DiSquawk: 512 cores, 512 memories, 1 JVM

Foivos S. Zakkak
FORTH-ICS
Heraklion, Crete, Greece
zakkak@ics.forth.gr

Polyvios Pratikakis
FORTH-ICS
Heraklion, Crete, Greece
polyvios@ics.forth.gr

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.

This paper presents 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 evaluation results demonstrating 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.

CCS Concepts

•**Theory of computation** → **Operational semantics;** •**Software and its engineering** → *Memory management; Virtual machines;*

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, 34, 7], despite several attempts to design less costly or more scalable coherence protocols [23, 25].

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.

This paper presents an implementation of the Java Memory Model (JMM) [22] 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, demonstrating that our implementation scales with the number of cores. To prove our implementation's adherence to the JMM, 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.

PPPJ '16, August 29-September 02, 2016, Lugano, Switzerland

ISBN 978-1-4503-4135-6/16/08…$15.00

DOI: http://dx.doi.org/10.1145/2972206.2972212

- 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 [24, 4] or software distributed shared memory [35, 33, 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 [22] is agnostic about such operations, we base our work on the Java Distributed Memory Model (JDMM) [36].

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.* [26] 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 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.

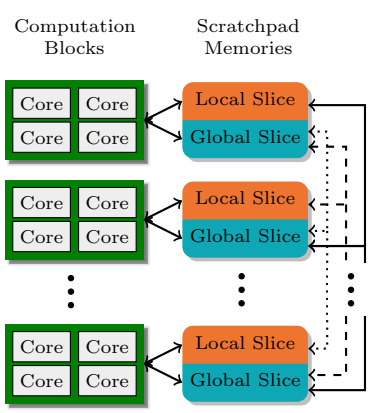

Figure 1: Memory abstraction.

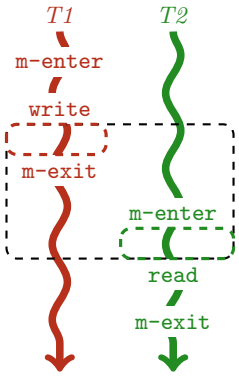

Figure 2: Time window example.

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 [36, §3 and §4.2]; we briefly present them here. **WF-1**–**WF-9** were first introduced in [22].

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.

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 [36, §5.2]. That is:

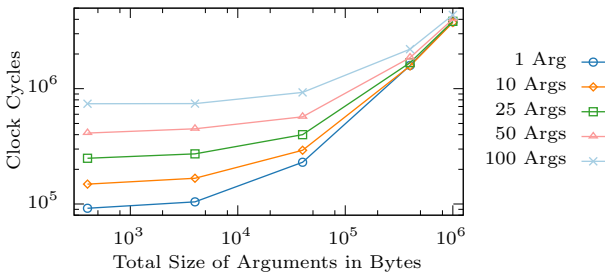

Figure 3: Performance impact of arguments size

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.

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* [18] 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 [24] —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 variable 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. [28] 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^{TM}-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 acquired then a write is performed, and a different monitor is now acquired. 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 this reduction of flexibility in 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 [32] 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 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 array block accessing the previous and next neighboring blocks. To ensure 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 shows the speedup for the benchmarks on DiSquawk running on the formic-cube and HotSpot running on a 4-chip NUMA machine with 16 cores per chip, totaling 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. We perform this comparison on up to 64 cores, since we do not have access to a

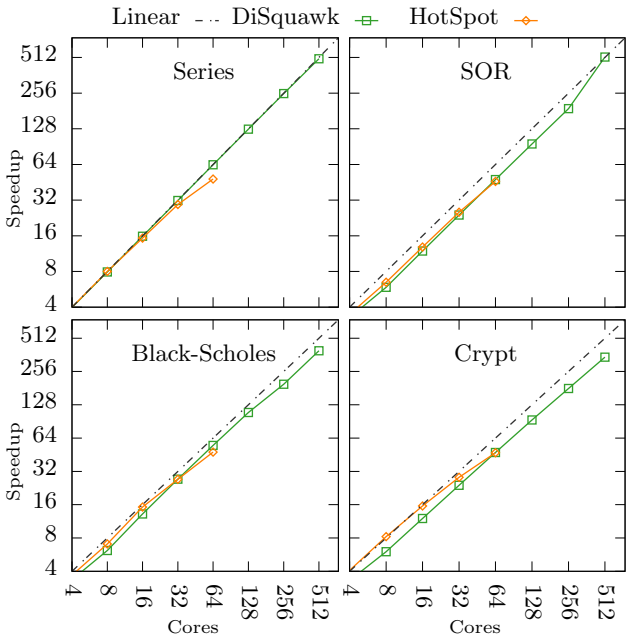

Figure 4: Speedup Results

cache-coherent architecture with more cores. Although we cannot compare with HotSpot beyond the limit of 64 cores, we present results from DiSquawk to show that it keeps scaling with the number of cores. 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 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.

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. In shared-memory cache coherent architectures Java monitors are usually implemented using atomic operations like *compare and swap*, 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 Java monitors on such architectures we use 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 about a single monitor end up in the corresponding synchronization manager's hardware message queue, from where they are handled by the synchronization manager in the order they arrived. We thus delegate the synchronization of requests to the architecture's network-on-chip and provide mutual exclusion via 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 sent 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 [27, §3.1.4] —meaning 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.

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

Figure 5: Abstract syntax of DJC

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 $\mathsf{class}\ C(\overrightarrow{f:\tau})\{e\}\{\vec{M}\}$ 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 $m(\overrightarrow{x:\tau})\{\mathsf{return}\ e;\} : \tau$ where m is the method's name; $\overrightarrow{x:\tau}$ is the set of formal arguments; e is the 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 $\mathsf{new}\ C(\vec{e})$; 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 $\mathsf{let}\ x : \tau = e\ \mathsf{in}\ e$. Note that the evaluation of e may have side-effects. Conditional expressions are expressed as $\mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$; 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 $e.\mathsf{monitorenter}$ and $e.\mathsf{monitorexit}$, 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.

A thread instance is defined as $c\langle r, \mathsf{start}\rangle$ or $c\langle r, e\rangle$, where

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
$\vec{X} \downarrow \vec{x}$	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

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 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 caches per core $\vec{\mathcal{C}}$, and the write buffers per core $\vec{\mathcal{D}}$. The heap is a map from references r to objects o and their monitor l. 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 $\mathcal{D}$. The object cache $\mathcal{C}$ is a map from references r to objects o. The write buffer $\mathcal{D}$ is a map from object fields $r.f$ to values v.

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

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

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

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

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

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

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

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

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

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

$$[\textsc{New}]\ \frac{\begin{array}{cc} \multicolumn{2}{c}{r - fresh} \\ \mathcal{H}(r) = C(\overrightarrow{f \mapsto 0}) & \mathsf{class}\ C(\overrightarrow{f:\tau})\{e\}\{\vec{M}\} \in J \end{array}}{\begin{array}{c} \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, \mathsf{new}\ C(\vec{v})\rangle \rightarrow \\ \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, \mathsf{let}\ \_ : Unit = e[\vec{v}/\vec{f}][r/\mathsf{this}]\ \mathsf{in}\ r\rangle \end{array}}$$

Figure 7: Semantics of Local Operations

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 included in the extended version of this paper [37], 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.

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 FIELD and FIELDDIRTY. FIELD handles non-volatile field accesses when the field is in the object cache while FIELDDIRTY handles non-volatile field accesses when the field is not 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 *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 copy of the field in its object cache, and the cached value is v. The last premise requires that the field is not 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 $dom\,(X)$ 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 in the write buffer. The only difference from FIELD is that we require f to be in the write buffer and get its value from there instead of the object cache.

ASSIGN handles non-volatile field writes. 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 $k \notin dom\,(X')$ 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*" [22, §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 $r - fresh$ to show that there is no other reference to that object already.

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. An invalidation results in the removal of the object referenced by r from the object cache, $\mathcal{C}$, of the core executing the invalidation. Note that invalidations do not affect the write buffer. 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. FINISH handles the completion of a thread. Note that a thread reaches completion when its thread body is equal to the unit value $()$. FINISH, as a release action, requires the write buffer to be empty, and changes the state of the thread.

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

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

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

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

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

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

Figure 8: Semantics of Implicit Operations

4.2.3 Semantics of Synchronization Operations

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 9: Semantics of Synchronization Operations

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. Thus, any concurrent actions operating on cached data used in the critical section would be the result of a data-race, meaning that the program is not DRF. Then 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. Racy accesses are not guaranteed to see the latest write if their thread 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 interrupted 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 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. In such cases

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

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

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

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

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

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

Figure 10: Global Semantics

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.

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.

Spawn handles thread spawns, *i.e.* `Thread.start()` invocations. 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. 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'. 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 requires the two sets of cores that perform a step in the parallel transition to be disjoint. This way we model that each core is running a single thread and performs a single step each time. Additionally, its eighth and ninth premise only allow 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 threads may write to 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. The extended version of this paper contains a detailed proof of adherence [37]. Intuitively, the correctness property can be expressed as:

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

To prove Theorem 1, 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.

The proof first shows DJC's local operational semantics generates only well-formed execution traces. We then show that lifting a well-formed execution trace from the local operational semantics to the global operational semantics preserves the well-formedness of the execution. Last, we show that the global operational semantics preserves the well-formedness of the execution, thus DJC's operational semantics generates only well-formed execution traces.

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 [24]. 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 [26, 31], and ARM [3] processors, without focusing on a specific language semantics or memory model. Sarkar *et al.* [30] 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.* [29] 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 Java.

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 JMM 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 architectures at hand.

Jagadeesan *et al.* [15] also describe an operational semantics for JMM. 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 [32]. 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.

Acknowledgments

We would like to acknowledge the support of the European Commission under the 7th Framework Programs through the *EuroServer* (FP7-ICT-610456) project. We would also like to express our deepest gratitude to Christi Symeonidou for her valuable review on this work.

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

[23] 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.

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

[25] 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.

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

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

[28] 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.

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

[30] 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*, 2012.

[31] 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.

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

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

[34] 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.

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

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

[37] F. S. Zakkak and P. Pratikakis. DiSquawk: 512 cores, 512 memories, 1 JVM. Technical Report TR-470, FORTH-ICS, June 2016.

[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.