth and ninth premise of ParG, every thread steps through the Lift, Spawn, or Migrate, and specifically migrations step through Migrate. **WFE-2** is satisfied by the premises of Migrate. That is, at migration actions there are no dirty data at the *old* core, in the two transitions in isolation. As a result, only the parallel evaluation of a migration and a write action at the *old* core could break this rule. However, since those two actions should be performed by the same thread this is not possible. By **WFH-7** on $\Sigma \rightarrow^* \Sigma'$ each spawned thread is assigned to a single core, and by **WFH-8** each thread appears only on a single set of threads, and by **WFH-9** the contents of the object cache and the write buffer of each core are altered only

by that core. As a result, since **WFE-2** holds by MIGRATE it is also true for the whole transition, since the core performing the migration is the only that can step the thread at hand and it cannot perform another action in parallel with itself (first premise of PARG).

WFH-1: By induction on the eighth and ninth premise of PARG, every thread steps through the LIFT, SPAWN, or MIGRATE, and specifically write-backs step through LIFT. According to Lemma 7 every step performed by LIFT is well formed and thus satisfies **WFH-1**. Since $\Sigma \rightarrow^* \Sigma'$ is well-formed we also know that it satisfies **WFH-1** as well. As a result we only need to examine new write-back actions. By induction on the eighth premise of PARG, only one core may modify the heap, thus there can only be one write-back in the transition at hand. As a result **WFH-1** is satisfied

WFH-2: By induction on the eighth and ninth premise of PARG, every thread steps through the LIFT, SPAWN, or MIGRATE, and specifically fetches and write-backs step through LIFT. According to Lemma 7 every step performed by LIFT is well formed and thus satisfies **WFH-2**. By **WFH-7** on $\Sigma \rightarrow^* \Sigma'$ each spawned thread is assigned to a single core, and by **WFH-8** each thread appears only on a single set of threads, and by **WFH-9** the contents of the object cache and the write buffer of each core are altered only by that core. As a result, since **WFH-2** is true for the single step it is also true for the whole transition, since the core performing the fetch or write-back is the only that can modify the object cache and it cannot perform another action in parallel with itself (first premise of PARG).

WFH-3: By induction on the eighth and ninth premise of PARG, every thread steps through the LIFT, SPAWN, or MIGRATE, and specifically writes step through LIFT. According to Lemma 7 every step performed by LIFT is well formed and thus satisfies **WFH-3**. By **WFH-7** on $\Sigma \rightarrow^* \Sigma'$ each spawned thread is assigned to a single core, and by **WFH-8** each thread appears only on a single set of threads, and by **WFH-9** the contents of the object cache and the write buffer of each core are altered only by that core. As a result, since **WFH-3** is true for the single step it is also true for the whole transition, since the core performing the write is the only that can modify the write buffer and it cannot perform another action in parallel with itself (first premise of PARG).

WFH-4: By induction on the eighth and ninth premise of PARG, every thread steps through the LIFT, SPAWN, or MIGRATE. According to Lemma 7 every step performed by LIFT is well formed and thus satisfies **WFH-4**. SPAWN and MIGRATE are of no interest since they do not alter the object cache. As a result, **WFH-4** is also satisfied in the whole transition since it is satisfied by every step in the transition.

WFH-5: By induction on the eighth and ninth premise of PARG, every thread steps through the LIFT, SPAWN, or MIGRATE. According to Lemma 7 every step performed by LIFT is well formed and thus satisfies **WFH-4**. SPAWN and MIGRATE are of no interest since they do not alter the values of any variables. As a result, **WFH-5** is also satisfied in the whole transition since it is satisfied by every step in the transition.

WFH-6: Since $\Sigma \rightarrow^* \Sigma'$ is well-formed we also know that it satisfies **WFH-6** as well. As a result we only need to examine new volatile writes. By induction on the eighth premise of PARG,

only one core may modify the heap, thus there can only be one volatile write in the transition at hand. As a result **WFH-6** is satisfied

WFH-7 and **WFH-8**: Since $\Sigma \to^* \Sigma'$ is well-formed we also know that it satisfies **WFH-7** and **WFH-8** as well. As a result we only need to examine new spawns. By induction on the eighth premise of ParG, only one core may modify the heap, thus there can only be one spawn in the transition at hand. By induction on the eighth premise of ParG, we see that a spawn can only step through Spawn. The spawned thread is assigned to a single core and the old thread remains assigned to its core. The spawned thread also gets marked as spawned in order to forbid future re-spawns of the same thread (first and second premise of Spawn). As a result, **WFH-7** and **WFH-8** are satisfied, since they are also satisfied in $\Sigma \to^* \Sigma'$.

WFH-9: Since **WFH-9** is satisfied by $\Sigma to^* \Sigma'$ we examine how the current transition alters object caches and write buffers. By induction on the eighth and ninth premise of ParG, we see that all actions altering the object caches and write buffers are evaluated by Lift. According to Lemma 7 every step performed by Lift is well formed and thus satisfies **WFH-9**. Since **WFH-9** is satisfied by Lift, it is also true for the whole transition, since the object caches and write buffers are disjoint.

□