Building a Java™ Virtual Machine for Non-Cache-Coherent Many-core Architectures

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

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

ABSTRACT

This work presents the key challenges that Java Virtual Machines (JVMs) implementers face when targeting future non-cache-coherent architectures. It discusses the techniques used to overcome these challenges by distributed JVMs in the literature, and examines their applicability on future non-cache-coherent architectures. It presents new algorithms for software caching, monitor management, and thread scheduling, that take advantage of the hierarchical nature of future non-cache-coherent architectures with coherent-islands. It also builds a proof-of-concept JVM that runs on a 512-core, non-cache-coherent, many-core prototype and present early evaluation results for parts of the proposed algorithms.

CCS Concepts

•**Software and its engineering** → **Virtual machines;**

Keywords

Java Virtual Machine; Non Coherent Memory; Many-core

1. INTRODUCTION

Contemporary multicore processors rely on hardware cache coherence to implement shared memory abstractions. However, recent literature shows that existing coherence implementations are not scaling well with the number of processor cores. They incur large energy and area costs, increase on-chip traffic, or limit the number of cores per chip [10, 28, 8]. Despite attempts to design less costly or more scalable coherence protocols [19, 21], this problem is yet to be solved.

At the same time hardware architects try to cope with the constantly growing number of cores per processor, proposing modular non-cache-coherent architectures. Such architectures delegate the memory coherency to the software, avoiding the extra overhead –in terms of energy, area, and validation complexity– of coherence protocols. Such examples are the architectures Intel® Runnemede [8], Formic [17], and EUROSERVER [12]. These architectures are designed in a modular way, allowing the combination of various smaller modules to create a big many-core processor. Each module is self-contained and able to interface with other modules. All the architectures provide DMA-like support for transferring data across modules. Each module features multiple cores and a memory module that is shared among them. EUROSERVER, by way of exception, provides cache coherency between the cores of a module.

The lack of cache-coherence renders the software responsible for performing the necessary data transfers to ensure data coherency in parallel programs. However, in high productivity languages, such as Java, the memory hierarchy is abstracted away by the process virtual machines, rendering the latter responsible for the data transfers. Process virtual machines provide the same language guarantees to the developers as in cache-coherent shared-memory architectures. Implementing process virtual machines on top of non-cache-coherent architectures requires special handling of the memory to ensure that the corresponding language's memory model is adhered.

This work focuses on the development of Java Virtual Machines (JVMs) on non-cache-coherent architectures. Specifically, this work makes the following contributions:

- We present and discuss the key challenges in the implementation of a JVM targeting non-cache-coherent many-core architectures.
- We propose algorithms to overcome those challenges and improve the state-of-the-art. Our algorithms exploit the spatial locality and memory coherency of partially coherent architectures like EUROSERVER.
- We implement the proposed inter-coherent-island techniques, and present early evaluation results.

2. RELATED WORK

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) [18]. JMM

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

ISBN 978-1-4503-4800-3/16/08…$15.00

DOI: http://dx.doi.org/10.1145/2990509.2990510

defines a set of rules that a Java program execution must satisfy in order to be considered valid. Since multithreaded Java programs are not always deterministic —they can have more than one possible execution— JVMs must restrict the possible executions to those considered valid by JMM.

Next, we briefly present the most noticeable JVM implementations on clusters or heterogeneous systems. Note however that these JVMs, except from Hera-JVM [20], do not adhere to the latest JMM [18]. Note also that we omit implementations relying on Software Distributed Shared Memory (SDSM). SDSM implementations are not as relaxed as the JMM and most of them operate on page granularity while JMM is defined on field granularity (64bit values or less). 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.

Cluster JVM (cJVM) [4] aims to provide a Single System Image (SSI) view of clusters. Each node in the cluster runs an instance of cJVM and all together form the VM. In cJVM, objects are allocated locally, on each node's heap, and proxies to them are provided to other nodes at remote method invocations. 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, by exploiting data locality. The authors call this technique *method shipping*. *Method shipping* is implemented using a set of server threads that handle the *shipped* methods, per cJVM instance. 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. A limitation of cJVM is that in order to scale on a large number of cores it requires parallel programs with balanced data distribution.

JESSICA2 [31] is a Distributed Java Virtual Machine, that employs a memory abstraction layer, called Global Object Space (GOS). In GOS, 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, while *cached heap area* accesses are handled by the GOS layer. Each thread has a private cache directory, where it caches objects that reside in remote *cache heap areas*. At cache misses JESSICA2 initiates a data transfer and yields the thread, until the transfer is completed. Furthermore, at synchronization points, cached objects are flushed and re-fetched to conform to JMM. JESSICA2 also 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 in increased data traffic at synchronization points. To reduce this overhead when a thread's number of accesses to an object dominate its total number of accesses, this object is migrated to the thread's node and the remaining nodes are informed about the object's new *home*. For synchronization actions JESSICA2 sends a message to the *home* node of the synchronization object to handle the synchronization action, something analogous to cJVM's method shipping. This approach is expected to slow down threads requesting a lock from a busy *home* node.

CellVM [23] is a JVM targeting IBM's Cell B.E. [25]. Cell B.E. 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 256KB of non-cache-coherent, dedicated local storage. CellVM abstracts away the heterogeneity of the processor and the absence of cache-coherence on the SPEs. CellVM employs a centralized design, where the PPE runs a special VM instance, called the ShellVM, and the SPEs run a lightweight VM, called the CoreVM. The ShellVM is responsible for handling the majority of the CellVM's internal data structures. On the contrary, CoreVM is mainly a bytecode interpreter. The ShellVM performs all the bookkeeping for CellVM and schedules work to the CoreVMs. However, some special instructions (*i.e.* memory allocation, synchronization, locking etc.) are not handled by the CoreVM. In such cases the corresponding CoreVM requests the ShellVM to process them. This design is not expected to scale on large number of cores, since it resides on a single coordinator, and increasing the number of coordinators is not trivial.

Hera-JVM [20] is another virtual machine targeting Cell B.E. Hera-JVM differs from CellVM in that: *a*) it offers transparent migration of Java threads between the two core types; *b*) does not have special types of VMs depending on the core type; *c*) it improves scalability by enabling SPEs to execute synchronization instructions, allocate memory, etc.; *d*) it adheres to JMM. Hera-JVM, similarly to CellVM, uses a data and an instruction cache. On a data 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 is initiated, that copies the new data to the main memory. To conform to JMM, a Java thread blocks until all DMAs are finished before either releasing a lock, writing to a volatile variable, context switching or migrating to another core. Hera-JVM is written in Java and delegates some of its synchronization to its Java implementation with the cost of some additional overhead. For instance, it implicitly writes-back data at context switches due to the fact that context switching is implemented by a synchronized method. Note, however, that context switches, in contrast to thread migrations, do not require any cache operations to adhere to JMM [29].

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. Additionally, they allow for explicit asynchronous DMA transfers from and to the whole system's address space. On the contrary, super-computer clusters with interfaces supporting DMA transfers only allow access to a subset of the system's address space. On the other hand, Cell B.E. shares common characteristics with that of the proposed architectures [8, 17, 12]. 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. 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 Cell B.E. to such an architecture may not result in the expected scalability.

3. KEY CHALLENGES

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

3.1 Memory Management

One of the key challenges is memory management. Java uses a heap for objects and a stack per thread for local variables. JMM defines that the Java heap is global per application. That is, every thread is able to access the whole Java heap. 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 Java heap is accessible from every thread in the application and that all threads get a coherent view of the Java heap. Java also uses garbage collection for dynamic memory management. The design and implementation of a distributed non-stop-the-world garbage collector is an open problem. However, it is orthogonal yet complementary to this work, and outside the scope of this paper.

Early attempts to implement distributed JVMs delegated the memory management to SDSM,which as we state in §2 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 [20, 3, 31]. When using software caching, to access a remote object the JVM *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. Since memory accesses are very common, software caches need to be efficient and 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 on top of software caches.

JDMM [29] is an extension of JMM that exposes such operations to the programming model and argues 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* [15] this read. For instance, in Figure 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 the 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.

As a result, JVMs targeting 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 1. The red dashed rectangle on the upper left corner of the big rectangle is the time window in which the write-back can be executed. Respectively the green 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.

In this work we present a software cache scheme that aims to minimize memory transfers while adhering to JMM.

3.2 Synchronization

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

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 §3.1) but also needs to ensure that monitor acquisitions are consistent with mutual exclusion. That is, a monitor may only be owned by a single thread at any time, allowing for that single thread to own the monitor multiple times—re-entrant acquisition. 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, etc. Such instructions are usually not available across coherent-islands in distributed memory architectures, thus JVM implementers need to provide a solution independent of those instructions for synchronizing cores across different coherence-islands. In this work we propose the use of dedicated cores that handle monitors.

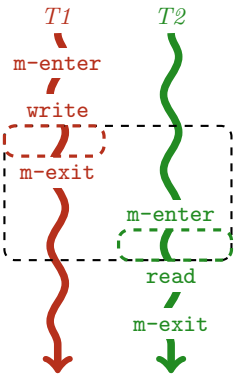


Figure 1: Time window example.

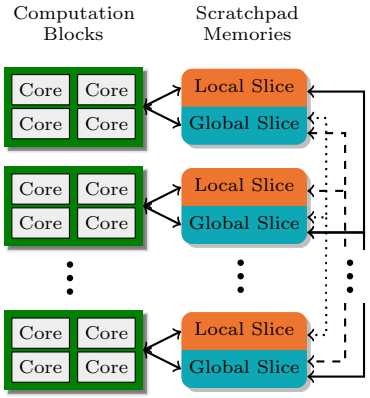


Figure 2: The memory abstraction.

Note that we do not consider `java.util.concurrent` here. This library is targeted to shared-memory machines and is usually implemented directly with atomic primitives provided by shared memory architectures. We believe that on non-cache-coherent architectures `java.util.concurrent` should be implemented from scratch, to make it efficient, and explicitly perform the needed software cache operations where needed, which is out of this paper's scope.

3.3 Thread Scheduling

On large numbers of cores, thread scheduling is another key challenge. 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 be comprised of multiple OS instances that rely on the application to perform the thread scheduling [13]. When implementing a JVM, such schemes must be abstracted away, since Java does not expose architectural and OS details. 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.

Following task-based programming models, we propose the use of light-weight tasks in applications with irregular parallelism. Related works studying Java extensions for tasks show good results [7, 14]. On shared memory architectures the dominating load balancing algorithm for task-based applications is that of work stealing [6]. 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 [9] 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 [11, 30, 22] has presented a few implementations of the work-stealing algorithm for clusters with an RDMA interconnect.

Although task-based programming appears to be promising, the scheduling problem of legacy code remains open. In this work we present a hybrid algorithm that allows work-dealing —coordinated thread exchange— across coherent-islands and work-stealing within coherent-islands.

4. DESIGN

This section examines the applicability of existing techniques to non-cache-coherent architectures, regarding the key challenges discussed in §3. We explain cases where existing techniques are inefficient and propose alternative solutions.

4.1 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. In our design we follow the JESSICA2 [31] paradigm and 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. Figure 2 gives an overview of the resulting scheme, excluding the *native* memory part. 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. 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 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 which case a request is sent to a remote *global 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. 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 [20] 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 1KB 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 and helps to better understand the application's behavior.

JESSICA2 [31] uses a separate *cache heap area* per Java thread. The authors argue 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 work we propose a hybrid approach, where read accesses are cached into a shared per-core cache, while write accesses are cached into a private per-Java-thread cache. In Java, the distinction between read and write accesses is easy without compiler support, since there are different bytecodes for each (*e.g.* getfield/putfield). The shared cache increases the cache hit rate, while the private cache reduces the delay of cache flushes. When reading a remote object, the JVM first queries the private cache, and in the case of a miss continues by querying the shared cache. If both queries fail, then the JVM fetches the remote object to the shared cache. When writing on a remote object, the JVM first queries the thread's private cache, and in the case of a miss fetches the object to it. Write-backs are straight forward; the JVM for each dirty field in the private cache, writes it back, copies it to the shared cache and invalidates it in the private cache. This way, we ensure that the private cache only holds dirty data. To avoid the invalidation of the whole shared cache at synchronization points, we use a bitmap for each shared cache entry to mark which threads had used that entry. Note that in our design it is possible for different threads on the same coherence-island to observe different cached values because of the private caches. This is safe, since in non data-race-free (DRF) programs the reader thread will have to synchronize with the writer first, and thus obtain the latest value.

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. To avoid long blocking release actions, we set an upper bound

of dirty data in the private caches. This sets an upper bound to the release actions' blocking time. When this threshold is reached, the JVM initiates asynchronous write-backs of the dirty data in the *releasing* thread's private cache. A lower threshold results in better communication and computation overlap, as well as shorter release actions. However, a low threshold may result in multiple write-backs of the same data, if that data are written more than once in the corresponding critical section. On the other hand, higher thresholds tend to result in longer release actions for critical actions that write many data, and less computation and communication overlap to hide the overhead. Since the fine tuning of the threshold depends on the nature of its critical section, a possible optimization is the use of the just-in-time (JIT) compiler to set a different threshold per case. JIT compilers are ubiquitous in modern JVMs and generate optimized code by profiling code segments.

Regarding the replacement policy of the cache, we avoid the book-keeping overhead of popular cache algorithms, like LRU, LFU etc. Since shared caches are invalidated at *acquire* operations, we expect the case of running out of space in the shared cache to be rare. As a result, when there is not enough space left in it, we invalidate the whole shared 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 private caches we rely on the aforementioned threshold on the number of dirty data in the private caches.

4.2 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 hardware implementation of interrupts.

In this work, we propose the use of specialized cores that act as *synchronization managers* (SMs). Each SM is responsible for a subset of objects in the heap. These subsets need to be evenly distributed across SMs and comprise non-contiguous memory addresses to better distribute the requests. The SM employs a message queue that it constantly polls for requests. To further improve energy efficiency, we suggest an interruption driven model where the SM 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. Furthermore, in architectures with heterogeneous cores or configurable hardware, SMs can be run on low-end energy efficient cores, since the computation requirements are low.

To enter a monitor, threads send a request to the corresponding SM and then yield. SMs hold a record for each monitor they are responsible for. This record holds the *owner* of the monitor. When serving a request, the SM 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. 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 assigned to a different thread, instead of sending a negative reply, we add the requesting thread to the acquisition queue. Later, when the monitor becomes available, the SM 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 core of the requester to wake up the requester and notify it about the successful monitor acquisition. Since the requester thread cannot make any progress until that message arrives, the JVM does not reschedule it until then. If there are no runnable threads in the threads queue the requester thread will be immediately scheduled to run when the reply arrives. If, however, the threads queue is not empty then another thread will start running and the requester thread will just be placed back to the threads queue when the reply arrives.

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 threads that run on the same coherent-island. A monitor acquired by a thread may be directly assigned to another thread running on the same coherent-island without notifying the SM. T is used to provide fairness among threads running on different coherent-islands and avoid starvation. To achieve this, we introduce the *local monitor*, 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.

When a thread requests ownership of an object's monitor, it first checks the concurrent data-structure for a local monitor. In the trivial case, where the local-monitor exists and is owned by the current thread, the JVM locally increases the nesting level. If a local-monitor does not exist the threads creates one and tries to add it to the data-structure. We rely on the data-structure's implementation to atomically insert the new monitor and in case of failure due to a race, to return an error. Then if the local-monitor is available, the requesting thread races with other threads, from the same coherent-island trying to enter the same monitor, to atomically set its owner. The winner then increases the local-monitor's nesting level and sends a request to the SM before yielding until it gets a positive reply to its request. The remaining threads (losers) 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 JVM needs to check for replies regarding threads in the acquisition queue, although these threads never explicitly send a request to the SM. This is a side-effect of the monitor release that we describe next.

When a thread releases a monitor, it first decreases locally the nesting level. If the nesting level is not zero it 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 the owner, and sends a release request to the SM. 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 a release request followed by an acquire request for the dequeued thread to the SM. 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. This is why the entering threads yield until they either get the monitor from an exiting thread, or get a positive reply from the SM.

This design reduces the contention on the SMs and the network in three different ways: *a*) limiting the requests per coherent-island—each coherent-island has up to one outstanding acquire request per object; *b*) combining monitor acquisitions—each monitor can be reused up to T times before returning it to the SM; *c*) distributing monitor management—each local monitor keeps information about nested acquisition eliminating redundant messages to the SM.

Another benefit 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 do 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. Since the JVM is aware of the monitor acquisition sequence, it is able to transfer data between the private caches of the corresponding threads. Directly copying between the private caches has the benefit of transferring data within the coherent-island, which is more efficient than writing them back to the main memory and fetching them again. Finally, when the monitor is released to the SM only the releasing thread needs to issue write-backs to remote memories for its software cache dirty entries.

Since object-wait and object-notify are tightly coupled with monitors, we delegate them to the SM as well. We further extend the monitor records in the SMs 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 SM. The SM enqueues the thread to the waiters queue and releases the monitor. Respectively, when a thread invokes `notify()`, a notify request is sent to the SM, 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()`, a slightly different request is sent to the SM, which dequeues and notifies all threads in the waiters queue.

4.2.1 Volatile Variables

Another challenging part is the support of volatile variables. In Java, 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 [16].

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 [24, §3.1.4] —meaning they are being mostly read during their life-cycle.

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

4.3 Thread Scheduling

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 [2]. Our work-dealing algorithm differs from that of [2] 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. In our design, the coordination happens through messages. That implies, however, that idle cores need to wait for active cores to reply back, essentially delaying the load-balancing process. On the other hand, idling cores' replies will be immediate, meaning that an idle core ends up waiting only when there is work on the remote core. Since such message exchanges impose overhead to the network, we propose the use of the half-steal approach, which [11] shows 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.

4.3.1 Load Balancing

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 some 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. If X deques on an island are empty, then there is probably not enough work to utilize every core on the island. As a result, after attempting X steals without success, the JVM tries to find work on remote islands. We heuristically use the ceiling of the square root of the number of cores C per coherent-island as the value of X, $X = \lceil\sqrt{C}\rceil$. We base our choice on the reasoning that if a core observes $\lceil\sqrt{C}\rceil$ idle cores, then these cores have probably also observed $\lceil\sqrt{C}\rceil$ idle cores totalling approximately C 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. 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 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, at random, a core from the same coherent-island and sends a schedule request to it, if its deque is not full. Otherwise it picks, at random, an island and sends a schedule request to it. The *Schedule Request Handler* procedure is responsible for handling 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.

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 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 SM. Thread join can be implemented with the use of wait and 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 SM dequeues and notifies all joiners in the queue.

5. EVALUATION

To evaluate the proposed algorithms we implement them in DiSquawk [1], our open-source proof-of-concept JVM, that targets non-cache-coherent many-core processors. 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 [17], a hardware processor prototype emulating such an architecture. Note however that 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 §4, 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.

Formic-cube consists of 64 computation blocks, each featuring 8 cores, totaling 512-cores. Each computation block has direct access to a private scratchpad of 128 MB, and RDMA access to the remote scratchpads. Each core is equipped with a mailbox—a hardware queue where it can receive messages from other cores on the system. Each core also features a two-level, non-coherent, cache hierarchy. Formic-cube does not provide atomic operations on memory addresses.

DiSquawk is implemented as a combination of modified Squawk VM [26] 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*. 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.

To implement the distributed Java heap as described in §4.1, we create a PGAS scheme. We use the most significant bits (MSBs) of a memory address as the unique identification of a scratchpad memory. The remaining bits are used as the relative address inside that scratchpad memory. The number of bits reserved for the scratchpad memories identification is calculated by $\lceil log_2(N)\rceil$, where N is the number of scratchpad memories in the system. Since Formic-cube does not provide coherent-islands, we further slice the *global heap areas* in 8 equal slices, one per core.

5.1 Software Caches

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 9 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.

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.

5.2 Synchronization Manager

For the SMs we reserve one of the Formic-boards. On each core of this board we deploy a SM. The SMs constantly poll their mailbox for incoming requests and directly serve any available requests. To find the number of SMs needed to efficiently handle the requests from 504 cores —the cores left after reserving a board for the SMs— we create a micro-benchmark with 504 threads that each requests to enter a different monitor. We run it with a single SM and present the results in Figure 3a. The results show that a single SM is able to handle up to 256 cores; after that point it starts to become a bottleneck. More interestingly, as shown in Figure 3b, the throughput of a single SM drops suddenly at certain numbers of cores. This is caused by the mailbox getting full. Each request is of 8 B 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 16 B in the SM's hardware queue. Since the hardware queue is of 4 KB size, in the worst case scenario, where the SM 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 3c we present the results from varying the number of SMs in the same micro-benchmark when running with 504 threads. The results show that the peak throughput —aggregated throughput of all the SMs in the system— is reached when using 3 SMs. However, we 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 SMs is needed to handle 504 cores.

5.2.1 Impact of Queuing Requests

To evaluate the impact of queuing requests in the SM, instead of replying with NACKs and retrying (see §4.2), 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 4a presents the total execution time in millions of clock cycles, when queuing is enabled (blue bars), and when it is disabled (orange bars). We observe a maximum 3× 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 SM 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 4b presents the total throughput, as monitor enters per one thousand clock cycles, measured at the SM for the same runs. We observe that when queuing is enabled for runs with more than 15 threads, the SM is able to handle 9 `monitorenter`s per one million clock cycles. On the contrary when queuing is disabled, the SM'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 SM's mailbox is constantly full, resulting in the SM being mostly busy with sending NACKs.

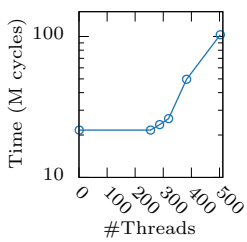
Note that the SM's throughput is much higher than 9 operations per one million clock cycles, as shown in Figure 3b. In this micro-benchmark however, the throughput is bound by the rate at which a thread is able to acquire and release a monitor, since all threads race for the same monitor. Our measurements show that a SM 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 SM does not perform queue operations.

5.3 Scheduler

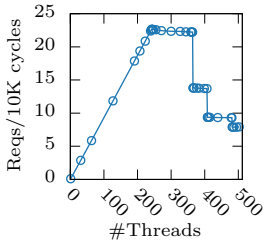
Due to the lack of coherent-islands we are not able to implement the hybrid load balancing mechanism presented in §4.3. Instead, we use a variation of the *Start New Thread* and *Schedule Request Handler* procedures. In our implementation the procedures pick, at random, a core from the whole system to send the schedule request.

To improve performance we use the `Thread` class as a wrapper for the internal Squawk's `VMThread` class. `Thread`'s instances hold the thread's state, and references to the target `Runnable` and the corresponding `VMThread` instance. When the thread's state is `NEW`, its `VMThread` reference is `null`. After scheduling the thread, the core that serves the schedule request is responsible to create a `VMThread` instance, associate it with the thread's `Thread` instance, and change its state to running. The `VMThread` instance is holding the thread's stack and runtime information for that thread. 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 to `DEAD`, without acquiring the monitor, writes back the cached `Thread` instance, and notifies any joiners through the SM. Note that this is safe, since only one core —the one running the thread— can change its state.

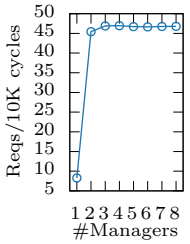
To evaluate the performance of our approach we use a micro-benchmark that creates a single thread and schedules it to a remote idling core, then it joins on it and completes. Our measurements show that from the invocation of



(a) Execution Time (1 SM) vs #Threads

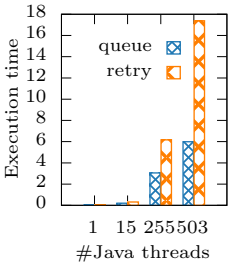


(b) SM Throughput vs #Threads

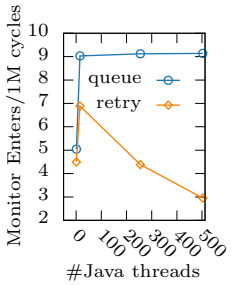


(c) Throughput vs #SM

Figure 3: Number of synchronization managers (SMs) vs Performance



(a) Execution Time



(b) SM Throughput

Figure 4: Queuing vs Retrying

`Thread.start()` to the actual run of the thread on an idle core it takes 18640 clock cycles on average. Of this time DiSquawk spends: 1.6% to choose a remote core and to construct and issue the schedule request; about 0.5% for the schedule request transfer itself; 28% to allocate and construct the new *VMThread* instance; 55% to initialize the `VMThread` instance; 4% to allocate the stack; 1% to add it to the runnable's queue; 8% waiting for the JVM to schedule it; and the remaining 2% in other more fine-grained tasks. Our experiment shows that most time is spent in computations performed in Java, since Squawk is written in Java. We expect even faster thread scheduling in modern JVMs, that support JIT compilation. 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 Disquawk spends about 15% for the bookkeeping and the remaining 85% 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× speedup in `join()`.

5.4 Overall

To evaluate the overall scalability of DiSquawk we use the Crypt, SOR, and Series benchmarks from the Java Grande [27] suite and the Black-Scholes benchmark from the PARSEC suite [5], ported to Java. Due to the lack of garbage collection and the upper limit of 4 GB heap we are unable to run reasonable workloads with the rest of the Java Grande benchmarks. These benchmarks require larger than 4 GB datasets to produce meaningful results on a large number of cores and some of them also create objects with short lifespans, relying on garbage collection to reclaim their memory. Series and Black-Scholes are embarrassingly parallel benchmarks. Each thread operates on a different subset of data from an input set and creates a new set with the corresponding results. 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. 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.

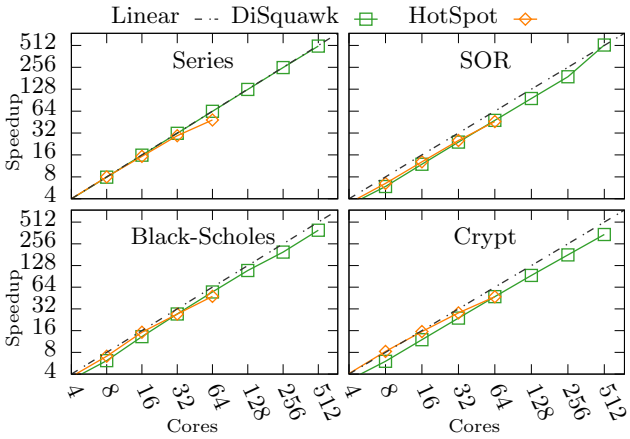


Figure 5: Speedup Results

Figure 5 presents the speedup of the four benchmarks on both DiSquawk, running on the Formic-cube, and HotSpot running on a 4-chip NUMA machine with 16 cores per chip, totalling 64 cores. Since Formic-cube is a prototype clocked at 10MHz, a comparison of the throughput or the execution time is not possible, thus we chose to compare the applications' scaling on both architectures. The presented speedups are over the performance of the application running on a single core on each architecture respectively. Since DiSquawk does not support JIT compilation, we also disable it in HotSpot (using the `-Xint` flag), this allows us to better understand the applications' behavior on both architectures. The number of threads, one per core, is placed on the x-axis, and the speedup is placed on the y-axis. We observe that all benchmarks scale with the number of cores in both architectures. Black-Scholes and Series scale better on DiSquawk than HotSpot when using 32 or more cores, while Crypt performs better on HotSpot than DiSquawk when using up to 32 cores.

6. CONCLUSIONS

This work discusses the emerging key challenges of implementing a JVM on top of non-cache-coherent architectures with hundreds of cores. We propose new mechanisms, inspired by and improving on the existing literature, to overcome these challenges. We implement part of the proposed mechanisms in DiSquawk [1], the first, to the best of our knowledge, experimental open-source JVM that is capable of

running on top of a non-cache-coherent many-core architecture with hundreds of cores, and evaluate their performance on a FPGA-prototype, 512-core processor. Our measurements show that our proposed JVM design is reasonable and efficient even on limited, prototype hardware.

Acknowledgments.

We would like to thank Christi Symeonidou and the anonymous reviewers for the valuable reviews and constructive feedback. We also acknowledge the support of the *GreenVM* project on Energy-Efficient Runtimes for Scalable Multicore Architectures (project #1643), co-funded by the European Social Fund (ESF) and Greek National Resources.

7. REFERENCES

[1] https://github.com/CARV-ICS-FORTH/disquawk.

[2] U. A. Acar, A. Charguéraud, and M. Rainey. Scheduling Parallel Programs by Work Stealing with Private Deques. In *PPoPP*, 2013.

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

[4] Y. Aridor, M. Factor, and A. Teperman. cJVM: A Single System Image of a JVM on a Cluster. In *ICPP*, 1999.

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

[6] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, 1999.

[7] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

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

[9] D. Chase and Y. Lev. Dynamic Circular Work-stealing Deque. In *SPAA*, 2005.

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

[11] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable Work Stealing. In *SC*, 2009.

[12] Y. Durand, P. Carpenter, S. Adami, A. Bilas, D. Dutoit, A. Farcy, G. Gaydadjiev, J. Goodacre, M. Katevenis, M. Marazakis, E. Matus, I. Mavroidis, and J. Thomson. EUROSERVER: Energy Efficient Node for European Micro-Servers. In *DSD*, 2014.

[13] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC*, 2010.

[14] Y. hun Eom, S. Yang, J. C. Jenista, and B. Demsky. DOJ: Dynamically Parallelizing Object-Oriented Programs. In *PPoPP*, 2012.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.

[16] D. Lea. The JSR-133 cookbook for compiler writers, 2008.

[17] S. Lyberis, G. Kalokerinos, M. Lygerakis, V. Papaefstathiou, D. Tsaliagkos, M. Katevenis, D. Pnevmatikatos, and D. Nikolopoulos. Formic: Cost-efficient and Scalable Prototyping of Manycore Architectures. In *FCCM*, 2012.

[18] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, 2005.

[19] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why On-chip Cache Coherence is Here to Stay. *Commun. ACM*, 55(7), 2012.

[20] R. McIlroy and J. Sventek. Hera-JVM: A Runtime System for Heterogeneous Multi-core Architectures. In *OOPSLA*, 2010.

[21] L. G. Menezo, V. Puente, and J. A. Gregorio. The Case for a Scalable Coherence Protocol for Complex On-chip Cache Hierarchies in Many Core Systems. In *PACT*, 2013.

[22] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In *PGAS*, 2011.

[23] S.-c. Multiprocessor, A. Noll, A. Gal, and M. Franz. CellVM: A homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor. *Workshop on Cell Systems and Applications*, 2008.

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

[25] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, et al. The design and implementation of a first-generation CELL processor–a multi-core SoC. In *ICICDT*, 2005.

[26] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java™on the Bare Metal of Wireless Sensor Devices: The Squawk Java Virtual Machine. In *VEE*, 2006.

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

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

[29] F. S. Zakkak and P. Pratikakis. JDMM: A Java Memory Model for Non-cache-coherent Memory Architectures. In *ISMM*, 2014.

[30] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale. Hierarchical load balancing for Charm++ applications on large supercomputers. In *ICPPW*, 2010.

[31] W. Zhu, C.-L. Wang, and F. C. M. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *CLUSTER*, 2002.