Java™ on Scalable Memory Architectures

by

Foivos Zakkak

University of Crete

Department of Computer Science

Dissertation

submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree

Java™ on Scalable Memory Architectures

Foivos Zakkak

October 2016

University of Crete
Computer Science Department

Doctoral Committee:

Angelos Bilas, Professor, University of Crete, (Advisor)
Polyvios Pratikakis, Assistant Researcher, ICS-FORTH, (Advisor)
Manolis G.H. Katevenis, Professor, University of Crete
Panagiota Fatourou, Assistant Professor, University of Crete
Dimitrios S. Nikolopoulos, Professor, Queen's University of Belfast
Evangelos Markatos, Professor, University of Crete
Vasileios Koutavas, Assistant Professor, Trinity College Dublin

This work was performed at the Computer Architecture and VLSI Systems (CARV) Laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH), and was financially supported by a FORTH-ICS scholarship, including funding by the European Social Fund (ESF) and Greek National Resources, through the *GreenVM* (#1643) project, which has been implemented under the *ARISTEIA* action of the "Operational Programme on Education and Lifelong Learning", and the European Commission under the 7th Framework Programs through the *EuroServer* (FP7-ICT-610456) project.

This thesis was typeset in Liberation Sans and Liberation Mono
with the help of KOMA-Script and LuaLaTeX.
Figures were generated using TikZ and Inkscape.
Plots were generated using gnuplot.
Source files were edited with GNU Emacs.

University of Crete
Department of Computer Science

Java™ on Scalable Memory Architectures

Dissertation submitted by

Foivos Zakkak

in partial fulfillment of the requirements for the
Doctor of Philosophy degree in Computer Science

Author:

Foivos Zakkak, University of Crete

Examination Committee:

Angelos Bilas, Professor, University of Crete

Polyvios Pratikakis, Assistant Researcher, ICS-FORTH

Manolis G.H. Katevenis, Professor, University of Crete

Panagiota Fatourou, Assistant Professor, University of Crete

Dimitrios S. Nikolopoulos, Professor, Queen's University of Belfast

Evangelos Markatos, Professor, University of Crete

Vasileios Koutavas, Assistant Professor, Trinity College Dublin

Departmental approval:

Antonis Argyros, Professor, Director of Graduate Studies

Heraklion, October 2016

Acknowledgements

This thesis would not be possible without the contributions of a number of people. I deeply thank my family and friends for being there for me. I specially thank my parents for their support all these years and making me the person I am today, my sister for creating the logo of this thesis and setting an example for me to follow (and compete with), and Marilena for respecting my dislike of talking about my work at home, for brightening up my days, and for much more.

I thank everyone who took part in the discussions about the design decisions and implementation difficulties regarding the JVM presented in this thesis. Namely, those are[1] Panagiota Fatourou, Nikolaos Kallimanis, Lena Kanellou, Manolis Katevenis, Spyros Lyberis, Vassilis Papaefstathiou, Polyvios Pratikakis, Antonis Psathakis, Christi Symeonidou, and Evaggelos Vasilakis.

I especially thank Polyvios Pratikakis, Angelos Bilas, Manolis Katevenis, Panagiota Fatourou, and Dimitrios Nikolopoulos for their advice and support.

Special thanks also go to Christi Symeonidou for reviewing all my papers that got published during the period of this PhD thesis, and Antonis Psathakis for being my eyes on the Formic-Cube, and sometimes my hands as well, when I was working remotely and I needed help to debug link failures.

Additionally, I thank all those who contributed in the implementation of the Formic 520-core prototype, which was used as the evaluation platform of this thesis. Especially, Spyros Lyberis who, apart from having a major role in the hardware design process, also developed the core C libraries for Formic. I also thank Manolis Marazakis and Michalis Ligerakis for their support and contribution in the administration of the various systems used in this thesis.

Last but not least, I would like to thank the free and open source community for providing the world with so many great tools, without which this thesis would not have been possible.

Funding: The work presented in this thesis was performed at the Computer Architecture and VLSI Systems (CARV) laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH), and it was financially supported by a FORTH-ICS scholarship, including funding by:

[1] in alphabetical order

1. the European Social Fund (ESF) and Greek National Resources, through the *GreenVM* (#1643) project, which has been implemented under the *ARISTEIA* action of the “Operational Programme on Education and Lifelong Learning”;
2. and the European Commission under the 7th Framework Programs through the *EuroServer* (FP7-ICT-610456) project.

Abstract

The beginning of the new millennium signaled the need for new ways to keep improving the performance of processors. The approach of increasing the frequency as transistors got smaller and smaller is no longer viable due to the increased power-leakage and heat generation. In an effort to maintain the performance increase steady despite the difficulties, processor manufacturers turned to multi-core computing. This way the industry keeps taking advantage of the smaller transistors by packing more compute elements in the same area, rather than making the circuits more complex and increasing the frequency. However, as the number of cores increases new challenges come up. One such challenge regards cache-coherence. Managing hardware caches and keeping the data coherent across them is become more and more complex and energy inefficient as the number of cores grow. To tackle this issue hardware architects are experimenting with modular non cache coherent and partially coherent architectures. Such architectures delegate the memory coherency to the software.

In this thesis we study how high productivity languages can be run efficiently on such architectures. 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 process virtual machines, like the Java virtual machine, to provide consistent behavior across different platforms. We focus our work on the Java virtual machine since it is one of the most popular and widely studied process virtual machines on top of which tens of languages are being implemented, with the most distinguishable being Java and Scala.

Java virtual machine implementations need to adhere to the Java language specifications and the Java memory model. In this thesis we study the Java memory model and present an extension of it that exposes explicit memory transfers between caches. This extension eases the process of arguing about the adherence of a Java virtual machine targeting a non cache coherent architecture by providing explicit rules regarding the ordering of memory transfers in respect to other operations in an Java execution. We sketch the proof that our extension complies to the original Java memory model and allows the same optimizations.

We present a Java virtual machine design targeting non cache coherent and partially coherent architectures. Our design aims to minimize the number of memory transfers and messages exchanged while still adhering to the Java memory model. Our design also takes advantage of partial coherence by sharing some structures across different cores on the same coherence island. Based on our design we implement a Java virtual

machine and evaluate it on an emulator of a non cache coherent architecture. The results show that our implementation scales up to 500 of cores and its scalability is comparable to that of the state of the art Java virtual machine running on a cache-coherent architecture.

Last but not least we model our implementation in the operational semantics of a Java core calculus that we define for this purpose. We then prove that these operational semantics produce only well-formed executions according to the Java memory model. Since the operational semantics model our implementation we argue that the latter also produces only well-formed executions, thus it adheres to the Java memory model.

Περίληψη

Η αρχή της χιλιετίας που διανύουμε σήμανε την ανάγκη αναζήτησης νέων μεθόδων για τη συνέχιση της βελτίωσης της απόδοσης των επεξεργαστών. Η μέθοδος της αύξησης της συχνότητας των επεξεργαστών καθώς τα τρανζίστορ γίνονται ολοένα και μικρότερα δεν είναι πλέον εφικτή, λόγο της ολοένα αυξανόμενης διαρροής ρεύματος και παραγωγής θερμότητας. Στην προσπάθεια τους να διατηρήσουν τους ρυθμούς βελτίωσης της απόδοσης των επεξεργαστών, οι κατασκευαστές επεξεργαστών στράφηκαν προς τον σχεδιασμό πολυπύρηνων επεξεργαστών. Με αυτό τον τρόπο η βιομηχανία εξακολουθεί να εκμεταλλεύεται το ολοένα μικρότερο μέγεθος των τρανζίστορ, περιλαμβάνοντας περισσότερα επεξεργαστικά στοιχεία στον ίδιο χώρο αντί να αυξάνουν την πολυπλοκότητα των κυκλωμάτων και την συχνότητα εναλλαγής καταστάσεων. Καθώς όμως το πλήθος των πυρήνων αυξάνεται εμφανίζονται νέες προκλήσεις. Μία τέτοια πρόκληση αφορά την παροχή συνέπειας μνήμης. Η διαχείριση των κρυφών μνημών και η διατήρηση συνεπών αντιγραφών σε αυτές γίνεται ολοένα και πιο περίπλοκη και ενεργειακά μη αποδοτική καθώς το πλήθος των πυρήνων αυξάνεται. Για να αντιμετωπίσουν αυτό το πρόβλημα, οι αρχιτέκτονες υπολογιστών πειραματίζονται με νέες αρχιτεκτονικές που παρέχουν συνέπεια μνήμης μόνο μεταξύ ενός υποσυνόλου των κρυφών μνημών ή και καθόλου. Αυτές οι αρχιτεκτονικές αναθέτουν τη διαχείριση μνήμης στο λογισμικό.

Στη διατριβή αυτή μελετάμε πως μπορούν να τρέξουν γλώσσες προγραμματισμού υψηλού επιπέδου σε τέτοιες αρχιτεκτονικές. Οι γλώσσες προγραμματισμού υψηλού επιπέδου, όπως η Java, είναι σχεδιασμένες να αποκρύπτουν τις λεπτομέρειες της αρχιτεκτονικής από τους προγραμματιστές. Με αυτόν τον τρόπο επιτρέπουν στους προγραμματιστές να επικεντρωθούν στην υλοποίηση των αλγορίθμων τους και όχι στην διαχείριση της εκάστοτε αρχιτεκτονικής. Οι γλώσσες προγραμματισμού υψηλού επιπέδου βασίζονται σε εικονικές μηχανές για να παρέχουν τα ίδια αποτελέσματα σε διαφορετικές αρχιτεκτονικές. Στην εργασία μας επικεντρωνόσαστε στην εικονική μηχανή της Java. Η εικονική μηχανή της Java είναι μία από τις πιο δημοφιλείς και πολυμελετημένες εικονικές μηχανές η οποία υλοποιεί δεκάδες γλώσσες προγραμματισμού, από τις οποίες ξεχωρίζουν κυρίως η Java και η Scala.

Οι υλοποιήσεις εικονικών μηχανών Java πρέπει να συμμορφώνονται υπακούν στον ορισμό της γλώσσας προγραμματισμού Java (Java language specification) και στο μοντέλο μνήμης της (Java Memory Model). Στην εργασία αυτή μελετάμε το μοντέλο μνήμης της Java και παρουσιάζουμε μία επέκταση του που εκφράζει ρητά τις μεταφορές δεδομένων μεταξύ των κρυφών μνημών. Η επέκταση αυτή παρέχει ρητούς κανόνες που αφορούν

τις μεταφορές δεδομένων μεταξύ των κρυφών μνημών οι οποίοι διευκολύνουν την επιχειρηματολογία σχετικά με την συμμόρφωση μίας εικονικής μηχανής Java στο μοντέλο μνήμης της Java. Ακόμη αποδεικνύουμε ότι η επέκταση αυτή είναι συμβατή με το αρχικό μοντέλο μνήμης της Java και επιτρέπει τις ίδιες βελτιστοποιήσεις κώδικα.

Επιπροσθέτως σχεδιάζουμε μία εικονική μηχανή Java για αρχιτεκτονικές με μερική ή μηδενική συνέπεια μνήμης. Ο σχεδιασμός μας στοχεύει στην ελαχιστοποίηση των μεταφορών δεδομένων και ανταλλαγών μηνυμάτων που χρειάζονται για τη συμμόρφωση στο μοντέλο μνήμης της Java. Ο σχεδιασμός μας εκμεταλλεύεται αρχιτεκτονικές με μερική συνέπεια μνήμης χρησιμοποιώντας μερικές κοινές δομές μεταξύ πυρήνων που βρίσκονται στην ίδια συστάδα πυρήνων με συνέπεια μνήμης. Βασιζόμενοι στον σχεδιασμό αυτό υλοποιούμε μία εικονική μηχανή Java και την αξιολογούμε τρέχοντας πειράματα σε έναν εξομοιωτή αρχιτεκτονικής χωρίς συνέπεια μνήμης. Τα αποτελέσματα δείχνουν ότι η υλοποίηση μας κλιμακώνεται μέχρι και σε 500 πυρήνες και η κλιμακωσιμότητα της είναι συγκρίσιμη με αυτή της πιο πρόσφατης εικονικής μηχανής Java (HotSpot VM) σε μία αρχιτεκτονική με συνέπεια μνήμης.

Τέλος εκφράζουμε με μαθηματικό τρόπο την υλοποίησή μας και αποδεικνύουμε ότι παράγει μόνο σωστές εκτελέσεις, σύμφωνα με το μοντέλο μνήμης και τον ορισμό της γλώσσας.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1.

Introduction

In 1965 Gordon Moore predicted that the number of components in CPUs would double roughly every year for the next decade. Later, in 1975, he revised his prediction and stated that after 1980 he expects that the components in CPUs will double roughly every two years. These predictions where used to create the infamous “Moore’s law” which states that the *performance* –not the number of transistors– of CPUs doubles every one and a half years. Satisfying the “Moore’s law”, processor manufacturers for many years kept improving the performance of CPUs by creating more complex circuits using more, smaller transistors and increasing the frequency they operate on. However, as transistors get smaller, their power leakage and heat generation get higher, putting a limit on the frequency increase. As a result, to achieve the targets set by “Moore’s lay” processor, in the new millennium manufacturers turned to chips with multiple, slower, but at the same time more energy efficient, cores. In 2005 AMD and Intel released the first dual-core desktop CPUs (*Athlon 64 x2* and *Pentium D* respectively). Since then, processor designers are trying to fit more and more cores in a single chip. Multi-cores might not be as fast as single-cores but can handle more workload at the same time, resulting in better overall performance. This move, from frequency and circuit complexity increase to multiple cores, however, has led to a reduction in the rate in which the performance of CPUs doubles.

> “The last two technology transitions have signalled that our cadence today is closer to two and a half years than two.”
>
> *– Brian Krzanich, CEO of Intel, 2015*

Nevertheless, as the number of transistors per chip continue to increase, it is expected that the number of cores per processor will follow in a similar pace. Esmaeilzadeh et al. [31] estimate, that processors will reach the hundreds of cores per chip in the next decade. As the number of cores per chip continuously grows, processor designers are trying to keep energy consumption low or even reduce it while improving the overall performance. Keeping the energy consumption at low levels allows for longer battery life for mobile devices, as well as, for reduced operational costs of data-centers and supercomputers. Trends suggest that future many-core machines will have hundreds of mid-range cores with very fast communication channels. At such high number of cores,

memory management becomes a major issue. Implementing efficient cache coherency protocols over thousands of cores is not trivial. To make matters worse, previous work has shown that cache coherency protocols as we know them today are not going to scale in terms of energy consumption. Kaxiras et al. [53] show that directory coherence protocols do not scale in terms of power and performance. Choi et al. [24] also argue that cache coherence protocols imply large performance and energy overhead.

To overcome this issue, processor designs proposed by recent literature use limited or no hardware cache coherency. Intel has presented two examples. First, the Single-chip Cloud Computer (SCC) [39], a homogeneous 48-core non-cache-coherent architecture. The SCC uses a full cache hierarchy per core, includes an on-chip buffer for core-to-core communication, and is programmed mainly using MPI or an alternative custom-tailored message passing programming model. Second, Runnemede [20], a hierarchical non-cache-coherent many-core architecture design that can scale up to hundreds of cores. Runnemede has four levels of scratchpad memories and three levels of computational modules. Furthermore, in each execution engine, the finer computation module features a 32K software-managed non-coherent cache. All the scratchpads map to a unique address range from a global address space. Runnemede offers DMA-like transfers that can work on cache-line granularity. In related literature, Lyberis et al. [66, 67] introduce the Formic board and built the Formic-Cube, a 512-core prototype using 64 such boards. Each Formic-board features eight MicroBlaze-based, non-cache-coherent cores, while the prototype features DMA capabilities and full network-on-chip in a 3D-mesh topology. Last but not least, the EUROSERVER architecture [29] uses the *chiplet* as its basic module. Each *chiplet* provides 8 ARMv8 cores with full cache coherent memory accesses, forming coherent-islands, and is equipped with a DMA engine. Multiple *chiplets* can communicate through a multi-level global interconnect which allows for remote memory accesses.

1.1. Motivation

In distributed or non-cache-coherent memory architectures like SCC, Runnemede and the Formic-Cube, the software needs to explicitly transfer data across nodes and make sure the transfers happen in the correct order to get the desired results. Exposing all architectural details to the programmer and giving her the full control of the architecture, allows her to develop sophisticated and efficient programs. Unfortunately though, it also slows down the development process and reduces productivity, since the programmer has to argue about the correctness of operations, not directly related to the programming language. Furthermore, it significantly limits the portability of applications because they depend heavily on the underlying system architecture.

Runnemede and the Formic-Cube both propose using a task-based programming model as a higher-level abstraction to MPI, using task annotations to declare memory usage

and a runtime system to schedule computations to data or set up all required data transfers among memories or caches in software. In such programming models developers do not need to explicitly move data or work, as long as they correctly annotate their programs. In some cases, however, properly annotating the program is not trivial, e.g., due to deep nesting levels which makes it hard for the developer to find all possible memory accesses of a task. Additionally, task-based programming models are not yet widely adopted and are mostly focused to high performance computing (HPC).

The computer science community often categorizes programming languages based on their abstraction level. Low-level languages are closer to the hardware specification and enable the programmer to directly interact with the hardware. C is an example of such a low-level language. These languages are very popular in the embedded systems and kernel development eras. In such development environments the program often needs to have direct access to the hardware, i.e., hardware drivers.

On the other hand, high-level programming languages hide any underlying architectural details. Such programming languages are a better fit for developing user level or computational applications. High-level programming languages allow the programmer to focus on the problem she needs to solve instead of the underlying architecture characteristics. As a result, high-level programming languages increase productivity. Thus, they are also called high-productivity languages. Such programming languages are usually implemented using process virtual machines, that act as the middle-ware between the language and the underlying operating system.

Java[1] is such a high-level, object oriented, general-purpose programming language. Some of the more interesting features of Java are its builtin concurrency support and its portability. In terms of concurrency, Java provides builtin threads and synchronization mechanisms. Of special interest the fact that the behavior of concurrent Java programs is formally defined by a memory model. Java was one of the first programming languages to define a memory model to clearly define the interaction of concurrent threads through the memory. In terms of portability, Java has been advertised as a “write once, run anywhere” (WORA) language. Java achieves this by abstracting away any architectural details, providing a uniform experience across different platforms through the Java Virtual Machine (JVM). The JVM, along with the Java compiler, are essentially the implementation of the Java language. The Java compiler translates the Java source code to Java bytecode that is then interpreted or compiled just-in-time (JIT) by the JVM. The Java bytecode is common for every architecture, meaning that a Java program may be compiled once on any platform and then be run on any other platform, as long as there is a JVM implementation for that platform.

JVMs are usually developed using a low-level language and take care of the thread scheduling, memory management etc. As of today, JVMs are mostly targeted to shared-memory, cache-coherent systems. There are a few JVMs targeting distributed systems, which essentially target clusters of computers and not single chips [5, 68, 102, 107]. The

[1] Java is a trademark of Oracle.

emerging many-core architectures, however, differ from super-computer clusters in that all the communication is performed on chip. As a result, they feature significantly lower network latency and possibly higher network throughput than super-computer clusters. Additionally, they allow for explicit asynchronous fetches through DMA transfers from and to the whole system's address space. On the contrary, super-computer clusters with interfaces supporting DMA transfers usually only allow access to a subset of the system's address space. There are also two JVMs [73, 77] targeting the IBM's Cell B.E. processor [81]. Cell B.E. is a heterogeneous multiprocessor featuring one POWER Processing Unit (PPU) and eight Synergistic Processing Units (SPUs). The cores can access each other's memory through DMA transfers or remote accesses, but there is no cache coherency between the SPUs' scratchpads. Cell B.E. shares many common characteristics with that of the proposed future many core architectures [20, 29, 66]. However, the low number of cores, on Cell B.E., allows for centralized designs, like CellVM [77], that use the more powerful PPE as a server, handling synchronization requests.

In this thesis we explore how JVMs can be implemented on non-cache-coherent architectures with hundreds of cores. We focus our work on JVMs because they are of special interest, since apart from Java itself they run Scala and other languages as well. Java itself is a popular high-level programming language used in many fields. One of its popular use cases is that of Hadoop [90], a distributed file system that scales to a large number of cores. Scala [78], on the other hand, is a younger programming language with built-in support for *Actors*, data-parallelism, asynchronous programming with *Futures* and *Promises*, software transnational memory, and event streams. Scala, despite its young –compared to Java– age, has been adopted by the industry and the academia for its performance in data analytics.

1.2. Contributions

Specifically, in this thesis we make the following contributions:

- We present the Java Distributed Memory Model (JDMM), an algebraic formalization of the Java Memory Model for non-cache-coherent and distributed memory architectures, and argue about its adherence to the original Java Memory Model. Our formalization exposes the memory management mechanisms, enabling JVM implementers to better understand how to design and implement the memory management mechanisms of their JVM in order to adhere to JMM, when targeting non-cache-coherent or partially cache-coherent architectures. (Chapter 3)
- We design algorithms for software caching of Java objects, and synchronization management. The software caching algorithms provide a memory access layer that adheres to the Java Memory Model. The synchronization management algorithms ensure mutual exclusion for threads synchronizing through Java monitors.

Additionally they ensure the proper ordering between threads that synchronize in a point-to-point manner without the use of Java monitors. Our algorithms take advantage of partial cache coherence, when the latter is provided by the architecture. (Chapter 4)

- We implement our algorithms in DiSquawk, a proof-of-concept Java virtual machine. DiSquawk targets the Formic-Cube architecture, a non-cache-coherent, 512-core, architecture. Since Formic-Cube does not offer coherent-islands, we limit our implementation to the inter-coherent-island part of our algorithms, excluding any logic regarding optimizations within coherent-islands. Each core runs an instance of an interpreter-based JVM. These instances implicitly communicate with each other and exchange data to provide a single system image to the Java application. DiSquawk is licensed under the GNU General Public License (GPL) and can be downloaded at `https://github.com/CARV-ICS-FORTH/disquawk`. (Chapter 5)
- We evaluate DiSquawk using a set of benchmarks and micro-benchmarks. The micro-benchmarks help us to better understand the behavior and performance of specific mechanisms of DiSquawk, while the benchmarks help us measure its overall scalability. The results show that DiSquawk scales with the number of cores in a similar manner to the state-of-the-art, HotSpot JVM. (Chapter 5)
- We define a Java core calculus and model DiSquawk in its operational semantics. Our calculus is a minimal core calculus for the Java language, which is based on the Featherweight Java [43] variant introduced by Johnsen et al. [48]. In this thesis we extend it by replacing the explicit lock support with synchronization operations, *e.g.*, monitor-enter, join, etc. Similarly to Johnsen et al. [48], we also omit Java inheritance, subtyping, and typecasting. Based on our calculus' operational semantics, we sketch a proof of adherence of the proposed algorithms to the Java Distributed Memory Model and thus to the original Java Memory Model. (Chapter 6)

1.3. Outline

This thesis is organized as follows. Chapter 2 discusses the trends in emerging processor architectures and presents the state of the art in process virtual machines that are most closely related to our proposal; Chapter 3 presents the Java Distributed Memory Model (JDMM), the JMM extension we create to expose cache management operations; Chapter 4 presents the algorithms we design for software caching of Java objects and synchronization management that adhere to JDMM while taking advantage of partial coherence when provided by the architecture; Chapter 5 presents DiSquawk, a Java Virtual Machine (JVM) implementing the proposed algorithms for architectures without partial coherence; Chapter 6 presents DJC, a Java core calculus we define along with

its operational semantics that models DiSquawk; Chapter 7 discusses related work; and Chapter 8 concludes and discusses open research problems and future work.

Chapter 2.

Background and State of the Art

This chapter discusses the trends in future processor architectures and the state of the art in process virtual machines targeting systems with similar properties as those of the future processor architectures.

2.1. Emerging Processor Architectures

As we discuss in Chapter 1, to overcome the power wall and continue increasing the performance of integrated circuits, processor architects pack multiple cores in a single chip. Processors in the next decade are expected to contain hundreds of cores [31]. At that high number of cores, cache management and fast memory access in general, becomes a major issue. Previous work has shown that cache coherency protocols as we know them today are not going to scale in terms of energy consumption [24, 53]. Inspired by such research results, processor designs proposed by recent literature use limited or no hardware cache coherency.

2.1.1. Intel Runnemede

Intel Runnemede [20] is a hierarchical non-cache-coherent many-core architecture design that can scale up to hundreds of cores. Runnemede has four levels of scratchpad memories and three levels of computational modules. The basic module of Runnemede is the *block*. Each *block* consists of a single Control Engine (CE), several Execution Engines (XE) and an L2 scratchpad. Execution engines are general purpose mid-range cores aiming for higher performance over watt ratios. As their name implies they are designed to execute the application code. On the contrary, control engines, as their name implies, are designed to manage (control) the execution engines and are aiming for high performance in an effort to reduce the latency of critical non-parallel operations. This design is tailored after the master-servant model, where the master core manages the servant cores and schedules work to them. Each control engine and execution engine features a 32K software-managed non-coherent cache. Execution engines also feature

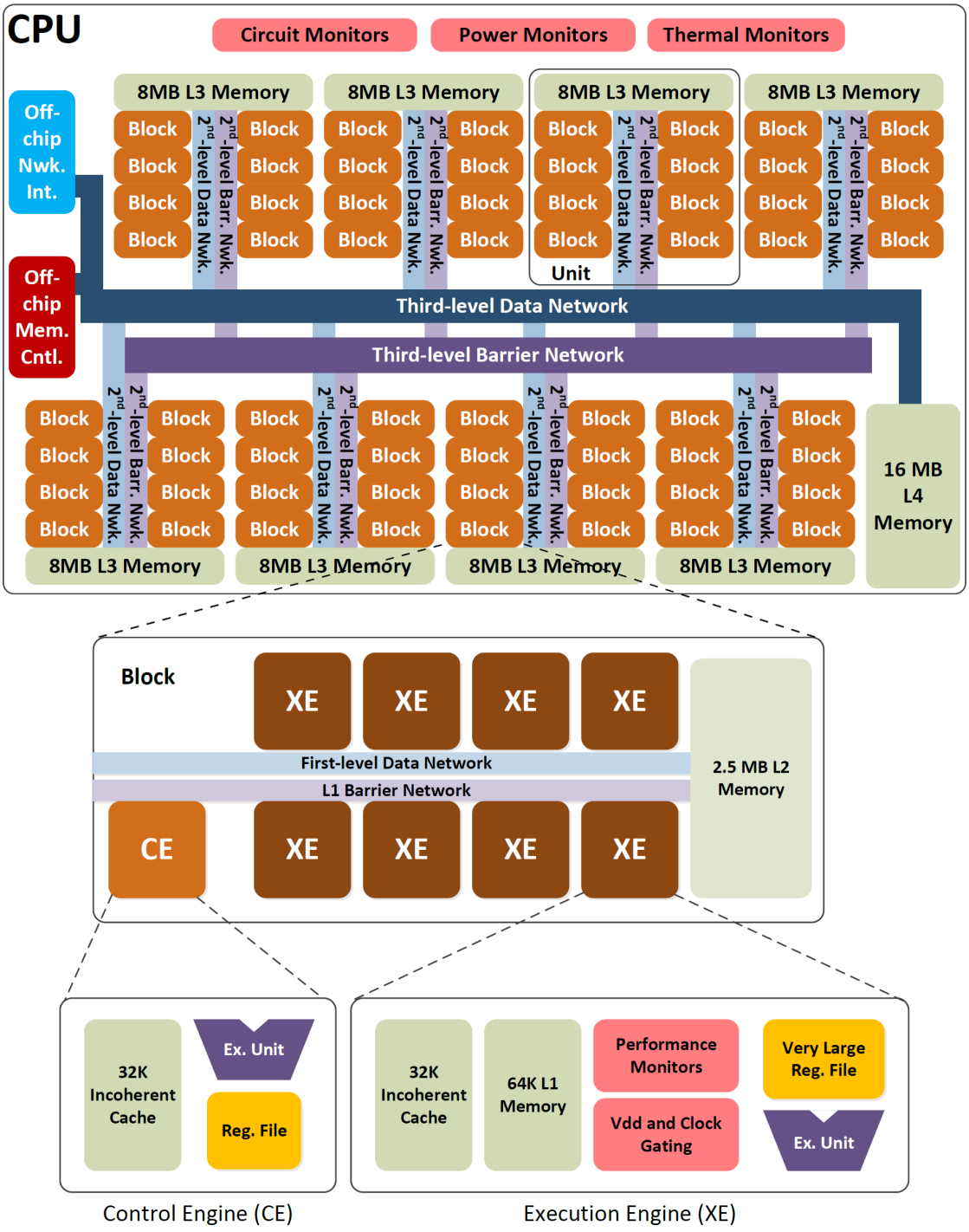


Figure 2.1.: Overview of the Intel Runnemede Architecture

a 64K L1 scratchpad. Multiple *blocks* can be combined in a *unit*, featuring a shared L3 scratchpad. A Runnemede chip can have multiple *units* that share an L4 scratchpad. A schematic overview of the Runnemede architecture is presented in Figure 2.1[1]. All the scratchpads, register files, and DRAMs in Runnemede are addressable by every core in the system using a 64-bit global address space. Runnemede offers DMA-like transfers from and to any scratchpad, register file, and DRAM in the system.

2.1.2. Formic

In related literature, Lyberis et al. [66, 67] introduce the Formic-Board. The Formic-Board is another modular design that allows scaling up by combining multiple modules. Each Formic-Board features 128MiB of DRAM and 8 cores, each equipped with a private 8KiB L1 cache and an also private 256KiB L2 cache. Combining 64 Formic boards, Lyberis et al. built the Formic-Cube, a 512-core prototype, which they show to scale well running a representative set of benchmarks. The prototype features DMA capabilities and full network-on-chip in a 3D-mesh topology. Similarly to Runnemede all the caches and DRAMs are addressable by every core in the system using a global address space. A schematic overview of the Formic-Board is presented in Figure 2.2a and a picture of the Formic-Cube is presented in Figure 2.2b.

2.1.3. EUROSERVER Architecture

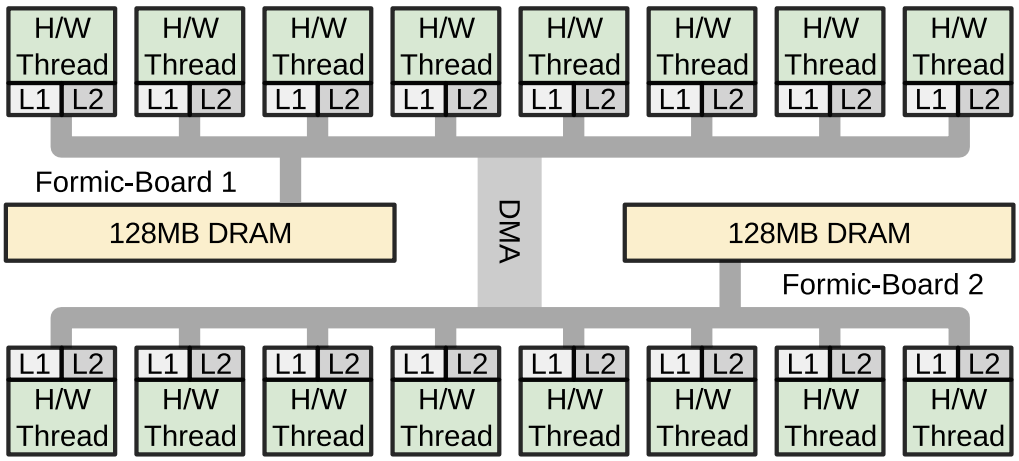
Last but not least, the EUROSERVER architecture [29] uses the *chiplet* as its basic module. Each chiplet combines 2 quad-core ARMv8 processors, totaling 8 cores. In contrast to Runnemede and Formic-Board the caches of the chiplet are coherent, forming coherence-islands. Chiplets are also equipped with DMA engines that allow for data transfers across chiplets. Multiple chiplets can also communicate through a multi-level global interconnect which allows for remote memory accesses. A schematic overview of the EUROSERVER architecture is presented in Figure 2.3.

2.1.4. Assumptions About Future Processors

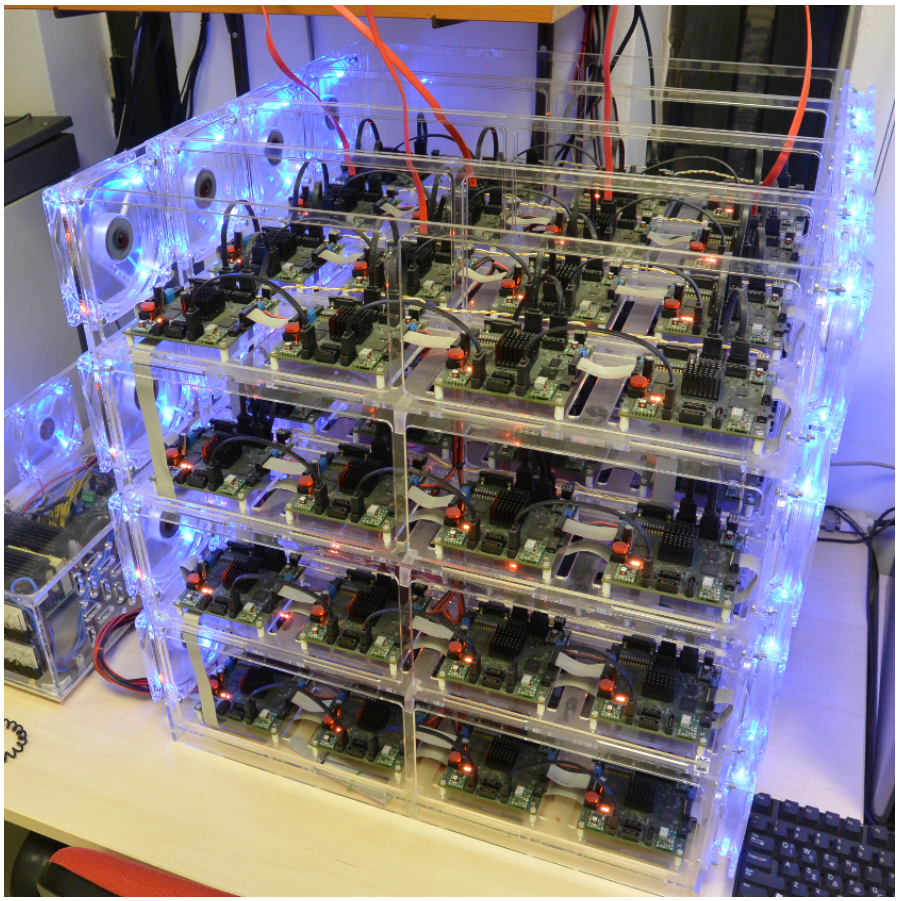
Inspired by these architectures, we assume that in the following decade:

- Processors will be featuring hundreds of CPU cores in a single chip. A few chips will be able to be combined in a single board, forming a node with thousands cores.

[1] Figure's source: [20]



(a) Formic-Board Overview (Example with two Formic-Boards)



(b) The Formic-Cube

Figure 2.2.: The Formic-Board and Formic-Cube

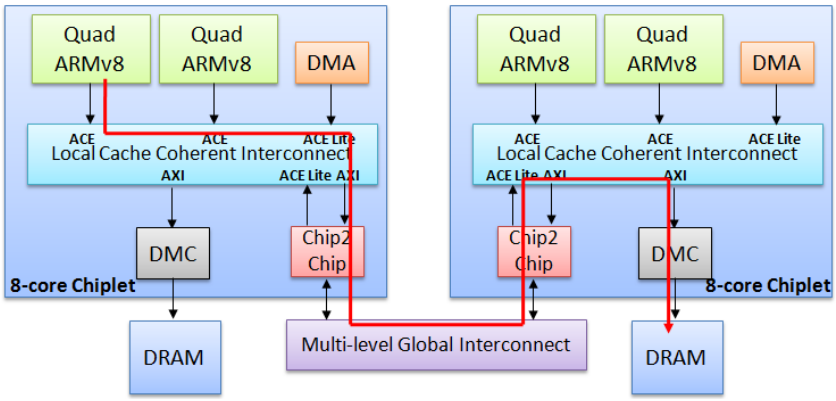


Figure 2.3.: Overview of the EUROSERVER Architecture

- Processors will be of heterogeneous nature. Many-core processors will consist of a mix of strong, high-performance, energy-hungry cores, and weaker, power-efficient cores. The number of weaker cores will be about an order of magnitude more than the number of strong ones in a many-core processor.
- Cache-coherence will be partial. Groups of cores, called *coherence-islands*, will share cache coherent caches. Coherence between caches of different coherence-islands will not be available. Any memory transfers across coherence-islands will be explicit and managed by software.

2.2. Java Virtual Machines

This thesis focuses on Java Virtual Machines (JVMs). Despite the fact that the Java Virtual Machine (JVM) was mainly designed to implement Java [62], it has been used to implement other languages as well [10, 26, 41, 78]. The popularity of Java as a programming language has motivated researchers to study JVMs in detail and propose new techniques that improve their performance. Although, most studies and projects focus on JVMs targeting shared-memory cache-coherent architectures, in the literature there are several JVM implementations that go beyond a single node or a single address space. Most of these implementations target super-computer clusters, while a few others target heterogeneous systems, but all of them aim to deliver a single system image to the programmer to ease development. Such implementations are relevant to the one proposed in this work, since both clusters and heterogeneous systems impose similar challenges to those imposed by non-cache-coherent many-core architectures. The lack of coherent shared memory creates the need for explicit software memory management to ensure coherence across the system and adherence to the Java Memory Model (JMM) [72].

In the following sections, we briefly present the most noticeable JVM implementations on clusters or heterogeneous systems. In this process we omit implementations relying on Software Distributed Shared Memory (SDSM), since they are stricter than the JMM and usually operate on page granularity while JMM is defined on variable granularity. As a result SDSM is expected to introduce significant performance and energy overhead by transferring more data than needed, due to false sharing and redundant coherence traffic (re-fetches and invalidations).

2.2.1. Java/DSM

Java/DSM [102] is a combination of Java with Distributed Shared Memory (DSM), aiming to hide the heterogeneity and the lack of memory coherence of distributed systems. Yu and Cox implement Java/DSM as a parallel JVM over DSM, where all objects are treated as local objects through DSM. The parallel VM consists of a JVM instance per node and does not support thread migration across nodes. DSM provides a coherent Java heap across the JVMs, while JVMs are responsible for translating the data across different architectures (i.e big/little endian). Applications written using synchronization and locking wherever necessary can run over Java/DSM without the need of any modifications. Garbage collection is operating separately per node taking in account possible remote references to avoid premature reclamation.

2.2.2. Hyperion

Hyperion is a high-performance distributed VM written in C as a runtime [4, 68]. To execute Java, Hyperion first uses, `java2c`, a source-to-source compiler, to translate Java bytecode to C. The produced code is afterwards compiled and linked with the Hyperion runtime. Hyperion targets clusters of homogeneous computer nodes. Its aim is to abstract away the lack of memory-coherence across nodes. To achieve this, Hyperion implements a software cache over the Partitioned Global Address Space (PGAS) programming model.

Hyperion uses an *Object Table* (OT) which acts like a cache directory and is replicated on each node. The OT uses object references as indices and has two fields per index, a local and a remote object pointer. Both fields are initially set to zero. Each node holds a replica of the OT, which is partitioned in equal portions, so that each node *owns* a segment of the global address space. An OT entry is not allowed to have both fields set at the same time. When a new object is created it is given one of the empty entries' reference (index), from those *owned* by the node allocating the object, and its local object pointer, on the local OT, is updated to point to the allocated memory. When an object is referenced, Hyperion first checks if the local object pointer is set on the corresponding OT entry. If not set, the reference is a remote one, so Hyperion checks the remote object pointer. If the remote object pointer is also not set the node copies the actual data

from the node *owning* them to the local memory and updates the remote object pointer. To provide a coherent view of the memory, Hyperion tracks writes using bitfields that indicate which object bytes where modified for the cached remote objects. At synchronization points Hyperion transmits all local modifications to the node *owning* the original data and invalidates all the cached data. When a thread writes to a `volatile` variable the data are transmitted immediately after the local write and the cached copy is purged.

In Hyperion Java threads are arbitrarily assigned to system nodes by the runtime. In order to detect the program's completion, a central node is holding a live thread counter which is updated at thread start and completion. For nested threads their parent is responsible to issue the thread start event to avoid races between the start event of the child and the completion event of the parent.

2.2.3. cJVM

Cluster JVM (cJVM) aims to provide a Single System Image (SSI) view of clusters [5]. Each node in the cluster runs an instance of cJVM and all together form the VM.

cJVM uses a distributed heap model and the proxy pattern to provide the illusion of a single heap across the cluster nodes. Objects are allocated locally on each node's heap and proxies are provided to other nodes at remote method invocations. To improve performance cJVM supports multiple proxy implementations per object. This way different policies can apply on different object instances depending on the object access pattern. cJVM performs code analysis to infer the different object access patterns and implement the appropriate proxies.

When a node invokes a method through a proxy, if it accesses any of the object's variables, cJVM redirects execution to the node owning the object to improve performance. Aridor et al. call this technique *method shipping*. *Method shipping* is implemented using a set of server threads per cJVM instance. These server threads are handling the *shipped* methods. *Method shipping*, although efficient, results in distributed threads. Distributed threads also have distributed stacks, however Java expects to find the whole stack at the local memory. cJVM solves this issue by keeping information about the thread that requested a method shipment. Methods changing a Java thread's state and traversing the stack frames are using this information to concatenate the distributed stacks. Another issue is thread identification using `Thread.currentThread()`. This method returns a reference to the thread object invoking it. To provide the correct reference cJVM passes the global memory address of the thread object with every method shipment. `Thread.currentThread()` is then able to refer to the current thread through its global memory address.

cJVM also hijacks `new` so that, allocation of new threads, contrary to that of new objects, can be performed on a remote node. A load balancing routine is invoked at thread allocation to find the *best* node to create the master object.

2.2.4. JESSICA2

JESSICA2 is a Distributed Java Virtual Machine (DJVM) employing just-in-time compilation to improve performance [107]. JESSICA2, like Hyperion (see Section 2.2.2), is based on the partitioned global address space (PGAS) model and implements software caching to improve efficiency.

JESSICA2 introduces the Global Object Space (GOS) layer. In GOS, similar to the Hyperion PGAS model, each cluster node contributes a portion of the Java heap. Each node reserves a *global heap area* and a *cache heap area*. All threads within a node access the *global heap area* directly, without the need of any special handling. On the other hand, *cached heap area* accesses are handled by the GOS layer. Each thread has a private data structure acting as a cache directory. At cache misses JESSICA2 initiates a data transfer and yields the thread until the transfer is completed. This way, when there is sufficient work to execute the communication overhead is overlapped with execution. Furthermore, at synchronization points, cached objects are invalidated and re-fetched to conform to the Java Memory Model [70, 72]. JESSICA2 introduces the *adaptive object home migration* concept, to improve performance for migrated threads. When a thread is migrated to another node it caches its data to the new node. This results to increased data traffic at synchronization points. In order to reduce this overhead JESSICA2 also supports object migration. When a thread's number of accesses to an object dominates its total number of accesses, the corresponding object is migrated to that thread's node and the rest nodes are informed about the object's new *home*.

JESSICA2 also supports thread migration across heterogeneous cluster nodes. To combine this feature with JIT compilation JESSICA2 limits the points where a thread migration is possible. At these points migration requests are checked and any machine registers used to improve performance are written to the corresponding memory slots. In JESSICA2, Java threads are migrated along with their stack. Additionally at these points JESSICA2 stores information about the types of the objects in the stack. This information is mandatory for heterogeneous systems, since type sizes vary among nodes. The spacial cost of this information is four bits per object.

2.2.5. CellVM

CellVM [77] is a virtual machine targeting the IBM's Cell BE processor [23, 81]. The Cell BE is a heterogeneous chip multiprocessor. It consists of a single Power Processing Element (PPE) and 8 Synergistic Processor Elements (SPEs). Each SPE is equipped with 256KiB of non-coherent, dedicated local storage. CellVM abstracts away the heterogeneity of the processor and the absence of cache-coherence on the SPEs.

CellVM actually consists of two separate modules, the ShellVM and the CoreVM. The ShellVM is running as a resource manager on the PPE. ShellVM is responsible for handling the majority of the CellVM's data structures. On the contrary, CoreVM is mainly

a bytecode interpreter. A CoreVM instance is run on each SPE and manages the local Java stack, JVM registers and local caches. The Java applications are essentially run by the CoreVMs on the SPEs, while ShellVM coordinates the. Note also that some instructions cannot be executed by the CoreVM (e.g. memory allocation, synchronization, locking etc.), in such cases CoreVM requests the ShellVM to process them.

To abstract away the lack of cache coherence in the SPEs CellVM employs a software instruction cache (I-Cache) as well as a software data cache (D-Cache). The local memory of the SPEs is divided into three segments. About half of the memory is reserved for the CoreVM's binary. One of the remaining two quarters is reserved for the I-Cache and the D-Cache, while the last quarter is used for the Java stack.

The I-Cache keeps copies of whole methods, thus a fixed cache block size does not apply. The I-Cache is fully associative and uses the method's address in the main memory as a tag. In the case of a I-Cache miss if there is no sufficient space to fetch the requested method all cached methods are invalidated.

The D-Cache on the other hand is direct mapped to reduce the lookup latency. The D-Cache configuration (number of cache lines and cache block size) is performed at compile time. Write backs occur in two cases. When there is a conflict miss and when the code switches from the CoreVM to the ShellVM.

Note that, according to Noll et al. CellVM fails to provide a coherent Java heap, thus it does not strictly comply to the Java memory model [70, 72]. Noll et al. claim that this was a design decision to improve performance. Additionally, its design does not scale on a large number of cores since it resides on a single coordinator, the ShellVM, and requires all CoreVMs to communicate with it. Such master-servant models are efficient up to a certain number of servants, on a large number of cores the coordinator is expected to become a bottleneck.

2.2.6. Hera-JVM

Hera-JVM [73] is another virtual machine targeting the IBM's Cell BE processor. Hera-JVM differs from CellVM in that

- it offers transparent migration of Java threads between the two core types (PPEs and SPEs),
- it is designed as a single VM,
- it improves scalability by enabling SPEs to execute synchronization instructions, allocate memory, etc.,
- it adheres to the Java memory model [70, 72].

Hera-JVM implements a JIT compiler targeting both the PPE and the SPEs. The bytecode of a method is compiled to the instruction set of the target core before migration. Java thread migrations are only allowed at method invocations to reduce complexity. Hera-JVM achieves up to $13\times$ speedup when compared with single core executions on the PPE.

In Hera-JVM each SPE core acts as a *virtual processor*. Each virtual processor has a private run queue with Java threads. Java threads run for a full time quantum unless they block (e.g. acquiring a lock). When a thread consumes its time quantum or blocks, the next thread is scheduled for execution in a round robin fashion. Furthermore, at scheduling points, the virtual processor checks the other virtual processors queues and load balances the system if needed by transferring threads accordingly.

In Hera-JVM the Java thread stack, contrary to CellVM, is located in the main memory and only the top part of it (16KiB) is locally available. This is done to enable migration without having to move all the data around. In the case of a (local) stack overflow the current the current block is transferred to the main memory and the next one takes its place.

Hera-JVM also uses a data cache for data in the Java heap. Hera-JVM caches objects or array blocks. There are four different Java bytecodes to access memory; `getfield`, `setfield`, `iaload` and `iastore`. The `getfield` and `setfield` bytecodes are used to access an object's field. In these cases a cache miss results in the whole object cached. While, the `iaload` and `iastore` bytecodes are used to access an array element. In these cases Hera-JVM does not cache the whole array, but a 1KiB block of it, as it could be too large. The software cache is implemented as a 1024-entry hashtable, using as a key the main memory address of an object instance or an array block, while the value is the local memory address of the cached data. On a cache miss a DMA transfer is initiated to fetch the data and the Java thread blocks until the DMA is completed. The data cache is using a write through policy. Whenever a thread writes to a cached address, a non-blocking DMA, copying the new data to the main memory, is initiated. However, in order to conform to the Java memory model, a Java thread blocks until all DMAs are finished before releasing a lock, writing to a volatile variable, context switching or migrating to another core.

Additionally, Hera-JVM implements an instruction software cache. Similarly to CellVM caching operates on methods. Hera-JVM takes advantage of the standard method of supporting virtual methods in a JVM. Hera-JVM uses a per SPE table of contents (TOC) which acts as a cache directory for type information blocks (TIB) instead of methods. TIBs are found in each object's instance and contain an entry for each declared method in the class. When a virtual method is invoked its class's TIB is looked up in the local TOC. If the class containing the class's TIB is not cached the TOC points to the TIB's main memory address. When Hera-JVM caches a method, it copies the class instance's TIB to local memory; it updates the TOC with the TIB's local memory address; it copies

the method's code to the local memory; and updates the local TIB copy to point to the local copy of the method.

In this work we target future many-core architectures with hundreds of cores. Such architectures differ from super-computer clusters in that all the communication is performed on chip. As a result, they feature significantly lower network latency and possibly higher network throughput than super-computer clusters. On the other hand, Cell B.E. shares many common characteristics with that of the proposed future many core architectures [20, 29, 66]. However, the low number of cores, on Cell B.E., allows for centralized designs, like CellVM, that use the more powerful PPE as a server, handling synchronization requests. Moreover, although Hera-JVM appears to be better distributed than CellVM, due to the expected large number of cores in future architectures, porting Hera-JVM from the Cell B.E. to a future many-core architecture is not certain to provide the expected scalability.

Chapter 3.

The Memory Model

In this chapter we present the Java Distributed Memory Model (JDMM). JDMM is an extension of the Java Memory Model (JMM) that exposes cache management operations and explicitly defines rules that govern the order in which those operations must appear in respect to other Java operations in order for an execution to be valid. We provide the formalization of JDMM, argue that JDMM-compliant executions are also JMM-compliant, and finally use JDMM to show that Hera-JVM [73] complies to JMM.

Parts of the work presented in this chapter have been published in the proceedings of the *2014 ACM SIGPLAN International Symposium on Memory Management* (ISMM 2014) [103].

3.1. Introduction

As different architectures implement different consistency models, the same program may behave differently on different machines. This results in reduced code portability and requires the programmer to understand the consistency model of every architecture she wants her program to run on. To solve this issue some languages, with builtin multi-threading support, define their own memory model. Such examples are the Unified Parallel C (UPC) [25], C/C++ [51, 52], Java [70, 72], etc. The memory model is essentially a description of the interaction between parallel threads through the memory. It defines the flow of the data from one thread, as defined by the language, to another thread according to the synchronization between them. Every implementation of a language, with a memory model specification, must adhere to the corresponding memory model, irrespective of the consistency model of the underlying architecture. A language's programming model can also be seen as a contract between the programmers and the language implementers, since it specifies what a programmer should expect, and what the language runtime system or compiler guarantees.

Java was one of the first programming languages to define a memory model. The Java Memory Model (JMM) is a relaxed memory model, according to which, data-race-free (DRF) programs are guaranteed to be sequentially consistent. With sequential consistency we refer to the property that requires that:

> “ [...] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. ”
>
> *– Lamport [56]*

Additionally, non DRF programs are guaranteed to be *safe*. In this context, *safe* means that data-races do not result in any thread observing an *out-of-thin-air* value (except for reads of long and double values). That is, all reads in a Java program return values that have been written, to the corresponding variable, and not arbitrary ones. Under JMM, non-DRF programs may result in non sequential consistent values of object fields, static fields, or array elements that are being accessed without proper synchronization. Note however, that properly synchronized accesses are sequentially consistent even in non-DRF programs.

JMM is essentially a lazy release consistency model [55]. Lazy release consistency assumes the existence of two special synchronization operations, called *acquire* and *release*. In correctly synchronized programs, before modifying an object the program needs to acquire it, and after modifying it needs to release it. In JMM *acquire* and *release* correspond to a number of synchronization mechanisms provided by the language. JMM abstracts program operations to *actions* (e.g., reads, writes, interrupts). The *actions* concern only operations one could observe by monitoring the interface between the processor and the memory. Some of these *actions* are grouped in *synchronization* actions and *external* actions. *Synchronization* actions, as their name implies, are responsible for inter-thread communication and synchronization. *Synchronization* actions are further grouped in *acquire* and *release* actions. Without the use of *synchronization* actions, a write might never be seen by a different thread than the writer-thread. However, this is by no means guaranteed, nor is the order in which such *leaked* writes are observed by other threads. JMM, based on the synchronization actions and their ordering, defines the possible values that each read may observe. In general, any writes visible to a thread executing a *release* action must become visible to the thread executing the corresponding *acquire* action. *External* actions are those observable out of the program (i.e., I/O).

3.1.1. Motivation

In the x86 Total Store Order (x86-TSO) memory model [79], a memory fence is sufficient to make any writes in the write-buffer and/or registers visible to any subsequent action, and thus satisfy JMM. However, in the context of non-cache-coherent architectures, a memory fence is not enough. The result of a memory fence in such architectures is observable only on the local node. For the writes to become visible to all nodes, at *release* actions, any writes must be committed to the main memory. To achieve this,

the JVM needs to explicitly transfer data from the local node's scratchpad to the main memory. Then, the corresponding *acquire* action also needs to perform explicit data transfers to get the data from the main memory.

To implement JMM in architectures like Runnemede, the Java Virtual Machine (JVM) needs to explicitly move data across nodes. Unfortunately, moving data across nodes for each remote memory access is prohibiting, due to the increased time and energy overheads of these transfers compared to the corresponding overheads of local accesses. There is a lot of existing work on implementing a JVM over non-cache-coherent machines and distributed computer systems [4, 5, 32, 68, 73, 100, 102, 107, 108]. Most implementations are older than JMM and they are based on the second edition of the Java Language Specification (without the updates introduced by JSR-133). These implementations follow three different approaches, regarding how they access remote data. Aridor et al. [5] and Zigman et al. [108] avoid transferring data by using object proxies; instead of transferring data, the operations on the data are performed on the node having the corresponding object on its local heap-slice. Other implementations [4, 73] use a custom software cache and implement some form of a hardware cache coherence protocol or global directory in software. The rest [32, 100, 102, 107, 108] use Software Distributed Shared Memory (SDSM). SDSM and custom software caches have two common properties: (i) to access an object, they need to create a local copy; and (ii) to make writes visible to other processors they need to write-back the data to the main memory and also make sure that any local copies of these data are updated.

The SDSM approach is the simplest to implement, since the developers can design the JVM as they would do for a shared memory machine with cache coherency. SDSM is responsible to ensure that all nodes have a coherent, according to some coherence protocol, view of the shared data. SDSM systems, however, produce more traffic than needed by JMM. JMM is a relaxed memory model, while SDSMs usually implement more strict coherence protocols. The model we propose in this work can help future JVM implementers to choose the more relaxed available coherence protocol for their SDSM based implementation.

The use of custom software caches, on the other hand, enables JVM developers to improve performance by moving data, only when that is mandatory to comply with JMM. Unfortunately, the efficient implementation of the custom software cache as well as the data movement mechanisms is a tedious and error prone process. Furthermore, it requires good understanding of JMM to minimize the data movement while preserving JMM's properties. JMM's abstract nature does not make clear when those data transfers should be performed. The work presented in this chapter aims to demystify JMM and the ways it can be implemented using software managed data caches. We achieve this by exposing cache management operations to the memory model and introducing new rules that dictate when such cache management operations should be performed in order to preserve JMM's properties.

3.1.2. Approach

To minimize the effort needed to understand JDMM we build it by extending JMM. This way we are able to use well established, in the Java era, definitions, notations, and mechanisms.

Previous work on proving JMM's DRF guarantee [72] has shown that JMM is correct [6, 7, 42, 45, 65, 89, 97], in the sense that correctly synchronized programs have sequentially consistent semantics. However, Aspinall and Ševčík [7, §5] show that there are still some counterexamples to JMM regarding the *out-of-thin-air* guarantee. Specifically, the following transformations are allowed under JMM, but might cause *out-of-thin-air* values to be seen under specific circumstances:

1. Reordering of Independent statements (also mentioned in [21, §7]).
2. Reordering of memory accesses with external actions.
3. Moving memory accesses into synchronized blocks. Also known as "roach motel" ordering [70, §3.5.1].

Additionally, Manson et al. have introduced a set of *causality tests* [83] to describe a series of examples of unacceptable causal loops and seemly causal loops that must be allowed by the semantics. According to Manson et al. [72] JMM should satisfy all the *causality test* cases. However, Aspinall and Ševčík [7] claim that test cases 17–20 [83] are not satisfied by JMM, while Torlak et al. [97] show that only the test cases 19 and 20 [83] are not satisfied by JMM.

To make the first group of transformations safe under JMM and to make it pass the causality tests, Aspinall and Ševčík [6] propose a change to JMM that Lochbihler [65] proves correct. Furthermore, Lochbihler shows that the Java Language Specification (JLS) [36] and Java API define extra communication channels between Java threads than those JMM covers. In his work, Lochbihler, covers these communication channels as well as dynamic memory allocation, thread spawning and joining, infinite executions, the wait-notify mechanism and interruption. Since Lochbihler's work provides a more complete and concise definition of JMM properties, in our thesis we use some of its definitions and formalization over the original definition of the JMM [70] for clarity. Moreover, at the time of writing there is an OpenJDK project in progress aiming to reformulate the base model of JMM [47]. The main targets of this project are:

- The improvement of JMM's formalization to make it machine checkable, as well as, more human readable.
- The fix of existing errors as reported by Aspinall and Ševčík and Torlak et al. [97].
- The coverage of JVM related aspects, like class initialization. Currently JMM focuses on the Java programming language and not to its bytecode. This results in

ambiguous definitions about some JVM operations and the use of the Java bytecode by other languages.

- The coverage of `java.util.concurrent`'s parts (e.g. `AtomicInt.weakCompareAndSet`), as well as, any extensions that my arise from forthcoming JDK Enhancement Proposals (JEPs).

- To provide compatibility with the C11 and C++11 standards, aiming to provide a consistent behavior across Java and C/C++ native libraries.

- To provide a technical document to guide JVM implementors, JDK library developers, and developers, explaining how JMM impacts particular problems and solutions.

- To provide tests for conformance to JMM.

- To provide an interface or hints for analysis tools that check for data-races and security properties across multiple threads.

Since, until the time of this writing, the proposed fixes concern only the validation procedure of JMM, in our work we choose to not introduce any modifications to it. Instead we build JDMM on JMM by extending a few definitions and especially that of the *well-formed executions*. Under JMM an execution is considered to be *well-formed* if a set of conditions is satisfied, and only well-formed executions are considered for validation. In this work we essentially define how cache management actions should be ordered in respect to the other actions of the execution, by introducing new well-formedness conditions. That said, we believe that any future alternations of the validation procedure of JMM are orthogonal to our work and will not impact its correctness nor its contributions.

3.2. The formalization of JMM

This section presents a summary of the formal definition of JMM, as introduced by Manson [70] and Lochbihler [65], that we use in the rest of this paper, since our memory model builds on the existing JMM formalization. Note that the formal definitions presented in this section have no modifications over the original definitions, introduced by Manson and Lochbihler, except for renaming of variables for clarity. Intuitively, JMM defines the legal executions of a program P as a set A of *actions*, restricting the order in which actions can become visible, or *committed* during the execution. In the following definitions, unless explicitly specified otherwise, when referring to actions x and y, we mean $\forall x, y \in A$.

3.2.1. Definitions

Variable: According to JMM [72, §4.1] a variable can be: a static variable of a loaded class, a field of an allocated object, or an element of an allocated array. In general, a variable is a memory location in the Java heap. Variables can contain references to objects or primitive values. As a result, their size depends on the JVM implementation and the underlying system.

Actions: JMM abstracts thread operations as actions [70, §5.1]. An action is a tuple $\langle t, k, v, u \rangle$, 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.

In Table 3.1 we present the abbreviations we use to describe all possible kinds of actions. Our abbreviations are similar to those introduced in [64, §1.1].

Table 3.1.: Abbreviations for JMM Action Kinds

Abbreviation	Description
R	Non-volatile read
W	Non-volatile write
In	Initialization write
Vr	Volatile read
Vw	Volatile write
L	Monitor enter
U	Monitor exit
St	Thread start
Fi	Thread final action
Ir	Thread interrupt
Ird	Interrupt detection
Sp	Thread spawn (`Thread.start()`)
J	Thread join (`Thread.join()`)
Ex	External actions and I/O

Synchronization Actions: Any actions with kind In, Ir, Ird, Vr, Vw, L, U, St, Fi, Sp, or J are *synchronization actions*, which form the only communication mechanism between threads according to JMM. We use $x \in \mathrm{SA}(A)$ to show that x is a synchronization action in A, where A is a set of actions:

$$\mathrm{SA}(A) = \{x \in A : x.k \in \{In, Ir, Ird, Vr, Vw, L, U, St, Fi, Sp, J\}\}$$

Program Order: The partial order $\leq_{po}$ among actions A of an execution that defines a total order over all the actions executed by any single thread t is the *program order*. We use $x \leq_{po} y$ 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:

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

Synchronization Order: A total order over all the synchronization actions of a program execution. Note that JMM considers only synchronization orders consistent with the program order to preserve intra-thread semantics. We use $x \leq_{so} y$ to show that x comes before y according to the synchronization order. Every pair of synchronization actions are ordered by synchronization order.

$$\big((x \neq y) \wedge (x, y \in \mathrm{SA}(A))\big) \Leftrightarrow \big((x \leq_{so} y) \vee (y \leq_{so} x)\big)$$

Synchronizes-With: We use $x \leq_{sw} y$ to show that x synchronizes-with y, where $x \neq y$. Note that $x \leq_{sw} y \Rightarrow x \leq_{so} y$. In the synchronizes-with pairs to follow, 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 $x.t$.

An action x synchronizes-with an action y, written $x \leq_{sw} y$, when:

- x is the initialization of variable v (to `zero`, `false`, or `null`) and y is the first action of any thread:

$$\big((x.k = In) \wedge (y.k = St)\big)$$

- y is any subsequent (according to synchronization order) read of the volatile variable written by x:

$$(x.k = Vw) \wedge (y.k = Vr) \wedge (x \leq_{so} y)$$

- y is any subsequent (according to synchronization order) 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:

$$(x.k = Sp) \wedge (y.k = St) \wedge (x.v = y.t)$$

- y is an invocation to `Thread.join()` or `Thread.isAlive()` that returns `false` and x is the final 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 a synchronizes-with pair $x \leq_{sw} y$, the x action is called a *release* action and the y action is called an *acquire* action. According to JMM a *release* action must make all writes, visible to the executing thread, visible to the actions following (according to the transitive closure of the program order and the synchronizes-with order) the *acquire* action.

Happens-Before Order: The happens-before notion is the one introduced by Lamport in [57]. In the context of JMM this is the transitive closure of the program order and the synchronizes-with order. We use $x \leq_{hb} y$ to show that x happens-before y.

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

Value-written Function: The value-written function $V(w)$ returns the value written for every write action w; every read r reads the value $V\big(W(r)\big)$.

Execution: JMM defines an execution E as a tuple:

$$E = \langle P, A, \leq_{po}, \leq_{so}, W(), V(), \leq_{sw}, \leq_{hb} \rangle$$

We summarize the definitions of the notation used in this tuple in Table 3.2.

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.

Table 3.2.: Definition of JMM Notation

Notation	Description
E	Program execution
P	Java program
A	Set of actions
$\leq_{po}$	Program order
$\leq_{so}$	Synchronization order
$W()$	Write-seen function
$V()$	Value-written function
$\leq_{sw}$	Synchronizes-with order
$\leq_{hb}$	Happens-before order

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 Executions: JMM considers only well formed executions. We use $WF(E)$ to show that the execution E is well formed. According to JMM, an execution

$$E = \langle P, A, \leq_{po}, \leq_{so}, W(), V(), \leq_{sw}, \leq_{hb} \rangle$$

is well-formed under the following conditions (refer to [70, §5.3] for a more detailed description):

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

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

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

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

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

$$\forall y \in \mathrm{SA}(A) : \#\{x \in \mathrm{SA}(A) : x \leq_{so} y\} < \infty$$

WF-4: Synchronization order is consistent with program order:

$$\forall x, y \in A : \big((x.t = y.t) \wedge (x \leq_{so} y)\big) \Rightarrow (x \leq_{po} y)$$

WF-5: Lock operations are *consistent with mutual exclusion*. The number of lock actions performed by any thread t' before the lock action l performed by thread t on the monitor m, according to the synchronization order, must be equal to the number of unlock actions performed by thread t' before l on the monitor m:

$$\begin{aligned}\forall x \in A : \forall t \in T : (x.k = L) \wedge (x.t \neq t) \\ \Rightarrow \#\{y \in A : (y.t = t) \wedge (y.k = L) \wedge (y.v = x.v) \wedge (y \leq_{so} x)\} \\ = \#\{z \in A : (z.t = t) \wedge (z.k = U) \wedge (z.v = x.v) \wedge (y \leq_{so} x)\}\end{aligned}$$

where T is the set of all the execution threads:

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

WF-6: The execution obeys *intra-thread consistency*.

$$\forall r \in A : \Big(\neg\big(r \leq_{po} W(r)\big) \wedge \nexists w \in A : (w.v = r.v) \wedge \big(W(r) \leq_{po} w \leq_{po} r\big)\Big)$$

WF-7: The execution obeys *synchronization-order consistency*:

$$\begin{aligned}\forall r \in A : (r.k = Vr) \Rightarrow \Big(\neg\big(r \leq_{so} W(r)\big) \wedge \nexists w \in A : (w.k = Vw) \\ \wedge (w.v = r.v) \wedge \big(W(r) \leq_{so} w \leq_{so} r\big)\Big)\end{aligned}$$

WF-8: The execution obeys happens-before consistency:

$$\forall r \in A : \Big(\neg\big(r \leq_{hb} W(r)\big) \wedge \nexists w \in A : (w.v = r.v) \wedge \big(W(r) \leq_{hb} w \leq_{hb} r\big)\Big)$$

Lochbihler in [65, §2.4.3] additionally requires that:

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

$$\forall x, y, z \in A : \big((x.k = In) \wedge (y.k = St) \wedge (z.k \notin \{S, In\})\big) \Rightarrow (x \leq_{hb} y \leq_{hb} z)$$

Intuitively, an execution of a Java program can be visualized as a graph, where the actions are nodes connected by synchronizes-with and program order edges, as in Figure 3.1. An action x happens-before an action y if and only if there is a path in the graph that connects x and y. Figure 3.1 visualizes program order edges using solid arrows and synchronizes-with edges using dashed arrows. As implied by the program order, actions in the same row are executed by a single thread. To maintain JMM properties, any code optimization must preserve all the synchronizes-with edges, the happens-before edges and the *intra-thread* consistency of the program.

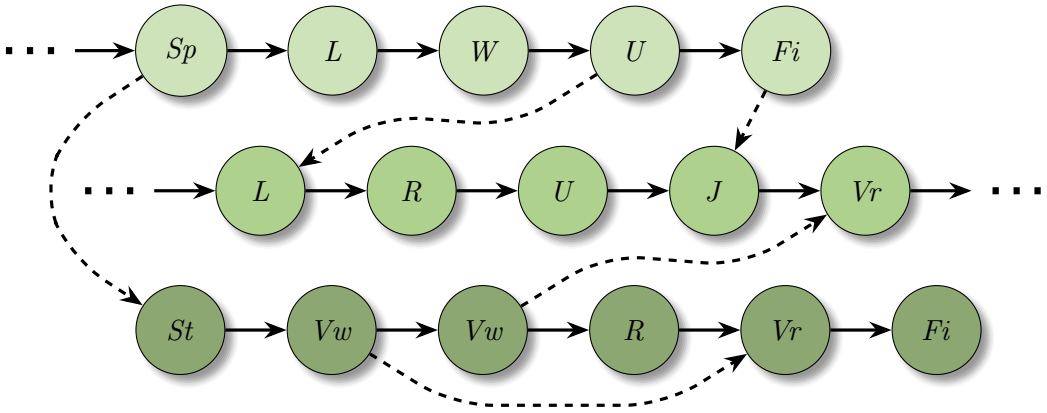


Figure 3.1.: Actions ordering visualization

3.2.2. Validation procedure:

As described in [70, §5.4], JMM, validates well-formed executions by *committing* actions from A. If all actions can be committed then the execution is considered *legal*. The validation procedure aims to enforce the happens-before order while allowing concurrent execution of non-ordered actions. The validations procedure starts with an empty set of actions, C_0, and performs a sequence of steps, committing a set of actions at each step. The set of actions that have been committed at step i is denoted by C_i. The set of actions committed at each step is a superset of the set of actions committed in the previous step, $C_i \subset C_{i+1}$. To enforce the ordering of actions JMM defines a set of restrictions regarding which actions may be committed at each step. These restrictions essentially state that:

1. the actions in a set C_i must exist in A,
2. the commit order needs to preserve the partial orders obtained by restricting the happens-before order to the actions in the corresponding commit set C_i,
3. the commit order must preserve the partial orders obtained by restricting the synchronization orders to the actions in the corresponding commit set C_i,
4. the values written by actions in C_i must be the same as those written by these actions in E,
5. the values read in by actions in C_{i-1} must be the same as those read by these actions in E,
6. the values read in by actions in $C_i - C_{i-1}$ must be written by a write in C_{i-1} but are not restricted to be the same as those in E, moreover, they must be written by a write that happens-before the corresponding read in C_i,
7. release-acquire pairs appearing in a step i must persist in later steps $j \geq i$,

8. if an action is committed, all external actions that happen-before it are also committed.

3.2.3. JMM Guarantees

JMM also gives a number of guarantees to compiler developers, JVM implementers, and Java programmers [72, §3]. We shortly present these guarantees in this section.

DRF: Data-race-free programs, also referred as correctly synchronized, have sequentially consistent semantics.

Reorder 1: Adjacent statements that are independent can be reordered.

Reorder 2: Actions that a compiler can detect that always end up happening, with the same side-effects, can be re-ordered regardless of dependencies.

Happens-Before (HB): Volatile writes are ordered before subsequent volatile reads of the same variable. Monitor exits are ordered before subsequent monitor enters of the same monitor.

Redundant-Synchronization (RS): Synchronization actions that only introduce redundant happens-before edges can be treated as if they don't introduce any happens-before edges.

Roach Motel: Accesses outside of a critical section can be moved inside it, but not the opposite.

Volatile Atomicity: All accesses to volatile variables are performed in a total order.

Strong Volatile: There must be a happens-before relationship from each write to each subsequent read of that volatile.

Thin-air 1: As long as the early execution of a write does not result in subsequent reads seeing non-sequentially consistent values, that write can occur earlier in an execution than it appears in program order.

Thin-air 2: As long as the early execution of a write does not result in reads seeing values via a data race, that write can occur earlier in an execution than it appears in program order.

Isolation: Consider a partitioning of the threads and variables of a program, such that all threads that accessed a variable are grouped in the same partition P along with that variable, as well as any other variables they access. Given P, one is able to understand the execution of the threads and the values of the variables contained in that partition without examining any other partition of the program.

Observable: The only reason that an action visible to the external world (e.g., a file read/write, program termination) might not be observable is if there is an infinite sequence of actions that might happen before it or come before it in the synchronization order.

The aim of this work is to provide an extension of JMM that exposes cache management operations, making their ordering requirements clear to the reader, while providing the same guarantees with JMM as described here.

Note:

For in depth understanding of JMM the reader is referred to the following resources:

- The JSR-133 Cookbook for Compiler Writers [96]
- JSR-133 Java Memory Model and Thread Specification [50]
- SPECIAL POPL ISSUE: The Java Memory Model (not published) [71]
- Manson's Ph.D. Thesis [72]

3.3. The Distributed Model

In this section we present our Java Distributed Memory Model (JDMM). We essentially extend the formalization of JMM by making it aware of cache management operations. We follow a similar approach to the x86 Total Store Order definition [79]. We first define an abstract machine model and then use it to define well-formed executions for JDMM.

3.3.1. The JDMM's Abstract Machine Memory Model

As stated in Chapter 1, larger numbers of cores lead processor designers towards using non-cache-coherent memories, also known as scratchpad memories. This approach simplifies the processor's design and verification, and improves performance and energy efficiency. On the other hand, it renders the application responsible for keeping the memory consistent. With the latter being a tedious and error prone process; especially in hierarchical architectures, featuring multiple levels of scratchpad memories, like SCC [39], Runnemede [20] and the Formic-Cube [66, 67].

In our abstract machine memory model, we assume a single level of scratchpad memories for simplicity. Although simpler, this abstract machine model maintains the same properties as a machine with multiple levels of scratchpad memories regarding consistency. Multiple levels of scratchpad memories are mainly used to improve performance with reasonable cost. The low level scratchpad memories are faster and more expensive per byte while the higher level ones are slower and the cost per byte is lower. Note that we omit hardware data caches from our abstract machine memory model. Hardware caches in non coherent architectures are usually software managed. As a result, we can treat them like software data caches. In the rest of this section, when we refer to caches we mean both hardware and software data caches.

To achieve better performance, the design of a software cache is usually heavily connected to the underlying design of the actual memory hierarchy. In our work we aim to abstract over the actual memory architecture as much as possible, while targeting distributed memory or non-cache-coherent memory systems. To do that, we keep a simplified model of a non-cache-coherent memory, where software caches copy remote data to and from a local scratchpad memory. Such a design improves performance by reducing network on chip (NoC) traffic and access latency. We assume three actions that the memory system can take to change the state of a software cache:

- **fetch**: copies data from remote scratchpads to the local scratchpad. The data copies in the local scratchpad are called *cached*.
- **write-back**: writes back *dirty* cached data to the remote scratchpad from where they were fetched. *Dirty* are any cached data that have been written in the local scratchpad and not yet written back.
- **invalidate**: removes cached data from the local scratchpad.

These actions are abstract enough to capture the behavior of a wide range of systems, including existing coherence protocols (i.e., MESI, MSI, MOESI). We believe that these actions could not be more abstract without hiding the communication or more concrete without supposing a specific memory hierarchy or messaging protocol.

Figure 3.2 presents an instance of our abstract machine. 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. We slice the scratchpad memory into

a local and a global slice. The local slice is used to *cache* remote data while the global slice is used as a slice of a global address space, similarly to Partitioned Global Address Space (PGAS) models. 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 slice; after finishing the job it can transfer back the new data, if any. In this abstract machine the state of the memory, marked as a light gray rectangle with dashed surrounding, is driven by the computation units. The only way to modify the state of the abstract machine's memory is by committing fetch, write-back and invalidate actions, as described above.

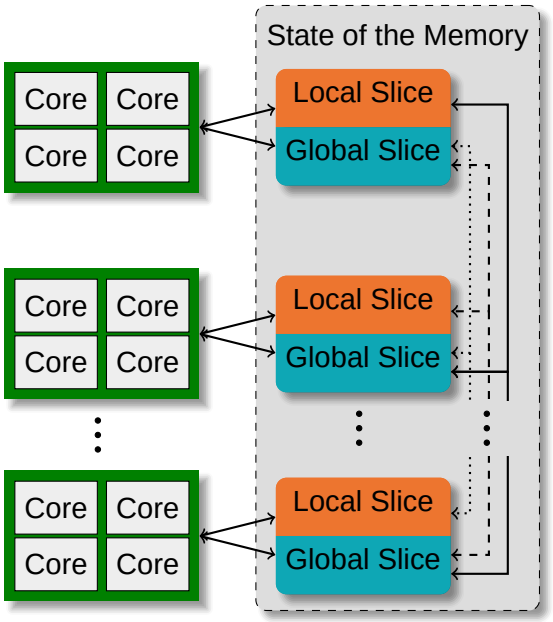


Figure 3.2.: The abstract machine

Although the local slice is used for caching remote data to increase efficiency and reduce data-transfers, it can also be used for all the local data (i.e., Java stacks). Overall, the Java Heap is split and stored across all global slices, so that a total —virtual— Java Heap consists of all the contents of all global slices, similarly to Partitioned Global Address Space (PGAS) models.

3.3.2. The Java Distributed Memory Model

To simplify the definition of the core JDMM and the reasoning about its correctness, we initially assume that all accesses to heap-based locations go through the cache, even when they reside in the local heap-slice. Furthermore, we assume that threads may not be context switched or migrated, thus caches are private per Java thread. We also do not take into account final fields, which are specially handled by JMM. We later

examine and discuss these subjects in Section 3.3.3, Section 3.3.4, Section 3.3.5, and Section 3.3.7.

Regarding the way that software managed caches operate we make the following assumptions aiming to cover as many cache models as possible. As in hardware caches, when reading a variable, the system first checks if the variable at hand is present in the cache. When present, the cache immediately returns the cached data, whereas if not present it produces a cache miss. In hardware caches, cache misses are usually handled automatically by initiating a data transfer from the main memory to the cache while blocking until this transfer completes. When done the cache returns the fetched value and the execution proceeds. In software managed caches however it is not necessary to handle cache misses this way, thus in our model we assume that cache misses are explicitly handled by the cache. As a result we also assume that a cache can explicitly issue write-backs, fetches, and self-invalidations. Additionally, we allow writes directly on the cache without the need to previously fetch the corresponding variable, essentially assuming that the cache operates at variables' granularity or finer. Each cache slot is assumed to feature a dirty and a valid bit, to show whether it needs to be written back and if it is valid. Self-invalidates set the valid bit to zero, while write-backs set the dirty bit to zero and transfer the dirty data back to the main memory. Cache slots hold up to a single value per variable, meaning a cache can only hold a single version per variable.

Our JDMM definition and formalization extends the definition of JMM discussed in Section 3.2. We extend the notion of an action with additional kinds that capture the extra functionality of the abstract machine described in Section 3.3.1, and extend the definition of an execution with additional constructs that capture relations between these additional actions.

In the rest of this section, we refer to hardware threads as *cores* and to Java threads as *threads*. Furthermore, unless explicitly specified otherwise, we use variable t for threads, v for variables, r for read actions (including volatile reads), w for write actions (including volatile writes), i for initialization actions, s for thread start actions, f for fetch actions, p for invalidation actions, b for write-back actions, and x, y, z to name actions of any kind. We summarize the meaning of each symbol in Table 3.3.

To define JDMM we extend JMM as follows.

Cache Actions: We extend the available action kinds defined in Section 3.2 by adding three new action kinds that can change the cache state (see Section 3.3.1). Namely, these action kinds are:

- F for fetches of heap-based variables,
- B for write-backs of heap-based variables,
- Iv for invalidations of cached variables.

Table 3.3.: Commonly used symbols

Symbol	Description
t	Thread
v	Variable
r	Read action
w	Write action
i	Initialization action
s	Thread start action
f	Fetch action
p	Invalidate action
b	Write-back action
x, y, z	Actions of any kind

Note that actions of kind F or B are also synchronization actions, since they implement a communication channel between threads, so we adapt the definition of the set SA accordingly. We denote the set of all actions in an execution of the JDMM as A_D. In Table 3.4 we present a revised version of Table 3.1 including the new action kinds.

Functions: We introduce four new functions, similar to $V()$ and $W()$:

- The *cache-action-seen* function $Cs(r)$, which for each read r gives the fetch or write action, that cached $r.v$, seen by r. Note that $Cs(r) \leq^d_{po} r$, $Cs(r).v = r.v$, and $Cs(r).k \in \{W, F\}$.

- The *write-back-fetched* function $Bf(f)$, which for each fetch f gives the write-back action whose data f fetches. $Bf(f) \leq^d_{so} f$, where $\leq^d_{so}$ is the analogous of $\leq_{so}$ for A_D instead of A, $Bf(f).v = f.v$, and $Bf(f).k = B$.

- The *action-written-back* function $Ab(b)$, which for each write-back b gives the initialization write, write, or volatile write action that b writes back. $Ab(b) \leq^d_{po} b$, $Ab(b).v = b.v$, and $Ab(b).k \in \{In, W, Vw\}$.

- The *action-invalidated* function $Ai(p)$, which for each invalidation p gives the write or fetch action that cached the data that p invalidates. $Ai(p) \leq^d_{po} p$, $Ai(p).v = p.v$, and $Ai(p).k \in \{W, F\}$.

Note also that for each function f above, $f(x).v = x.v$.

Table 3.4.: Abbreviations for JDMM Action Kinds

Abbreviation	Description
R	Non-volatile read
W	Non-volatile write
In	Initialization write
Vr	Volatile read
Vw	Volatile write
L	Monitor enter
U	Monitor exit
St	Thread start
Fi	Thread final action
Ir	Thread interrupt
Ird	Interrupt detection
Sp	Thread spawn (`Thread.start()`)
J	Thread join (`Thread.join()`)
Ex	External actions and I/O
F	Fetch action
B	Write-back action
Iv	Invalidation action

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; A_D is a set of actions (including cache actions); the program order $\leq^d_{po}$ is a relation on A_D defining the order of actions per thread; the synchronization order $\leq^d_{so}$ is a relation on A_D defining a global ordering among all synchronization actions in A_D; the function $W()$ on A_D returns the write action seen by every read action in A_D; the function $V()$ on A_D returns the value written by every write action in A_D; the functions Cs, Bf, Ab, and Ai act as described in the previous paragraph; the distributed synchronizes-with order $\leq^d_{sw}$ is the equivalent of $\leq_{sw}$ for the set of actions A_D, defining which actions in A_D synchronize; and the happens-before order $\leq^d_{hb}$ is the equivalent of $\leq_{hb}$ for the set of actions A_D, it defines a partial order among actions in A_D and is the transitive closure of $\leq^d_{po}$ and $\leq^d_{sw}$. We summarize the definitions of the notation used in this tuple in Table 3.5.

Well-Formed Distributed Execution: JDMM defines well-formed executions similarly to JMM, by extending the set of well-formedness constrains with ten intuitive rules. The additional rules we present are derived from the abstract machine memory model

Table 3.5.: Definition of JDMM Notation

Notation	Description
E_D	Distributed program execution
P	Java program
A_D	Set of actions
$\leq^d_{po}$	Program order
$\leq^d_{so}$	Synchronization order
$W()$	Write-seen function
$V()$	Value-written function
$Cs()$	Cache-action-seen function
$Bf()$	Write-back-fetched function
$Ab()$	Action-written-back function
$Ai()$	Action-invalidated function
$\leq^d_{sw}$	Synchronizes-with order
$\leq^d_{hb}$	Happens-before order
$\leq^d_{co}$	Cache order

discussed in Section 3.3.1. For brevity, in our formal definitions we use A_D instead of $E_D.A_D$ to denote a distributed execution's set of actions. We use $WF_D(E_D)$ to show that the distributed execution E_D is well formed. Specifically, a distributed execution E_D is well-formed when:

WF-1 – WF-9: It satisfies conditions **WF-1** through **WF-9** as defined in Section 3.2, for the new definitions of action set A_D, functions $W()$, $V()$, and relations $\leq^d_{po}$, $\leq^d_{so}$, $\leq^d_{sw}$ and $\leq^d_{hb}$ over A_D. For completeness we repeat conditions **WF-1** – **WF-9** below.

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

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

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

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

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:

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

WF-5: Lock operations are *consistent with mutual exclusion*. The number of lock actions performed by any thread t' before the lock action l performed by thread t on the monitor m, according to the synchronization order, must be equal to the number of unlock actions performed by thread t' before l on the monitor m:

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

where T is the set of all the execution threads:

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

WF-6: The execution obeys *intra-thread consistency*.

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

WF-7: The execution obeys *synchronization-order consistency*:

$$\begin{aligned}\forall r \in A_D : (r.k = Vr) \Rightarrow \Big(\neg\big(r \leq_{so} W(r)\big) \wedge \nexists w \in A_D : (w.k = Vw) \\ \wedge (w.v = r.v) \wedge \big(W(r) \leq^d_{so} w \leq^d_{so} r\big)\Big)\end{aligned}$$

WF-8: The execution obeys happens-before consistency:

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

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

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

WF-10: Every read is preceded, according to program order, by a write or fetch action, acting on the same variable as the read. Formally:

$$\forall r \in A_D : \exists x \in A_D : \big(x \leq^d_{po} r\big) \wedge x.v = r.v) \wedge \big(x.k \in \{W, F\}\big)$$

Recall that in our model all reads of heap-based variables see cached values, and that context switching and thread migration are not supported. As a result, all reads must find a copy of the corresponding variable in the cache. Since data get cached only through writes or fetches, every read must be preceded, according to program order, by at 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, according to program order, the action that cached it and the read that sees it. Formally:

$$\forall r, x \in A_D : \big(x = Cs(r)\big) \Rightarrow \nexists y \in A_D : (y.k \in \{Iv, F, W\}) \wedge (x.v = y.v) \wedge (x \leq^d_{po} y \leq^d_{po} r)$$

Since the cache can only hold a single version per variable, this will be the last write or fetch of that variable, according to program order. Additionally an invalidation results in the removal of the cached copy from the cache. As a result, there cannot be any invalidation, update, or overwrite of a variable's cached value between the action that cached it and the read that sees it.

In our memory model we are reasoning about cached data invalidations, while there is no obvious use of them in JMM. In an ideal machine with infinite caches, invalidations would be redundant. However, in real machines the cache size is limited. At some point the cache becomes full and to fetch a variable it is obligatory to invalidate another cached variable. If invalidations were left out of the memory model we would not be able to reason about its correctness, especially when such evictions and invalidations are not implemented in hardware, but are rather part of the JVM.

WF-12: Fetch actions are preceded, according to synchronization order, by at least one write-back of the corresponding variable. Formally:

$$\forall f \in A_D : \exists b \in A_D : \big(b.v = f.v\big) \wedge (b \leq^d_{so} f)$$

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.

WF-13: Write-back actions are preceded, according to program order, by at least one write to the corresponding variable. Formally:

$$\forall b \in A_D : \exists w \in A_D : (b.v = w.v) \wedge (w \leq^d_{po} b)$$

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

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

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

Since the cache can only hold a single version per variable, this will be the last write or fetch of that variable, according to program order. As a result, there cannot be any other writes to the same variable between a write and its write-back.

WF-15: Only cached variables are invalidated. Formally:

$$\forall p \in A_D : \exists x \in A_D : \nexists p' \in A_D : (Ai(p) = x) \wedge \big(Ai(p) = Ai(p')\big)$$

WF-16: Reads that see writes performed by other threads are preceded, according to program order, by a fetch action that fetches the data of the corresponding write, which were written back, and there is no other write-back of the corresponding variable happening between the write-back and the fetch, according to synchronization order. Formally:

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

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.

WF-17: Volatile writes are immediately written back, in the sense that no other action happens between the volatile write and its write-back, according to the program order. Formally:

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

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 JMM.

WF-18: A fetch of the corresponding variable happens immediately before each volatile read, in the sense that no other action happens between the corresponding fetch and the volatile read, according to the program order. Formally:

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

The unconditional fetch of volatile variables before volatile reads is mandatory to ensure that the volatile read will *see* the latest, according to happens-before order, volatile write to the corresponding variable. Additionally, 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 JMM.

Note that in well-formedness constraints **WF-17** and **WF-18**, we chose not to embed the write-back and fetch actions in the corresponding synchronization actions, e.g., consider volatile writes to be of both kinds Vw and B. This way, the model remains simple and as non-intrusive as possible to JMM. However, the JDMM requires that the write-backs of volatile writes and the fetches of volatile reads are executed immediately after and before the corresponding volatile action, respectively.

WF-19: Initialization writes are immediately written back. Formally:

$$\forall x \in A_D : (x.k = In) \Rightarrow \exists b \in A_D : \nexists y \in A_D : \Big(b = Ab(x)\Big) \wedge (b \leq^d_{po} y \leq^d_{po} x)\Big)$$

Although counter-intuitive this rule is needed to conform to **WF-9**.

Fetching already cached variables (re-fetch): Up to this point we did not mention the case where a thread needs to fetch a variable which is already cached. According to the well-formedness conditions, a thread needs to fetch a variable:

1. to read it, if not cached (see condition **WF-10**);
2. to read a value written to it by a different thread (see condition **WF-16**);
3. at every volatile read (see condition **WF-18**).

Case 1 is trivial, since **WF-10** is satisfied for already cached variables and there is no need to re-fetch the variable. In case 2, when a thread performs a read r that sees a write $W(r)$ performed by another thread, according to **WF-16** r is preceded by a fetch action f that fetches the data of $W(r)$, which were written back, and there is no other write-back of the corresponding variable $r.v$ happening between the write-back and the fetch, according to synchronization order. In this case, if $r.v$ is already cached before f and that cached value is dirty, then there exists a write w, where $r.v = w.v \wedge w.t = r.t$ that happens-before r. Under DRF programs w and $W(r)$ are ordered through happens-before and since $r.t$ needs to re-fetch $r.v$ then w happens-before $W(r)$ and $V(w)$ can be dropped, since it is not the latest value, according to happens-before order. For non-DRF programs, there is no guarantee that incorrectly synchronized writes will be seen by some read, so if w is not ordered with $W(r)$ through happens-before order, dropping the dirty data is acceptable since the value seen by r is the result of a data race. As a

result, we can unconditionally fetch variables when needed and overwrite any cached data. Finally in case 3, volatile variables are only written through volatile writes, which immediately write-back the corresponding variable (see condition **WF-17**). As a result, a volatile variable cannot be observed as dirty by any action.

3.3.3. No Local Caching Optimization

In Section 3.3.2, we assume that all accesses to heap-based locations go through the cache. However, caching data for the local heap-slice increases the memory traffic of the system and slows down the application. We argue that JMM implementers can avoid data caching for data that reside in a core's local heap-slice. For writes, it is straightforward to show that not using a cache will have the same behavior. Assume a write-through and blocking cache, under this configuration, writes are immediately written to the local heap-slice as they would do if they were not cached. Regarding reads, not caching data from the local-heap slice is identical to fetching the data before every read. This behavior is similar to having a tiny cache that results in the invalidation of the cached data at every fetch. As a result, not caching data from the local heap-slice does not impact JMM's properties and JVM implementers could follow this approach when more efficient.

It is important to note that the above reasoning is valid even if we choose not to cache data residing in a remote heap-slice. This enables the JVM implementer to have a hybrid system where some cores use data caches while others do not. Such a hybrid JVM could be utilized on heterogeneous architectures where some cores have access to coherent hardware caches while others do not.

3.3.4. Context Switching and Cache Sharing

Context switching is the process where a thread stops running to allow another thread to run on the same core. To the best of our knowledge, in all available systems caches are shared between the different software threads of an application. As a result, we expect JVM implementers to also allow their caches to be shared between Java threads of the same application. That said, context switching can also affect an execution and the writes seen by reads. A problem may arise by context switching when a thread stops executing right after fetching a variable and continues its execution right after another thread has invalidated this variable. In these cases an immediate read of this variable may return an *out-of-thin-air* value for non cache coherent systems. In cache coherent shared memory systems this issue is implicitly resolved by producing a cache miss and fetching the variable. On the contrary, on non cache coherent systems this needs to be explicitly resolved. To model this in JDMM we need a way to argue about the ordering of actions acting on the same cache. To do this we introduce a new partial order over

cache and memory access actions and modify some of the introduced functions and well formedness rules to use this new partial order instead of the program order.

Cache order: In order to argue about the ordering of cache and memory access actions —actions of kind F, B, Iv, R, W, Vw, Vr, or In— acting on the same cache we introduce the partial order $\leq^d_{co}$. The cache order $\leq^d_{co}$ defines a total order over all cache and memory access actions acting on a single cache. We use $x \leq^d_{co} y$ to show that x and y act on the same cache, and that x comes before y according to the cache order. Note that in implementations with private per thread caches the cache order is equal to the program order.

Functions: Additionally, we modify the *cache-action-seen*, *action-written-back*, and *action-invalidated* functions to use the cache order instead of the program order. We modify these functions as follows:

- The *cache-action-seen* function $Cs(r)$, which for each read r gives the fetch or write action, that cached $r.v$, seen by r. $Cs(r) \leq^d_{co} r$, $Cs(r).v = r.v$, and $Cs(r).k \in \{W, F\}$.
- The *action-written-back* function $Ab(b)$, which for each write-back b gives the initialization write, write, or volatile write action that b writes back. $Ab(b) \leq^d_{co} b$, $Ab(b).v = b.v$, and $Ab(b).k \in \{In, W, Vw\}$.
- The *action-invalidated* function $Ai(p)$, which for each invalidation p gives the write or fetch action that cached the data that p invalidates. $Ai(p) \leq^d_{co} p$, $Ai(p).v = p.v$, and $Ai(p).k \in \{W, F\}$.

Well Formedness Conditions: Finally we change the following well formedness rules to use the cache order instead of the program order.

WF-10: Every read is preceded, according to cache order, by a write or fetch action, acting on the same variable as the read. Formally:

$$\forall r \in A_D : \exists x \in A_D : x \leq^d_{co} r \wedge x.v = r.v \wedge x.k \in \{W, F\}$$

This change enables different threads to fetch data that another thread may access from the shared, among them, cache.

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

$$\forall r, x \in A_D : \big(x = Cs(r)\big) \Rightarrow \nexists y \in A_D : (y.k \in \{Iv, F, W\}) \wedge (x.v = y.v) \wedge (x \leq^d_{co} y \leq^d_{co} r)$$

This change ensures that all threads see the latest, according to the cached order, state of a cached value. Additionally it forbids (by making them not well formed) executions where a thread stops executing right after fetching a variable and continues its execution right after another thread has invalidated this variable, like the one discussed earlier.

WF-13: Write-back actions are preceded, according to cache order, by at least one write to the corresponding variable. Formally:

$$\forall b \in A_D : \exists w \in A_D : (b.v = w.v) \wedge (w \leq^d_{co} b)$$

This change enables different threads to write-back writes performed by another thread sharing the same cache.

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

$$\forall b, w \in A_D : \big(w = Ab(b)\big) \Rightarrow \nexists w' \in A_D : \big((w'.v. = w.v) \wedge (w \leq^d_{co} w' \leq^d_{co} b)\big)$$

This change ensures that all threads write-back the latest, according to the cached order, cached value.

WF-16: Reads that see writes acting on a different cache are preceded, according to cache order, by a fetch action that fetches the data of the corresponding write, which were written back, and there is no other write-back of the corresponding variable happening between the write-back and the fetch, according to synchronization order. Formally:

$$\begin{aligned}\forall r \in A_D : \neg \big(W(r) \leq^d_{co} r\big) \\ \Rightarrow \exists b, f \in A_D : \Big(Ab(b) = W(r) \wedge Bf(f) = b \wedge f \leq^d_{co} r \\ \wedge \big(\nexists b' : b'.v = b.v \wedge b \leq^d_{so} b' \leq^d_{so} f\big)\Big)\end{aligned}$$

With the above modifications a context switch can cause a thread to see the data of another thread, but is this allowed under JDMM? JMM allows threads to see writes performed by other threads even when they are not ordered by happens-before. The only case where this can cause a problem, is when a thread t' changes the write that a different thread t would otherwise observe. There are three ways to cause this:

a) Thread t' writes the variable in question (e.g., v), and thread t sees this write.

b) Thread t' invalidates a variable v and thread t fetches it from the main memory, where its value is different from the old cached value.

c) Thread t' fetches a variable v from the main memory, where again its value is different from the old cached value, and thread t sees this new value.

Note that all three scenarios are only possible under non-DRF programs. In the first scenario, thread t sees the value written by thread t'. The corresponding write in thread t' and read in thread t are conflicting accesses. If the data-race does not occur, then thread t' happens-before thread t and thread t sees thread's t' write anyway (assuming there are no other threads running). The second and third scenarios, produce the same behavior, they both update the cached value. If thread t sees a different value from the one it would see if there was no context switch, then there is a write w that does not happen-before the read r performed by thread t. This implies a data race between w and r. Under non-DRF programs the only guarantee that should hold, is the *out-of-thin-air* guarantee. Since, under well formed executions, initialization actions happen-before the first action of any thread context switching cannot result in *out-of-thin-air* values.

As a result, our model allows context switching without violating JMM. Context switching enables multiple threads to share the same core and the same cache. As a result, a cache can be shared by multiple cores having the same access time to it. This way, memory can be better utilized, by reducing the number of variable replicas across the system.

3.3.5. Thread Migration

Thread migration can also affect an execution and the writes seen by reads. Thread migration occurs when a thread t is moved to a different core, than the one it is currently running, to continue its execution. At thread migration, there are three scenarios that can cause a thread t to observe a different value than the one it would observe if it would not migrate, these are:

a) Thread t accesses a dirty cached variable at its *new* cache.

b) Thread t tries to access a variable that it had previously cached in its *old* cache, but does not find the variable cached and fetches it from the main memory, with a different value from the one cached at its *old* cache

c) Thread t accesses a variable that is already cached to its *new* cache, but with a different value than the one cached at its *old* cache.

In the first scenario, thread t sees the value written by another thread t'. This is exactly the same scenario with scenario a) discussed for context switching in Section 3.3.4. The second scenario again produces the same behavior with scenarios b) and c) discussed for context switching. The third scenario, however, is different. In the case of thread migration, thread t can see an older value than the one cached at its *old* cache. This may result in executions inconsistent to the happens-before order and is possible under both DRF and non-DRF programs. As a result, to preserve the happens-before order, migrating threads need to re-fetch all variables before accessing them for the first time after the migration. To express this formally we introduce a new (synthetic) action kind

that denotes the migration of a thread. We use the abbreviation M for such actions and $M.v$ is the migrating thread. To allow migration under JDMM while preserving its adherence to JMM we introduce two new well-formedness conditions.

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

$$\forall m, r \in A_D : \big((m.k = M) \wedge (m \leq^d_{po} r)\big) \Rightarrow \exists x \in A_D : \big((x = Cs(r)) \wedge (m \leq^d_{co} x \leq^d_{co} r)\big)$$

WFE-2: Additionally, to ensure that the fetched value is the latest according to the happens-before order, any dirty data on the *old* cache need to be written back. Formally:

$$\forall m, w \in A : \big((m.k = M) \wedge (w \leq^d_{po} m)\big) \Rightarrow \exists b \in A : \big((w = Ab(b)) \wedge \big(w \leq^d_{co} b \leq^d_{co} m\big)\big)$$

3.3.6. Garbage Collection

The use of caches introduces some extra overhead to copying garbage collectors. JMM requires that the relocation of a variable or its reuse is not observable by the semantics. To achieve this, any relocated and garbage collected addresses must be invalidated. If a cache does not invalidate these objects, reads might see *out-of-thin-air* values. The example in Table 3.6 demonstrates such a case.

Table 3.6.: Garbage Collection Example

	Class Definition	Thread 1	Thread 2
1	`class A {`	`// ...`	`// ...`
2	`  public int v;`	`A a = new A(1);`	`int e = a.v;`
3	`  A(int i) { v = i; }`	`  a = null;`	`int f = b.v;`
4	`}`	`A b = new A(4);`	

`e == f == 1` is unacceptable

To simplify our reasoning we assume that the compiler does not optimize the code. The possible values of a and b are a valid memory address or `null`. As a result, one would expect this program to throw a `NullPointerException` or have one of the following states at the end of the run (`e == 1, f == 4`), (`e == 1, f == 0`) and (`e == 0, f == 0`). The zero values can be observed if the constructor's store does not commit before the corresponding access from *T2*. Note that this is acceptable since this is not a DRF program and according to JMM there is no ordering between the *construction* of an object and its accesses. However, `e == f == 1` is not acceptable. `f` can only take

values zero and four. Zero is the default value of `b.v` while four is the value that `b.v` gets from the constructor. That said, any other value (i.e., one) is *out-of-thin-air*.

If a garbage collection happens after executing Thread's 1 line 3 (e.g., by a reference counting garbage collector), the memory address previously referenced by variable `a` can be garbage collected and reused in Thread's 1 line 4. If this address is cached in Thread's 2 cache, which means that `e = a.v` in Thread 2 happened before the garbage collection and the cache is not invalidated, then the execution of `f = b.v` will hit the cache and read a bogus value, leftover from `a`. As a result, `f` is assigned the value one.

3.3.7. Final Fields

As in JMM, we chose to omit final fields from the core model and discuss them separately in this section. Intuitively, in Java a final field may only be written once by the constructor of the object it belongs to, with the exception of some cases, like objects implementing the `java.io.Serializable` interface, where an object's final fields need to be updated after construction. Similarly, static final fields may only be written by the corresponding class's initializer. JMM also handles specially some of the static final fields. Namely those are `java.lang.System.in`, `java.lang.System.out`, `java.lang.System.err`. These fields, although static and final, may be altered through the functions `java.lang.System.setIn`, `java.lang.System.setOut`, `java.lang.System.setErr`, thus JMM does not treat them the same as other static final fields allowing them to be altered after construction.

Manson [70, §7] introduces a *freeze* action to define the semantics of final fields. *Freeze* is used to mark a final field as frozen, meaning it may not be written any more. Each final field gets frozen at the end of the constructor in which it is set (even if the constructor exits abnormally). Note that in some cases, e.g., deserialization, multiple freezes are allowed per final field. This way *serializable* objects may freeze their final fields at construction and later at deserialization. Note, however, that this is not a generic rule and only applies to special cases like deserialization. In this work we use Fr to denote the kind of freeze actions.

JMM guarantees that threads that only read references to an object that were written after the last freeze of its fields will always see the frozen values of the final fields. Manson calls such references *correctly published*, because they are only published after the object is initialized. Note that since final fields may be references as well, there may be objects that are reachable through them. In such cases JMM guarantees that reads of those objects will see values at least as up to date as they were when the freeze of the final field was performed.

In the case where a thread reads a reference to an object that was written before the last freeze of its fields, JMM requires that the above guarantees are enforced through

the happens-before order. That is, it requires that any reads to the object's final fields come after, according to happens-before, the freeze actions of that final fields.

In the process of formalizing the above semantics Manson extends executions by introducing two additional partial orders of actions, dereference chain ($\leq_{dc}$) and memory chain ($\leq_{mc}$). The dereference chain order is used to show the order between an object's field or element access x and the read r that sees the address of that object ($r \leq_{dc} x$). Note that the $\leq_{dc}$ order is reflexive, thus r can be the same as x. According to JMM, for every action x that accesses a field, or element in the case of arrays, of an object o that was not constructed by $x.t$ there exists a read r, where $r.t = x.t$, that sees the address of o such that $r \leq_{dc} x$. The memory chain is a bit more complex. According to JMM:

- $\forall r \in A : W(r) \leq_{mc} r$,
- $\forall r, x \in A : r \leq_{dc} x \Rightarrow r \leq_{mc} x$,
- For every action w that is a write of the address of an object o that was not constructed by $w.t$ there exists a read r, where $r.t = w.t$ such that $r \leq_{mc} w$. [1]

Having defined these two additional partial orders, Manson extends JMM to define which writes can be seen by reads of final fields. The happens-before order is used as previous except for the cases where a read acts on a final field and either the write of that field is performed by a different thread or it is the result of a special mechanism such as deserialization. In such cases,

$$\forall w, x, y, r, r' \in A : \left((x.k = Fr) \wedge (x.v = r.v) \wedge w \leq_{hb} x \leq_{hb} y \leq_{mc} r \leq_{dc} r'\right) \Rightarrow w \leq_{hb} r'$$

where y is not a read of a final field and $w \leq_{hb} r'$ is not transitively closed with other $\leq_{hb}$ orderings. As in the core model a read r can see a write w if $w \leq_{hb} r$ and $\nexists w' \in A : w \leq_{hb} w' \leq_{hb} r$.

Since the extensions of JMM for final fields builds on the happens before order it is also applicable on JDMM without the need of any alternations.

3.3.8. Direct Transfers Across Local Memories

In some architectures it is possible to directly transfer data from one local slice to another, essentially avoiding the transfer to and from the main memory in some cases. In our abstract machine memory model (Section 3.3.1) we omit such communication channels however. This allows us to keep JDMM simple and make clear when the JVM should perform write-backs to and fetches from the main memory, which we consider to be the most common case when designing a software cache based JVM. We acknowledge, however, that in some cases a JVM could exploit the direct memory access

[1] JMM defines one more case regarding *final field safe contexts*. In this work, we do not include this case as well as a discussion of *final field safe context*, since this mechanism is optional and implementation specific.

support to optimize performance and energy efficiency. To argue about the correctness of executions with such optimizations, where a variable may be directly fetched from another software cache instead of the main memory, it suffices to consider these direct transfers as a combination of a write-back action and a fetch action, to and from an artificial main memory. This is essentially equivalent to considering the write-back a no-op and allowing fetch actions to fetch data from a different place than the main memory. Note that, although this would be sufficient to allow such optimizations under JDMM, JDMM is stricter than necessary since **WF-13** only allows write-backs that are preceded by at least one write to the corresponding variable. That said, JDMM does not allow the propagation of a cached variable from caches of threads that do not write the variable, limiting the optimization window.

3.4. On JDMM's adherence to JMM

In this section we argue that JDMM adheres to JMM, while allowing a more detailed modeling of non cache coherent or distributed memory implementations. The argument consists of a construction of a shared memory well-formed execution from a given distributed memory well-formed execution.

We show that given a distributed well-formed execution trace, E_D, we can transform it to a shared-memory well formed execution E.

We use $[\![E_D]\!] = E$ to show that E is constructed from E_D. Then we show that:

$$\forall E_D : WF_D(E_D) \Rightarrow WF([\![E_D]\!])$$

Assuming a JDMM execution:

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

such that $WF_D(E_D)$, the construction $[\![E_D]\!]$ results in a new JMM execution:

$$[\![E_D]\!] = \langle P, A, \leq_{po}, \leq_{so}, W(), V(), \leq_{sw}, \leq_{hb} \rangle$$

where A is the set of actions in A_D excluding all actions x with kind F, B, Iv, or M:

$$A = \{x \in A_D : x.k \notin \{F, B, Iv, M\}\}$$

In the same fashion, the relations $\leq_{po}$, $\leq_{so}$, $\leq_{sw}$ and $\leq_{hb}$ are the projections of $\leq^d_{po}$, $\leq^d_{so}$, $\leq^d_{sw}$ and $\leq^d_{hb}$, respectively, that only refer to actions in A.

$$\leq_{po} = \leq^d_{po} |_A$$
$$\leq_{so} = \leq^d_{so} |_A$$
$$\leq_{sw} = \leq^d_{sw} |_A$$
$$\leq_{hb} = \leq^d_{hb} |_A$$

To show that $[\![E_D]\!]$ is well-formed it suffices to show that the well-formedness conditions **WF-1** – **WF-9** as defined in Section 3.2 are satisfied by it. From assumption $WF_D(E_D)$ we have that conditions **WF-1** – **WF-9** hold for A_D, $\leq^d_{po}$, $\leq^d_{so}$, $W()$, $V()$, $\leq^d_{sw}$ and $\leq^d_{hb}$. As none of these conditions refers to actions with kinds F, B, Iv, M, and relations $\leq^d_{po}$, $\leq^d_{so}$, $\leq^d_{sw}$, $\leq^d_{hb}$ are supersets of $\leq_{po}$, $\leq_{so}$, $\leq_{sw}$, $\leq_{hb}$, respectively, then none of the actions removed from A_D to A cause any of the conditions **WF-1** – **WF-9** to break.

3.5. On JDMM's expressiveness over JMM

In this section we argue that JDMM is as expressive as JMM, while allowing a more detailed modeling of non cache coherent or distributed memory implementations. The argument consists of a construction of a distributed memory well-formed execution from a given shared memory well-formed execution.

We show that given a shared-memory well-formed execution trace, E, we can transform it to a distributed well formed execution E_D.

We use $[\![E]\!]' = E_D$ to show that E_D is constructed from E. Then we show that:

$$\forall E : WF(E) \Rightarrow WF_D([\![E]\!]')$$

Assuming a JMM execution:

$$E = \langle P, A, \leq_{po}, \leq_{so}, W(), V(), \leq_{sw}, \leq_{hb} \rangle$$

such that $WF(E)$, the construction $[\![E]\!]'$ results in a new JDMM execution:

$$[\![E]\!]' = \langle P, A_D, \leq^d_{po}, \leq^d_{so}, W(), V(), Cs(), Bf(), Ab(), Ai(), \leq^d_{sw}, \leq^d_{hb} \rangle$$

where:

- A_D is the set of actions in A enriched with one write-back action per write action, and one fetch, one invalidate and another fetch action per read action.
- $\leq^d_{po}$ is a superset of $\leq_{po}$, $\leq^d_{po} \supseteq \leq_{po}$, enriched with edges between the actions in A and the additional actions in A_D. Formally:

$$\forall x, y \in A_D : x \leq_{po} y \Rightarrow x \leq^d_{po} y \tag{3.1}$$

$$\forall w, y \in A_D : w \leq_{po} y \Rightarrow \left(\exists b \in A_D : (w.v = b.v) \wedge (w \leq^d_{po} b \leq^d_{po} y)\right) \tag{3.2}$$

$$\forall x, r \in A_D : x \leq_{po} r \Rightarrow \left(\exists f, p, f' \in A_D : (r.v = f.v = p.v = f.v) \wedge (x \leq^d_{po} f \leq^d_{po} p \leq^d_{po} f' \leq^d_{po} r)\right) \tag{3.3}$$

$$\forall x, y, z \in A_D : x \leq^d_{po} y \leq^d_{po} z \Rightarrow x \leq^d_{po} z \tag{3.4}$$

$\leq^d_{so}$ is a superset of $\leq_{so}$, $\leq^d_{so} \supseteq \leq_{so}$, enriched with edges between the synchronization actions in A and the additional actions in A_D. Formally:

$$\forall x, y \in A_D : x \leq_{so} y \Rightarrow x \leq^d_{so} y \tag{3.5}$$

$$\forall x \in \mathrm{SA}(A) : \forall y \in \mathrm{SA}(A_D) : x \leq^d_{po} y \Rightarrow x \leq^d_{so} y \tag{3.6}$$

$$\forall x \in \mathrm{SA}(A) : \forall y \in \mathrm{SA}(A_D) : y \leq^d_{po} x \Rightarrow y \leq^d_{so} x \tag{3.7}$$

$$\forall x, y \in \mathrm{SA}(A) : \forall z \in \mathrm{SA}(A_D - A) : (x \leq_{so} y) \wedge (x \leq^d_{so} z) \Rightarrow (x \leq^d_{so} z \leq^d_{so} y) \tag{3.8}$$

$$\forall x, y \in \mathrm{SA}(A_D) : x \leq^d_{po} y \Rightarrow x \leq^d_{so} y \tag{3.9}$$

$$\forall x, y, z \in A_D : x \leq^d_{so} y \leq^d_{so} z \Rightarrow x \leq^d_{so} z \tag{3.10}$$

$Cs(r)$ returns the last, according to program-order, write or fetch (whichever comes last) that comes before r and acts on the same variable.

$$\begin{aligned} Cs(r) = x : (x \in A_D) \wedge (x.k \in \{W, F\}) \wedge (x.v = r.v) \wedge (x \leq^d_{po} r) \\ \wedge \left(\exists x' \in A_D : (x'.k \in \{W, F\}) \wedge (x'.v = r.v) \wedge (x \leq^d_{po} x' \leq^d_{po} r)\right) \end{aligned} \tag{3.11}$$

$Bf(f)$ returns the last, according to synchronization-order, write-back that comes before f and acts on the same variable.

$$\begin{aligned} Bf(f) = b : (b \in A_D) \wedge (b.v = f.v) \wedge (b \leq^d_{so} f) \\ \wedge \left(\exists b' \in A_D : (b'.v = f.v) \wedge (b \leq^d_{so} b' \leq^d_{so} f)\right) \end{aligned} \tag{3.12}$$

$Ab(b)$ returns the last, according to program-order, write that comes before b and acts on the same variable.

$$\begin{aligned} Ab(b) = w : (w \in A_D) \wedge (w.v = b.v) \wedge (w \leq^d_{po} b) \\ \wedge \left(\exists w' \in A_D : (w'.v = b.v) \wedge (w \leq^d_{po} w' \leq^d_{po} b)\right) \end{aligned} \tag{3.13}$$

$Ai(p)$ returns the last, according to program-order, write or fetch that comes before p and acts on the same variable.

$$\begin{aligned} Ai(p) = x : (x \in A_D) \wedge (x.k \in \{W, F\}) \wedge (x.v = p.v) \wedge (x \leq^d_{po} p) \\ \wedge \left(\exists x' \in A_D : (x'.k \in \{W, F\}) \wedge (x'.v = p.v) \wedge (x \leq^d_{po} x' \leq^d_{po} p)\right) \end{aligned} \tag{3.14}$$

$\leq_{sw}^{d}$ is equivalent to $\leq_{sw}$. Formally:

$$\forall x, y \in A_D : x \leq_{sw} y \Rightarrow x \leq_{sw}^{d} y \tag{3.15}$$

$\leq_{hb}^{d}$ is a superset of $\leq_{hb}$, $\leq_{hb}^{d} \supseteq \leq_{hb}$, enriched with edges between the actions in A and the additional actions in A_D. Formally:

$$\forall x, y \in A_D : x \leq_{hb} y \Rightarrow x \leq_{hb}^{d} y \tag{3.16}$$

$$\forall x, y \in A : \forall z \in A_D : (x \leq_{hb} y \wedge y \leq_{po}^{d} z) \Rightarrow x \leq_{hb}^{d} z \tag{3.17}$$

$$\forall x, y \in A : \forall z \in A_D : (x \leq_{hb} y \wedge z \leq_{po}^{d} x) \Rightarrow z \leq_{hb}^{d} y \tag{3.18}$$

$$\forall x, y \in A_D : x \leq_{po}^{d} y \Rightarrow x \leq_{hb}^{d} y \tag{3.19}$$

$$\forall x, y, z \in A_D : x \leq_{hb}^{d} y \leq_{hb}^{d} z \Rightarrow x \leq_{hb}^{d} z \tag{3.20}$$

To show that $[\![E]\!]'$ is well-formed it suffices to show that the well-formedness conditions **WF-1** – **WF-19** as defined in Section 3.3.2 are satisfied by it. From assumption $WF(E)$ we have that conditions **WF-1** – **WF-9** hold for A, $\leq_{po}$, $\leq_{so}$, W, V, $\leq_{sw}$ and $\leq_{hb}$.

WF-1, **WF-2**, and **WF-5**: hold for A_D, $\leq_{po}^{d}$, $\leq_{so}^{d}$, $\leq_{sw}^{d}$ and $\leq_{hb}^{d}$. since they do not depend on any of the $\leq_{po}$, $\leq_{so}$, $\leq_{sw}$ and $\leq_{hb}$.

WF-3: $[\![E]\!]'$ extends $\leq_{so}$ by introducing relations between the synchronization actions A and the new actions in $A_D - A$. Since the new relations are added in a manner that does not create circles and the number of new actions is finite, *the number of synchronization actions preceding another synchronization action y is finite* in $[\![E]\!]'$ as well.

WF-4: $[\![E]\!]'$ extends $\leq_{po}$ and $\leq_{so}$ by introducing relations between the synchronization actions A and the new actions in $A_D - A$. $\leq_{po}$ is extended by including relations between the writes and the introduced write-backs, as well as between the introduced fetches and invalidates, and the reads. $\leq_{so}$ is extended by including all the $\leq_{po}^{d}$ relations between synchronization actions and orders the new actions $A_D - A$ by placing them between the synchronization actions of A by respecting $\leq_{po}^{d}$. As a result, *synchronization order is consistent with program order*.

WF-6: $[\![E]\!]'$ extends $\leq_{po}$ by including relations between the writes and the introduced write-backs, as well as between the introduced fetches and invalidates, and the reads. The program-order is not altered in any other way and the reads and writes do not get re-ordered. As a result, *the execution obeys intra-thread consistency*.

WF-7: By **WF-4**, $\leq_{so}^{d}$ is consistent with $\leq_{po}^{d}$ and no reads or writes get re-ordered, that is no volatile reads or writes are re-ordered as well. As a result, *the execution obeys synchronization-order consistency*.

WF-8:$[\![E]\!]'$ extends $\leq_{po}$ by including relations between the writes and the introduced write-backs, as well as between the introduced fetches and invalidates, and the reads. $\leq_{sw}^{d}$ is equal to $\leq_{sw}$, the $\leq_{hb}^{d}$ is the same as $\leq_{hb}$ extended with relations between the new actions $A_D - A$ that are consistent with $\leq_{po}^{d}$ and no reads or writes get re-ordered. As a result, *the execution obeys happens-before consistency*.

WF-9: $\leq_{hb}^{d}$ is the same as $\leq_{hb}$ extended with relations between the new actions $A_D - A$ that are consistent with $\leq_{po}^{d}$ and no reads or writes get re-ordered. As a result, *every thread's start action happens-before its other actions except for initialization actions* as it did in E.

WF-10: According to $\leq_{po}^{d}$ (Equation (3.3)) there exists a fetch action right before each read action, according to program-order. As a result, *every read is preceded, according to program order, by a write or fetch action, acting on the same variable as the read*.

WF-11: According to $Cs()$ (Equation (3.11)) there is no other write (overwrite) or fetch (update) action between $Cs(r)$ and r. Additionally, according to $\leq_{po}^{d}$ (Equation (3.3)), invalidations appear between two fetches. As a result, *there is no invalidation, update, or overwrite of a variable's cached value between, according to program order, the action that cached it and the read that sees it*.

WF-12: According to $\leq_{po}^{d}$ (Equation (3.2)) there exists a write-back action b right after each write action (including initialization actions). According to **WF-9** *every thread's start action happens-before its other actions except for initialization actions*, $(i \leq_{hb}^{d} s \leq_{po}^{d} x)$, where s is the thread start action and x is any other action except for initialization. As a result there is at least a write (initialization) and a write-back, of a variable, that happen-before each read of that variable. Additionally, according to $\leq_{po}^{d}$ (Equation (3.3)) there exists a fetch action right before each read action. Since happens-before is the transitive closure of $\leq_{sw}^{d}$ and $\leq_{po}^{d}$, $\exists x, y \in \mathrm{SA}(A_D) : b \leq_{po}^{d} x \leq_{sw}^{d} y \leq_{po}^{d} f \wedge b \leq_{so}^{d} f$. As a result, *fetch actions are preceded, according to synchronization order, by at least one write-back of the corresponding variable*.

WF-13: According to $\leq_{po}^{d}$ (Equation (3.2)) there exists a write-back action right after each write action. As a result, *Write-back actions are preceded, according to program order, by at least one write to the corresponding variable*.

WF-14: According to $\leq_{po}^{d}$ (Equation (3.2)) there exists a write-back action right after each write action. As a result, *there are no other writes to the same variable between a write and its write-back, according to program order*.

WF-15: According to $\leq_{po}^{d}$ (Equation (3.3)) there exists a fetch action right before each invalidation action. As a result, *only cached variables are invalidated*.

WF-16: According to $\leq_{po}^{d}$ (Equation (3.2)) there exists a write-back action right after each write action, and a fetch action right before each read. Additionally, according to $Bf(f)$ a f fetches the last, according to synchronization order, write-back that comes before f and acts on the same variable. As a result *reads that see writes performed by other threads are preceded, according to program order, by a fetch action that fetches the data of the corresponding write, which were written back, and there is no other write-back of the corresponding variable happening between the write-back and the fetch, according to synchronization order*.

WF-17: According to $\leq_{po}^{d}$ (Equation (3.2)) there exists a write-back action b right after each write action (including volatile writes). As a result, *volatile writes are immediately written back, in the sense that no other action happens between the volatile write and its write-back, according to the program order*.

WF-18: According to $\leq_{po}^{d}$ (Equation (3.3)) there exists a fetch action right before each read action (including volatile reads). As a result, *a fetch of the corresponding variable happens immediately before each volatile read, in the sense that no other action happens between the corresponding fetch and the volatile read, according to the program order*.

WF-19: According to $\leq_{po}^{d}$ (Equation (3.2)) there exists a write-back action b right after each write action (including initialization actions). As a result, *initialization writes are immediately written back*.

Note that the construction $[\![E]\!]'$, described above, is not the only possible one. For instance JDMM allows multiple writes without the need to write-back each one of them when there is no release action between them, according to program-order. However, our construction $[\![E]\!]'$ places a write-back after every write, producing just one of many possible well-formed executions.

Furthermore, we examine whether JDMM satisfies the causality test cases [83] introduced by Manson et al. and whether it allows the same reorderings as JMM.

3.5.1. Causality Tests

The causality test cases [83] presented by Manson et al. mainly refer to compiler optimizations, thus they are satisfied by JDMM. However, for completeness we examine the cases one by one in tables 3.7–3.26. In each table we first show the test case, starting with the initial memory state and code segment on each thread, and ending with the behavior in question. For allowed behavior, we present the possible executions that could result in the behavior in question. We also show for each case whether it is allowed or not, both under JMM and JDMM. For each allowed test case we shortly discuss the execution producing the behavior in question both under JMM and JDMM. To present the execution we interleave the instructions executed by the different threads in each example. We present the interleaved execution as a table where each thread

is assigned to a single column and each row is considered a different time step, with the higher rows/steps appearing earlier in the execution time. In cases where the exact order of some instructions is not important to the case at hand we might place in the same row instructions of various threads, implying that these instructions need not be interleaved. For non allowed test cases we shortly discuss how the behavior in question is forbidden both by JMM and JDMM. According to JMM test cases 4, 5, 10, 12-15 are forbidden, while test cases 1-3, 6-9, 11, 16-20 are allowed.

Causality Test Case 1

Table 3.7.: Causality Test Case 1

Initially `x == y == 0`

Thread 1	**Thread 2**
`r1 = x`	`r2 = y`
`if r1 >= 0`	`x = r2`
`y = 1`	

Result `r1 == r2 == 1` ?

JMM	**JDMM**
Allowed	Allowed

Possible Execution

Thread 1	**Thread 2**
`y = 1`	
	`r2 = y`
	`x = r2`
`r1 = x`	
`if true ;`	

In Test Case 1 (Table 3.7) an inter-thread analysis of the code could determine that variables `x` and `y` cannot be negative, and move `y = 1` early. Under JDMM, `y = 1` may get written back by Thread 1, and `y` may get (re)fetched by Thread 2 before the execution of `r2 = y`, resulting to `r2` being assigned the value `1`. Similarly, `x = r2` may get written back by Thread 2, and `x` may get (re)fetched by Thread 1 before the execution of `r1 = x`, resulting in `r1` being assigned the value `1`. As a result, `r1 == r2 == 1` is allowed under both JMM and JDMM.

Causality Test Case 2

In Test Case 2 (Table 3.8) the behavior in question is allowed, since in sequentially consistent executions `r1` and `r2` are always equal and the compiler could determine this and move `y = 1` early in the execution. Under JDMM, `y = 1` may get written back by Thread 1, and `y` may get (re)fetched by Thread 2 before the execution of `r3 = y`, resulting to `r3` being assigned the value `1`. Similarly, `x = r3` may get written back by Thread 2, and `x` may get (re)fetched by Thread 1 before the execution of `r1 = x` and `r2 = x`, resulting in `r1` and `r2` being assigned the value `1`. As a result, `r1 == r2 == r3 == 1` is allowed under both JMM and JDMM.

Table 3.8.: Causality Test Case 2

Initially `x == y == 0`

Thread 1	Thread 2
`r1 = x`	`r3 = y`
`r2 = x`	`x = r3`
`if r1 == r2`	
`  y = 1`	

Result `r1 == r2 == r3 == 1` ?

JMM	JDMM
Allowed	Allowed

Possible Execution

Thread 1	Thread 2
`y = 1`	
	`r3 = y`
	`x = r3`
`r1 = x`	
`r2 = x`	
`if true ;`	

Causality Test Case 3

Table 3.9.: Causality Test Case 3

Initially `x == y == 0`

Thread 1	Thread 2	Thread 3
`r1 = x`	`r3 = y`	`x = 2`
`r2 = x`	`x = r3`	
`if r1 == r2`		
`  y = 1`		

Result `r1 == r2 == r3 == 1` ?

JMM	JDMM
Allowed	Allowed

Possible Execution

Thread 1	Thread 2	Thread 3
`y = 1`		`x = 2`
	`r3 = y`	
	`x = r3`	
`r1 = x`		
`r2 = r1`		
`if true ;`		

In Test Case 3 (Table 3.9) the behavior in question is allowed, since redundant read elimination could result in the compiler determining that `r1` is always equal to `r2` and move `y = 1` early in the execution. Note that in this case the execution of `x = 2` by Thread 3 nay happen anytime as long as it does not become visible to the other threads. Under JDMM, `y = 1` may get written back by Thread 1, and `y` may get (re)fetched by Thread 2 before the execution of `r3 = y`, resulting in `r3` being assigned the value `1`. Similarly, `x = r3` may get written back by Thread 2, and `x` may get (re)fetched by Thread 1 before the execution of `r1 = x`, resulting in `r1` and `r2` being assigned the value of `1`. Note that since there are is no synchronization in this test, `x = 2` may never be written back and thus be observed by Thread 1. As a result, `r1 == r2 == r3 == 1` is allowed under both JMM and JDMM.

Causality Test Case 4

Table 3.10.: Causality Test Case 4

Initially `x == y == 0`

Thread 1	Thread 2
`r1 = x`	`r2 = y`
`y = r1`	`x = r2`

Result `r1 == r2 == 1`?

JMM	JDMM
Forbidden	Forbidden

In Test Case 4 (Table 3.10) since `1` is never written to `y` nor to `x`, if observed then it would be out-of-thin-air, thus this case is forbidden in JMM. Since JDMM respects the no out-of-thin-air guarantee this case is forbidden in JDMM as well.

Causality Test Case 5

Table 3.11.: Causality Test Case 5

Initially `x == y == z == 0`

Thread 1	Thread 2	Thread 3	Thread 4
`r1 = x`	`r2 = y`	`z = 1`	`r3 = z`
`y = r1`	`x = r2`		`x = r3`

Result `r1 == r2 == 1` and `r3 == 0`?

JMM	JDMM
Forbidden	Forbidden

In Test Case 5 (Table 3.11), the value 1 only appears in Thread 3, where it gets assigned to `z`. Since in the behavior in question `r3` holds the value 0, Thread 4 may not observe the write performed by Thread 3. However, `z` is not read by any other Thread in the execution, thus `1` is never written to `x` nor to `y`. As a result, the only way to observe `r1 == r2 == 1 and r3 == 0` would be by `r1` or `r2` being assigned out-of-thin-air values, thus this case is forbidden both in JMM and JDMM.

Causality Test Case 6

In Test Case 6 (Table 3.12) the behavior in question is allowed, since an intra-thread analysis of the code could determine that `A` is always set to `1`, and move `B = 1` early in the execution. Under JDMM, `B = 1` may get written back by Thread 1, and `B` may get (re)fetched by Thread 2 before the execution of `r2 = B`, resulting in `r2` being assigned the value `1`. Similarly, `A = 1` may get written back by Thread 2, and `A` may get (re)fetched by Thread 1 before the execution of `r1 = A`, resulting in `r1` and `r2` being

Table 3.12.: Causality Test Case 6

Initially `A == B == 0`

Thread 1	Thread 2
`r1 = A`	`r2 = B`
`if r1 == 1`	`if r2 == 1`
`  B = 1`	`  A = 1`
	`if r2 == 0`
	`  A = 1`

Result `r1 == r2 == 1` ?

JMM	JDMM
Allowed	Allowed

Possible Execution

Thread 1	Thread 2
`B = 1`	
	`r2 = B`
	`if r2 == 1`
	`  A = 1`
`r1 = A`	`if r2 == 0`
`if true ;`	`  A = 1`

assigned the value of `1`. As a result, `r1 == r2 == 1` is allowed under both JMM and JDMM.

Causality Test Case 7

Table 3.13.: Causality Test Case 7

Initially `x == y == z == 0`

Thread 1	Thread 2
`r1 = z`	`r3 = y`
`r2 = x`	`z = r3`
`y = r2`	`x = 1`

Result `r1 == r2 == r3 == 1` ?

JMM	JDMM
Allowed	Allowed

Possible Execution

Thread 1	Thread 2
	`x = 1`
`r2 = x`	
`y = r2`	
	`r3 = y`
	`z = r3`
`r1 = z`	

In Test Case 7 (Table 3.13) the behavior in question is allowed, since an intra-thread analysis of the code could determine that `r1 = z` could be moved after all other assignments of Thread 1, and that `x = 1` could be moved before all other assignments of Thread 2, since there are no dependencies between these assignments and the other assignments of the corresponding threads. Under JDMM, `x = 1` may get written back by Thread 2, and `x` may get (re)fetched by Thread 1 before the execution of `r2 = x`, resulting in `r2` being assigned the value `1`. Then, `y = r2` may get written back by Thread 1, and `y` may get (re)fetched by Thread 2 before the execution of `r3 = y`. Similarly, `z`

= r3 may get written back by Thread 2, and z may get (re)fetched by Thread 1 before the execution of r1 = z, resulting in r1, r2, and r3 being assigned the value of 1. As a result, r1 == r2 == r3 == 1 is allowed under both JMM and JDMM.

Causality Test Case 8

Table 3.14.: Causality Test Case 8

Initially x == y == 0

Thread 1	Thread 2
r1 = x	r3 = y
r2 = 1+(r1*r1)-r1	x = r3
y = r2	

Result r1 == r2 == 1 ?

JMM	JDMM
Allowed	Allowed

Possible Execution

Thread 1	Thread 2
y = 1	
	r3 = y
	x = r3
r1 = x	
r2 = 1	

In Test Case 8 (Table 3.14) the behavior in question is allowed, since an intra-thread analysis of the code could determine that x and y can only be equal to 0 or 1, and thus determine that r2 can only be equal to 1, allowing y = 1 to be moved early in the execution. Under JDMM, y = 1 may get written back by Thread 1, and y may get (re)fetched by Thread 2 before the execution of r3 = y, resulting in r3 being assigned the value 1. Similarly, x = r3 may get written back by Thread 2, and x may get (re)fetched by Thread 1 before the execution of r1 = x, resulting in r1 and r2 being assigned the value of 1. As a result, r1 == r2 == 1 is allowed under both JMM and JDMM.

Causality Test Case 9

In Test Case 9 (Table 3.15) the behavior in question is allowed, since an intra-thread analysis of the code could determine that Thread 3 is scheduled after Thread 2 (due to some constraint not present here), meaning that the read of x by Thread 2 will never see the write of Thread 3. Similarly to test case 8 (Table 3.14), the intra-thread analysis of the code could then determine that r1 can only be equal to 0 or 1, and thus determine that r2 can only be equal to 1, allowing y = 1 to be moved early in Thread 1. Under JDMM, as in test case 8, y = 1 may get written back by Thread 1, and y may get (re)fetched by Thread 2 before the execution of r3 = y, resulting in r3 being assigned the value 1. Similarly, x = r3 may get written back by Thread 2, and x may get (re)fetched by Thread 1 before the execution of r1 = x, resulting in r1 and r2 being assigned the value of 1. As a result, r1 == r2 == 1 is allowed under both JMM and JDMM.

Table 3.15.: Causality Test Case 9

Initially `x == y == 0`

Thread 1	Thread 2	Thread 3

`r1 = x`
`r2 = 1+(r1*r1)-r1`
`y = r2`
	`r3 = y`
	`x = r3`
		`x = 2`

Result `r1 == r2 == 1` ?

JMM	JDMM
Allowed	Allowed

Possible Execution

Thread 1	Thread 2	Thread 3
`y = 1`		
	`r3 = y`	
	`x = r3`	
`r1 = x`		
`r2 = 1`		`x = 2`

Causality Test Case 10

Table 3.16.: Causality Test Case 10

Initially `x == y == z == 0`

Thread 1	Thread 2	Thread 3	Thread 4

`r1 = x`
`if r1 == 1`
`y = 1`
	`r2 = y`
	`if r2 == 1`
	`x = 1`
		`z = 1`	`r3 = z`
			`if r3 == 1`
			`x = 1`

Result `r1 == r2 == 1` and `r3 == 0` ?

JMM	JDMM
Forbidden	Forbidden

In Test Case 10 (Table 3.16), similarly to test case 5 (Table 3.11), since in the behavior in question `r3` holds the value 0, Thread 4 may not observe the write performed by Thread 3. However, `z` is not read by any other Thread in the execution, thus 1 is never written to `x` nor to `y`. As a result, the only way to observe `r1 == r2 == 1 and r3 == 0` would be by `r1` or `r2` being assigned out-of-thin-air, thus this case is forbidden both in JMM and JDMM.

Causality Test Case 11

In Test Case 11 (Table 3.17), similarly to test case 7 (Table 3.13), the behavior in question is allowed, since an intra-thread analysis of the code could determine that independent assignments could be reordered. Under JDMM, `x = 1` may get written back by Thread 2, and `x` may get (re)fetched by Thread 1 before the execution of `r2 = x`, resulting

Table 3.17.: Causality Test Case 11

Initially `x == y == z == 0`

Thread 1	Thread 2
`r1 = z`	`r4 = w`
`w = r1`	`r3 = y`
`r2 = x`	`z = r3`
`y = r2`	`x = 1`

Result `r1 == r2 == r3 == r4 == 1` ?

JMM	JDMM
Allowed	Allowed

Possible Execution

Thread 1	Thread 2
	`x = 1`
`r2 = x`	
`y = r2`	
	`r3 = y`
	`z = r3`
`r1 = z`	
`w = r1`	
	`r4 = w`

in `r2` being assigned the value `1`. Then, `y = r2` may get written back by Thread 1, and y may get (re)fetched by Thread 2 before the execution of `r3 = y`, resulting in `r3` being assigned the value `1`. Later, `z = r3` may get written back by Thread 2, and `z` may get (re)fetched by Thread 1 before the execution of `r1 = z`, resulting in `r1` being assigned the value `1`. Similarly, `w = r1` may get written back by Thread 1, and w may get (re)fetched by Thread 2 before the execution of `r4 = w`, resulting in `r1`, `r2`, `r3`, and `r4` being assigned the value `1`. As a result, `r1 == r2 == r3 == r4 == 1` is allowed under both JMM and JDMM.

Causality Test Case 12

Table 3.18.: Causality Test Case 12

Initially `x == y == 0`, `a[0] == 1`, and `a[1] == 2`

Thread 1	Thread 2
`r1 = x`	`r3 = y`
`a[r1] = 0`	`x = r3`
`r2 = a[0]`	
`y = r2`	

Result `r1 == r2 == r3 == 1` ?

JMM	JDMM
Forbidden	Forbidden

In Test Case 12 (Table 3.18), array a is only accessed by Thread 1, thus the code for Thread 1 should be equivalent to the code in Figure 3.3. Similarly to test case 4 (Table 3.10), since in sequentially consistent executions `r1` can never be assigned the value `1`, if observed then it would be out-of-thin-air, thus this case is forbidden in both

```
r1    = x
a[r1] = 0
if r1 == 0
  r2 = 0
else
  r2 = 1
y = r2
```

Figure 3.3.: Test case 12 thread 1 equivalent code

JMM and JDMM.

Causality Test Case 13

Table 3.19.: Causality Test Case 13

Initially `x == y == 0`	
Thread 1	**Thread 2**

`r1 = x`
`if r1 == 1`
`  y = 1`
	`r2 = y`
	`if r2 == 1`
	`  x = 1`

Result `r1 == r2 == 1` ?	

JMM	**JDMM**
Forbidden	Forbidden

In Test Case 13 (Table 3.19), since there is no sequentially consistent execution where `x` or `y` are written, the only sequentially consistent result is `r1 == r2 == 0`. Additionally, since `x` and `y` are only read concurrently, the program is correctly synchronized, meaning that non sequentially consistent executions are not acceptable both in JMM and JDMM.

Causality Test Case 14

In Test Case 14 (Table 3.20), since there is no sequentially consistent execution where `r1` is equal to `1` and the program is correctly synchronized, non sequentially consistent executions, where `r1` is equal to `1`, are not acceptable both in JMM and JDMM.

Table 3.20.: Causality Test Case 14

Initially `a == b == y == 0` and `y` is `volatile`

Thread 1	Thread 2
`r1 = a`	`do {`
`if r1 == 0`	`  r2 = y`
`  y = 1`	`  r3 = b`
`else`	`} while (r2+r3 == 0)`
`  b = 1`	`a = 1`

Result `r1 == r3 == 1, r2 == 0` ?

JMM	JDMM
Forbidden	Forbidden

Causality Test Case 15

Table 3.21.: Causality Test Case 15

Initially `a == b == x == y == 0`, and `x` and `y` are `volatile`

Thread 1	Thread 2	Thread 3
`r0 = x`	`do {`	`x = 1`
`if r0 == 1`	`  r2 = y`	
`  r1 = a`	`  r3 = b`	
`else`	`} while (r2+r3 == 0)`	
`  r1 = 0`	`a = 1`	
`if r1 == 0`		
`  y = 1`		
`else`		
`  b = 1`		

Result `r0 == r1 == r3 == 1, r2 == 0` ?

JMM	JDMM
Forbidden	Forbidden

In Test Case 15 (Table 3.21), since there is no sequentially consistent execution where `r1` is equal to `1` and the program is correctly synchronized, non sequentially consistent executions, where `r1` is equal to `1`, are not acceptable both in JMM and JDMM.

Causality Test Case 16

In Test Case 16 (Table 3.22), since the program is not correctly synchronized and values 2 and 1 are not out-of-thin-air for `r1` and `r2` respectively, this is an acceptable result

Table 3.22.: Causality Test Case 16

Initially `x == 0`

Thread 1	Thread 2
`r1 = x`	`r2 = x`
`x = 1`	`x = 2`

Result `r1 == 2, r2 == 1`?

JMM	JDMM
Allowed	Allowed

under JMM. Note that the code in both threads features an anti-dependency (write-after-read), meaning that the compiler cannot reorder the instructions. That said, to observe the result `r1 == 2, r2 == 1` it needs to be an artifact of out-of-order execution. Under JDMM this behavior could be observed if Thread 1 would proceed to the write `x = 1` and its write-back without waiting for the fetch of x to reach completion. Assuming the same happens on Thread 2, it is possible for the write-back of `x = 2` to be observed by `r1 = x` on Thread 1 and the write-back of `x = 1` to be observed by `r2 = x` on Thread 2. Note that the anti-dependency is not broken, since the reads of x do not see the writes of x performed subsequently by the same thread. As a result, the outcome `r1 == 2, r2 == 1` is valid in both JMM and JDMM.

Causality Test Case 17

Table 3.23.: Causality Test Case 17

Initially `x == y == 0`

Thread 1	Thread 2
`r3 = x`	`r2 = y`
`if r3 != 42`	`x = r2`
`  x = 42`	
`r1 = x`	
`y = r1`	

Result `r1 == r2 == r3 == 42`?

JMM	JDMM
Allowed	Allowed

Possible Execution

Thread 1	Thread 2
`r1 = 42`	
`y = r1`	
	`r2 = y`
	`x = r2`
`r3 = x`	
`if r3 != 42`	
`  x = 42`	

In Test Case 17 (Table 3.23), an analysis of the code could result in the compiler to determine that x will always see the value `42` in `r1 = x`. That said, the compiler could change `r1 = x` to `r1 = 42` and allow it along with the `y = r1` assignment to be moved earlier. Under JDMM, `y = r1` may get written back by Thread 1, and y may

get (re)fetched by Thread 2 before the execution of `r2 = y`, resulting in `r2` being assigned the value `42`. Similarly, `x = r2` may get written back by Thread 2, and `x` may get (re)fetched by Thread 1 before the execution of `r3 = x`, resulting in `r1`, `r2`, and `r3` being assigned the value `42`. As a result, `r1 == r2 == r3 == 42` is allowed under both JMM and JDMM.

Causality Test Case 18

Table 3.24.: Causality Test Case 18

Initially `x == y == 0`

Thread 1	Thread 2
`r3 = x`	`r2 = y`
`if r3 == 0`	`x = r2`
`  x = 42`	
`r1 = x`	
`y = r1`	

Result `r1 == r2 == r3 == 42` ?

JMM	JDMM
Allowed	Allowed

Possible Execution

Thread 1	Thread 2
`r1 = 42`	
`y = r1`	
	`r2 = y`
	`x = r2`
`r3 = x`	
`if r3 == 0`	
`  x = 42`	

In Test Case 18 (Table 3.24), an analysis of the code could result in the compiler to determine that `x` can only have values `0` or `42`, thus `r3 != 0` implies that `r3` is equal to `42`. That said, as in test case 17 (Table 3.23) the compiler could change `r1 = x` to `r1 = 42`, allowing it along with the `y = r1` assignment to be moved earlier. Under JDMM, `y = r1` may get written back by Thread 1, and `y` may get (re)fetched by Thread 2 before the execution of `r2 = y`, resulting in `r2` being assigned the value `42`. Similarly, `x = r2` may get written back by Thread 2, and `x` may get (re)fetched by Thread 1 before the execution of `r3 = x`, resulting in `r1`, `r2`, and `r3` being assigned the value `42`. As a result, `r1 == r2 == r3 == 42` is allowed under both JMM and JDMM.

Causality Test Case 19

In Test Case 19 (Table 3.25), an analysis of the code could result in the compiler to determine that Thread 1 will always see the value `42` in `r1 = x`. That said, the compiler could change `r1 = x` to `r1 = 42`, allowing it along with the `y = r1` assignment to be moved earlier, removing the happens-before relation between the `x = 42` by Thread 2 and `r1 = x` by Thread 1. After the reordering taking place, test case 19 is similar to test case 17 (Table 3.23) and should be allowed both under JMM and JDMM. However, as Aspinall and Ševčík [7] and Torlak et al. [97] claim this test case is not allowed by

Table 3.25.: Causality Test Case 19

Initially `x == y == 0`

Thread 1	Thread 2	Thread 3
`join Thread 3`	`r2 = y`	`r3 = x`
`r1 = x`	`x = r2`	`if r3 != 42`
`y = r1`		`x = 42`

Result `r1 == r2 == r3 == 42` ?

JMM	JDMM
Desirable	Desirable

Possible Execution

Thread 1	Thread 2	Thread 3
`r1 = 42`		
`y = r1`		
	`r2 = y`	
	`x = r2`	
		`r3 = x`
		`if r3 != 42`
		`x = 42`
`join Thread 3`		

JMM due to stronger than needed rules in the validation process. Since JDMM relies on JMM's validation process it also forbids this test case. The changes on the validation rules proposed by Aspinall and Ševčík [7], however, fix this issue for both JMM and JDMM.

Causality Test Case 20

In Test Case 20 (Table 3.26), an analysis of the code could result in the compiler to determine that `x` can only have values `0` or `42`, thus `r3 != 0` implies that `r3` is equal to `42`. That said, as in test case 18 (Table 3.24) the compiler could change `r1 = x` to `r1 = 42`, allowing it along with the `y = r1` assignment to be moved earlier, removing the happens-before relation between the `x = 42` by Thread 2 and `r1 = x` by Thread 1. After the reordering taking place, test case 20 is similar to test case 18 (Table 3.24) and should be allowed both under JMM and JDMM. However, as Aspinall and Ševčík [7] and Torlak et al. [97] claim this test case, similarly to test case 19 (Table 3.25) is not allowed by JMM due to stronger than needed rules in the validation process. Since JDMM relies on JMM's validation process it also forbids this test case. The changes on the validation rules proposed by Aspinall and Ševčík [7], however, fix this issue for both JMM and JDMM.

Table 3.26.: Causality Test Case 20

Initially `x == y == 0`

Thread 1	Thread 2	Thread 3
`join Thread 3`	`r2 = y`	`r3 = x`
`r1 = x`	`x = r2`	`if r3 == 0`
`y = r1`		`  x = 42`

Result `r1 == r2 == r3 == 42` ?

JMM	JDMM
Desirable	Desirable

Possible Execution

Thread 1	Thread 2	Thread 3
`r1 = 42`		
`y = r1`		
	`r2 = y`	
	`x = r2`	
		`r3 = x`
		`if r3 == 0`
		`  x = 42`
`join Thread 3`		

To sum up, JDMM satisfied all the causality tests satisfied by JMM, and is expected to also satisfy test cases 19 and 20 if some of the rules in the validation process of JMM relax as shown by Aspinall and Ševčík [7].

3.5.2. Code optimization: Reordering

Manson et al. prove that reordering two independent statements, when it does not affect the happens-before relation of any other actions, is legal under JMM [70, §6.2.1]. Specifically, [70, Theorem 1] shows that two adjacent statements s_x and s_y, where x and y are the corresponding actions of the statements, can be reordered as long as:

1. their reordering does not eliminate any transitive happens-before edges in any valid execution
2. they are not conflicting accesses
3. they are not both synchronization or external actions
4. their reordering does not move an action before an infinite loop
5. their reordering preserves the intra-thread consistency

We argue that the reorderings allowed by [70, THEOREM 1] under JMM are the same as the ones allowed under JDMM. In JDMM, there are four newly introduced (synthetic) actions, namely fetch, write-back, invalidate and migrate. These actions do not map to statements and may appear between the corresponding actions of adjacent statements. As a result, the reordering of two independent statements might also require the reordering of some synthetic actions happening between them, according to the program order. Assuming an execution E_D where $\exists x, y, z \in A_D : (x \leq^d_{po} y \leq^d_{po} z) \wedge y.k \in \{F, B, Iv, M\}$, and x and y are independent, we examine the allowed reorderings, depending on y's kind and relation to x and z.

1. In the case where y fetches the value that z sees ($Cs(z) = y$), then to preserve $y \leq^d_{co} z$ we only allow the reordering $y \leq^d_{po} z \leq^d_{po} x$.

2. In the case where y write-backs the value that x writes ($Ab(y) = x$), then to preserve $x \leq^d_{co} y$ we only allow the reordering $z \leq^d_{po} x \leq^d_{po} y$.

3. In the case where y migrates the thread to a different core ($y.k = M$), we need to preserve **WFE-1** and **WFE-2** in the post-reorder execution. As a result, depending on the reordering we might need to introduce some additional synthetic actions not previously present in the execution. Thus, changing the order of the actions around y can also affect the program's performance.

   In the case of reordering to $y \leq^d_{po} z \leq^d_{po} x$, any write-backs happening before y, in the original execution, must preserve their ordering, in respect to y, except for the write-back of x if it happens to be a write ($x.k = W$). If x is a read $x.k = R$, then we need to add a fetch action f, such that $y \leq^d_{co} f \leq^d_{co} x$.

   In the case of reordering to $z \leq^d_{po} x \leq^d_{po} y$, any fetches previously happening after z, in the original execution, must preserve their ordering, in respect to z, except for the fetch of z if it happens to be a read ($z.k = R$). If z is a write $z.k = W$, then we need to write it back before y.

   Finally, in the case of reordering to $z \leq^d_{po} y \leq^d_{po} x$; if x is a read ($x.k = R$), a fetch $f = Cs(x)$ needs to be added so that $y \leq^d_{co} f \leq^d_{co} x$; if x is a write ($x.k = W$) and x gets written back before y, its write-back must be moved after y, according to cache order.

4. In all other cases reordering of synthetic actions is not necessary.

Our model allows the same reorderings as JMM, as long as some synthetic actions are also reordered as described above.

3.6. Case Study

As a use case example we chose to use JDMM to verify the correctness of existing JVMs regarding their compliance to JMM. To select the JVMs to examine, we went through all publications citing [72], [71] or [70], looking for JVM implementations on non memory coherent architectures that comply with JMM. We also searched for open-source JVMs targeting non coherent architectures. We narrowed our search only to open-source projects, since proprietary JVMs are not accompanied with enough details to extract how they implement JMM. Surprisingly we found only one JVM claiming that it fully complies with JMM (as defined by JSR-133 in the current Java Language Specification) and targets non-cache-coherent architectures.

Note that in our verification procedure we omit the *out-of-thin-air* guarantee. The *out-of-thin-air* guarantee depends on multiple factors orthogonal to caching, such as the semantics of the underlying hardware, the correctness of the garbage collector implementation etc. As a result, to verify that Hera-JVM does not allow *out-of-thin-air* a thorough investigation of a large number of components would be necessary. Since JDMM focuses on caching and how to orchestrate the memory transfers, we chose to only examine the Hera-JVM compliance to JMM regarding these aspects.

Hera-JVM [73], also discussed in Section 2.2.6 targets the IBM's Cell B.E. processor. Cell B.E. is a heterogeneous multiprocessor featuring one POWER Processing Unit (PPU) and eight Synergistic Processing Units (SPUs). The cores can access each other's memory through DMA transfers and there is no cache coherency. Hera-JVM uses a software cache for heap-based variables, which caches whole objects or array blocks of 1KiB. In the case of a cache miss, the JVM fetches a whole object or a 1KiB block of an array. To do that, Hera-JVM initiates a DMA transfer to fetch the data and blocks the execution of the Java thread until the DMA is complete. The cache uses a write through policy: Whenever a thread writes to a cached address, Hera-JVM initiates a non-blocking DMA, copying the new data to the main memory. To conform to JMM, a Java thread blocks until all DMAs complete before releasing a lock, writing to a volatile variable, context switching or migrating to another core. Furthermore, Hera-JVM purges (invalidates) the caches before every monitor-enter (acquire lock) or volatile read, causing any future reads to fetch the data from the main memory.

Examining the text in [73] we were able to verify that most of the JDMM constraints are satisfied by Hera-JVM. Hera-JVM appears to handle caches properly as far as it concerns context switching, synchronized blocks and volatile variables. We were unable, however, to verify that thread migration is properly handled. Specifically, Hera-JVM claims that it writes back all dirty data before a thread migrates off an SPE core, but does not explicitly state that the cache is invalidated. Section 3.3.5 describes a scenario where this is not enough and may result in executions that are not consistent to the happens-before order. In short, consider a thread that migrates to another core and attempts to read a variable, which happens to be already cached on that core.

To verify that Hera-JVM adheres to JMM we had to examine its source code and contact the authors. We found that the source code does not explicitly invalidate the cache. Note, however, that in Hera-JVM the context switching and the thread migration mechanisms are written in Java and rely on synchronized methods. This implicitly satisfies **WFE-1** and **WFE-2**, since the runtime synchronized method call for the context switch or thread migration will invalidate the cache contents. Note here that Hera-JVM also invalidates the caches for context switches, that as we show in Section 3.3.4 is not mandatory and may result in additional energy and performance overheads.

Chapter 4.

Designing a JVM for hundreds of incoherent cores

In this Chapter we discuss the key challenges of developing a Java Virtual Machine (JVM) targeting a non-cache-coherent prototype. We propose techniques and algorithms to overcome those challenges and scale Java to hundreds of cores. Our algorithms exploit the spatial locality and memory coherency of coherent-islands in architectures like EUROSERVER.

Parts of the work presented in this chapter have been published in the proceedings of the *14th International Workshop on Java Technologies for Real-Time and Embedded Systems* (JTRES '16) [33].

4.1. Key Challenges

The related literature, discussed in Chapter 2, underlines a number of key challenges that JVM implementers face when the underlying architecture does not provide a coherent shared memory abstraction. In this section we present and discuss these challenges.

4.1.1. Memory Management

One of the key challenges is memory management, mainly due to the lack of a coherent shared memory abstraction. Java uses a heap for objects and a stack per Java thread for local variables. JMM defines that the Java heap is global per application. That is, every thread of a Java program must be able to access the whole Java heap. Thus, Java developers do not need to worry about memory localities when writing their programs. When the underlying hardware provides a shared-memory abstraction, the Java heap is implicitly accessible by every java thread, since the whole memory is addressable and the hardware is responsible for performing the actual access. In the absence of a hardware shared-memory abstraction, the JVM is responsible to ensure that the whole

Figure 4.1.: Time window example.

Java heap is accessible from every Java thread in the application. Additionally, the JVM is responsible to ensure that all threads get a coherent view of the Java heap.

Keeping the Java Heap Coherent

Early attempts to implement distributed JVMs delegate the memory management to Software Distributed Shared Memory (SDSM) [32, 100, 102, 107]. As we state in Chapter 2, however, SDSM is not expected to perform well, both performance and energy wise. In order to reduce network traffic and execution time, distributed and heterogeneous JVMs implement some kind of software caching [4, 73, 107].

When a JVM employs software caching, to access a remote object it *fetches* a local copy; to make dirty copies globally visible it writes them back (*write-back*); and to free space in its cache or force an update on the next access it invalidates local copies (*self-invalidate*). Since memory accesses are very common, software caches not only need to be efficient, but they also require careful management to avoid redundant operations. JMM defines when the Java heap should be updated so that all threads get a coherent view of it. However, cache operations are not exposed to JMM, making it hard to understand and implement software caches on non-cache-coherent memory architectures.

In Chapter 3 we present JDMM, an extension of JMM exposing such operations to the programming model and argue about when they should commit to ensure adherence to JMM. JDMM's rules aim to be as relaxed as possible, so as to accept all legal executions that adhere to the JMM. For instance, the JDMM intuitively states that a write-back and its corresponding fetch may be executed anytime in the time window between a write and the corresponding read, given that the write happens-before this read [57]. For instance, in Figure 4.1 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 JDMM allows a write-back and its corresponding fetch to be performed is 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 prohibiting, 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.

Software caching has been previously studied in depth to improve performance and usability of distributed systems, heterogeneous multiprocessors, and GPUs [19, 54, 59, 60, 61, 86, 87, 94, 104, 106]. These approaches aim to provide a software shared virtual memory (SVM), a very similar task to providing a global heap view when the underlying memory is actually distributed. Such SVM approaches, however, are page-based and thus not suitable for an object-oriented language, where caching at the object level is expected to be more efficient. Additionally, being page-based many of the SVM approaches utilize hardware Memory Management Units (MMUs) to detect when a page becomes dirty. Unfortunately, MMUs are not capable of providing such information at the object granularity, since objects are a notion introduced by object oriented languages and is not yet understood by common MMUs. Additionally, the coherence protocols discussed in these works are more relaxed and fail to adhere to JMM. Some of these software caches delegate the coherence procedures to the application, while others rely on reference counting to detect when a cached line is no longer referenced to write it back and release it. In the case of Java, however, reference counting does not suffice to ensure a coherent view of the Java heap, as defined by JMM.

As a result, JVM implementations targeting many-core non-cache-coherent architectures end up writing back all dirty data before a release operation and invalidating all data at an acquisition operation, in order to force a re-fetch of the cached data. This approach is safe and sound, but shrinks the time window, thus limiting the optimization space. A visualization of the shrunk time windows is presented in Figure 4.1. The small orange 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 cyan dashed rectangle on the lower right corner is the time window in which the corresponding fetch can be executed. However, even this way, the software caching benefits are significant, making software caches a key component of JVMs targeting non-cache-coherent architectures. 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 are out of the scope of this work. In this thesis we present a software cache scheme that aims to minimize memory transfers while adhering to JMM.

Garbage Collection

Process virtual machines implementing safe languages use garbage collection (GC) for dynamic memory management. Garbage collectors cooperate with the memory al-

location mechanisms and perform automatic deallocation of unreachable objects—the *garbage*. Garbage collectors in many cases are also tightly coupled with the memory allocator to improve performance.

There are several classes of garbage collectors with different characteristics and behavior. The state-of-the-art garbage collectors follow the tracing garbage collector model. A tracing garbage collector is invoked periodically or when the VM runs out of free memory. It traverses the Java heap and the thread stacks to find references to heap objects and reclaims any non referenced objects. *Mark-and-sweep* garbage collectors are typical tracing garbage collectors. They first trace object references and mark unreachable objects which are then swept. To improve spatial locality, and simplify allocation, some tracing garbage collectors, called *copying* garbage collectors, shift data or move them to another memory segment to de-fragment the heap.

The use of software caches, however, hinders the process of garbage collection since it introduces new *roots* for tracing garbage collectors. Normally, tracing garbage collectors perform traversals following references in the Java heap and the Java stacks, and marking the referenced objects as reachable to prevent their collection. With the introduction of software caches, the garbage collector needs to follow all references in each cache as well, to mark remotely accessible objects. To make matters worse, copying garbage collectors need a way to update remote cache entries to reflect the new location of an object. A stop-the-world garbage collector may trivially request the write-back and invalidation of all the software caches before garbage collection. However, stop-the-world garbage collectors require that all threads block before garbage collection, resulting in long pauses.

Generational garbage collectors split the heap space in segments based on the objects' *age*. This approach is based on the *generational hypothesis* that recently allocated objects are more likely to be unreachable in the near future. In this scenario, objects surviving a number of garbage collection cycles are promoted to older heap spaces. The split heap space enables garbage collection to run on each segment independently. Older heap segments need to be garbage collected only when the younger ones fail to provide enough free memory, resulting in more efficient garbage collection.

Blackburn et al. [12] study the performance of three basic garbage collection algorithms, *copying semi-space*, *mark-sweep* and *reference counting* along with their generational counterparts and conclude that generational garbage collectors are outperforming their counterparts. They also conclude that all applications benefit from the improved data locality of young objects. That said, a copying garbage collector enabling contiguous allocation is the best fit for the young heap segments, regardless of whether the application follows the generational hypothesis. On the contrary, the choice of the garbage collector for the mature heap segment is more complex and depends on the access and mutation rate of the segment.

Implementing these techniques efficiently without a coherent shared memory abstraction is not trivial. Although the implementation of garbage collectors for non-cache-

coherent architectures is orthogonal yet complementary to this work, and outside the scope of this thesis, we discuss how the JVM can provide some properties to help in the implementation of the garbage collector, and how the garbage collector can help the JVM become more efficient.

4.1.2. Synchronization

The JMM defines several synchronization pairs, among which the only explicit ones during a thread's lifetime are `monitorenter`–`monitorexit`, and `wait`–`notify`. The first pair ensures mutual exclusive accesses to critical sections, while the second explicitly orders concurrent threads. The `wait` and `notify` operations are tightly coupled with monitors since a wait operation may only be executed by a thread owning the corresponding object's monitor.

Note that we do not consider the `java.util.concurrent` library in this thesis. This library is targeted to shared-memory machines and is usually implemented directly with atomic primitives provided by shared memory architectures. Furthermore, its interaction with JMM is not yet fully defined.

In Java, each object can be used as a lock, which is achieved with *monitors*. In most JVMs this implies one extra field per object that is used as the monitor. At monitor acquisition (enter) and release (exit) points the JVM not only needs to ensure proper data handling (see Section 4.1.1) but also needs to ensure that monitor acquisitions are consistent with mutual exclusion. That is, a monitor may only be owned by a single Java thread at any time, allowing for that single thread to own the monitor multiple times—re-entrant acquisition. In Java there are three ways to acquire an object's monitor:

a) by executing a `synchronized` instance method of that object,

b) by executing the body of a `synchronized` block that synchronizes on the object, and

c) for objects of type `Class`, by executing a `synchronized` static method of that class.

In shared memory machines, monitors are implemented using instructions like load-link/store-conditional, compare-and-swap, fetch-and-add, etc. Such instructions are not always present in distributed memory architectures, thus JVM implementers need to provide a solution independent of those instructions. This work proposes the use of dedicated cores to act as synchronization managers and handle monitors.

4.1.3. Thread Scheduling

On a large number of cores, thread scheduling is a key challenge for JVM implementers. Most JVMs on shared-memory machines delegate this job to the operating system (OS). However, in non-cache-coherent architectures the OS is expected to comprise multiple OS instances that rely on the application to perform the thread scheduling [39]. When implementing a JVM, such schemes must be abstracted away and get hidden from the programmer, since Java does not expose architectural and OS details.

Threads are a basic construct of parallel applications in Java. Threads are usually used to create multiple servers, most often one per core, that execute some workloads and exchange information through synchronization. Thread over-subscription is often suggested for applications with blocking threads, to improve utilization by executing another thread while another is blocked. Such issues that cannot be solved by the OS and increase the complexity of a JVM for non-cache-coherent architectures include thread synchronization, over-subscription of threads to cores, blocking and context-switching threads, load balancing of threads to cores, and dynamic or non-balanced thread creation and destruction.

However, there are also applications that vary the number of threads during their lifetime. In such cases, the JVM needs to be able to adapt and distribute the workload to load balance the system. Moreover, parallel applications with irregular or dynamic parallelism require continuous dynamic load balancing to improve resource utilization and thus improve energy efficiency and performance.

Following task-based programming models, we propose the use of light-weight tasks in applications with irregular parallelism. The main difference between threads and tasks is that the latter are much more light-weight than the first. As a result, tasks are more suitable for applications with irregular parallelism as their creation overhead is low and schedulers can take advantage of their fine granularity to achieve better load balancing [8, 14, 15, 28, 93, 98].

On shared memory architectures the dominating load balancing algorithm for task based applications is the work stealing algorithm introduced by Blumofe and Leiserson [16]. Work stealing on shared-memory machines is implemented using a lock-free double-ended queue (deque), essentially relying on atomic operations like compare-and-swap and fetch-and-add [22] that might not be available across coherent-islands in non-cache-coherent architectures. The deque owner pops and pushes tasks to the bottom of the deque, while idle cores pop tasks from the top of the deque.

Previous work [28, 76, 105] has presented a few implementations of the work-stealing algorithm for clusters with an RDMA interconnect. These implementations, however, are coupled with the task-based programming model. Task-based programming models make assumptions about task properties which are not compliant with the properties of Java threads. Such assumptions are that:

a) each task is described by a *task descriptor* which among others contains information about the memory footprint of the task—the memory it is going to access;

b) each task runs to completion without blocking.

While attempts to extend Java with constructs to satisfy such assumptions have been proposed in the past [17, 30, 46, 99], such extensions still leave the scheduling problem of legacy code open. As a result, to support legacy code, JVM implementers need to implement load balancing techniques for threads as well. This work presents a hybrid algorithm that allows work-stealing within coherent-islands and work-dealing –coordinated thread exchange– across coherent-islands, through message passing.

4.2. Design

This section examines the applicability of previously proposed techniques, regarding the key challenges discussed in Section 4.1, to non-cache-coherent many-core architectures, with the exception of the orthogonal problem of garbage collection. We discuss cases where existing techniques are inefficient and propose alternative solutions towards designing a JVM for non-cache-coherent many-core architectures.

4.2.1. Memory Management

Since future non-cache-coherent architectures are expected to feature tens or hundreds scratchpad memories we need a way to distribute the Java heap on them. To manage local memories, we follow the JESSICA2 [107] paradigm and we split the local memories in three parts. The first two parts are the *global heap area* and the *cache heap area*. The third part is reserved for the needs of the JVM itself, *i.e.*, Java stacks, native stacks and native heap. Note that this scheme matches that of Figure 3.2 (redrawn in Figure 4.2) in Chapter 3, without the *native* memory part. Each scratchpad memory is split in a *cache heap area*, marked in orange, and a *global heap area*, marked in cyan. Caches fetch or write-back items from or to remote global heap areas, respectively. The conjunction of all the *global heap areas*, forms the Java heap, similarly to Partitioned Global Address Space (PGAS) models. Accesses to the local *global heap area* are direct while accesses to remote *global heap areas* get cached in the *cache heap area*. New objects are allocated on the local *global heap area*, unless it runs out of space. In such cases, a request is sent to a remote *global heap area*.

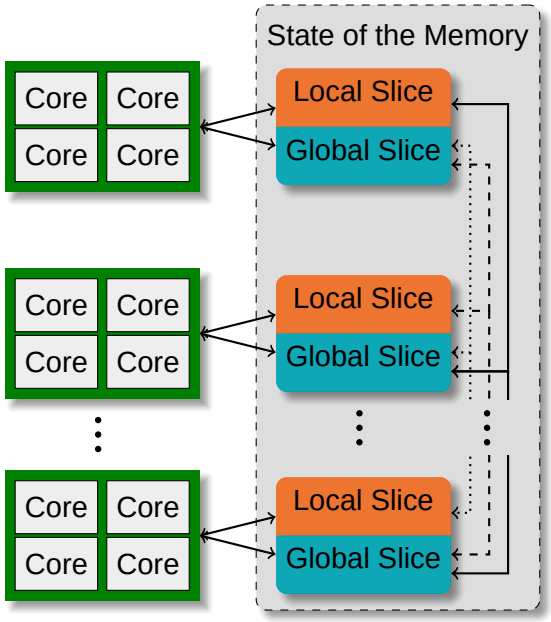


Figure 4.2.: The abstract machine (redraw of Figure 3.2)

Cache Heap Area

Partitioning the Java heap raises the need for caching to improve performance. To access an object that is located in a remote *global heap area*, a DMA transfer for each access would be too expensive, in terms of time and energy overhead. To reduce the overhead, we propose the use of software caching. The *cache heap area* is managed by a software cache implementation, working at object granularity. We choose to go with object granularity, since in object oriented programming languages the fields of an object are usually tightly coupled, and accesses to them usually occur near to each other. Caching whole objects exploits the spatial and temporal locality of objects' field accesses. In the case of arrays, we follow the approach of McIlroy and Sventek [73] and propose the fetch of array chunks instead of the whole array at once. Though, instead of setting a byte size limit to the chunks, we suggest setting a threshold to the number of elements, *e.g.*, chunks of 100 elements instead of 1KiB chunks. This way the performance of loops accessing array elements is bound by the number of iterations instead of the size of the elements, which we believe is more intuitive to developers and helps them to better understand the application's behavior. Note that JMM is a relaxed memory model, allowing the cache operations to be performed any time between a write and the read that sees the value written by that write. That said, the software cache design space is rather big, allowing for various approaches and policies. In this work we propose a simple yet efficient software cache that avoids the overhead of extensive bookkeeping.

In hierarchical architectures that comprise coherent islands, *cache heap areas* can be

either shared, among different cores, or private. Shared *cache heap areas* have the benefit of more efficient resource utilization through sharing a single copy of commonly accessed objects instead of creating multiple copies of that same objects. However, shared *cache heap areas* require more bookkeeping, introducing performance overhead to cache operations and creating a contention point in coherent-islands. Shared *cache heap areas* need information about the *owner* of each cache entry, so that they can only write-back or invalidate the entries associated with a single thread and not the whole *cache heap area* at synchronization points. Private *cache heap areas*, on the contrary, allow for totally distributed caching, without any contention points.

Zhu et al. [107] suggest the use of a separate private *cache heap area* per Java thread in JESSICA2, arguing that:

a) this approach is closer to the JMM definition,

b) it prevents threads from invalidating cached objects of other threads, and

c) the smaller cache size results in more efficient parsing, *e.g.*, at flush operations.

In this thesis we propose a hybrid approach, where read accesses are cached into a shared per-core cache, which we call *object cache*, while write accesses are cached into a private per-thread cache, which we call *write-buffer*. As we show in Section 3.3.3, JVM implementations allowing the sharing of a cache among different Java threads still adhere to JDMM. Additionally, since in Java there are different bytecodes for read and writes (*i.e.* `getfield`, `setfield`, etc.) the distinction of the two comes at no additional runtime overhead. The object cache increases the cache hit rate and reduces the number of object copies on the system, while the write-buffer reduces the delay of cache flushes. 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

1. set an upper limit on the release operations' blocking time;
2. allow for overlapping write-backs with computation when the threshold is met;
3. allow for bulk transfer of contiguous data, *e.g.*, written elements of an array; and
4. allow for multiple writes to the same variable without the need to write back every time.

Algorithms 1 and 2 present the main procedures of the software cache mechanisms. When reading a remote object, the JVM first queries the write-buffer, and in the case of a miss continues by querying the object cache. If both queries fail, then the JVM fetches the remote object to the object cache. When writing on a remote object, the JVM first queries the Java thread's write-buffer, and in the case of a miss fetches the

object to it. Write-backs of the cache are straight forward; the JVM writes back each dirty field of the objects in the write-buffer and then moves those entries to the object cache. This way, we ensure that the write-buffer only holds dirty data. To avoid the invalidation of the whole object cache at synchronization points, we use a bitmap for each object cache entry to mark which java threads had used that entry. Since the coherent-islands are expected to be in the scale of tens of cores, we consider the space overhead of the bitmap acceptable.

Adherence to the JMM To adhere to the JMM, we propose the intuitive approach of ensuring that at release actions all dirty data are written back and at acquire actions all cache entries are invalidated. However, this may result in long blocking release actions for critical sections that perform writes on a large memory segments.

To demonstrate the overhead of such operations we perform a simple experiment, where a single core transfers a given data set from another core's scratchpad to its own. Figure 4.3 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.

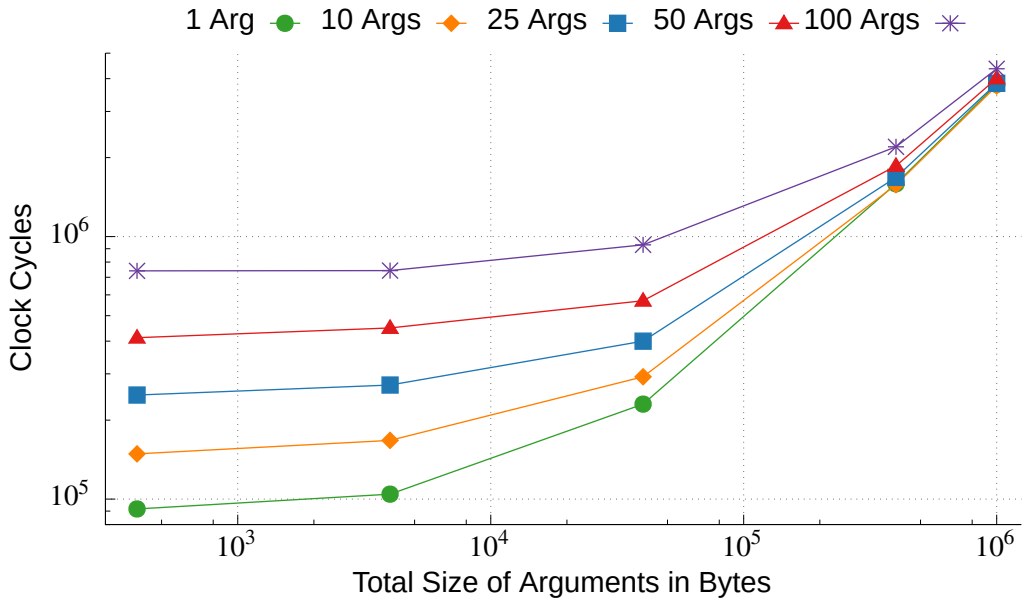


Figure 4.3.: Impact of arguments size and number on delay

That said, the size of the write-buffer might significantly impact the performance and

Algorithm 1: Hybrid software cache

Write-Buffer : A private, per Java thread, cache holding dirty copies of remote objects, written by that thread
Object Cache: A shared, per core, cache holding clean copies of remote objects

Procedure read(object)

Input: The address of the object to be read
Output: The translated address

```
1 if the object resides in the current thread's global heap space then
2 |   return the address of the object;
3 else if the object is cached in the write-buffer then
4 |   return the address of the object's cached copy in the write-buffer;
5 else if the object is not cached in the object cache then
6 |   Fetch the object to the object cache;
7 end
8 Set the tag bit, corresponding to the current thread, on the cache entry's bitmap;
9 return the address of the object's cached copy in the object cache;
```

Procedure write(object)

Input: The address of the object to be written
Output: The translated address

```
1 if the object resides in the current thread's global heap space then
2 |   return the address of the object;
3 else if the object is cached in the write-buffer then
4 |   return the address of the object's cached copy in the write-buffer;
5 end
6 Fetch the object to the write-buffer;
7 return the address of the object's cached copy in the write-buffer;
```

Procedure write-back cache

Result: All dirty data in the thread's cache are written back and invalidated

```
1 foreach entry in the current thread's write-buffer do
2 |   Write-back the dirty fields of the entry;
3 |   Copy the entry to the object cache;
4 |   Remove the entry from the write-buffer;
5 end
```

energy consumption of a program. A large write-buffer may allow for a bulk transfer of some data, reducing the total transfer cost, while a small write-buffer results in faster release operations at the cost of possibly more expensive in total data transfers. Additionally, a small write-buffer may result in multiple write-backs of the same data, if that

Algorithm 2: Hybrid software cache (cont.)

Procedure invalidate cache
Result: All cached entries are invalidated

```
1 foreach entry in the current thread's write-buffer do
2 |   Invalidate the entry;
3 end
4 foreach entry in the object cache do
5 |   if the tag bit corresponding to the current thread is set in the entry's bitmap then
6 |   |   Invalidate the entry;
7 |   end
8 end
```

data are being written more than once in the corresponding critical section. Since the fine tuning of the write-buffer size depends on the nature of each program, a possible optimization is the use of the just-in-time (JIT) compiler to set a write-buffer size per application or even per code segment. JIT compilers are ubiquitous in modern JVMs and generate optimized code by profiling code segments.

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.

Replacement Policy Regarding the replacement policy of the software cache, we avoid the book-keeping overhead of popular cache algorithms, like LRU, LFU etc. Blackburn et al. [13] show that the total size of live objects in the benchmarks of the DaCapo suite is on average about 6KiB, while the worst case scenario does not exceed 72 MiBs. Given that the benchmarks are parallel and the threads do not access all the data at once without any synchronization, we believe that using object caches on the granularity of MiBs should suffice. Additionally, since object caches are being invalidated at *acquire* operations, we expect the case of running out of space in the object cache to be rare. As a result, when there is not enough space left in it, we invalidate the whole object cache. Emptying the cache increases the number of fetches, but simplifies the allocation algorithm, since there is no de-allocation or fragmentation. In the case of write-buffers we simply flush and invalidate it, as we do in *release* operations. This way we also expect to reduce the write-backs needed to be performed at synchronization points.

Garbage Collection

As mentioned in Section 4.1.1, garbage collection on non-cache-coherent architectures is a complicated issue, mostly orthogonal to the work presented in this thesis. In this work, we briefly discuss how the JVM design can help in the design and implementation of the garbage collector, and how the garbage collector can potentially improve the JVM's performance.

Distributed JVMs usually distribute garbage collection at the node level—each node may garbage collect its own heap. The lack of the global memory view from a single thread makes garbage collection distribution necessary to achieve good performance. Each computation block has direct access to its *global heap area*, meaning it could potentially perform a garbage collection on it. However, looking at a small portion of the heap does not provide enough information about the reachability of an object. For instance, if an object A in global heap area X, is referenced by an object B in global heap area Y, a garbage collection cycle on the global heap area X could result in A being garbage collected since there are no references to it known to the corresponding computation unit.

To allow for local garbage collection cycles we propose the introduction of an additional property per object, marking whether this object has been shared with any other computation units or not. Objects marked as shared are considered *reachable*, when performing local garbage collections, ensuring that they will not be garbage collected while they might be still reachable from remote *global heap areas* or remote Java threads' stacks. Since new objects are normally allocated on the local *global heap area*, by default new objects are marked as non shared. On the less common case where the local *global heap area* runs out of space and memory is requested from a remote *global heap area*, the new object is marked as shared immediately. A non shared object gets marked as shared in the following cases:

a) when it is reachable from a thread that gets migrated to a remote computation block, or

b) when a reference to it is created in a shared object.

This approach comes with the drawback that in applications with long reference graphs, the marking overhead might be significant. Note, however, that all reachable objects, from a shared object, are also shared. As a result the marking traversal does not traverse already marked objects, reducing the marking overhead as the application progresses. Additionally, following the *generational hypothesis*, the majority of the objects are expected to have short lifetimes and thus not get shared during their lifetime.

A generational garbage collector could take advantage of the object marking and use it for local garbage collection cycles on the young generation. For the second generation a copying garbage collector performing adaptive home migration, as presented in

JESSICA2 [107], is expected to reduce remote memory accesses and improve the performance and energy efficiency of the JVM.

There are also garbage collectors that employ heuristics to further improve spatial locality. Huang et al. [40] introduce the Online Object Reording (OOR) enhancement to copying garbage collectors. OOR analyzes the accesses of frequently invoked methods and identifies the frequently accessed fields. These fields are then copied to contiguous memory addresses to further improve spatial locality and reduce execution time. OOR introduces at most 2% overhead and yields up to 25% better performance compared to the standard copying garbage collectors.

Since remote memory accesses are not coherent, we believe that a message passing approach is needed to implement an incremental garbage collector, but further discussion on this topic is out of this thesis' scope.

4.2.2. Synchronization

To provide mutual exclusive access to monitors, a centralization point per object is necessary. In shared memory architectures, this centralization point is the monitor field of the object. Multiple threads compete to acquire ownership of that field through atomic operations. In non-cache-coherent architectures, however, atomic operations are not always available, but even when they are, high contention on a single memory address is expected to reduce performance. A trivial approach is to move the centralization point from the object's monitor to the computation block attached to the *global heap area* where the object resides. This, however, is expected to slow down monitor acquisition when the corresponding computation block happens to be busy, since the requester will need to wait for the computation block to serve its request. An interrupt driven approach could reduce this delay at the cost of interrupting the application's workload, which might add variable overhead to the application, depending on the underlying hardware implementation of interrupts.

In this work, we propose the use of specialized cores that will act as *synchronization managers*. Each *synchronization manager* is responsible for a subset of the objects in the heap. Subsets need to be evenly distributed across *synchronization managers* and not be responsible for objects in contiguous memory addresses, to better distribute the requests. The synchronization manager employs a message queue that constantly polls for requests and serves them. To further improve energy efficiency, we suggest an interruption driven model where the synchronization manager idles until a message arrives. This model, however, relies on the efficiency of the interruption handling mechanisms and may significantly affect performance, depending on the underlying architecture.

Although the use of specialized cores has the disadvantage of sacrificing some computing resources, we consider it a fair trade-off. Specialized cores reduce the latency of monitor acquisition by being highly available. In architectures with heterogeneous cores

or configurable hardware, synchronization managers can be run on low-end energy efficient cores, since the computation requirements are low.

Monitor Handling

To enter a monitor, Java threads send a request to the corresponding synchronization manager and then yield. Synchronization managers hold a record for each monitor they are responsible for. This record holds the *owner* of the monitor. When serving a request, the synchronization manager checks whether the object is available —its owner is `null`. If available, then it proceeds with updating its owner and sends back a message with the current owner. If the current owner is different than the requesting thread, then the monitor is already acquired and the thread needs to retry.

Since monitor-acquisition is blocking, the program only proceeds after the monitor is acquired. To reduce network traffic, contention, and energy consumption, we extend the record by adding an *acquisition queue*. This way, when a monitor is already taken, instead of replying with the current owner, we add the requesting thread to the acquisition queue. Later, when the monitor becomes available, the synchronization manager checks the acquisition queue and if it is not empty de-queues the oldest requester and assigns it as the new owner of the monitor. Finally, a message is sent back to the requester to notify it about the successful monitor acquisition.

To further reduce network traffic, contention on the synchronization manager, and energy consumption, we propose the reuse of a monitor up to a threshold of T times, from Java threads that run on the same coherent-island. A monitor acquired by a Java thread may be directly assigned to another Java thread running on the same coherent-island without notifying the synchronization manager. The threshold T is used to provide some fairness among threads running on different coherent-islands and avoid starvation.

To achieve this, we introduce the *local monitor*, a record resembling a monitor. It is essentially an extension of the record used in the synchronization manager, that is comprised of:

a) the monitor's owner;

b) a concurrent queue, called *acquisition queue*, that holds the threads waiting to acquire the monitor;

c) the nesting level; and

d) a counter of continuous acquisitions of that monitor.

Local monitors are stored in a concurrent, local per coherent-island, data structure associating objects with local monitors.

Algorithms 3 and 4 present the algorithm for synchronization management with local monitors. Procedure *Monitor Enter* presents the process of acquiring a monitor. When

Algorithm 3: Synchronization Management with Local Monitors

Procedure Monitor Enter(object)

Input: The object to enter its monitor

```
1  monitor ← get local monitor associated with the object;
2  if monitor is null then                          // there is no local monitor
3  |  monitor ← create new local monitor for object;
4  |  Try to atomically associate monitor with the object;
5  |  if failed then          // Another thread associated a local monitor
6  |  |  Destroy monitor;
7  |  |  monitor ← get local monitor associated with the object;
8  |  end
9  end
10 if monitor's owner is null then                  // the monitor is not acquired
11 |  Try to atomically assign current thread as the monitor's owner;
12 |  if succeeded then
13 |  |  Send enter request to synchronization manager;
14 |  |  Yield until a positive reply arrives;
15 end
16 if the current thread is the monitor's owner then
17 |  Increase monitor's nesting;
18 else                                  // The monitor is owned by another thread
19 |  Append current thread to monitor's acquisition queue;
20 |  Yield until the monitor gets assigned to the current thread by a releasing thread or
   |   a positive reply, for this thread arrives;
21 end
```

a Java thread requests ownership of an object's monitor, it first creates a local monitor if one does not exist. Introducing a new local monitor requires an update of the concurrent data structure that associates objects to monitors. We rely on the data structures implementation to atomically insert the new monitor and in case of failure due to a race, to return an error. Then if the monitor is available, the JVM proceeds by requesting the monitor from the corresponding synchronization manager, assigning the local monitor to the current thread, and increasing the nesting level. Since multiple threads may try to acquire an available monitor concurrently, we synchronize them on the local monitor's owner field. The threads race to atomically set the owner. The winner sends a request to the synchronization manager and yields until it gets a positive reply to its request. The remaining threads get added to the local monitor's acquisition queue and yield until the monitor gets assigned to them by a releasing thread or they get a positive reply to their request. The same happens when the monitor is already owned by another thread. Note that the threads in the acquisition queue need to check for replies

Algorithm 4: Synchronization Management with Local Monitors (cont.)

Procedure Monitor Exit(object)

Input: The object to exit its monitor

```
1  monitor ← get local monitor associated with the object;
2  Decrease monitor's nesting;
3  if monitor's nesting > 0 then
4  |  return;
5  else if monitor's acquisition queue is empty then
6  |  Set monitor's acquisition counter to zero;
7  |  Send release message to synchronization manager;
8  else
9  |  thread ← dequeue thread from monitor's acquisition queue;
10 |  if monitor's acquisition counter < T then
11 |  |  Increase monitor's acquisition counter;
12 |  |  Assign thread as the monitor's owner;
13 |  else
14 |  |  Set monitor's acquisition counter to zero;
15 |  |  Send release message, combined with an acquire message for thread, to
   |  |   synchronization manager;
16 |  |  Set monitor's owner to ;
17 |  end
18 end
```

from the synchronization manager, although they never explicitly send a request to it. This is related to the monitor exit procedure that we describe next. In the trivial case, where the monitor is already owned by the current thread, the JVM locally increases the nesting level.

Procedure *Monitor Enter* presents the process of releasing a monitor. When a thread finally releases a monitor, it first decreases the nesting level. If the nesting level is not zero then returns, since the monitor is still owned by the releasing thread. If the nesting level reaches zero, the JVM checks if there are any threads on the same coherent-island waiting to acquire the monitor. If not, the JVM resets the acquisition counter and sends a release request to the synchronization manager. In the case that there are other threads waiting for the monitor on the same coherent-island, the JVM dequeues a thread from the queue and checks whether the monitor reuse threshold is reached. If not, it proceeds by assigning the monitor to the dequeued thread and increasing the reuse counter. If the threshold is reached it sends an acquire request for the dequeued thread to the synchronization manager. This last step keeps the remote messages count low since, for each object, only a single outstanding acquire request is allowed per coherent-island. This request, however, is generated for a different thread than the current one;

thus, that other thread needs to check for the reply. This is why the entering threads yield until they either get the monitor from an exiting thread, or until they get a positive reply from the synchronization manager.

This design reduces the contention on the synchronization manager and the network in three different ways:

a) limiting the acquire requests per coherent-island—each coherent-island may have only a single outstanding acquire request per object;

b) combining monitor acquisitions—each monitor can be reused up to T times before returning it to the synchronization manager;

c) distributing monitor management—each local monitor keeps information about nested acquisition eliminating redundant messages to the synchronization manager about nesting.

Another benefit of this design is that it reduces remote memory transfers caused by software cache write-backs. JMM defines that all writes performed before a release operation, must become visible to reads performed after a subsequent acquire operation of the same monitor. As long as the monitor ownership stays in a single coherent-island and there are no other interleaving synchronization operations, the writes performed in the critical section protected by that monitor does not need to be written back to the remote memory. They only need to become visible to the thread, within the coherent-island, that acquires the monitor next. When reusing a monitor, the releasing thread may just propagate its dirty data to the cache of the thread to which it is assigning the monitor. Since the JVM is aware of the monitor acquisition sequence, it is able to transfer data between the write-buffers of the corresponding threads. Data accessed within a critical section protected by a monitor are expected to be accessed by another critical section protected by the same monitor. In such cases, writing dirty data back to the remote location, and then fetching them back to the object cache is a waste of resources, energy, and time. Directly copying between the write-buffers has the benefit of transferring data within the coherent-island, which is more efficient. Finally, when the monitor is released to the synchronization manager only the releasing thread needs to issue write-backs to remote memories for its software cache dirty entries.

Object Wait-Notify

Since object-wait and object-notify are tightly coupled with monitors, we delegate them to the synchronization manager as well. We further extend the monitor records in the synchronization managers to include a *waiters queue*. When a thread invokes `wait()` on an object, a release request combined with a wait request is sent to the corresponding synchronization manager. The synchronization manager enqueues the thread to the waiters queue and releases the monitor. Respectively, when a thread invokes `notify()`, a notify request is sent to the synchronization manager, which dequeues a

thread from the waiters queue and notifies it. The notification itself may be implemented as a message, a remote write, or even a remote interrupt. In the case of `notifyAll()`, as slightly different requests is sent to the synchronization manager, which dequeues and notifies all threads in the waiters queue.

4.2.3. Thread Scheduling

Following the trends towards task-based programming models (see Section 4.1.3), we suggest the use of tasks to scale an application on a large number of cores. However, threads are a basic construct of parallel applications in Java, especially in legacy code. One of the main differences between threads and tasks is that the latter are much lighter than the first. As a result, tasks are more suitable for applications with irregular parallelism as their creation overhead is low, and there is enough research on scheduling them accordingly to achieve better load balancing. Threads on the other hand are usually used to create multiple servers, most often one per core, that perform some workloads and exchange information through synchronization. Thread over-subscription is often suggested for applications with blocking threads, to improve utilization by executing another thread while another is blocked.

To solve the thread scheduling problem, we propose the use of lock free deques within coherent-islands, and message passing across coherent-islands. The use of deques aims to allow for efficient work-stealing within the coherent-islands. Each core may queue and dequeue threads to and from the bottom of its deque. In case its deque becomes empty, it tries to steal threads from other cores in its coherent-island. If after a number of attempts it fails to find work it attempts to get work from another coherent-island.

For inter-coherent-island scheduling we propose the use of a work-dealing algorithm instead of work-stealing, similarly to Acar et al. [1]. Our work-dealing algorithm differs from that of Acar et al. [1] in that it solely relies on message passing and does not depend on atomic operations. In work-dealing algorithms, the cores coordinate and decide together how to balance work. That implies, however, that idle cores need to wait for active cores to reply back, essentially delaying the load-balancing process. The positive side is that the replies from other idling cores will be immediate, meaning that a thread essentially ends up waiting only when there is work on the remote thread. Since such message exchanges are expected to take a lot of time, we propose the use of the half-steal approach, which Dinan et al. [28] show to be a good fit for distributed architectures. In half-steal, instead of taking a single thread, the requester takes half the threads from the remote queue. This way, it is less likely to run out of work in a short period of time. Additionally, fetching more than one thread, allows for other threads, in the requester's coherent-island, to steal from its deque. To further improve performance we suggest the use of heuristics when choosing which threads to hand over to the requester thread. For instance, threads that have not been started yet should be

preferred over threads that have started and yielded. However, at the time of writing we do not yet use such heuristics in our scheduling algorithm.

Load Balancing

Algorithms 5 and 6 present the proposed algorithm for load balancing Java threads. When a thread yields, the JVM invokes procedure *Schedule Next Thread*, which first checks the current thread's deque for available threads. On failure and while the deque remains empty, it tries to steal work from neighboring threads on the same coherent-island. When a few of the threads on an island are idle –their deques are empty–, it means that there is not enough work for everyone on this island. As a result, we put a threshold X to the number of steal attempts from neighbors. We heuristically use the ceiling of the square root of the number of cores per coherent-island as the value of X, $X = \left\lceil \sqrt{\#\text{Cores per island}} \right\rceil$. We base our choice on the reasoning that if a core observes X idle cores, then these cores have probably also observed X idle cores totalling approximately #Cores per island observed idle cores. Since at least X more threads are also looking for work, we need to periodically come back and check our neighbors' deques as well. To achieve this, we set a threshold Y on the number of failed remote requests. Following the reasoning above, we suggest the use of the ceiling of the square root of the number of coherent-islands as the value of Y, $Y = \left\lceil \sqrt{\#\text{Coherent islands}} \right\rceil$. Note that apart from stealing, the deque might get some work from a new thread started by another thread, as we discuss below. The *Steal Request Handler* procedure is straightforward. If the receiving thread is idle, it sends back a NACK, notifying the requester it has no available work to hand over. On the other hand, if it has some threads in its deque, it dequeues half of them and sends them to the requester.

To further improve load balancing, we employ an algorithm to push fresh threads to other cores. This way we expect to improve the performance of applications without nested threads—threads that create other threads recursively to inherently distribute threads. When starting a new thread, the *Start New Thread* procedure is invoked. The JVM picks a random core from the same coherent-island and sends a schedule request to it if its deque is not full. Otherwise it picks a random island and sends a schedule request to it. The *Schedule Request Handler* procedure is responsible to handle the request accordingly. If the receiver's deque happens to be full, it randomly picks another island and forwards the request to that island. Alternatively, if its deque is not full it adds the new thread to its deque. Since the handler never sends back a reply, there is no need for the threads sending schedule requests to wait for a reply, allowing for faster thread creation and start, when using non-nested threading.

Thread join: In distributed environments the implementation of thread *joins* is also far from trivial. In shared memory architectures, at high level, a thread joining on another

Algorithm 5: Hybrid load balancing

Deque: A, per core, deque holding runnable Java threads

Procedure Schedule Next Thread

Result: A runnable Java thread is scheduled for execution

```
1  while deque is empty do
2      repeat ⌈√(#Cores per island)⌉ times
3          randomthread ← pick randomly a core in the same island;
4          if randomthread's deque is not empty then
5              thread ← Dequeue a thread from randomthread's deque top;
6              Schedule thread to be executed next;
7              return;
8          end
9      end
10     repeat ⌈√(#Coherent islands)⌉ times
11         randomisland ← pick randomly an island;
12         Send a steal request to randomisland;
13         if the reply is positive then
14             Add the threads from the reply to the bottom of the deque;
15             break;
16         end
17     end
18     Handle incoming requests (if any);
19 end
20 thread ← Dequeue a thread from deque's bottom;
21 Schedule thread to be executed next;
```

Procedure Steal Request Handler

Result: An incoming steal request is handled

```
1  if deque is empty then
2      Send back a NACK;
3  else
4      Dequeue half of deque's threads from its bottom and send back a message with
         their addresses;
5  end
```

thread can yield and periodically wake up to check the status of that thread. When the status reflects that the thread finished, then it may proceed. On non-cache-coherent environments, however, this is not efficient. Once again we delegate this process to the synchronization manager. Thread join can be implemented with the use of wait and

Algorithm 6: Hybrid load balancing (cont.)

Deque: A, per core, deque holding runnable Java threads

Procedure Start New Thread

Result: A newly created thread is added to the deque of some core

```
1 randomthread ← pick randomly a core in the same island;
2 if randomthread's deque is full then
3 |   randomisland ← pick randomly an island;
4 |   Send a schedule request to randomisland;
5 else
6 |   Send a schedule request to randomthread;
7 end
```

Procedure Schedule Request Handler

Result: An incoming schedule request is handled

```
1 if deque is full then
2 |   randomisland ← pick randomly an island;
3 |   Forward the schedule request to randomisland;
4 else
5 |   Add the new thread, from the request, to the bottom of the deque;
6 end
```

notify. A thread calling the join function essentially waits on that object, which notifies all waiters when it finishes. Following that scheme, we extend monitor records in the synchronization manager to include a new *joiners queue*. Similarly to `wait()`, when a thread invokes `join()` it gets added to the joiners queue of the corresponding `Thread` object's monitor. On thread completion, an analogous to `notifyAll` is invoked and the synchronization manager dequeues and notifies all joiners in the queue.

Chapter 5.

DiSquawk: 512 Cores, 64 Memories, 1 JVM

To evaluate the algorithms proposed in Chapter 4 we develop DiSquawk, a proof-of-concept JVM that targets non-cache-coherent many-core processors, and implement our algorithms in it. In this Chapter we discuss implementation details of DiSquawk and evaluate it on Formic-Cube, an emulator of a non-cache-coherent many-core processor.

Parts of the work presented in this chapter have been published in the proceedings of the *14th International Workshop on Java Technologies for Real-Time and Embedded Systems* (JTRES '16) [33].

5.1. Formic-Cube's architecture overview

Due to the lack of access to commercially available non-cache-coherent many-core processors on the scale of hundreds of cores, we deploy DiSquawk on Formic-Cube [66], a hardware processor prototype emulating a 512-core non-cache-coherent architecture. The Formic-Cube consists of 64 Formic Boards[66] with a total of 512 MicroBlaze™ [74] cores.

Each Formic board features:

- 8 MicroBlaze™ 32-bit RISC CPUs running at 10MHz
- 128MiB of memory (400MHz DDR DRAM)

Each MicroBlaze™ core features:

- a private non-coherent two-way 4KiB instruction L1 cache
- a private non-coherent two-way 8KiB data L1 cache
- a private non-coherent eight-way 256KiB L2 unified cache
- a DMA engine, supporting 64 outstanding DMAs (32 incoming and 32 outgoing)
- a 4KiB mailbox (messages can be of one or two words or even one cache-line)

- 128 counters, that support atomic increment, decrement and read (but no *fetch and add*, *compare and swap*, *etc.*)

Formic-Cube supports a global address space in hardware and every core can access every virtual address in the system. Each board implements a full network-on-chip. The boards are interconnected in a 3D-mesh using GTP links. Cores can communicate through multiple mechanisms. Using its DMA engine, a core can access any core's cache or DRAM in the system. Cores can issue cache-to-DRAM, DRAM-to-cache and DRAM-to-DRAM DMAs. Mailboxes can also be used to transfer data to another core. Another mean of communication is the use of the hardware counters.

To write-back parts of a cache (and not the whole cache) the programmer may issue a DMA transfer from the cache to the local DRAM. DMA transfers operate on cache line granularity (and alignment). Note, however, that invalidation on cache line granularity is not supported. Instead, the whole cache must be invalidated.

Limitations Imposed by Formic-Cube

Formic-Cube as a prototype exhibits some limitations. One of them is the lack of coherent islands. As a result, DiSquawk does not implement all the mechanisms discussed in Section 4.2, but focuses on the ones we consider mostly unexplored by previous literature—the inter-coherent-island mechanisms. Intra-coherent-island mechanisms mainly focus on the use of atomic operations and concurrent data structures, that are thoroughly studied in the literature.

Additionally, due to the absence of an OS, it limits the range of benchmarks it can currently execute, to *e.g.*, applications not using sockets or a file system. Formic-Cube also lacks of persistent memory. The whole VM, along with the class files of the application, is currently loaded in-memory at boot time. Formic-Cube's clock is scaled down to 10Mhz to emulate a 512-core with a high speed network on chip (NoC). As a result the use of long running compute intensive benchmarks is prohibiting.

5.2. Base Virtual Machine

To avoid the overhead of developing a JVM from scratch, we build DiSquawk on top of the Squawk VM [91]. Squawk is an interpreter based JVM, designed to implement the Connected Limited Device Configuration (CLDC) on embedded systems with limited resources. The main reason we chose Squawk over other more advanced JVM's, like the JikesRVM [3] or the MaxineVM [101], is that Squawk is the only, to the best of our knowledge, available VM designed to run on the bare metal—it does not rely on the existence of an OS. Squawk is mainly written in Java, with a minimal core written in C to

bootstrap the system and execute system specific operations. Existing operating systems do not support the Formic-Cube architecture, so running on the bare metal allows us to avoid porting an OS on Formic-Cube, an open research question in itself. Additionally, not relying on an OS increases the portability of DiSquawk to future prototypes that may not be supported by existing operating systems as well. Finally, avoiding the OS layer allows us to better understand the VM's behavior and evaluate our work, by reducing noise in the measurements.

Squawk comes only with an interpreter and no JIT (Just-In-Time) compilation support. Although JIT is considered the state-of-the-art implementation for byte-compiled programming languages we consider it to be an optimization and independent of the problems we are trying to solve. Our contributions in this thesis are applicable to virtual machines with JIT support as well. Additionally, as we discuss in Section 5.4.5 JIT has the potential to improve the performance of our mechanisms and reduce the overhead they introduce.

DiSquawk is implemented as a combination of modified Squawk VM instances, each running on a different core of Formic-Cube, utilizing all the available cores. These instances only differ from each other in that they each have a unique ID equal to the core ID, and access a different *global heap area* and *cache heap area*. At application start, the instance with core ID zero executes the `Main` function, while the rest instances enter an infinite loop polling their mailboxes for incoming requests. To avoid the overhead of transferring bytecodes to caches, we keep a copy of the application in the native memory of each modified Squawk instance.

5.3. Implementation

5.3.1. Memory Management

The Formic-Cube's total memory is 8GiB (64×128MiB). However, the MicroBlaze™ processor can address up to 4GiB of memory (32-bit CPU). In DiSquawk's design we take advantage of this feature and split the memory to two segments. On each Formic Board, 64MiB are dedicated to the *global heap area* and are addressable and accessible from every core on the system, through a virtual global address space. The remaining 64MiB are used as follows. The DiSquawk reserves 4MiB for the binary code (read-only). The DiSquawk reserves 1MiB per core for the JVM's and the Java's stacks. Finally, the DiSquawk reserves 6.5MiB per core as the *cache heap area* for caching objects from *remote global heap areas*, as discussed in Chapter 4.

Figure 5.1 is a visualization of the memory partitioning. The dark block is the reserved space for the binary code. The turquoise scales are the private memory segments used for the JVM's and the Java's stacks as well as for caching. Each scale denotes a different core. Finally the light green block is the memory partition that serves as a heap slice.

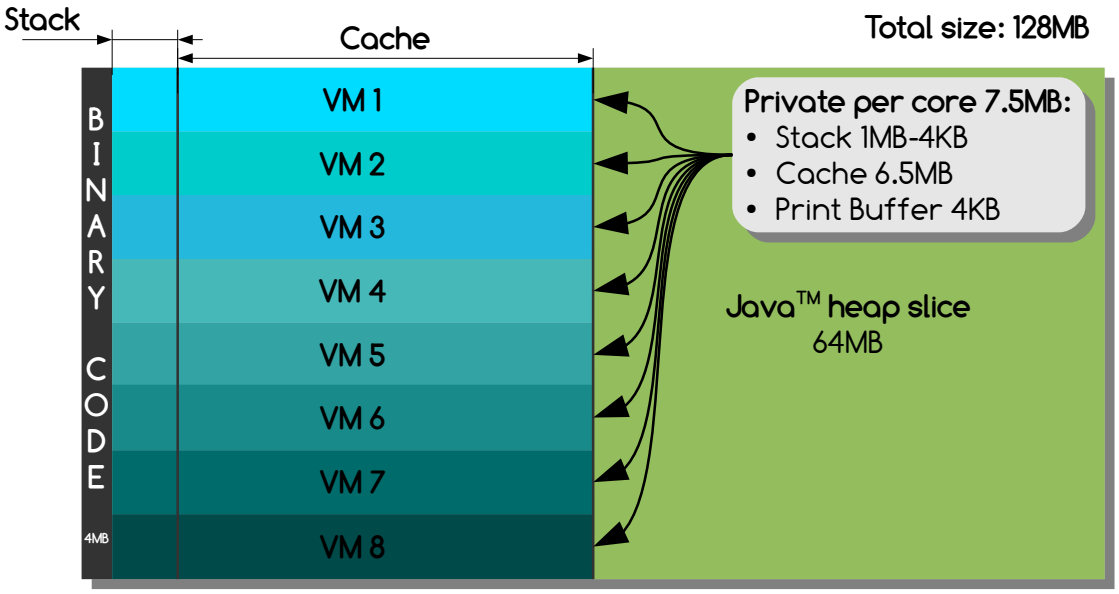


Figure 5.1.: Memory Partitioning

The DiSquawk further splits the Java heap slice in eight artificial sub-slices —one per core. This further splitting defines which core is responsible for which memory range, for special operations such as garbage collection, memory allocation, and so on.

Figure 5.2 shows the interpretation of a virtual address in the global address space. The 6 most significant bits (MSBs) of the virtual address are interpreted as the board's id that owns the physical memory that this virtual address maps to. The rest 26 bits are the offset in the physical memory of that board, where this virtual address maps. Because Formic-Cube does not support DMA transfers with finer granularity than the size of a cache-line, we make DiSquawk's allocator to work on cache line granularity, to avoid implications regarding software caching as we discuss in Section 5.3.2. This way we restrict each cache-line to a single object. As a result, on the Formic-Cube the 6 least significant bits (LSBs) are always zero in the virtual addresses. Additionally, the 3 MSBs of the offset can be used to find the core ID of the physical core that is responsible for the memory range in which the virtual address maps to.

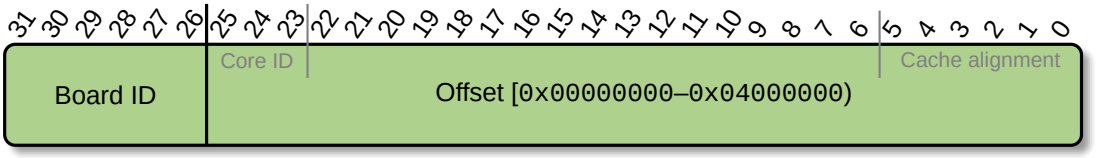


Figure 5.2.: Virtual address interpretation

To calculate the physical address on the corresponding board, DiSquawk adds an extra offset of 64MiB (`0x04000000`) to the interpreted offset, since the heap slices are mapped in the higher 64MiB of the physical memory. As a result, the actual offset is in the range $[\texttt{0x04000000} - \texttt{0x08000000})$.

In this implementation each board needs to address a total of 4GiB (global Heap) plus 64MiBs (local storage). Unfortunately the 32 bits of the MicroBlaze™ are not enough to address this size. As a result, Virtual addresses with their 6 MSBs set to zero conflict with the local physical addresses, in the range $[\texttt{0x00000000} - \texttt{0x04000000})$, of the first board with ID 0. To resolve this issue, DiSquawk assigns IDs in the range $[1 - 64)$ instead of the range $[0 - 64)$. As a result we lose 64MiBs of the global Heap memory (those of the 64th board). Note, that this is a limitation of our architecture supporting up to 32bit addresses. Most architectures nowadays support up to 64bit addresses. In such architectures there is no need to "sacrifice" any memory.

Note also that the range $[\texttt{0x04000000} - \texttt{0x08000000})$ still conflicts with translated addresses which happen to map in the current board's heap slice. This is a problem only if DiSquawk tries to translate an already translated address. DiSquawk performs translation on a late stage, thus avoiding this behavior.

Memory allocation

DiSquawk is built on the bare-metal, thus `malloc` is not available. DiSquawk uses the memory partitioning described above. When a thread needs to allocate a chunk of memory, it gets it from its own Java-heap slice. Whenever a chunk of memory is requested, DiSquawk rounds it up to the cache-line size. As a result, two different objects never share the same cache-line. As described in Section 5.3.2 this is mandatory to avoid implications regarding software caching.

As a first implementation, DiSquawk can only allocate memory from its sub-slice. This way memory allocation is completely distributed to the eight cores and can be done without any synchronization. However, this limits the size of memory a thread can allocate to 8MiBs. To overcome this limitation we also support remote allocation. In case that a DiSquawk instance cannot allocate the needed space locally it can request the missing memory from a remote DiSquawk instance.

5.3.2. Software Cache

Partitioning the Java Heap across 64 boards raises the need for 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 access an object that is located in a remote DRAM module, a DMA transfer for each access would be too expensive. To reduce the overhead, DiSquawk is based on software

caching at object granularity, as we describe in Chapter 4. Caching, however, requires extra effort in order to provide a coherent view of the data, as it is defined in Chapter 3.

Since the 4GiB Java heap is sliced, to find out in which heap slice a heap address resides, DiSquawk right-shifts the corresponding heap address by 26 bits. The resulting number is the Java heap slice ID. On the Formic-Cube we use one-to-one mapping of Java heap slice IDs to board IDs as we describe in Section 5.3.1. In the case that the resulting Java heap slice ID is different than the board ID of the core trying to access the corresponding heap address, DiSquawk issues a remote DMA transfer to fetch the corresponding cache-line to the local software cache.

For the software cache implementation we employ a double hashing hashtable. The first hash function uses a combination of the object's relative placement in its home's global heap area, and its home ID. This way, we avoid collisions between objects with the same relative placement on different scratchpads. Such cases are expected to be common, since every DiSquawk instance is identical and expected to behave similarly. The second hash function uses the 10 MSBs of the address, where the first 6 MSBs denote the home scratchpad and the next 3 bits denote the home core. This way, objects from different homes follow different probes, reducing the collisions. Note that the scratchpad IDs start from 1 instead of 0 to avoid the case of the second hash function returning zero for objects with core ID 0 and scratchpad ID 0. As a result, the second hash function returns integers in the range $[8 - 512)$.

Normally, for only to-be-written memory chunks there is no need to perform a fetch before writing, since the fetched data will eventually be overwritten. Formic-Cube, however, only supports write-backs of cache-line granularity. As a result, DiSquawk needs to first fetch the cache-lines that will be written, to avoid writing back random data for the non-written part. This also restricts concurrent accesses to a single cache-line. Two threads are not allowed to write on different parts of a single cache-line, since at the write-back the second writer of the cache-line would overwrite any writes performed by the first one. We implement this by modifying DiSquawk's allocator to work on cache line granularity. This way we restrict each cache-line to a single object. However, correctly synchronized writes to different fields of an object may still produce invalid executions if the fields happen to be in the same cache-line and the writes are executed concurrently, since one or more of the writes might get lost. In our experiments we avoid such scenarios by using the object's monitor to access all the fields of a single object.

Note that other architectures like Cell B.E., support remote DMA transfers with finer granularity than a cache-line. On such architectures we can have many cores writing on different parts of a cache-line, enabling more flexibility and allowing finer locking mechanisms. Due to the object oriented nature of Java, however, it is not expected to have different locks to access different fields of the same object all stored within the same cache line. Thus, we do not expect enabling finer locking granularity than object granularity to significantly improve the performance of Java applications. The flexibility

however, to pack multiple concurrently accessible objects in a single cache-line could significantly reduce memory fragmentation in some programs.

Object Caching

DiSquawk's software cache operates on object granularity. When fetching an object, DiSquawk always fetches the whole object. In DiSquawk an object in memory can be visualized as an array. In Figure 5.3 we provide a visualization of the in memory structure of the stored data.

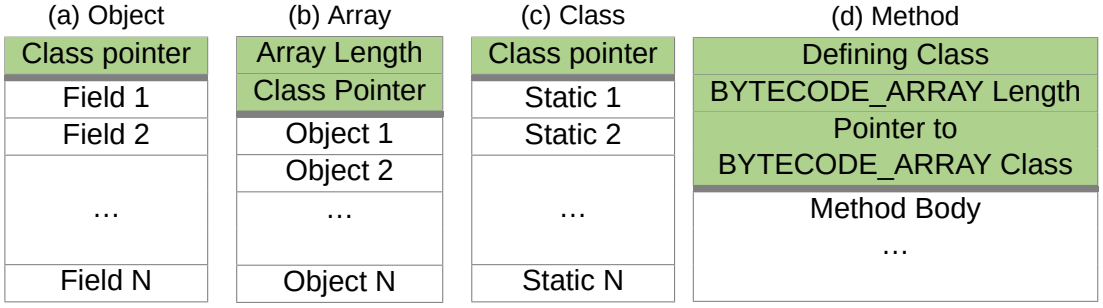


Figure 5.3.: Visualization of the stored data layout. The headers metadata are colored.

Field access are essentially accesses to predefined, per field, offsets in this array-like structure. Partially fetching an object requires an extra directory structure that maps field numbers to offsets. Since most objects usually fit in a single cache line [13], we consider the overhead of keeping such an extra directory prohibitive, both in terms of space and performance.

Fetch and Write-back

When a virtual address is dereferenced and its home node is different than the core dereferencing it and it is not present in the cache or the write buffer, DiSquawk issues an object fetch. The first step is to allocate a single cache-line in the software cache and fetch, with a DMA transfer, the first cache-line of the object. As shown in Figure 5.3, the first cache-line includes enough information to infer the object's class and figure out its instance size. If the instance size of the class is less than a cache-line, then the whole object has already been fetched. If, however, the class instance size is larger than a single cache-line then DiSquawk allocates a new memory chunk of that size in the software cache and issues a new DMA transfer of that size.

The write-back process is trivial. DiSquawk issues a DMA transfer of the cached object to its home node. Note here that DiSquawk could issue smaller DMA transfers and only

write-back dirty parts of the cached object instead. This is not possible, however, due to the lack of support for non cache-line granularity transfers by Formic-Cube.

5.3.3. Thread Scheduling

Since Squawk was not initially designed for many-core architectures, in DiSquawk we improve its thread scheduling mechanism. Due to the lack of coherent-islands we are not able to take advantage of the full, hybrid, load balancing mechanism presented in Section 4.2.3. Instead, we implement a variation of the *Start New Thread* and *Schedule Request Handler* procedures. In our implementation, since there are no coherent-islands, the procedures pick randomly a core from the system to send the schedule request.

After instantiating a thread object, when the `Thread.start()` function is invoked the DiSquawk scheduler peeks a core and schedules the thread in its thread queue. Specifically, the running thread instantiates the new thread in its core's memory. When the `Thread.start()` is invoked, that core sends a message to the core peeked by the scheduler. This message contains a special operation code (op-code), the sender's core ID, and the previously instantiated thread object's address. On the other end (i.e. the peeked core) at every reschedule (see below) the corresponding core checks its mailbox for new messages. In the case of a message with the previously mentioned special op-code the receiver: first fetches the remote thread object from the sender's hardware cache to the receiver's software cache; locally allocates a stack for the thread; and then adds it to the thread queue.

By allocating the stack locally, we make most of the thread related accesses local. Additionally, in Squawk and DiSquawk there is a JVM specific class, `VMThread`, that acts as the backend of the `java.lang.Thread` class. As a result, each `Thread` object is tightly coupled with a `VMThread` object. The `VMThread` instance is the one holding the thread's stack and runtime information for that thread. Since the `VMThread` object is only useful after starting (through `Thread.start()`) a `Thread` object, we instantiate it lazely together with the stack. As a result, by allocating a thread locally to the *global heap space* of the core where it was scheduled, we avoid using the cache for every memory access on its stack. Additionally, since the thread's state is only changed at the beginning and at the end of the thread's life we keep it at the thread's initial host core. This way, threads querying the state of another thread synchronize with the `Thread` instance instead of the `VMThread` instance. When the thread reaches completion it updates its `Thread` instance state, without acquiring the monitor, writes back the cached `Thread` instance, and notifies any joiners through the synchronization manager. Note that this is safe, since only a single core —the one running the thread— can change its state.

In Squawk and DiSquawk rescheduling happens when:

1. A thread's `start()` method is called
2. A thread's `yield()` method is called
3. A thread's `sleep()` method is called
4. A thread invokes the `join()` method of another thread
5. A thread waits for an event
6. A thread fails to enter a monitor
7. A thread exits a monitor and there are threads with higher priority waiting for the monitor
8. A thread invokes an object's `wait()` method
9. A thread reaches completion or aborts
10. A thread reaches its 1000th backward branch. That said, preemption is working based on backward branches quanta (1000) and not on time or bytecode quanta.

5.3.4. Java Monitors and The Synchronization Manager

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 [39, 66].

To implement the Java monitors on such architectures we use the synchronization managers described in Chapter 4. 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 [37], 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.

Since Formic-Cube does not provide coherent islands, in our implementation we use the local monitors to optimize monitor enter requests performed to the same monitor by multiple Java threads running on a single core, instead of a coherent island.

For the communication between the Squawk VM instances and the synchronization managers we rely on the hardware mailboxes.

As discussed in Chapter 4, out design extends the monitors in a way such that each object copy has a *local* monitor and there is a single *central* monitor per object used to ensure mutually exclusive acquisition. *Local* monitors are used to synchronize accesses to objects by Java threads running on the same core, while *central* monitors are used to synchronize accesses to objects by Java threads running on different cores.

The *central* monitors are managed by the *synchronization managers* which are running on dedicated cores. The monitors are sliced in N sets where N is the number of synchronization managers. All sets are independent and each slice is managed by a single synchronization manager. As a result an object's *central* monitor can only be managed by a single synchronization manager. With each object's *central* monitor being managed by one single manager, we can ensure that there is no possibility for two different, correctly synchronized, Java threads to enter the same critical region concurrently. The synchronization manager's mailbox acts as a serialization point for the incoming requests. It's the synchronization manager responsibility to handle these requests and reply appropriately to the corresponding *requester*.

Each request consists of two words packed in a single atomic message. The first word includes the request's operation code packed with the requester's core id and board id. A visualization of the packing is shown in Figure 5.4. The second word of a request includes the object we want the synchronization manager to act on.

Similarly, In the case of replies to monitor enter we send two words per reply. The first word includes the synchronization managers reply and the second word includes the object the synchronization manager acted on and relates to this reply.

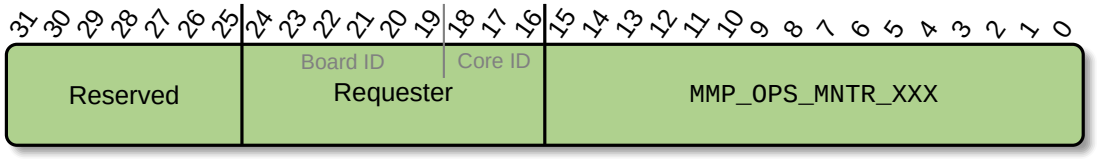


Figure 5.4.: Visualization of requests to synchronization manager

5.3.5. 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 [58].

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 [80, §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.

5.3.6. 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

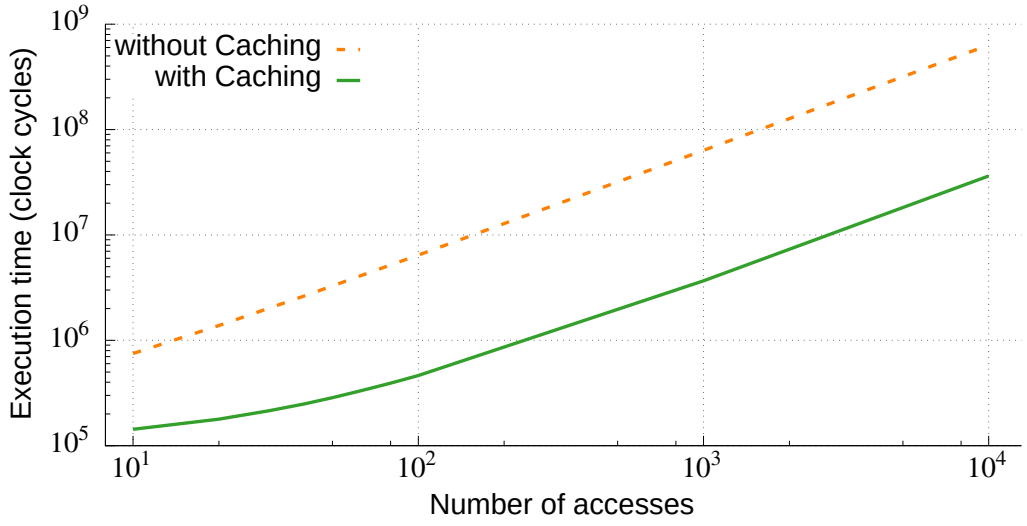


Figure 5.5.: The memory abstraction.

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.

5.4. Evaluation

5.4.1. Software Cache Impact

To demonstrate the impact of software caching we perform a simple experiment where a thread accesses a remote `Integer` object N times and compare the execution time when software caching is enabled and when it is disabled. Figure 5.5 presents the results of this experiment. On the x-axis is the number of accesses performed by the thread and on y-axis is the total execution time in clock cycles. The dashed orange line plots the execution time when software caching is disabled, and the solid blue line plots the execution time when software caching is enabled. We observe that the execution time is about an order of magnitude longer when software caching is disabled, even for objects of a simple class like `Integer`.

5.4.2. Scheduling

To evaluate the performance of the `Thread.start()` implementation we use a micro-benchmark that creates a single thread and schedules it to a remote idling core, then

it joins on it and completes. Our measurements show that from the beginning of the `Thread.start()` method to the actual run of the thread on an idle core, it takes 18640 clock cycles on average. Of this time: 1.6% is spent to choose a remote core and to construct and issue the schedule request; only about 0.5% is spent for the schedule request transfer itself; 28% is spent to allocate and construct the new *VMThread* instance; 55% is spent to initialize the `VMThread` instance; 4% is spent to allocate the stack; 1% is spent to add it to the runnable's queue; 8% is spent waiting for the JVM to schedule it; and the last 2% is spent in other more fine-grained tasks. The measurements show that the most time is spent in local computations performed in Java, since Squawk is written in Java. Squawk's interpreter, however, is not as efficient as the state-of-the-art JVMs with JIT compilation support. We expect even faster thread scheduling in modern JVMs. Regarding thread completion, our measurements show that from the end of a thread till one of its joiners gets notified, it takes around 20000 clock cycles. Of this time, about 15% is spent for the bookkeeping, while the remaining 85% is spent in message transfers and mostly in waking up the joiner. Compared to an early implementation not using the wrapper described above, we achieve a $10\times$ speedup in `join()`.

5.4.3. Synchronization Manager

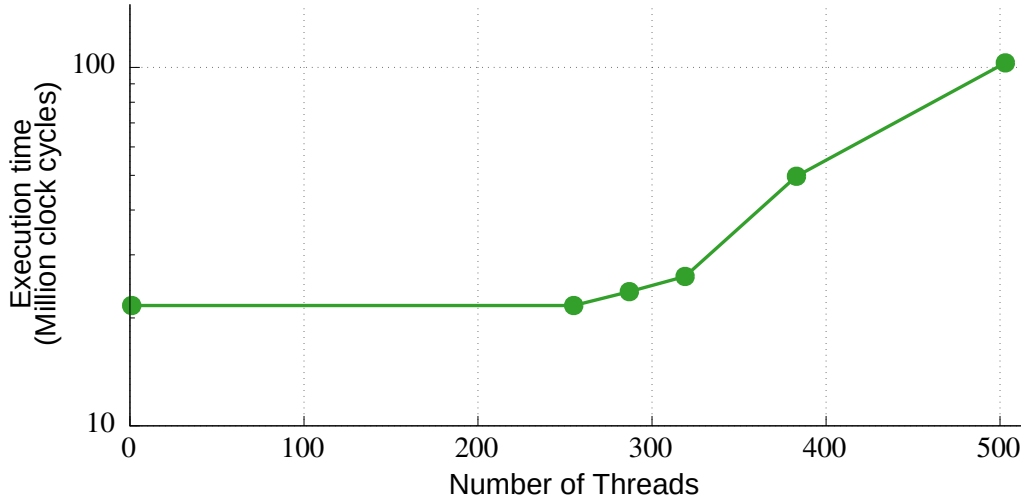


Figure 5.6.: Execution Time (1 Synchronization Manager) vs #Threads

For the synchronization managers we reserve one of the computational blocks. On each core of this block we deploy a synchronization manager. The synchronization managers constantly poll their hardware message queue for incoming requests and directly serve any available requests. To find the number of synchronization managers needed to efficiently handle the requests from 504 cores —the cores left after reserving a board for synchronization managers— we create a micro-benchmark with 504 threads

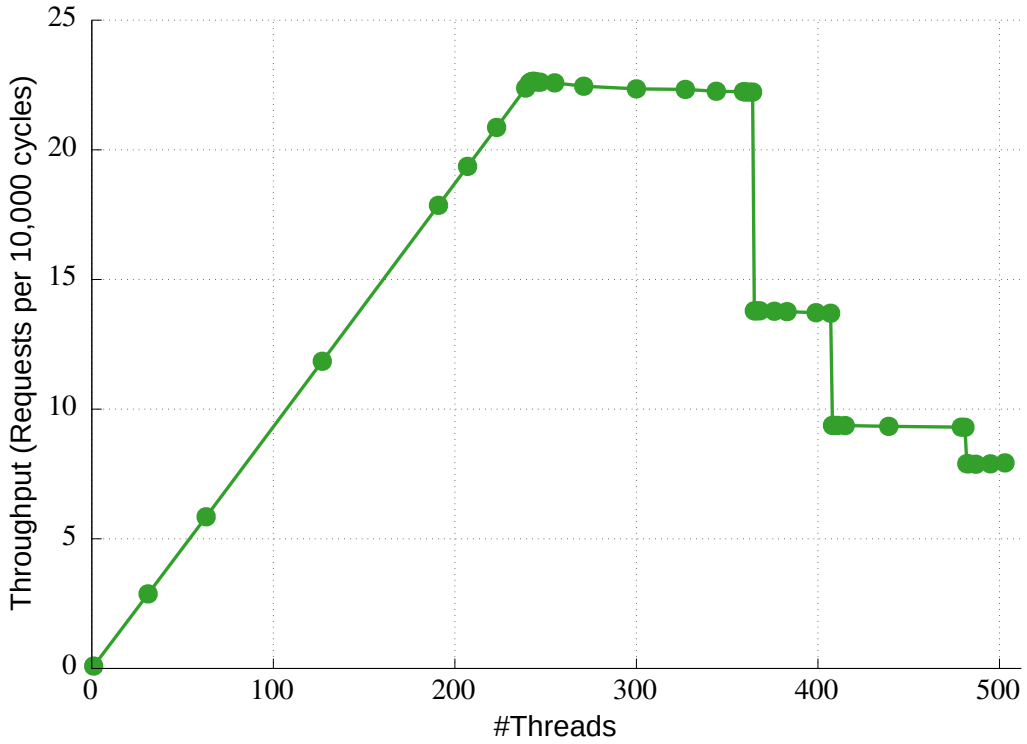


Figure 5.7.: Synchronization Manager Throughput vs #Threads

that each requests to enter a different monitor. We run it with a single synchronization manager and present the results in Figure 5.6. The results show that a single synchronization manager is able to handle up to 256 cores; after that point it starts to become a bottleneck. More interestingly, as shown in Figure 5.7, the throughput of a single synchronization manager drops suddenly at certain numbers of cores. This is caused by the hardware queue getting full. Each request is of 8B size. Each thread in the benchmark can have an outstanding monitor-exit request (non blocking) and an outstanding monitor-enter request (blocking). As a result in the worst case scenario, each thread has 16B in the synchronization manager's hardware queue. Since the hardware queue is of 4KiB size, in the worst case scenario, where the synchronization manager request handling rate is much lower than the requests generation rate by the threads, it can handle up to 256 threads. The measurements show, however, that we only manage to fill the hardware queue and start getting NACKS after 365 threads. As the number of cores increases further, we observe two more sudden drops, which we attribute to the increased number of NACKs and network packages in the network.

In Figure 5.8 we present the results from varying the number of synchronization managers in the same micro-benchmark when running with 504 threads. The results show that the peak throughput –aggregated throughput of all the synchronization managers in the system– is reached when using three synchronization managers. However, we

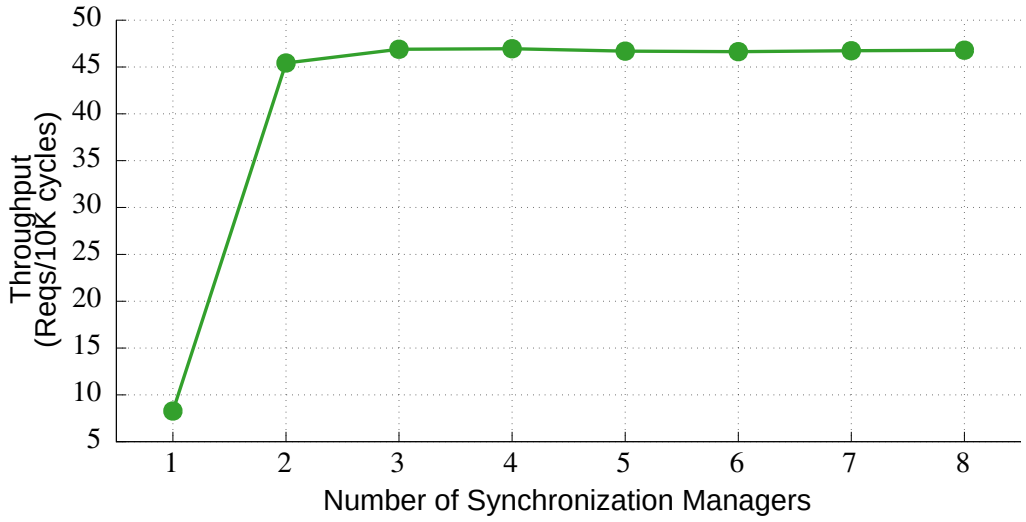


Figure 5.8.: Throughput vs Number of Synchronization Managers

believe that this is an artifact of the slow interpreter that fails to generate requests at a faster pace. Native measurements have shown that a number of 8 synchronization managers is needed to handle 504 cores.

Local Monitors

Due to the lack of coherent-islands we implement local monitors per core instead of per coherent island. This way a monitor may only be reused by several Java threads scheduled on the same core. Additionally, it is only possible to limit the outstanding monitor acquire requests to one per core instead of one per coherent-island. Unfortunately, this does not allow us to measure the impact of the proposed design on the network traffic and the overall performance.

Impact of Queuing Requests

To evaluate the impact of queuing requests in the synchronization manager, instead of replying with NACKs and retrying (see Section 4.2.2), we use a benchmark that spawns multiple threads, that each tries to acquire and release a monitor 100 times. To maximize contention all threads act on the same monitor and perform no other workload other than the acquires and releases.

Figure 5.9 presents the total execution time in billions of clock cycles, when queuing is enabled (blue bars), and when it is disabled (orange bars). We observe a maximum $3\times$

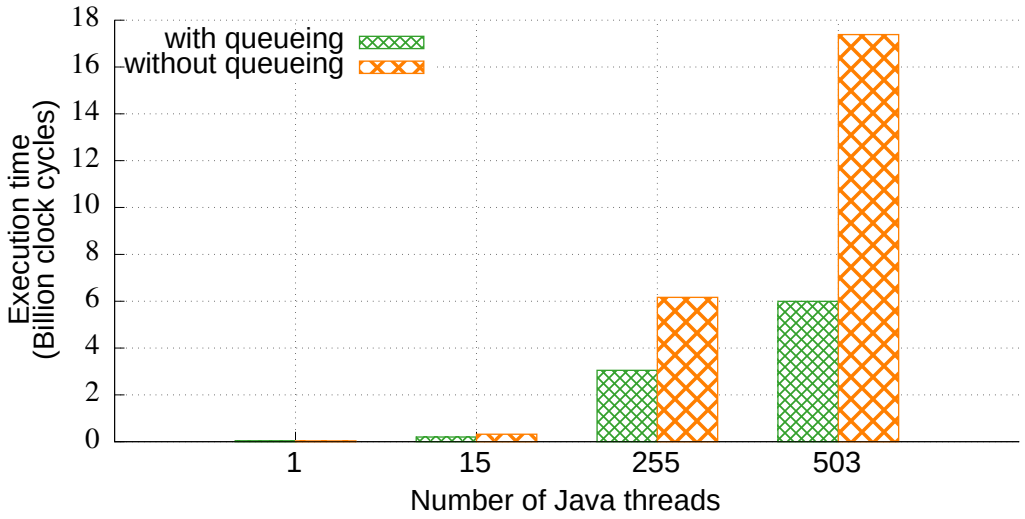


Figure 5.9.: Queuing vs Retrying: Impact on Application's Execution Time

speedup at 504 cores. We observe that the higher the contention, the higher the difference. At a low number of threads this happens because the synchronization manager always finds a request to serve in its queue and can immediately process it and notify the corresponding thread. At a high number of threads, in addition to the high availability of requests, we reduce the network contention by reducing the messages.

Figure 5.10 presents the total throughput, as monitor enters per one thousand clock cycles, measured at the synchronization manager for the same runs. We observe that when queuing is enabled for runs with more than 15 threads, the synchronization manager is able to handle 9 `monitorenters` per one million clock cycles. On the contrary when queuing is disabled, the synchronization manager's `monitorenter` handling rate starts to decrease after 15 cores. This is a result of the additional network traffic, the contention it creates, and the fact that the synchronization manager's mailbox is constantly full, resulting in the synchronization manager being mostly busy with sending NACKs.

Note that the synchronization manager's throughput is much higher than 9 operations per one million clock cycles, as shown in Figure 5.7. In this micro-benchmark however, the throughput is bound by the rate at which a Java thread is able to acquire and release a monitor, since all threads race for the same monitor. Our measurements show that a synchronization manager spends about 400 clock cycles to handle a `monitorenter` operation, and 600 clock cycles to handle a `monitorexit` operation under high contention. This cost is lower under no contention, since the synchronization manager does not perform queue operations.

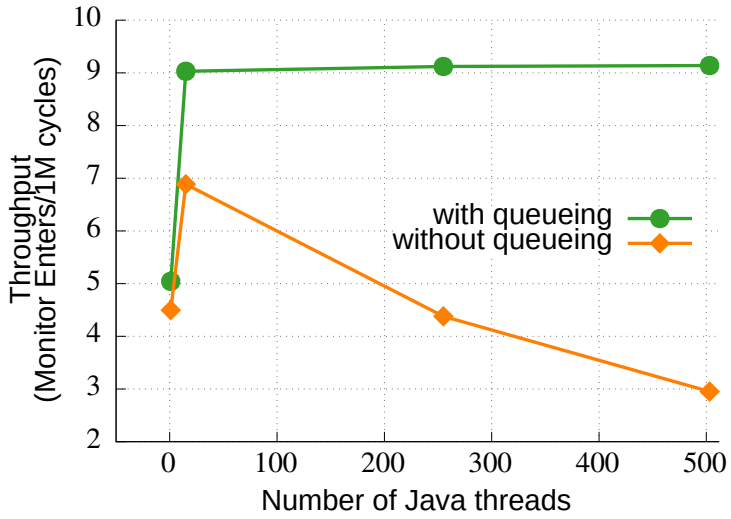


Figure 5.10.: Queuing vs Retrying: Impact on Synchronization Manager's Throughput

5.4.4. Overall Scalability

To evaluate the overall scalability of DiSquawk we use the Crypt, SOR (Successive Over-Relaxation), and Series benchmarks from the Java Grande [92] suite and the Black-Scholes benchmark from the PARSEC suite [11], ported to java. Due to the lack of garbage collection and the upper limit of 4 GiB heap we are unable to run reasonable workloads with the rest of the Java Grande benchmarks as well as more realistic workloads from other benchmark suites. These benchmarks require larger than 4 GiB 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.

For each benchmark we measure its performance 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. HotSpot is the current state-of-the-art Java implementation for shared memory architectures. Since DiSquawk does not support JIT compilation, we disable JIT in HotSpot (using the `-Xint` flag) as well. This allows us to better understand the applications' behavior on both architectures. Additionally, due to the large number of cores, the distributed memory nature, and the slow CPU clock of Formic-Cube we examine weak scaling instead of strong scaling. Strong scaling requires all runs to act on a large workload so that when we run with 512 threads there will be enough work for all the cores, which results on long runs when using only a few threads. Weak scaling on the other hand allows us to avoid such long runs by increasing the workload along with the number of threads. Weak scaling also allows us to run our benchmarks on a *non-distributed* Squawk instance on Formic-Cube to get a sequential baseline for our measurements, since the workload for one thread can fit in the memory of a single board.

Since Formic-Cube is a prototype clocked at 10MHz, a direct comparison of the throughput or the execution time on the two architectures is not possible, thus we compare the scaling of the applications' throughput instead of their absolute performance. To achieve this we measure each application's throughput when run with a different number of threads (one per core), and divide it by the throughput of the application when run with a single thread on the corresponding architecture.

$$\text{Throughput Scaling} = \frac{\text{Throughput with N threads}}{\text{Throughput with 1 thread}}$$

Especially for DiSquawk, we divide the throughput by the throughput of the application when run on a *non-distributed* Squawk instance on Formic-Cube, without our mechanisms for distributed execution. This way we avoid the overhead of our mechanisms and provide a more clear view of the scaling factor. We further discuss the overheads of our mechanisms in Section 5.4.5.

Black-Scholes and Series 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. Figure 5.11 and Figure 5.12 present our evaluation results for Black-Scholes and Series respectively. The number of Java threads, one per core, is placed on the x-axis, and the throughput scale factor is placed on the y-axis. Both axes are in logarithmic scale of base 2. Since we cannot run HotSpot on more than 64 cores, we only plot measurements up to 64 threads for HotSpot. We observe that Black-Scholes when run on HotSpot exhibits linear scaling up to 16 threads and starts to scale in a slower manner on higher number of threads. On the other hand, when run on DiSquawk we observe that it needs to run with more than 2 threads in order to hide the overhead of our mechanisms. However, when run with more than 2 threads it scales linearly with the number of threads. For Series we observe that it achieves close to linear speedup on both architectures, we attribute this to the embarrassingly parallel and compute intensive nature of this benchmark. Since our mechanisms mostly concern memory management and synchronization, in the case of Series the introduced overhead is low, allowing it to scale linearly in DiSquawk.

Figure 5.13 and Figure 5.14 present our evaluation results for Crypt and SOR respectively. Crypt comprises two embarrassingly parallel 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.

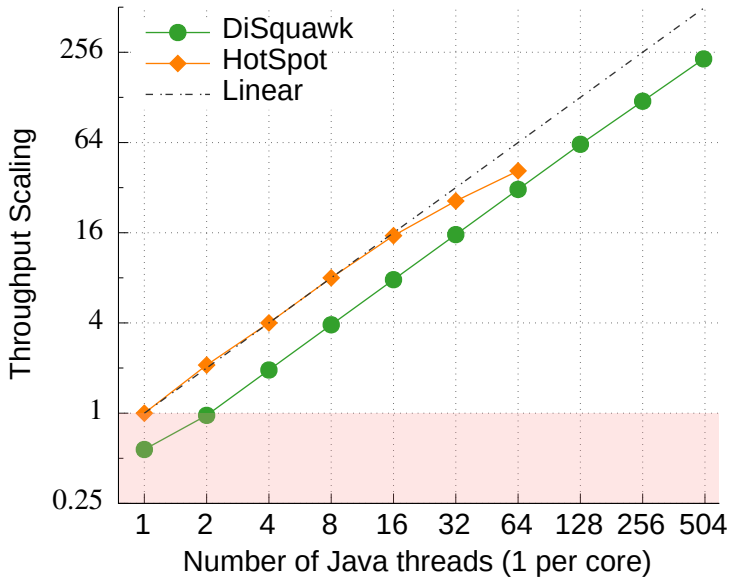


Figure 5.11.: Black-Scholes Scaling

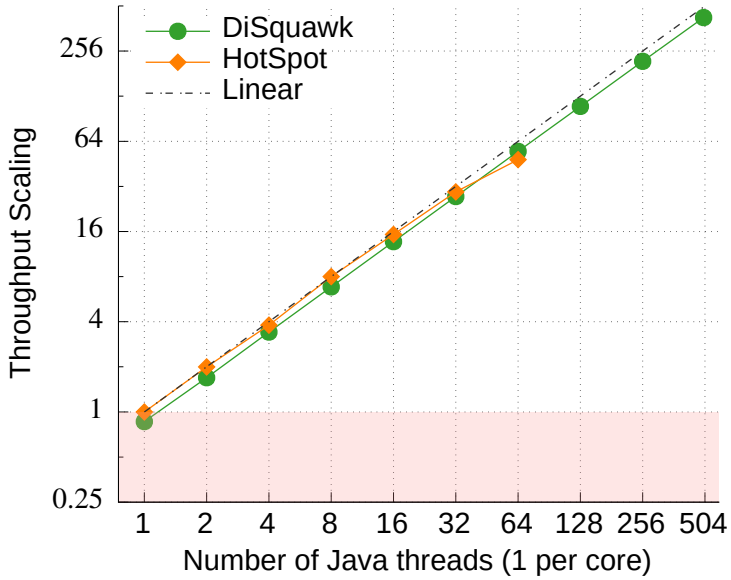


Figure 5.12.: Series Scaling

We observe that both benchmarks fail to scale until 4 threads. We attribute this behavior to the memory intensive nature of Crypt and the high communication rate of SOR.

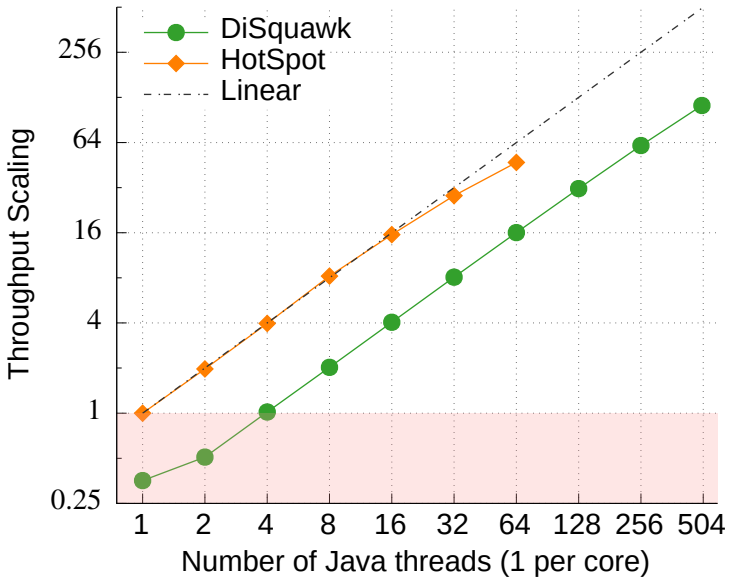


Figure 5.13.: Crypt Scaling

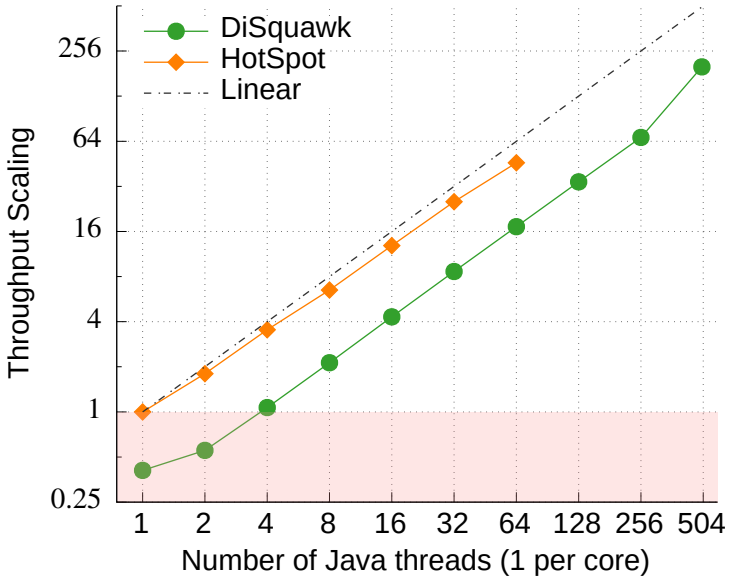


Figure 5.14.: SOR Scaling

5.4.5. Overhead

Measuring the added overhead of our mechanisms on the application's runtime is not trivial. To get an estimation of that overhead we run each application on a *non-distributed*

Squawk instance on Formic-Cube, without our mechanisms for distributed execution. Then we compare the execution time of this run with that of a run on DiSquawk with a single thread. The difference between the execution times of the two runs demonstrates a part of the added overhead, by our mechanisms. Figure 5.15 presents the results of our measurements on the four benchmarks, Black-Scholes, Series, Crypt, and SOR.

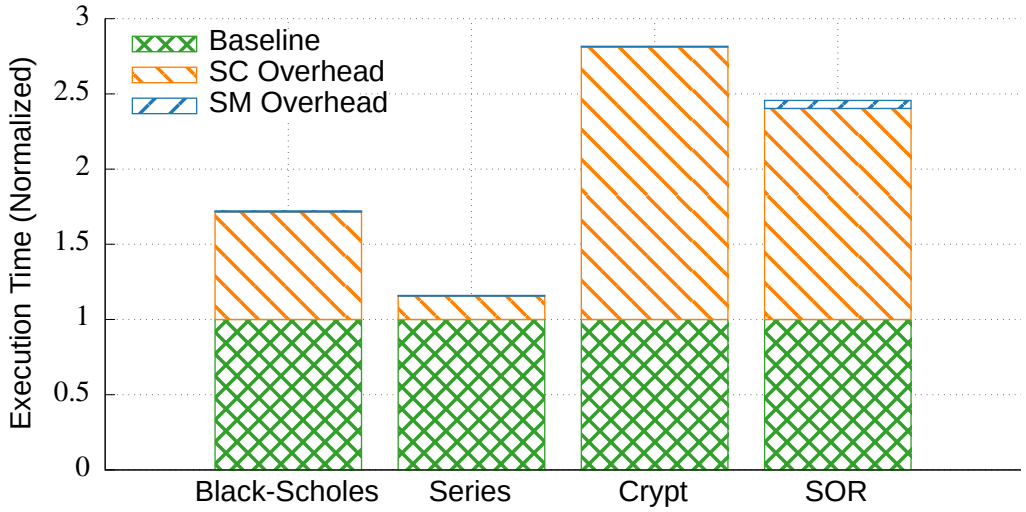


Figure 5.15.: DiSquawk Overheads

This approach however does not measure the total overhead, since it does not account the overhead of caching data, it only accounts the overhead of checking if a memory address is cacheable or not, and the overhead of communicating with a synchronization manager at each synchronization action.

We group the overheads introduced by our mechanisms in categories and discuss each separately.

Checks at pointer dereference

In software caching, to figure out whether a memory address is remote and needs to be cached we need to perform a check at each object reference (pointer) dereference. In DiSquawk we achieve this by shifting each object reference 26 bits right to get the home ID of that reference. When the home ID is equal to the our nodes ID it means that the corresponding object is local and does not need to be cached. However, we still need to strip the home ID from the object reference to access it. As a result, in the best case scenario where the object is in the local memory, we pay the overhead of one shift, one branch, and one logical and operation for every object dereference.

Software cache lookups

In the case where the object resides in a remote memory we pay the additional overhead of looking up the object in the software cache. This overhead might be non-constant per lookup, depending on the number of the already cached data that happen to collide in the hashtable we use as the cache directory.

Memory transfers

If a remote object is not found in the software cache we fetch it from a remote memory and then insert it in our software cache. The overhead of these operations is also non-constant, and depends on the size of the object and the number of the already cached data that happen to collide in the hashtable we use as the cache directory. Furthermore, in the case where the software cache is full we also invalidate it. Note that the memory transfer cost depends on the underlying architecture and cannot be reduced by optimizing the software. The only way to reduce the overhead in execution time is by prefetching data.

Synchronization

Finally, at synchronization points we pay the overhead of communicating with the synchronization manager and in some cases the additional overhead of flushing or invalidating the software cache.

The overheads of the checks and the synchronization are reflected in Figure 5.15, but the cache lookups and memory transfer overheads cannot be measured accurately. Due to the fine grained nature of the lookup and memory transfer operations, measuring them introduces significant noise to the measurements.

To reduce the overheads and improve performance we suggest the use of JIT compilation. Using JIT compilation we can update object references to their stripped counterparts when local or to their address in the software cache when remote and cached. This way hot spots in the application can avoid paying the overhead of checking, and stripping or looking up an object reference multiple times. However we still need to pay the cost of de-optimization when an object gets invalidated.

Chapter 6.

Distributed Java Calculus

In this Chapter we present a Java core calculus that we define along with its operational semantics that models DiSquawk. We use this calculus and its operational semantics to argue that it only generates well formed executions and thus it adheres to JDMM and consequently to JMM. This way we also show that DiSquawk adheres to JDMM and JMM, since the operational semantics of the calculus we define models it.

Parts of the work presented in this chapter have been published in the proceedings of the *13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (PPPJ '16) [34].

6.1. The Calculus

The calculus we define is a minimal core calculus for the Java language, which is based on the Featherweight Java [43] variant introduced by Johnsen et al. [48]. This variant omits inheritance, subtyping and type casts, and adds concurrency and explicit lock support. In this thesis we extend it by replacing the explicit lock support with synchronization operations, *e.g.*, monitor-enter, join, etc. Similarly to Johnsen et al. [48], we also omit Java inheritance, subtyping, and typecasting as they have been extensively studied previously and are orthogonal to cache management. The resulting calculus, called Distributed Java Calculus (DJC), is a minimal calculus that its operational semantics gives us the power to argue about the correctness of cache and monitor management in DiSquawk.

6.1.1. Syntax

The syntax of DJC is presented in Table 6.1. A Java program J consists of a sequence $\vec{D}$ of class definitions. A class is defined as 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

Table 6.1.: Abstract syntax of DJC

Program	J	::=	$\vec{D}$
Class Def.	D	::=	class $C(\overrightarrow{f : \tau})\{e\}\{\vec{M}\}$
Types	τ	::=	$C \mid Bool \mid Nat \mid Unit$
Methods	M	::=	$m(\overrightarrow{x : \tau})\{\text{return } e;\} : \tau$
Expressions	e	::=	$x \mid \text{new } C(\vec{e}) \mid e.f \mid e.f := e$
		|	let $x : \tau = e$ in e | if e then e else e | $e.m(\vec{e})$
		|	e.acquire | e.release | e.monitorenter | e.monitorexit
Values	v	::=	$r \mid () \mid \text{true} \mid \text{false} \mid n$
Contexts	$E(\bullet)$	::=	new $C(v, \ldots, \bullet, \ldots, e) \mid \bullet.f \mid e.f := \bullet \mid \bullet.f := v$
		|	let $x : \tau = \bullet$ in e | if $\bullet$ then e else e
		|	$e.m(v, \ldots, \bullet, \ldots, e) \mid \bullet$.monitorenter | $\bullet$.monitorexit
Threads	T	::=	$c\langle r, \text{start}\rangle \mid c\langle r, e\rangle \mid (T \parallel T) \mid \mathbf{0}$
Object	o	::=	$\langle C, \overrightarrow{f \mapsto v}\rangle \mid \langle C, \overrightarrow{f \mapsto v}, \text{started}\rangle \mid \langle C, \overrightarrow{f \mapsto v}, \text{spawned}\rangle$
		|	$\langle C, \overrightarrow{f \mapsto v}, \text{finished}\rangle \mid \langle C, \overrightarrow{f \mapsto v}, \text{interrupted}\rangle$
Heap	$\mathscr{H}$	$\doteq$	$\overrightarrow{r \mapsto (o, l)}$
Object Cache	$\mathscr{C}$	$\doteq$	$\overrightarrow{r \mapsto o}$
Write Buffer	$\mathscr{D}$	$\doteq$	$\overrightarrow{r.f \mapsto v}$
Cache per Core	$\vec{\mathscr{C}}$	$\doteq$	$\overrightarrow{c \mapsto \mathscr{C}}$
Buffer per Core	$\vec{\mathscr{D}}$	$\doteq$	$\overrightarrow{c \mapsto \mathscr{D}}$
Lock State	l	::=	$0 \mid r(n)$
Transition Labels	$\mathscr{L}$	::=	$\mathbf{0} \mid c \mapsto () \mid c \mapsto \alpha \mid \mathscr{L}; \mathscr{L}$
Actions	α	$\doteq$	$\langle r, k, f, n\rangle$
Action Kinds	k	::=	$R \mid W \mid In \mid Vr \mid Vw \mid L \mid U \mid St \mid Fi$
		|	$Ir \mid Ird \mid Sp \mid J \mid Ex \mid F \mid B \mid Iv \mid M$

the unit value $()$. A method is defined as $m(\overrightarrow{x : \tau})\{\text{return } e;\} : \tau$ 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 supports 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. The syntax also supports conditional expressions in the form of if e then e else e; method calls as $e.m(\vec{e})$, where m is the method name.

Additionally, the syntax supports the monitor enter and exit actions through the expressions e.monitorenter and e.monitorexit, respectively. Note that volatile accesses do not have separate bytecodes in Java, they appear as normal memory accesses that the

JVM checks at runtime whether they are volatile or not, thus we do not provide special expressions for volatile accesses.

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 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 $\|$.

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

The memory of the system is split in the Heap $\mathscr{H}$, the object cache $\mathscr{C}$, the write buffer $\mathscr{D}$, the object cache per core $\vec{\mathscr{C}}$, and the write buffer per core $\vec{\mathscr{D}}$. 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 $\mathscr{C}$. Similarly, the write buffer per core is a map from core ids c to write buffers $\mathscr{D}$.

To model mutual exclusion we 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.

In DJC's operational semantics we also use a *Labelled Transition System (LTS)* [75, §2.2] to associate steps with the corresponding cores performing the step and the corresponding JMM/JDMM actions. We use $\xrightarrow{c\mapsto\alpha}$ to show that the step is performed by core c and performs the JMM/JDMM action α. For steps that do not correspond to a JMM/JDMM action we use $\xrightarrow{c\mapsto()}$. We use this information in Appendix B to argue about the adherence of the operational semantics to JDMM.

JMM/JDMM actions are expressed as tuples $\langle r, k, f, n\rangle$, where r is the reference to the object the action acts on, k is the action kind (see Table 3.1), f is the field the action acts on, and n is a unique ID for the action.

6.1.2. Operational Semantics

The operational semantics of DJC are based on those introduced by Johnsen et al. [48]. 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 include `java.util.concurrent`, a Java library introducing more synchronization mechanisms, in our formalization since its interference with JMM is not yet fully defined.

Table 6.2.: Definition of Notation

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`

In Table 6.2 we present the summary of the notations, along with their definitions, that we use in the operational semantics of DJC. We discuss these definitions in more detail, where they appear, as we present the operational semantics. To improve readability, we split the operational semantics in four categories; the *fundamental* operational semantics, regarding the core language; the *synchronization* operational semantics, regarding volatile accesses, monitor handling, join, and interrupts; the operational semantics for *implicit operations* performed by the JVM; and the *global* operational semantics, regarding parallel execution.

The *fundamental* operational semantics of DJC, are presented in Figure 6.1. Following the notation of Johnsen et al., the local configurations are of the form $\mathscr{H};\mathscr{C};\mathscr{D} \vdash e$. 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 6.3 and Figure 6.4. We only use them to argue about the global operational semantics in Figure 6.5. 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. Structural rule Let handles substitution in the standard manner.

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

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

$$[\textsc{IfTrue}]\ \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, \text{if true then } e_1 \text{ else } e_2\rangle \xrightarrow{c\mapsto()} \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e_1\rangle$$

$$[\textsc{IfFalse}]\ \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, \text{if false then } e_1 \text{ else } e_2\rangle \xrightarrow{c\mapsto()} \mathcal{H};\mathcal{C};\mathcal{D} \vdash c\langle r_t, e_2\rangle$$

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

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

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

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

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

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

Figure 6.1.: Semantics of Local Operations

Structural 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 $m(\overrightarrow{x : \tau})\{\text{return } e;\}$, 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 Java, `this` is used to refer to the object owning the method, and it is usually passed, in an implicit manner, as the first argument of each non-static method.

In our implementation all memory accesses first go through the write buffer and if they miss 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. Structural rule FIELD handles non-volatile field accesses, when the field is cached in the object cache, while FIELDDIRTY handles non-volatile field accesses, when the field is cached in the write buffer.

In FIELD the first premise requires that the object, that the field being accessed belongs to, 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 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 $\mathscr{H}$, $\mathscr{C}$, and $\mathscr{D}$ as maps X, we use $X(k)$ to get the value of the cached object or field with key k. We also use $\mathscr{C}(r.f) = v$ as a shorter version of $\mathscr{C}(r) = \langle C, f'_1 \mapsto v'_1, \dots, f \mapsto v, \dots, f'_n \mapsto v'_n \rangle$ to show that f maps to v in the object returned by $\mathscr{C}(r)$. Additionally, we use $dom\,(X)$ to get all the map keys, i.e., references in the case of $\mathscr{H}$ and $\mathscr{C}$ or field names in the case of $\mathscr{D}$.

In a similar manner 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.

Structural 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 $k \notin dom\left(X'\right)$ and $X' \subseteq X$. Note that we use $\subseteq$ instead of $\subset$, since k might not be in the map in the first place.

Structural rule NEW returns a $fresh$ reference to an object $\langle C, \vec{v} \rangle$, and adds it to $\mathscr{H}$. Similarly to Johnsen et al., we use $C\left(\vec{v}\right)$ for instances of class C with field values $\vec{v}$, i.e., field f_i contains the value v_i. Note that according to JMM "*conceptually every object is created at the start of the program*" [72, §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 returns a reference to it. We use $r - fresh$ to show that there is no other reference to that object already.

In Figure 6.2 we present the operational semantics for implicit operations. These are operations performed implicitly by the virtual machine and do not map to language expressions. Structural 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 $\mathscr{C}$. Writing back a field $r.f$ requires that the object

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

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

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

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

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

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

Figure 6.2.: Operational Semantics for Implicit Operations

referenced by r is present in the heap $\mathscr{H}$ and the object cache $\mathscr{C}$, $r.f$ is not volatile, and there is a dirty copy of it in the write buffer $\mathscr{D}$. Writing-back a field results in the update of its value both in the heap $\mathscr{H}$ and the object cache $\mathscr{C}$. 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, $\mathscr{C}$, of the core executing the invalidation. Structural 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.

Structural 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.

In Figure 6.3 we present the operational semantics for volatile accesses.

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

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

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

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

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

Figure 6.3.: Semantics of Volatile Accesses

Structural rules VolatileReadL and VolatileRead handle volatile reads. Structural rules VolatileWriteL and VolatileWrite handle volatile writes. Note that 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 we describe in Section 5.3.5. 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. Note that we use $\emptyset$ for empty maps.

In Figure 6.4 we present the *synchronization* operational semantics. That is, rules about monitor handling, join, and interrupts.

Structural rules MonitorEnter and NestedMonitorEnter handle monitor acquisition, and MonitorExit and NestedMonitorExit handle monitor release. In these rules we use $r.l$

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

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

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

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

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

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

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

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

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

Figure 6.4.: Semantics of Synchornization Operations

—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 by Johnsen et al. [48]. 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.

Structural rule MonitorEnter requires that a monitor must be free before its acquisition. Structural rule NestedMonitorEnter requires that a monitor is already owned by some thread before it gets re-entered by that same thread. Structural rules MonitorExit and NestedMonitorExit ensure that a monitor is released only by its owner and the same number of times it was previously acquired.

Nested Monitor Acquisition: 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.

Structural 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.

Structural 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.

Structural rules InterruptedT and InterruptedF handle `Thread.interrupted()` invocations. 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.

$rinterrupted\, f$ 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 invalidating the object cache or the write buffer.

$$\boxed{\mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash T \xrightarrow{\mathcal{L}} \mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash T}$$

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

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

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

$$[\textsc{Blocked}]\ \frac{}{\mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash T_1 \xrightarrow{0} \mathcal{H};\vec{\mathcal{C}};\vec{\mathcal{D}} \vdash T_1}$$

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

Figure 6.5.: Global Operational Semantics

In Figure 6.5 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 both global and local configurations since it is shared between all cores.

Structural 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.

Structural 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. This is represented 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 spawned to mark that this thread has been spawned and forbid re-spawns of it.

Structural 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'.

Structural rule BLOCKED is essentially a no-op that allows threads to block and not step in every transitions in an execution trace, e.g., a thread joining on another thread that is still running.

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 $\vec{\mathscr{C}}$ and $\vec{\mathscr{D}}$. 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 $\vec{\mathscr{C}} \downarrow \vec{c}$ (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.

6.2. Proof Sketch

This section briefly describes the proof sketch of DJC's adherence to the JDMM. For the detailed proof sketch please refer to Appendix A and Appendix B. To improve readability we use S as an abbreviation for a global configuration, $\mathscr{H}; \vec{\mathscr{C}}; \vec{\mathscr{D}} \vdash T$. 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 $S \rightarrow^* S'$ where $\rightarrow^*$ denotes a sequence of steps, i.e., $S \rightarrow^* S' \equiv S \xrightarrow{\mathscr{L}_1} S_1 \xrightarrow{\mathscr{L}_2} S_2 \ldots S_n \xrightarrow{\mathscr{L}_n} S'$, we show that the trace after taking one more step, i.e., $S \rightarrow^* S_n \xrightarrow{\mathscr{L}_{n+1}} S_{n+1}$, is well-formed as well.

This amounts to a preservation proof for each rule. 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.

Chapter 7.

Related work

7.1. Memory Models

To the best of our knowledge JDMM is the first formalization of the JMM for non-cache-coherent and distributed memories. JDMM is a pure extension of the JMM, which means that all properties of the JMM are also properties of JDMM and all valid JMM reorderings are also valid JDMM reorderings.

Hoefler et al. [38] follow a similar approach to ours in the definition of the semi-formal semantics of the MPI remote memory access (RMA). They use the JMM notation to argue about the ordering of actions in an MPI execution, and algebraic formalization to provide consistency guarantees to the MPI-RMA users. The MPI-RMA specification relates to our work in that it targets similar architectures. Hoefler et al. work shows when MPI users should use synchronization or communication primitives provided by MPI-RMA to ensure data consistency as needed by their program. Similarly, in JDMM we show when JVM implementers should perform explicit memory transfers to ensure that their JVM adheres to JMM. The main difference between the two works is that Hoefler et al. define the semantics of an API targeting non-coherent memory architectures and how it should be used to produce valid programs, while our work defines how data need to be explicitly transferred by the JVM so that executions of Java programs on that JVM will adhere to JMM. Combined together the formalization of JDMM and MPI-RMA can be used to argue about the implementation of a JVM relying on MPI-RMA to perform the data transfers, instead of the hardware primitives.

Regarding the formalization of memory models through a language operational semantics, previous work describes the memory semantics for shared memory multicore processor architectures, such as Power [69], x86 [79, 85], and ARM [2] processors, without focusing on a specific language semantics or memory model. Sarkar et al. [84] 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. [82] 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 [18] define a relaxed memory model using an operational semantics for the Core ML language (without typing). Their work takes into account write buffers that must become empty before a lock release. That is, they treat lock releases as memory barriers. Although the handling of write buffers is similar to handling caches regarding the write backs, the fetching and invalidation handling part is not covered by their work. Additionally, Boudol and Petri only consider lock releases as synchronization points, while in the Java language, according to the JMM, there are multiple synchronization points.

Joshi and Prasad [49], based on Boudol's and Petri's work, define an operational semantics that accounts for caches. Their work fills the gap about update and invalidation cache operations, left from Boudol's and Petri's work. In their work they 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 works 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 builds on top of existing work.

Demange et al. [27] 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 not present in the architectures at hand.

Jagadeesan et al. [45] also describe an operational semantics for the Java Memory Model. This work, however, does not account for caches or buffers. They abstract away the hardware details and consider 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 (even temporarily) after any evaluation step. We thus consider our approach to be closer to the implementation.

Cenciarelli et al. [21] use a combination of operational, denotational, and axiomatic semantics to define JMM. In their work they show that all their generated executions adhere to JMM, but similarly to Jagadeesan et al. [45] they do not account for the memory hierarchy.

DJC's definition is based on the concurrent object-oriented calculus definition introduced by Johnsen et al. [48]. The way DJC handles monitors is inspired by the way Johnsen et al. handle re-entrant locks in their calculus. Johnsen et al. focus on the prevention of lock errors through a static type and effect system. DJC aims to prevent false cache

management in JVM implementations targeting architectures with non-cache-coherent memories.

7.2. Software Caching

Lee et al. [60] propose the *centralized release consistency (CRC)* model where the PPE is responsible for handling the pages and the locks to ensure memory coherence. Such centralized scenarios are not expected to scale with the number of cores. To tackle this issue Lee et al. [59] propose the *hierarchical centralized release consistency (HCRC)*. HCREC is essentially an extension of CRC, where each page is assigned to a system node and in each system node there is one processing element responsible for handling the pages and the locks to ensure memory coherence.

Balart et al. [9] present a compiler-assisted software cache implementation that allows for outstanding DMA transfers of write-back and fetches, in an effort to overlap communication with computation. The cache operates on cache-line granularity and employs reference counting to detect whether a cache line can be evicted or not. The proposed software cache, fetches data at cache misses and write-backs data when the reference counter of a cache line becomes zero. The software cache presented in this thesis differs in that it caches objects in contiguous memory locations, this way we are able to reduce the amount of lookups to one per object access instead of one per field access.

Seo et al. [88] perform an extensive evaluation of four different software caches with three different replacement policies each, first-in-first-out (FIFO), Clock, and least-recently-used (LRU). Namely, the four software caches are the fully associative cache (FAC), 4-way set associative cache (4WC), Indirect Index Cache (IIC), and Extended Set-Index Cache (ESC). Their evaluation examines cache-line sizes in the range of [1KiB – 8KiB] and the results show that the smaller the cache-line size, the larger the overhead.

Gonzàlez et al. [35] propose a compiler assisted hybrid approach to software caching. The compiler separates memory references, according to their access patterns, in two classes, *high-locality* and *irregular*. Each class is cached in a different cache. The *High-Locality Cache* utilizes a reference counting algorithm to pin cache lines and increase the hit ratio. The *Transactional Cache*, used for the irregular references, is a write-through cache that reduces hit and miss overheads by utilizing SIMD compares. Memory consistency is provided through the *Memory Consistency Block*, which handles all the data transfers two and from the caches. To improve the cache performance, Gonzàlez et al. also perform compile-time code transformations in loops according to the memory access classification.

The software cache presented in this thesis differs from the above in that it operates on object level granularity, and avoids fragmentation. The object level granularity, allows us to fetch all data of an object in contiguous memory and avoid looking up the software

cache for each field access. On the contrary, we perform a single lookup per object, and the reusing the value form the Java stack we are able to access the object's fields by adding the corresponding offset to the object's address in the software cache. Furthermore, the lack of fragmentation allows us to fully utilize the limited memory resources in Formic-Cube. On the other hand, our policies regarding capacity misses may result in unnecessary invalidations of data. Our approach is fully distributed and does not rely on processing units to handle the data and resolve conflicts in architectures that support write-backs of finer than the cache-line size granularity.

7.3. Java Virtual Machines

To the best of our knowledge, the only other JVM adhering to JMM [72] on a non-cache-coherent architecture is Hera-JVM [73]. 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 Section 5.3. 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 [39, 66].

On a broader scope, however, most of the JVMs presented in Section 2.2 are also related to this work, since as we state in Section 4.1 they help us spot the key challenges faced when designing a JVM for hundreds of non-cache-coherent cores. Additionally, some of the JVMs discussed in Section 2.2 provided the foundations for the algorithms we present in Section 4.2 and later implement in Chapter 5.

Chapter 8.

Conclusions

This thesis aims to help future Java Virtual Machine (JVM) implementers better understand the Java Memory Model (JMM) and the implications of its implementation in JVMs targeting incoherent memory architectures.

To achieve this we first introduce the Java Distributed Memory Model (JDMM), an extension to the Java Memory Model aiming to bridge the gap between the Java Memory Model (JMM) specification and the memory management mechanisms provided by future processor architectures that lack memory-coherency. JDMM exposes the memory management mechanisms, easing the process of implementing new Java Virtual Machines (JVMs), or porting existing ones, on such architectures. JDMM is a pure extension of the JMM, which means that all properties of the JMM are also properties of JDMM's and all valid JMM re-orderings are also valid JDMM re-orderings. To the best of our knowledge, JDMM is the first formalization of the JMM for non-cache-coherent and distributed memories. Using JDMM, we were able to verify that Hera-JVM, a JVM implementation for a distributed memory architecture, adheres to the JMM. We were also able to detect a redundant invalidation of the software cache in the case of context switching, which could potentially have a negative impact on the performance, as well as, the energy consumption of Hera-JVM. This result increases our confidence that JDMM can benefit JVM designers and developers in the future.

We propose novel algorithms for software caching of Java objects, and synchronization management, as defined by the Java Memory Model. The algorithms presented in this thesis adhere to the Java Memory Model and take advantage of coherent-islands, groups of coherent cores in an otherwise incoherent processor, to improve performance and reduce energy consumption. The software caching algorithms provide a memory access layer hiding any complications of the underlying hardware, while the synchronization management algorithms ensure mutual exclusion for threads synchronizing through Java monitors. Furthermore, our algorithms ensure proper ordering between threads that synchronize in a point-to-point manner without the use of Java monitors.

We design and build DiSquawk, a proof-of-concept Java virtual machine targeting the Formic-Cube non-cache-coherent, 512-core, architecture. We also implement the inter-coherent-island part of our algorithms in DiSquawk. In DiSquawk, each core runs an instance of Squawk, an interpreter-based JVM. Those instances implicitly communicate

with each other and exchange data, to provide a single system image to the Java application. We evaluate DiSquawk using a set of benchmarks and micro-benchmarks. The micro-benchmarks help us better understand the behavior and performance of specific mechanisms of DiSquawk, while the benchmarks help us measure its overall scalability. The results show that DiSquawk scales with the number of cores in a similar manner to the state-of-the-art, HotSpot JVM.

Finally, to show that our implementation adheres to JMM, we define DFJ, a Java core calculus that models DiSquawk in its operational semantics. Then, based on this operational semantics, we prove the adherence of our implementation and the proposed algorithms to the Java Distributed Memory Model and thus to the original Java Memory Model.

8.1. Further Work & Open Research Problems

During the models' definitions and the software design and implementation phases we have identified a few aspects of our work that could be improved. We discuss these aspects bellow.

8.1.1. Machine-Checked Proofs

Although we formally define our models and present proof regarding the claims we make, machine-checked proofs would significantly increase our confidence and allow the community to advance our work with less effort. There are currently two dominant *proof assistants* that provide formal languages to write mathematical definitions, Coq and Isabelle [44, 95]. The formal languages provided by these tools is intuitive to users familiar with language operational semantics. However, the modeling of various aspects of a system (*e.g.*, memory) in these formal languages is far from trivial. Lochbihler [63] has put some significant effort on mechanically checking the Java Memory Model and a number of fixes to some flows of it. We believe that a similar work for JDMM and DFJ would significantly strengthen our contributions.

8.1.2. Evaluation on Non-emulated Architectures With Coherent-islands

As we discuss in Section 5.1, the evaluation of our implementation on an emulated non-cache-coherent architecture imposes a number of limitations on both the implementation of our algorithms as well as their evaluation. A non-emulated architecture with coherent-islands, such as EUROSERVER [29], would allow us to fully implement and evaluate the algorithms proposed in Chapter 4. That way we could better understand the ratios of

the overheads imposed by the inter-coherent and intra-coherent parts of our algorithms and possibly improve our algorithms.

8.1.3. State-of-the-art Core VM

In this thesis, as discussed in Section 5.2 due to the need to run on the bare metal we choose the Squawk VM as our base VM and built on top of it. However, Squawk VM is not a state-of-the-art VM and lacks many optimizations that are present in other JVMs. Such an optimization is the Just In Time (JIT) compilation, that allows the JVM to optimize performance by translating commonly used sequences of Java bytecodes to native machine code. JIT compilation can significantly lower the execution time of a program and thus affect the communication over computation ratio. Additionally, as we discuss in Section 4.2.1, JIT compilation can be used to fine-tune the write-buffer on-the-fly, according to the behavior of different code segments.

8.1.4. Garbage Collection

In this thesis we do not cover the aspect of garbage collection. Garbage collection, as discussed in Chapter 4, is a major challenge when implementing Java on incoherent memory architectures. A garbage collector targeting a JVM running on such architectures needs to distribute the overhead of garbage collection, and avoid long pauses by reducing the execution time of stop-the-world phases or even eliminating them. Stop-the-world phases are phases where all application threads halt and only garbage collection threads run on the system. During this phases the application makes no progress, which is prohibiting in real-time applications. Although there is a lot of work on parallelizing garbage collection in the literature, these efforts focus on coherent memory architectures. The task of designing garbage collectors that run efficiently on incoherent memory architectures is not trivial and remains an open problem to work on.

8.1.5. `java.util.concurrent`

The `java.util.concurrent` package offers a number of fine-grained concurrent data structures and abstractions that perform well on memory coherent architectures. However, the implementation of this package using locks does not perform similar to the corresponding implementation using atomic operations. Many of the data structures and abstractions that rely on atomic operations perform badly on incoherent memory architectures. As a result, there is the need to modify this package or provide an alternative one that will provide similar data structures and abstractions that perform better on incoherent memory architectures. The *GreenVM* project has made the first steps in this

direction by designing, implementing, and evaluating some distributed data structures for that cause.

8.1.6. Java Memory Model Update

Although the memory model of a programming language needs to be architecture independent, it still needs to expose details related to the state-of-the-art hardware mechanisms to be more intuitive. As a result, as the hardware design advances and new hardware primitives are introduced (or deprecated) memory models need to adapt in order for their definitions and rules to better relate to their implementations using hardware-primitives. At the time of writing there is an ongoing discussion about the improvement of the Java Memory Model [47]. The main targets of this project are:

- The improvement of JMM's formalization to make it machine checkable, as well as, more human readable.
- The fix of existing errors as reported by Aspinall and Ševčík and Torlak et al. [97].
- The coverage of JVM related aspects, like class initialization. Currently JMM focuses on the Java programming language and not to its bytecode. This results in ambiguous definitions about some JVM operations and the use of the Java bytecode by other languages.
- The coverage of `java.util.concurrent`'s parts, as well as, any extensions that my arise from forthcoming JDK Enhancement Proposals (JEPs).
- To provide compatibility with the C11 and C++11 standards, aiming to provide a consistent behavior across Java and C/C++ native libraries.
- To provide a technical document to guide JVM implementors, JDK library developers, and developers, explaining how JMM impacts particular problems and solutions.
- To provide tests for conformance to JMM.
- To provide an interface or hints for analysis tools that check for data-races and security properties across multiple threads.

Bibliography

[1] Umut A. Acar, Arthur Chargueraud, and Mike Rainey. “Scheduling Parallel Programs by Work Stealing with Private Deques”. In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’13. Shenzhen, China: ACM, 2013, pages 219–228. ISBN: 978-1-4503-1922-5. DOI: 10.1145/2442516.2442538. URL: http://doi.acm.org/10.1145/2442516.2442538 (cited on page 89).

[2] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. “The Semantics of Power and ARM Multiprocessor Machine Code”. In: *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*. DAMP ’09. New York, NY, USA: ACM, 2008, pages 13–24 (cited on page 129).

[3] B Alpern, S Augart, and SM Blackburn. “The Jikes research virtual machine project: building an open-source research community”. In: *IBM Systems …* 44.2 (2005), pages 399–417. URL: http://ieeexplore.ieee.org/xpls/abs%5C_all.jsp?arnumber=5386722 (cited on page 94).

[4] Gabriel Antoniu, Luc Bougé, Philip J. Hatcher, Mark MacBeth, Keith McGuigan, and Raymond Namyst. “The Hyperion system: Compiling multithreaded Java bytecode for distributed execution”. In: *Parallel Computing* 27.10 (2001), pages 1279–1297. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(01)00093-X. URL: http://dx.doi.org/10.1016/S0167-8191(01)00093-X (cited on pages 12, 21, 72).

[5] Yariv Aridor, Michael Factor, and Avi Teperman. “cJVM: A Single System Image of a JVM on a Cluster”. In: *Proceedings of the International Conference on Parallel Processing*. ICPP. Wakamatsu, Japan: IEEE Computer Society, Sept. 1999, pages 4–11. ISBN: 0-7695-0350-0. DOI: 10.1109/ICPP.1999.797382 (cited on pages 3, 13, 21).

[6] David Aspinall and Jaroslav Ševčík. “Formalising Java’s Data Race Free Guarantee”. In: *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*. TPHOLs. Springer Berlin Heidelberg, 2007, pages 22–37 (cited on page 22).

[7] David Aspinall and Jaroslav Ševčík. “Java Memory Model Examples: Good, Bad and Ugly”. In: *1st International Workshop on Verification and Analysis of Multi-threaded Java-like Programs*. VAMP. 2007 (cited on pages 22, 65–67, 136).

[8] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”. In: *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Euro-Par ’09. Berlin, Heidelberg: Springer-Verlag, 2009, pages 863–874. ISBN: 978-3-642-03868-6. DOI: http://dx.doi.org/10.1007/978-3-642-03869-3_80. URL: http://dx.doi.org/10.1007/978-3-642-03869-3%5C_80 (cited on page 76).

[9] Jairo Balart, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, Zehra Sura, Tong Chen, Tao Zhang, Kevin O’brien, and Kathryn O’brien. “A novel asynchronous software cache implementation for the cell-be processor”. In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2007, pages 125–140 (cited on page 131).

[10] Nick Benton, Andrew Kennedy, and George Russell. “Compiling standard ML to Java bytecodes”. In: *ACM SIGPLAN Notices* 34.1 (1999), pages 129–140 (cited on page 11).

[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: *PACT’ 08*. 2008 (cited on page 109).

[12] Stephen M S.M. Stephen M. Blackburn, Perry Cheng, and Kathryn S K.S. Kathryn S. McKinley. “Oil and water? High performance garbage collection in Java with MMTk”. In: *Proceedings. 26th International Conference on Software Engineering* (May 2004), pages 137–146. DOI: 10.1109/ICSE.2004.1317436. URL: http://dl.acm.org/citation.cfm?id=998675.999420%20http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1317436 (cited on page 74).

[13] Stephen M. Blackburn, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Robin Garner, Daniel von Dincklage, Ben Wiedermann, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, and Daniel Frampton. “The DaCapo benchmarks”. In: *ACM SIGPLAN Notices* 41.10 (Oct. 2006), page 169. ISSN: 03621340. DOI: 10.1145/1167515.1167488. URL: http://dl.acm.org/citation.cfm?id=1167515.1167488 (cited on pages 82, 99).

[14] Filip Blagojevic, Dimitris S Nikolopoulos, Alexandros Stamatakis, and Christos D Antonopoulos. “Dynamic multigrain parallelization on the cell broadband engine”. In: *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP ’07. New York, NY, USA: ACM, 2007, pages 90–100. ISBN: 978-1-59593-602-8. DOI: http://doi.acm.org/10.1145/1229428.1229445. URL: http://doi.acm.org/10.1145/1229428.1229445 (cited on page 76).

[15] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, Yuli Zhou, E Leiserson, C Kuszmaul, and H Randall. “Cilk: An efficient multithreaded runtime system”. In: *PPOPP*. 1995, pages 207–216 (cited on page 76).

[16] Robert D. Blumofe and Charles E. Leiserson. “Scheduling Multithreaded Computations by Work Stealing”. In: *J. ACM* 46.5 (Sept. 1999), pages 720–748. ISSN: 0004-5411. DOI: 10.1145/324133.324234. URL: http://doi.acm.org/10.1145/324133.324234 (cited on page 76).

[17] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. “A Type and Effect System for Deterministic Parallel Java”. In: *OOPSLA ’09*. OOPSLA ’09. Orlando, Florida, USA: ACM, 2009, pages 97–116. ISBN: 978-1-60558-766-0. DOI: 10.1145/1640089.1640097. URL: http://doi.acm.org/10.1145/1640089.1640097 (cited on page 77).

[18] Gérard Boudol and Gustavo Petri. “Relaxed Memory Models: An Operational Approach”. In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’09. Savannah, GA, USA: ACM, 2009, pages 392–403. ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480930. URL: http://doi.acm.org/10.1145/1480881.1480930 (cited on page 130).

[19] John B. Carter, John K. Bennett, and Willy Zwaenepoel. “Implementation and Performance of Munin”. In: *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. SOSP ’91. Pacific Grove, California, USA: ACM, 1991, pages 152–164. ISBN: 0-89791-447-3. DOI: 10.1145/121132.121159. URL: http://doi.acm.org/10.1145/121132.121159 (cited on page 73).

[20] Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua B. Fryman, Ivan Ganev, Roger A. Golliver, Rob C. Knauerhase, Richard Lethin, Benoît Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, and Jianping Xu. “Runnemede: An architecture for Ubiquitous High-Performance Computing.” In: *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture*. HPCA. IEEE Computer Society, 2013, pages 198–209 (cited on pages 2, 4, 7, 9, 17, 32).

[21] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. “The Java Memory Model: Operationally, Denotationally, Axiomatically”. In: *Proceedings of the 16th European Symposium on Programming*. ESOP. Springer Berlin Heidelberg, 2007, pages 331–346. URL: http://dx.doi.org/10.1007/978-3-540-71316-6_23 (cited on pages 22, 130).

[22] David Chase and Yossi Lev. “Dynamic Circular Work-stealing Deque”. In: *SPAA ’05*. SPAA ’05. Las Vegas, Nevada, USA: ACM, 2005, pages 21–28. ISBN: 1-

58113-986-1. DOI: 10.1145/1073970.1073974. URL: http://doi.acm.org/10.1145/1073970.1073974 (cited on page 76).

[23] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. “Cell Broadband Engine Architecture and Its First Implementation-A Performance View”. In: *IBM Journal of Research and Development* 51.5 (2007), pages 559–572. DOI: 10.1147/rd.515.0559. URL: http://dx.doi.org/10.1147/rd.515.0559 (cited on page 14).

[24] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. “DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism”. In: *2011 International Conference on Parallel Architectures and Compilation Techniques* (Oct. 2011), pages 155–166. DOI: 10.1109/PACT.2011.21. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6113797 (cited on pages 2, 7).

[25] The UPC Consortium. *UPC language specification v1.3*. 2005. URL: https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf (cited on page 19).

[26] David A. Roberts. *LLJVM*. http://da.vidr.cc/projects/lljvm/. [Online; accessed 15-Apr-2013]. 2009 (cited on page 11).

[27] Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. “Plan B: A Buffered Memory Model for Java”. In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’13. Rome, Italy: ACM, 2013, pages 329–342. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429110. URL: http://doi.acm.org/10.1145/2429069.2429110 (cited on page 130).

[28] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. “Scalable Work Stealing”. In: *SC ’09*. SC ’09. Portland, Oregon: ACM, 2009, 53:1–53:11. ISBN: 978-1-60558-744-8. DOI: 10.1145/1654059.1654113. URL: http://doi.acm.org/10.1145/1654059.1654113 (cited on pages 76, 89).

[29] Y. Durand, P.M. 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: *17th Euromicro Conference on Digital System Design*. DSD. Aug. 2014, pages 206–213. DOI: 10.1109/DSD.2014.15 (cited on pages 2, 4, 9, 17, 134).

[30] Yong hun Eom, Stephen Yang, James C. Jenista, and Brian Demsky. “DOJ: Dynamically Parallelizing Object-Oriented Programs”. In: *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming - PPoPP ’12*. New York, New York, USA: ACM Press, Feb. 2012, page 85. ISBN: 9781450311601. DOI: 10.1145/2145816.2145828. URL: http://dl.acm.org/citation.cfm?id=2145816.2145828 (cited on page 77).

[31] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. “Dark Silicon and the End of Multicore Scaling”. In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA ’11. San Jose, California, USA: ACM, 2011, pages 365–376. ISBN: 978-1-4503-0472-6. DOI: 10.1145/2000064.2000108. URL: http://doi.acm.org/10.1145/2000064.2000108 (cited on pages 1, 7).

[32] Michael Factor, Assaf Schuster, and Konstantin Shagin. “JavaSplit: a runtime for execution of monolithic Java programs on heterogenous collections of commodity workstations”. In: *Proceedings of the International Conference on Cluster Computing*. CLUSTER. 2003, pages 110–117. DOI: 10.1109/CLUSTR.2003.1253306 (cited on pages 21, 72).

[33] Foivos S. Zakkak and Polyvios Pratikakis. “Building a Java Virtual Machine for Non-Cache-Coherent Many-core Architectures”. In: *Proceedings of the 14th International Workshop on Java Technologies for Real-Time and Embedded Systems*. JTRES ’16. Lugano, Switzerland: ACM, 2016. ISBN: 978-1-4503-4800-3. DOI: 10.1145/2990509.2990510. URL: http://dx.doi.org/10.1145/2990509.2990510 (cited on pages 71, 93).

[34] Foivos S. Zakkak and Polyvios Pratikakis. “DiSquawk: 512 Cores, 512 Memories, 1 JVM”. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ ’16. Lugano, Switzerland: ACM, 2016, 2:1–2:12. ISBN: 978-1-4503-4135-6. DOI: 10.1145/2972206.2972212. URL: http://doi.acm.org/10.1145/2972206.2972212 (cited on page 115).

[35] Marc Gonzàlez, Nikola Vujic, Xavier Martorell, Eduard Ayguadé, Alexandre E Eichenberger, Tong Chen, Zehra Sura, Tao Zhang, Kevin O’Brien, and Kathryn O’Brien. “Hybrid access-specific software cache techniques for the cell BE architecture”. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM. 2008, pages 292–302 (cited on page 131).

[36] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, 3rd Edition*. Addison-Wesley Professional, 2005. ISBN: 0321246780 (cited on page 22).

[37] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java(TM) Language Specification, Java SE 8 Edition*. 2015 (cited on page 102).

[38] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. “Remote Memory Access Programming in MPI-3”. In: *ACM Trans. Parallel Comput.* 2.2 (June 2015), 9:1–9:26. ISSN: 2329-4949. DOI: 10.1145/2780584. URL: http://doi.acm.org/10.1145/2780584 (cited on page 129).

[39] 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: *Proceedings of the International Solid-State Circuits Conference*. ISSCC. 2010, pages 108–109. DOI: 10.1109/ISSCC.2010.5434077 (cited on pages 2, 32, 76, 101, 132).

[40] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. “The Garbage Collection Advantage : Improving Program Locality”. In: *ACM SIGPLAN Notices* 39.10 (Oct. 2004), page 69. ISSN: 03621340. DOI: 10.1145/1035292.1028983. URL: http://dl.acm.org/citation.cfm?id=1035292.1028983 (cited on page 84).

[41] Jim Hugunin. “Python and Java: The best of both worlds”. In: *Proceedings of the 6th international Python conference* (1997). URL: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.1100%20http://www.hugunin.net/papers/hugunin97python.pdf (cited on page 11).

[42] Marieke Huisman and Gustavo Petri. “The Java Memory Model: a Formal Explanation”. In: *1st International Workshop on Verification and Analysis of Multi-threaded Java-like Programs*. VAMP. 2007, pages 81–96 (cited on page 22).

[43] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. “Featherweight Java: A Minimal Core Calculus for Java and GJ”. In: *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA ’99. Denver, Colorado, USA: ACM, 1999, pages 132–146. ISBN: 1-58113-238-7. DOI: 10.1145/320384.320395. URL: http://doi.acm.org/10.1145/320384.320395 (cited on pages 5, 115).

[44] *Isabelle*. URL, https://isabelle.in.tum.de/. URL: https://isabelle.in.tum.de/ (cited on page 134).

[45] Radha Jagadeesan, Corin Pitcher, and James Riely. “Generative Operational Semantics for Relaxed Memory Models”. In: *Proceedings of the 19th European Symposium on Programming*. ESOP. Springer, 2010, pages 307–326 (cited on pages 22, 130).

[46] James Christopher Jenista, Yong hun Eom, and Brian Charles Demsky. “OoOJava: Software Out-of-order Execution”. In: *PPoPP ’11*. PPoPP ’11. San Antonio, TX, USA: ACM, 2011, pages 57–68. ISBN: 978-1-4503-0119-0. DOI: 10.1145/1941553.1941563. URL: http://doi.acm.org/10.1145/1941553.1941563 (cited on page 77).

[47] *JEP-188: Java™Memory Model Update*. URL: http://openjdk.java.net/jeps/188 (cited on pages 22, 136).

[48] Einar Broch Johnsen, Thi Mai Thuong Tran, Olaf Owe, and Martin Steffen. “Safe locking for multi-threaded Java with exceptions”. In: *The Journal of Logic and Algebraic Programming* 81.3 (2012). The 22nd Nordic Workshop on Programming Theory (NWPT) 2010, pages 257–283. ISSN: 1567-8326. DOI: http://dx.doi.org/10.1016/j.jlap.2011.11.002. URL: http://www.sciencedirect.com/science/article/pii/S1567832611000968 (cited on pages 5, 115, 118, 123, 130).

[49] Salil Joshi and Sanjiva Prasad. “An Operational Model for Multiprocessors with Caches”. English. In: *Theoretical Computer Science*. Edited by CristianS. Calude and Vladimiro Sassone. Volume 323. IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2010, pages 371–385. ISBN: 978-3-642-15239-9. DOI: 10.1007/978-3-642-15240-5_27. URL: http://dx.doi.org/10.1007/978-3-642-15240-5_27 (cited on page 130).

[50] *JSR-133 Java™Memory Model and Thread Specification*. URL: http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf (cited on page 31).

[51] ISO JTC. *SC22/WG14. ISO/IEC 9899: 2011*. 2011. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853 (cited on page 19).

[52] ISO JTC. *SC22/WG21. ISO/IEC 14882:2011*. 2011. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372 (cited on page 19).

[53] Stefanos Kaxiras and Georgios Keramidas. “SARC coherence: Scaling directory cache coherence in performance and power”. In: *Micro, IEEE* (2010), pages 54–65. URL: http://ieeexplore.ieee.org/xpls/abs%5C_all.jsp?arnumber=5582068 (cited on pages 2, 7).

[54] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. “TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems”. In: *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*. WTEC’94. San Francisco, California: USENIX Association, 1994, pages 10–10. URL: http://dl.acm.org/citation.cfm?id=1267074.1267084 (cited on page 73).

[55] Pete Keleher, Alan L Cox, and Willy Zwaenepoel. “Lazy release consistency for software distributed shared memory”. In: ISCA. Volume 20. ISCA. 00876. ACM, 1992 (cited on page 20).

[56] L. Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *Computers, IEEE Transactions on* C-28.9 (1979), pages 690–691. ISSN: 0018-9340. DOI: 10.1109/TC.1979.1675439 (cited on page 20).

[57] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (1978), pages 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563 (cited on pages 26, 72, 154).

[58] Doug Lea. *The JSR-133 cookbook for compiler writers*. 2008 (cited on page 103).

[59] Jaejin Lee, Jun Lee, Sangmin Seo, Jungwon Kim, Seungkyun Kim, and Zehra Sura. “COMIC++: A software SVM system for heterogeneous multicore accelerator clusters”. In: *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE. 2010, pages 1–12 (cited on pages 73, 131).

[60] Jaejin Lee, Sangmin Seo, Chihun Kim, Junghyun Kim, Posung Chun, Zehra Sura, Jungwon Kim, and SangYong Han. “COMIC: A Coherent Shared Memory Interface for Cell Be”. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’08. Toronto, Ontario, Canada: ACM, 2008, pages 303–314. ISBN: 978-1-60558-282-5. DOI: 10.1145/1454115.1454157. URL: http://doi.acm.org/10.1145/1454115.1454157 (cited on pages 73, 131).

[61] Kai Li and Paul Hudak. “Memory Coherence in Shared Virtual Memory Systems”. In: *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*. PODC ’86. Calgary, Alberta, Canada: ACM, 1986, pages 229–239. ISBN: 0-89791-198-9. DOI: 10.1145/10590.10610. URL: http://doi.acm.org/10.1145/10590.10610 (cited on page 73).

[62] Tim Lindholm and Frank Yellin. *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc., 1999 (cited on page 11).

[63] Andreas Lochbihler. “A machine-checked, type-safe model of Java concurrency: language, virtual machine, memory model, and verified compiler”. PhD thesis. Karlsruhe Institute of Technology, 2012. ISBN: 978-3-86644-885-8. URL: http://d-nb.info/1026537800 (cited on page 134).

[64] Andreas Lochbihler. “Java and the Java Memory Model — A Unified, Machine-Checked Formalisation”. In: *Proceedings of the 21th European Symposium on Programming*. ESOP. Springer Berlin Heidelberg, 2012, pages 497–517. DOI: 10.1007/978-3-642-28869-2_25. URL: http://dx.doi.org/10.1007/978-3-642-28869-2_25 (cited on page 24).

[65] Andreas Lochbihler. “Making the Java Memory Model Safe”. In: *ACM Transactions on Programming Languages and Systems*. TOPLAS 35.4 (2014), pages 1–65. ISSN: 0164-0925. DOI: 10.1145/2518191. URL: http://doi.acm.org/10.1145/2518191 (cited on pages 22, 23, 28).

[66] 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: *Proceedings of the 20th Annual International Symposium on Field-Programmable Custom Computing Machines*. FCCM. 2012, pages 61–64. DOI: 10.1109/FCCM.2012.20 (cited on pages 2, 4, 9, 17, 32, 93, 101, 132).

[67] Spyros Lyberis. “Myrmics: A Scalable Runtime System for Global Address Spaces”. PhD thesis. Computer Science Department, University of Crete, 2013 (cited on pages 2, 9, 32).

[68] MW MacBeth, KA McGuigan, and PJ Hatcher. “Executing Java threads in parallel in a distributed-memory environment”. In: *…of the 1998 conference of the …* (1998). URL: http://dl.acm.org/citation.cfm?id=783176 (cited on pages 3, 12, 21).

[69] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. “An Axiomatic Memory Model for POWER Multiprocessors”. In: *Proceedings of the 24th International Conference on Computer Aided Verification*. CAV’12. Berlin, Heidelberg: Springer-Verlag, 2012, pages 495–512 (cited on page 129).

[70] Jeremy Manson. “The Java Memory Model”. PhD thesis. 2004. URL: http://drum.lib.umd.edu/bitstream/1903/1949/1/umi-umd-1898.pdf (cited on pages 14, 15, 19, 22–24, 27, 29, 47, 48, 67–69, 151).

[71] Jeremy Manson, William Pugh, and Sarita Adve. *SPECIAL POPL ISSUE: The Java™Memory Model*. URL: https://dl.dropboxusercontent.com/u/1011627/journal.pdf (cited on pages 31, 69).

[72] Jeremy Manson, William Pugh, and Sarita V. Adve. “The Java Memory Model”. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL. Long Beach, California, USA, Jan. 2005, pages 378–391. ISBN: 1-58113-830-X. DOI: 10.1145/1047659.1040336. URL: http://doi.acm.org/10.1145/1040305.1040336 (cited on pages 11, 14, 15, 19, 22, 24, 30, 31, 54, 69, 120, 132, 154, 156, 157).

[73] Ross McIlroy and Joe Sventek. “Hera-JVM: a runtime system for heterogeneous multi-core architectures”. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA ’10*. Volume 45. 10. New York, New York, USA: ACM Press, Oct. 2010, page 205. ISBN: 9781450302036. DOI: 10.1145/1869459.1869478. URL: http://dl.acm.org/citation.cfm?id=1869459.1869478 (cited on pages 4, 15, 19, 21, 69, 72, 78, 132).

[74] *MicroBlaze Soft Processor Core*. URL: http://www.xilinx.com/tools/microblaze.htm (cited on page 93).

[75] Robin Milner. *Operational and Algebraic Semantics of Concurrent Processes*. 1990. DOI: 10.1016/B978-0-444-88074-1.50024-X. URL: http://www.sciencedirect.com/science/article/pii/B978044488074150024X%7B%5C%%7D5Cnhttp://linkinghub.elsevier.com/retrieve/pii/B978044488074150024X (cited on page 117).

[76] Seung-Jai Min, Costin Iancu, and Katherine Yelick. “Hierarchical work stealing on manycore clusters”. In: *PGAS ’11*. 2011 (cited on page 76).

[77] Albert Noll, Andreas Gal, and Michael Franz. “CellVM: A homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor”. In: *Workshop on Cell Systems and Applications* (2008). URL: http://www.ics.uci.edu/~franz/Site/pubs-pdf/ICS-TR-06-17.pdf (cited on pages 4, 14, 15).

[78] Martin Odersky, Philippe Altherr, Vincent Cremet, and Burak Emir. “An overview of the Scala programming language”. In: Section 5 (2004). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.127.184%5C&rep=rep1%5C&type=pdf (cited on pages 4, 11).

[79] Scott Owens, Susmit Sarkar, and Peter Sewell. “A Better x86 Memory Model: x86-TSO”. In: *Proceedings of the 22th International Conference on Theorem Proving in Higher Order Logics*. TPHOLs. Springer Berlin Heidelberg, 2009, pages 391–407. ISBN: 978-3-642-03358-2 (cited on pages 20, 31, 129).

[80] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java concurrency in practice*. Pearson Education, 2006 (cited on page 103).

[81] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. 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. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005.* Feb. 2005, pages 184–592. DOI: 10.1109/isscc.2005.1493930. URL: http://dx.doi.org/10.1109/isscc.2005.1493930 (cited on pages 4, 14).

[82] Polyvios Pratikakis, Hans Vandierendonck, Spyros Lyberis, and Dimitrios S. Nikolopoulos. “A Programming Model for Deterministic Task Parallelism”. In: *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. MSPC ’11. San Jose, California: ACM, 2011, pages 7–12. ISBN: 978-1-4503-0794-9. DOI: 10.1145/1988915.1988918. URL: http://doi.acm.org/10.1145/1988915.1988918 (cited on page 129).

[83] William Pugh and Jeremy Manson. *Java Memory Model Causality Test Cases*. On http://www.cs.umd.edu/ as ~pugh/java/memoryModel/CausalityTestCases.html. 2004. URL: http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html (cited on pages 22, 54).

[84] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. “Synchronising C/C++ and POWER”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’12. ACM, 2012, pages 311–322 (cited on page 129).

[85] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. “The Semantics of x86-CC Multiprocessor Machine Code”. In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’09. New York, NY, USA: ACM, 2009, pages 379–391 (cited on page 129).

[86] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. “Shasta: A Low Overhead, Software-only Approach for Supporting Fine-grain Shared Memory”. In: *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VII. Cambridge, Massachusetts, USA: ACM, 1996, pages 174–185. ISBN: 0-89791-767-7. DOI: 10.1145/237090.237179. URL: http://doi.acm.org/10.1145/237090.237179 (cited on page 73).

[87] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. “Fine-grain Access Control for Distributed Shared Memory”. In: *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VI. San Jose, California, USA: ACM, 1994, pages 297–306. ISBN: 0-89791-660-3. DOI: 10.1145/195473.195575. URL: http://doi.acm.org/10.1145/195473.195575 (cited on page 73).

[88] Sangmin Seo, Jaejin Lee, and Zehra Sura. “Design and implementation of software-managed caches for multicores with local memory”. In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE. 2009, pages 55–66 (cited on page 131).

[89] Jaroslav Ševčík and David Aspinall. “On Validity of Program Transformations in the Java Memory Model”. In: *Proceedings of the 22nd European Conference on Object-Oriented Programming*. ECOOP. Springer, 2008, pages 27–51 (cited on page 22).

[90] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. “The Hadoop Distributed File System”. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. May 2010, pages 1–10. DOI: 10.1109/msst.2010.5496972. URL: http://dx.doi.org/10.1109/msst.2010.5496972 (cited on page 4).

[91] Doug Simon and C Cifuentes. “The squawk virtual machine: Java™ on the bare metal”. In: *Companion to the 20th annual ACM SIGPLAN …* (2005). URL: http://dl.acm.org/citation.cfm?id=1094908 (cited on page 94).

[92] L. A. Smith, J. M. Bull, and J. Obdrzálek. “A Parallel Java Grande Benchmark Suite”. In: *SC ’01*. Denver, Colorado, 2001. ISBN: 1-58113-293-X. DOI: 10.1145/582034.582042. URL: http://doi.acm.org/10.1145/582034.582042 (cited on page 109).

[93] Fengguang Song, Asim YarKhan, and Jack Dongarra. “Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC ’09. New York, NY, USA: ACM, 2009, 19:1–19:11. ISBN: 978-1-60558-744-8. DOI: http://doi.acm.org/10.1145/1654059.1654079. URL: http://doi.acm.org/10.1145/1654059.1654079 (cited on page 76).

[94] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. “Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-write Network”. In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. SOSP ’97. Saint Malo, France: ACM, 1997, pages 170–183. ISBN: 0-89791-916-5. DOI: 10.1145/268998.266675. URL: http://doi.acm.org/10.1145/268998.266675 (cited on page 73).

[95] *The Coq Proof Assistant*. URL, https://coq.inria.fr/. URL: https://coq.inria.fr/ (cited on page 134).

[96] *The JSR-133 Cookbook for Compiler Writers*. URL: http://gee.cs.oswego.edu/dl/jmm/cookbook.html (cited on page 31).

[97] Emina Torlak, Mandana Vaziri, and Julian Dolby. “MemSAT: Checking Axiomatic Specifications of Memory Models”. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. Toronto, Ontario, Canada: ACM, 2010, pages 341–350. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806635. URL: http://doi.acm.org/10.1145/1806596.1806635 (cited on pages 22, 65, 66, 136).

[98] George Tzenakis, Konstantinos Kapelonis, Michail Alvanos, Konstantinos Koukos, Dimitrios S Nikolopoulos, and Angelos Bilas. “{Tagged Procedure Calls} ({TPC}): Efficient Runtime Support for Task-Based Parallelism on the Cell Processor”. In: *HiPEAC*. Edited by Yale N Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell. Volume 5952. Lecture Notes in Computer Science. Springer, 2010, pages 307–321. ISBN: 978-3-642-11514-1 (cited on page 76).

[99] Rob V Van Nieuwpoort, Thilo Kielmann, and Henri E Bal. “Satin: Efficient parallel divide-and-conquer in java”. In: *Euro-Par 2000 Parallel Processing*. Springer. 2000, pages 690–699 (cited on page 77).

[100] R. Veldema, R.A.F. Bhoedjang, and H.E. Bal. “Distributed Shared Memory Management for Java”. In: *Proceedings of the 6th Annual Conference of the Advanced School for Computing and Imaging*. ASCI. 1999, pages 256–264 (cited on pages 21, 72).

[101] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. “Maxine: An Approachable Virtual Machine for, and in, Java”. In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), 30:1–30:24. ISSN: 1544-3566. DOI: 10.1145/2400682.2400689. URL: http://doi.acm.org/10.1145/2400682.2400689 (cited on page 94).

[102] Weimin Yu and Alan Cox. “Java/DSM: A platform for heterogeneous computing”. In: *Concurrency: Practice and Experience* 9.April (1997), pages 1213–1224. URL: http://onlinelibrary.wiley.com/doi/10.1002/(SICI)1096-9128(199711)9:11%3C1213::AID-CPE333%3E3.0.CO;2-J/abstract (cited on pages 3, 12, 21, 72).

[103] Foivos S. Zakkak and Polyvios Pratikakis. “JDMM: A Java Memory Model for Non-cache-coherent Memory Architectures”. In: *Proceedings of the 2014 International Symposium on Memory Management*. ISMM ’14. Edinburgh, United Kingdom: ACM, 2014, pages 83–92. ISBN: 978-1-4503-2921-7. DOI: 10.1145/2602988.2602999. URL: http://doi.acm.org/10.1145/2602988.2602999 (cited on pages 19, 154).

[104] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. “Software Write Detection for a Distributed Shared Memory”. In: *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*. OSDI ’94. Monterey, California: USENIX Association, 1994. URL: http://dl.acm.org/citation.cfm?id=1267638.1267646 (cited on page 73).

[105] Gengbin Zheng, Esteban Meneses, Abhinav Bhatele, and Laxmikant V Kale. “Hierarchical load balancing for Charm++ applications on large supercomputers”. In: *ICPPW ’10*. IEEE. 2010, pages 436–444 (cited on page 76).

[106] Yuanyuan Zhou, Liviu Iftode, and Kai Li. “Performance Evaluation of Two Home-based Lazy Release Consistency Protocols for Shared Virtual Memory Systems”. In: *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*. OSDI ’96. Seattle, Washington, USA: ACM, 1996, pages 75–88. ISBN: 1-880446-82-0. DOI: 10.1145/238721.238763. URL: http://doi.acm.org/10.1145/238721.238763 (cited on page 73).

[107] Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. “JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support”. In: *Proceedings of the IEEE International Conference on Cluster Computing*. CLUSTER. IEEE Computer Society, 2002, pages 381–388. ISBN: 0-7695-1745-5. URL: http://dl.acm.org/citation.cfm?id=792762.793331 (cited on pages 3, 14, 21, 72, 77, 79, 84).

[108] John N Zigman and Ramesh Sankaranarayana. “Designing a Distributed JVM on a Cluster”. In: *Proceedings of the 17th High Performance and Large Scale Computing Conference*. HP&LSC. 2002 (cited on page 21).

Appendix A.

JDMM Formal Definitions and DJC

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

In DJC, executions are expressed as sequences of state transitions. We use $S \xrightarrow{\mathscr{L}} S'$ to denote a transition from state S to state S', where $dom(\mathscr{L})$ is the set of cores involved and $rng(\mathscr{L})$ is the set of actions performed by the step. Additionally, we use $S \rightarrow^* S'$ to abbreviate a sequence of steps $S \xrightarrow{\mathscr{L}_1} S_1 \xrightarrow{\mathscr{L}_2} S_2 \ldots S_n \xrightarrow{\mathscr{L}_n} S'$, and $\mathscr{L} < \mathscr{L}'$ to show that $\mathscr{L}$ appears before $\mathscr{L}'$ in the sequence, e.g. $\mathscr{L}_1 < \mathscr{L}_2$ in the previous example. Note that in the rest of this document we refer to sequences of state transitions as DJC executions.

Actions: The JMM abstracts thread operations as actions [70, §5.1]. An action is a tuple $\langle r_t, k, r.f, u \rangle$, where r_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.

To get the set of actions A_D, from a program's DJC execution trace $S \rightarrow^* S'$, we take the union of the ranges $rng\left(\mathscr{L}_i\right)$, formally:

$$A_D = \bigcup_{\forall \xrightarrow{\mathscr{L}} \in \rightarrow^*} rng(\mathscr{L})$$

Program order $\leq_{po}^{d}$ is a relation on A_D defining a total order over all the actions executed by any single thread.

JDMM uses $x \leq_{po}^{d} y$ to show that x comes before y according to the program order. Every pair of actions executed by a single thread t are ordered by the program order:

$$\forall x, y \in A_D : \left((x \neq y) \wedge (x.t = y.t)\right) \Leftrightarrow \left((x \leq_{po}^{d} y) \vee (y \leq_{po}^{d} x)\right)$$

In DJC, $\leq_{po}^{d}$ can be obtained by the order in which actions appear in the execution. Formally, given a DJC execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}}, \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : \forall \alpha' \in rng\left(\mathscr{L}'\right) : \left(\alpha.t = \alpha'.t \wedge \mathscr{L} < \mathscr{L}'\right) \Rightarrow \alpha \leq_{po}^{d} \alpha'$$

Cache order $\leq_{co}^{d}$ is a relation on A_D defining a total order over all cache and memory access actions acting on a single cache. We use $x \leq_{co}^{d} y$ to show that x and y act on the same cache, and that x comes before y according to the cache order. Note that in implementations with private per thread caches the cache order is equal to the program order.

In DJC, $\leq_{co}^{d}$ can be obtained by the order in which actions performed by the same core appear in the execution. Note that, as we argue in Appendix B the operational semantics of DJC do not allow a single core to perform more than one action in a single step. Formally, given a DJC execution $S \rightarrow^{*} S'$:

$$\forall \xrightarrow{\mathscr{L}}, \xrightarrow{\mathscr{L}'} \in \rightarrow^{*} : \forall (c \mapsto \alpha) \in \mathscr{L} : \forall (c \mapsto \alpha') \in \mathscr{L}' : \\ \left(\alpha.k, \alpha'.k \in \{Iv, F, B, R, W, Vr, Vw\} \wedge \mathscr{L} < slabels'\right) \Rightarrow \alpha \leq_{co}^{d} \alpha'$$

Synchronization order $\leq_{so}^{d}$ is a relation on A_D defining a global order among all *synchronization actions* in A_D

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

$$\forall x, y \in \mathrm{SA}(A_D) \Leftrightarrow \left((x \leq_{so}^{d} y) \vee (y \leq_{so}^{d} x)\right)$$

In DJC, $\leq_{so}^{d}$ can be obtained by the order in which synchronization action appears in the execution. Formally, given a DJC execution $S \rightarrow^{*} S'$:

$$\forall \xrightarrow{\mathscr{L}}, \xrightarrow{\mathscr{L}'} \in \rightarrow^{*} : \forall \alpha \in rng\left(\mathscr{L}\right) : \forall \alpha' \in rng\left(\mathscr{L}'\right) : \\ \left(\alpha \in \mathrm{SA}(A_D) \wedge \alpha' \in \mathrm{SA}(A_D) \wedge \mathscr{L} < \mathscr{L}'\right) \Rightarrow \alpha \leq_{so}^{d} \alpha'$$

Since synchronization order is a global order, it must be defined for synchronization actions performed in the same step as well. Note that as long as the synchronization actions performed in the same step do not form a synchronizes-with pair their order does not impact the happens-before order which is the one describing which read a write may observe. In Appendix B we argue that DJC's operational semantics do not allow both actions of a synchronization-with pair to be performed in a single step. As a result, the order of any synchronization actions performed in a single step is not important regarding the correctness and the well-formedness of an execution, thus it can be chosen arbitrarily. In this work we arbitrarily order such synchronization actions according to the ID of the core that performed them, formally:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^{*} : \forall (c \mapsto \alpha), (c' \mapsto \alpha') \in \mathscr{L} : \left(\alpha \in \mathrm{SA}(A_D) \wedge \alpha' \in \mathrm{SA}(A_D) \wedge c < c'\right) \Rightarrow \alpha \leq_{so}^{d} \alpha'$$

***Cache-action seen* function** Ai, returns the write or fetch action that cached the data seen by a read, in A_D. Note that $Cs(r) \leq^d_{co} r$ and $Cs(r).k \in \{W, F\}$.

In DJC, $Cs(\langle r_t, R, r.f, u' \rangle)$ returns the write or fetch action $\langle r_t, W$ or $F, r.f, u \rangle$ writing or fetching the value that $\langle r_t, R, r.f, u' \rangle$ sees, according to the execution.

***Write-back fetched* function** Bf, returns the write-back action that whose data each fetch action fetches, in A_D. Note that $Bf(f) \leq^d_{so} f$ and $Bf(f).k = B$.

In DJC, $Bf(\langle r_t, F, r.f, u' \rangle)$ returns the write-back action $\langle r_t, B, r.f, u \rangle$ writing-back the value that $\langle r_t, F, r.f, u' \rangle$ fetches, according to the execution.

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

In DJC, $Ab(\langle r_t, B, r.f, u' \rangle)$ returns the initialization or write action $\langle r_t, Iv$ or $W, r.f, u \rangle$ that writes the data, that the action $\langle r_t, B, r.f, u' \rangle$ writes back, according to the execution.

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.

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

In DJC, $Ai(\langle r_t, Iv, r.f, u' \rangle)$ returns the write or fetch action $\langle r_t, W$ or $F, r.f, u \rangle$ writing or fetching the value that $\langle r_t, Iv, r.f, u' \rangle$ invalidates, according to the execution. 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.

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.

In DJC, given any execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}}, \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : \forall \alpha' \in rng(\mathscr{L}') : \mathscr{L} < \mathscr{L}' \Rightarrow \left(\alpha \text{ and } \alpha' \text{ can form a synchronizes with pair } \Leftrightarrow \alpha \leq^d_{sw} \alpha'\right)$$

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 [57]. 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.

Well-Formed Distributed Execution:
JDMM defines well-formed executions similarly to the JMM. In this paragraph we present the formal well-formedness rules for JDMM and define their equivalents DJC well-formedness rules. 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 [103, §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 [72, §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 $S \rightarrow^{*} S'$:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^{*} : \forall \alpha \in rng(\mathscr{L}) : \alpha.k \in \{R, Vr\} \Rightarrow \exists \xrightarrow{\mathscr{L}'} \in \rightarrow^{*} : \exists \alpha' \in rng\left(\mathscr{L}'\right) : \alpha'.k \in \{Iv, W, aVw\} \wedge \alpha.v = \alpha'.v$$

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 $S \rightarrow^{*} S'$:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^{*} : \forall \alpha \in rng(\mathscr{L}) : \alpha.k \in \{Vr, Vw\} \Leftrightarrow \alpha.v \text{ is volatile}$$

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

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

In DJC, this means that given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^* : \sum_{\forall \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \mathscr{L}' < \mathscr{L}} \#\{\alpha : \alpha \in rng\left(\mathscr{L}'\right) \wedge \alpha \in \mathrm{SA}(A_D)\} < \infty$$

WF-4 Synchronization order is consistent with program order:

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

In DJC this means that given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}}, \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \forall \alpha \in rng\left(\mathscr{L}\right) : \forall \alpha' \in rng\left(\mathscr{L}'\right) : \\ \left(\alpha \neq \alpha' \wedge \alpha, \alpha' \in \mathrm{SA}(A_D)\right) \Rightarrow \left(\alpha \leq^d_{so} \alpha' \Rightarrow \alpha \leq^d_{po} \alpha'\right)$$

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:

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

where T is the set of all the execution threads:

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

In DJC this means that given the execution $S \rightarrow^* S'$:

$$\begin{aligned} &\forall \xrightarrow{\mathscr{L}} \in \rightarrow^* : \forall \alpha \in rng\left(\mathscr{L}\right) : \forall t \in T : (t \neq \alpha.t \wedge \alpha.k = L) \\ &\quad \Rightarrow \sum_{\forall \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \mathscr{L}' < \mathscr{L}} \#\{\alpha' : \alpha' \in rng\left(\mathscr{L}'\right) \alpha'.k = L \wedge \alpha'.t = t\} \\ &\quad\quad = \sum_{\forall \xrightarrow{\mathscr{L}''} \in \rightarrow^* : \mathscr{L}'' < \mathscr{L}} \#\{\alpha'' : \alpha'' \in rng\left(\mathscr{L}''\right) \alpha''.k = U \wedge \alpha''.t = t\} \end{aligned}$$

WF-6 The execution obeys intra-thread consistency.

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

In DJC this means that given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : \alpha.k \in \{R, Vr\} \Rightarrow \Big(\neg\big(\alpha \leq^d_{po} W(\alpha)\big) \wedge \forall \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \nexists \alpha' \in rng\left(\mathscr{L}'\right) : (\alpha.k \in \{W, Vw\}) \wedge (\alpha.v = \alpha'.v) \wedge \big(W(r) \leq^d_{po} w \leq^d_{po} r\big)\Big)$$

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*" [72, §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 $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : \alpha.k = Vr \Rightarrow \Big(\neg\big(\alpha \leq^d_{so} W(\alpha)\big) \wedge \forall \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \nexists \alpha' \in rng\left(\mathscr{L}'\right) : (\alpha.k = Vw) \wedge (\alpha.v = \alpha'.v) \wedge \big(W(r) \leq^d_{so} w \leq^d_{so} r\big)\Big)$$

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 $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : (\alpha.k \in \{R, Vr\}) \Rightarrow \Big(\neg\big(\alpha \leq^d_{hb} W(\alpha)\big) \wedge \forall \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \nexists \alpha' \in rng\left(\mathscr{L}'\right) : (\alpha.k \in \{W, Vw\}) \wedge (\alpha.v = \alpha'.v) \wedge \big(W(r) \leq^d_{hb} w \leq^d_{hb} r\big)\Big)$$

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.*" [72, §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 $S_{init} \rightarrow^* S'_{init}$, where $\rightarrow^*$ contains only transitions performing the initialization actions and their write-backs, for every variable in the execution trace, and $S_{init} \rightarrow^* S'_{init}$ is well-formed —each initialization happens-before its write-back.

WF-10 Every read is preceded, according to cache order, by a write or fetch action, acting on the same variable as the read:

$$\forall r \in A_D : \exists x \in A_D : x \leq^d_{co} r \wedge x.v = r.v \wedge x.k \in \{W, F\}$$

In DJC given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^* : \forall (c \mapsto \alpha) \in \mathscr{L} : \alpha.k = R$$
$$\Rightarrow \exists \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \exists (c \mapsto \alpha') \in \mathscr{L}' : \big(\mathscr{L}' < \mathscr{L} \wedge \alpha.v = \alpha'.v \wedge \alpha'.k \in \{W, F\}\big)$$

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:

$$\forall r, x \in A_D : \big(x = Cs(r)\big) \\ \Rightarrow \nexists y \in A_D : (y.k \in \{Iv, F, W\}) \wedge (x.v = y.v) \wedge (x \leq^d_{co} y \leq^d_{co} r)$$

In DJC this means that given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : \alpha.k = R \Rightarrow \forall \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \nexists \alpha' \in rng\left(\mathscr{L}'\right) : \\ \left(\alpha'.k \in \{Iv, F, W\} \wedge \alpha.v = \alpha'.v \wedge Cs(\alpha) \leq^d_{co} \alpha' \leq^d_{co} \alpha\right)$$

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:

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

In DJC this means that given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : \alpha.k = F \\ \Rightarrow \exists \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \exists \alpha' \in rng\left(\mathscr{L}'\right) : \left(\alpha'.k = B \wedge \alpha.v = \alpha'.v \wedge \alpha' \leq^d_{co} \alpha\right)$$

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:

$$\forall b \in A_D : \exists w \in A_D : (b.v = w.v) \wedge (w \leq^d_{co} b)$$

In DJC this means that given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : \alpha.k = B \\ \Rightarrow \exists \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \exists \alpha' \in rng\left(\mathscr{L}'\right) : \left(\alpha'.k = W \wedge \alpha.v = \alpha'.v \wedge \alpha' \leq^d_{co} \alpha\right)$$

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

$$\forall b, w \in A_D : \big(w = Ab(b)\big) \Rightarrow \nexists w' \in A_D : \big((w'.v. = w.v) \wedge (w \leq^d_{co} w' \leq^d_{co} b)\big)$$

In DJC this means that given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}}, \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : \alpha.k = B$$
$$\Rightarrow \forall \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \nexists \alpha' \in rng\left(\mathscr{L}'\right) : \left(\alpha.v = \alpha'.v \wedge \alpha'.k = W \wedge Ab(\alpha) \leq^d_{co} \alpha' \leq^d_{co} \alpha\right)$$

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:

$$\forall p \in A_D : \exists x \in A_D : \nexists p' \in A_D : (Ai(p) = x) \wedge \big(Ai(p) = Ai(p')\big)$$

In DJC this means that given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : \alpha.k = Iv \Rightarrow \Big(\exists \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \exists \alpha' \in rng\left(\mathscr{L}'\right) :$$
$$\alpha' = Ai(\alpha) \wedge \forall \xrightarrow{\mathscr{L}''} \in \rightarrow^* : \nexists \alpha'' \in rng\left(\mathscr{L}''\right) : \left(\alpha''.k = Iv \wedge Ai(\alpha'') = Ai(\alpha)\right)\Big)$$

WF-16 Reads that see writes acting on a different cache are preceded, according to cache order, by a fetch action that fetches the data of the corresponding write, which were written back, and there is no other write-back of the corresponding variable happening between the write-back and the fetch, according to synchronization order. Formally:

$$\forall r \in A_D : \neg\left(W(r) \leq^d_{co} r\right)$$
$$\Rightarrow \exists b, f \in A_D : \Big(Ab(b) = W(r) \wedge Bf(f) = b \wedge f \leq^d_{co} r$$
$$\wedge \left(\nexists b' : b'.v = b.v \wedge b \leq^d_{so} b' \leq^d_{so} f\right)\Big)$$

In DJC this means that given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}}, \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \forall (c \mapsto \alpha) \in \mathscr{L} : \forall (c' \mapsto \alpha') \in \mathscr{L}' : \left(W(\alpha) = \alpha' \wedge c \neq c'\right)$$
$$\Rightarrow \exists \xrightarrow{\mathscr{L}_f}, \xrightarrow{\mathscr{L}_b} \in \rightarrow^* : \exists \alpha_f \in rng\left(\mathscr{L}_f\right) : \exists \alpha_b \in rng\left(\mathscr{L}_b\right) :$$
$$Ab(\alpha_b) = W(\alpha) \wedge Bf(\alpha_f) = \alpha_b \wedge \forall \xrightarrow{\mathscr{L}'_b} \in \rightarrow^* :$$
$$\nexists \alpha'_b \in rng\left(\mathscr{L}'_b\right) : \alpha_b \neq \alpha'_b \wedge Ab(\alpha'_b) = W(\alpha)$$

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, in the sense that no other action happens between the volatile write and its write-back, according to the program order. Formally:

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

In DJC this means that given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : \alpha.k = Vw$$
$$\Rightarrow \exists \xrightarrow{\mathscr{L}_b} \in \rightarrow^* : \exists \alpha_b \in rng\left(\mathscr{L}_b\right) : \forall \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \nexists \alpha' \in rng\left(\mathscr{L}'\right) :$$
$$\left(\alpha = Ab(\alpha_b) \wedge \alpha \leq^d_{po} \alpha' \leq^d_{po} \alpha_b\right)$$

WF-18 A fetch of the corresponding variable happens immediately before each volatile read, in the sense that no other action happens between the corresponding fetch and the volatile read, according to the program order. Formally:

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

In DJC this means that given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : \alpha.k = Vr$$
$$\Rightarrow \exists \xrightarrow{\mathscr{L}_f} \in \rightarrow^* : \exists \alpha_f \in rng\left(\mathscr{L}_f\right) : \forall \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \nexists \alpha' \in rng\left(\mathscr{L}'\right) :$$
$$\left(\alpha_f = Cs(\alpha) \wedge \alpha_f \leq^d_{po} \alpha' \leq^d_{po} \alpha\right)$$

WF-19 Initialization writes are immediately written back. Formally:

$$\forall x \in A_D : (x.k = In) \Rightarrow \exists b \in A_D : \nexists y \in A_D : \Big(b = Ab(x)\Big) \wedge (b \leq^d_{po} y \leq^d_{po} x)\Big)$$

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 $S_{init} \rightarrow^* S'_{init}$ 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 immediately written back.

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

$$\forall m, r \in A_D : \Big((m.k = M) \wedge (m \leq^d_{po} r)\Big) \Rightarrow \exists x \in A_D : \Big((x = Cs(r)) \wedge (m \leq^d_{co} x \leq^d_{co} r)\Big)$$

In DJC this means that given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}}, \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : \forall \alpha' \in rng\left(\mathscr{L}'\right) :$$
$$(\alpha.k = M \wedge \alpha'.k = R \wedge \alpha \leq^d_{po} \alpha') \Rightarrow \exists \xrightarrow{\mathscr{L}_f} \in \rightarrow^* : \exists \alpha_f \in rng\left(\mathscr{L}_f\right) :$$
$$\alpha_f = Cs(\alpha') \wedge \alpha \leq^d_{po} \alpha_f \leq^d_{po} \alpha'$$

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:

$$\forall m, w \in A : \Big((m.k = M) \wedge (w \leq^d_{po} m)\Big) \Rightarrow \exists b \in A : \Big((w = Ab(b)) \wedge \Big(w \leq^d_{co} b \leq^d_{co} m\Big)\Big)$$

In DJC this means that given the execution $S \rightarrow^* S'$:

$$\forall \xrightarrow{\mathscr{L}}, \xrightarrow{\mathscr{L}'} \in \rightarrow^* : \forall \alpha \in rng(\mathscr{L}) : \forall \alpha' \in rng\left(\mathscr{L}'\right) :$$
$$(\alpha.k = W \wedge \alpha'.k = M \wedge \alpha \leq^d_{po} \alpha') \Rightarrow \exists \xrightarrow{\mathscr{L}_b} \in \rightarrow^* : \exists \alpha_b \in rng\left(\mathscr{L}_b\right) :$$
$$\alpha = Ab(\alpha_b) \wedge \alpha \leq^d_{po} \alpha_b \leq^d_{po} \alpha'$$

Appendix B.

Proof sketch of adherence to JDMM

In this appendix we sketch the proof of 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 $S \rightarrow^* S'$, as described in Appendix A, we show that any execution $S \rightarrow^* S' \xrightarrow{\mathscr{L}} S''$ is well-formed as well. In our reasoning we use a number of lemmas that we argue to be true for any DJC execution. A few of these lemmas are shown to be true, by induction, along with the well-formedness rules, thus we mark those lemmas with **WFH-X** and skip the arguing about their correctness for now.

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

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

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

WFH-4: For every object r that appears in the execution, if it is present in the object cache, then there exists at least one fetch action that placed it there.

WFH-5: For every variable $r.f$, that appears in the heap or the caches, the value stored in them is the result of a write to $r.f$.

WFH-6: For every volatile variable $r.f$ in $\mathscr{H}$, its value is the one written by the last, according to synchronization order, volatile write action, acting on it, in that execution, 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.

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

$$\forall(\mathscr{H};\vec{\mathscr{C}};\vec{\mathscr{D}} \vdash T \xrightarrow{\mathscr{L}} \mathscr{H}';\vec{\mathscr{C}}';\vec{\mathscr{D}}' \vdash T') \in S \rightarrow^* S' : \forall r_t \in dom\left(\mathscr{H}'\right) :$$
$$\mathscr{H}'(r_t) = \langle C, \overrightarrow{f \mapsto v}, \text{spawned}\rangle \iff (\exists c\langle r_t, \_\rangle \in T' : \nexists c'\langle r_t, \_\rangle \in T' : c \neq c')$$

WFH-8: Each thread appears only on a single set of threads in a pair of set of threads. Formally:

$$\forall(\mathscr{H};\vec{\mathscr{C}};\vec{\mathscr{D}} \vdash T_1 \parallel T_2 \xrightarrow{\mathscr{L}} \mathscr{H}';\vec{\mathscr{C}}';\vec{\mathscr{D}}' \vdash T_1' \parallel T_2') \in S \rightarrow^* S' :$$
$$\forall r_t \in dom\left(\mathscr{H}'\right) : \mathscr{H}'(r_t) = \langle C, \overrightarrow{f \mapsto v}, \text{spawned}\rangle$$
$$\Rightarrow (\forall \_\langle r_t, \_\rangle \in T_1' \Rightarrow \nexists \_\langle r_t, \_\rangle \in T_2') \wedge (\forall \_\langle r_t, \_\rangle \in T_2' \Rightarrow \nexists \_\langle r_t, \_\rangle \in T_1')$$

WFH-9: The contents of the object cache and the write buffer of each core are altered only by that core.

WFH-10 DJC's operational semantics does not allow the execution of both synchronization actions of a synchronizes-with pair in the same step.

Having presented the **WFH-X** lemmas we continue with the rest of the lemmas.

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

Argument. Satisfied for every execution by the definition of the beginning of every execution in DJC to be $S_{Init} \rightarrow^* S'_{init}$ (see **WF-9** in Appendix A), where $\rightarrow^*$ contains only transitions performing the initialization actions and their write-backs, for every variable in the execution. □

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

Argument. In DJC this rule is always satisfied, since the beginning of every execution in DJC is $S_{init} \rightarrow^* S'_{init}$ where $\rightarrow^*$ contains only transitions performing the initialization actions and their write-backs, for every variable in the execution. □

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

Argument. 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.*

Argument. 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.*

Argument. In DJC this rule is always satisfied, since the beginning of every execution in DJC is $S_{init} \rightarrow^* S'_{init}$ where $\rightarrow^*$ contains only transitions performing the initialization actions and their write-backs, for every variable in the execution. As a result, in every execution 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 executions.*

Argument. We show, by induction on the number of steps, that for each well formed execution $S \rightarrow^* S'$, $S \rightarrow^* S' \xrightarrow{\mathscr{L}} S''$, where $\rightarrow^*$ and $\xrightarrow{\mathscr{L}}$ contain reductions of the local operational semantics, is also well-formed.

Rules CtxStep, IfTrue, IfFalse, Let, Call, Join, Interrupt, InterruptedT, and InterruptedF 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

$$S_{init} \rightarrow^* \mathscr{H}; \emptyset; \emptyset \vdash c\langle r_{main}, \mathsf{start}\rangle \xrightarrow{\mathscr{L}} S'$$

is well-formed.

In DJC the execution starts with a single thread –the main thread– and the beginning of any execution is:

$$S_{init} \rightarrow^* \mathscr{H}; \emptyset; \emptyset \vdash c\langle r_{main}, \mathsf{start}\rangle$$

where $\rightarrow^*$ contains only transitions performing the initialization actions and their write-backs, for every variable in the heap $\mathscr{H}$, and

$$S_{init} \rightarrow^* \mathscr{H}; \varnothing; \varnothing \vdash c\langle r_{main}, \mathsf{start}\rangle$$

is well-formed.

In the local operational semantics,

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

the only rule that can step, after the initialization, is START.

WF-3 is satisfied, since this is the first synchronization action, other than initialization actions, in the execution and the number of initialization actions is equal to the number of variables, in a program, which we assume to be 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-1 and **WFH-6** are satisfied since any variables in the heap are initialized and written back only once.

WFH-2–**WFH-4** are satisfied since there are no variables in the the object cache, or the write buffer.

WFH-5 is satisfied since any variables in the heap are initialized and written back.

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 $S \rightarrow^* S'$, $S \rightarrow^* S' \xrightarrow{\mathscr{L}} S''$ is also well-formed.

We examine each case for $S' \xrightarrow{\mathscr{L}} S''$ in the local operational semantics:

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

and show that it satisfies the well-formedness rules.

Case 6.1. Field

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

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

By the premises of FIELD:

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

WF-1: Since the value of $r.f$ is read from the object cache and $S \rightarrow^* S'$ 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. As a result, **WF-1** is satisfied.

WF-6: Since $r.f$ is present in the object cache and $S \rightarrow^* S'$ is well formed and according to **WFH-2**, it was either fetched or updated through a write-back. In both cases, since $S \rightarrow^* S'$ 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. 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. That said, **WF-6** is satisfied.

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

WF-11: Since the value of $r.f$ is read from the object cache and $S \rightarrow^* S'$ 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. 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, 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

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

where $r.f \in dom(\mathscr{D})$.

By the premises of FieldDirty:

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

WF-1: Since the value of $r.f$ is read from the write buffer and $S \rightarrow^* S'$ 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. As a result, **WF-1** is satisfied.

WF-6: Since the value of $r.f$ is read from the write buffer and $S \rightarrow^* S'$ 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. As a result, **WF-6** is satisfied.

WF-11: Since the value is read from the write buffer and $S \rightarrow^* S'$ 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. 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, 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

$$S \rightarrow^* \mathscr{H}; \mathscr{C}; \mathscr{D} \vdash c\langle r_t, e\rangle \xrightarrow{c \mapsto \langle r_t, W, r.f, u\rangle} \mathscr{H}; \mathscr{C}; \mathscr{D}' \vdash c\langle r_t, e'\rangle$$

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

By the premises of Assign:

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

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.

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

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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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

$$S \rightarrow^* \mathscr{H};\mathscr{C};\mathscr{D} \vdash c\langle r_t, e\rangle \xrightarrow{c \mapsto \langle r_t, Vr, r.f, u\rangle} \mathscr{H}';\mathscr{C};\mathscr{D} \vdash c\langle r_t, e'\rangle$$

where

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

WF-1: Since the value of $r.f$ is read from the heap and $S \rightarrow^* S'$ 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, or by the initialization action, acting on $r.f$, if there are no volatile write actions, acting on $r.f$, in that execution. As a result, **WF-1** and **WF-6** are satisfied.

WF-2 is satisfied since in $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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.

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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ and NestedMonitorExit requires that the monitor $r.l$ is already acquired more than one time 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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. As a result, **WF-12** is satisfied.

WF-3 is satisfied, since $S \rightarrow^* S'$ 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.

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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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, which acts on $r.f$ and is performed by c. As a result, **WF-13** and **WF-14** are satisfied.

WF-20: Since $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ (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.

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 $S \rightarrow^{*} S'$ 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 $S \rightarrow^* S'$ 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 executions. □

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

Argument. In DJC the lifting is performed by Lift. Lift does not introduce new modifications to the memory state or new actions in the execution, 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 1. *DJC's operational semantics generates only well-formed execution traces.*

Argument. We show, by induction on the number of steps, that for each well formed execution $S \rightarrow^* S'$, $S \rightarrow^* S' \xrightarrow{\mathscr{L}} S''$, 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 $S \xrightarrow{\mathscr{L}} S'$, is well-formed.

In DJC the execution starts with a single thread –the main thread– and the beginning of any execution is:

$$S_{init} \rightarrow^* \mathscr{H}; \varnothing; \varnothing \vdash c\langle r_{main}, \mathsf{start}\rangle$$

where $\rightarrow^*$ contains only transitions performing the initialization actions and their write-backs, for every variable in the execution, and

$$S_{init} \rightarrow^* \mathscr{H}; \varnothing; \varnothing \vdash c\langle r_{main}, \mathsf{start}\rangle$$

is well-formed.

As a result, $S = \mathscr{H}; \varnothing; \varnothing \vdash c\langle r_{main}, \mathsf{start}\rangle$

In the global operational semantics,

$$\mathscr{H}; \vec{\mathscr{C}}; \vec{\mathscr{D}} \vdash T \xrightarrow{\mathscr{L}} \mathscr{H}; \vec{\mathscr{C}}; \vec{\mathscr{D}} \vdash T$$

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 1.1. Lift

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

Case 1.2. 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$, $S \rightarrow^* S' \xrightarrow{\mathscr{L}} S''$ is also well-formed.

We examine each case in the global operational semantics:

$$\mathscr{H}; \vec{\mathscr{C}}; \vec{\mathscr{D}} \vdash T \xrightarrow{\mathscr{L}} \mathscr{H}; \vec{\mathscr{C}}; \vec{\mathscr{D}} \vdash T$$

and show that it satisfies the well-formedness rules.

Case 1.3. Lift

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

Case 1.4. Spawn

WF-4: Since $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$. 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 $S \rightarrow^* S'$.

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 1.5. Migrate

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

WFH-7: Since $S \rightarrow^* S'$ 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 1.6. Blocked

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

Case 1.7. ParG

WF-1: Since $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 happens-before the start of the program. Additionally, since $S \rightarrow^* S'$ 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 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 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 $S \rightarrow^* S'$ 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.

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 $S \rightarrow^* S'$ 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.

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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 structural rules. 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$ 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 $S \rightarrow^* S'$.

WFH-9: Since **WFH-9** is satisfied by $S \rightarrow^* S'$ 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.

□