

Space-Based Multi-Core Programming in Java

Stefan Gudenkauf

OFFIS - Institute for Computer Science, Escherweg 2,
26127 Oldenburg, Germany
Software Engineering Group, Department of Computer
Science, Christian-Albrechts-Universität zu Kiel,
Christian-Albrechts-Platz 4, 24118 Kiel, Germany
stefan.gudenkauf@offis.de

Wilhelm Hasselbring

Software Engineering Group, Department of Computer
Science, Christian-Albrechts-Universität zu Kiel,
Christian-Albrechts-Platz 4, 24118 Kiel, Germany
wha@informatik.uni-kiel.de

Abstract

Multi-core processors require programmers to exploit concurrency in software as far as possible. Unfortunately, our current concurrency abstractions make multi-core programming harder than necessary because we have to reduce unintended non-determinism on a very low level of abstraction, for instance via synchronisation mechanisms. In this experience paper we analyse if space-based systems can mitigate multi-core programming in the Java programming language and present the Procol programming model that introduces the space-based choreography of active components, which internally orchestrate fine-grained workflow activities. The main contributions are (1) the Procol programming model, (2) benchmark results of the scalability of different tuple space implementation techniques that we evaluated on different multi-core architectures, (3) benchmark results of the scalability of two equivalent Mandelbrot applications for best-case measurements – one implemented with the standard Java thread model, one with our Procol programming model. The conclusions drawn from these experiments are (1) tuple space data structures that scale reasonably well can be provided, (2) the performance overhead that Procol imposes is at least for the considered application a reasonable trade-off for the ease of programming provided for multi-core architectures.

Categories and Subject Descriptors D.1.3 [PROGRAMMING TECHNIQUES]: Concurrent Programming—Parallel programming

General Terms Space-Based Systems, Multi-Core Programming, Java

Keywords Space-Based Systems, Multi-Core, Java

1. Introduction

With the advent of multi-core processors in the consumer market in 2005, parallel systems became a commodity [1]. Industry today is relying on hyper-threading and increasing processor count since physical limitations impede further performance gains that are based on increasing clock speed and optimising execution flow [2]. These new performance drivers make it necessary to explicitly

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

PPPJ '11, August 24–26, 2011, Kongens Lyngby, Denmark.
Copyright © 2011 ACM 978-1-4503-0935-6...\$10.00

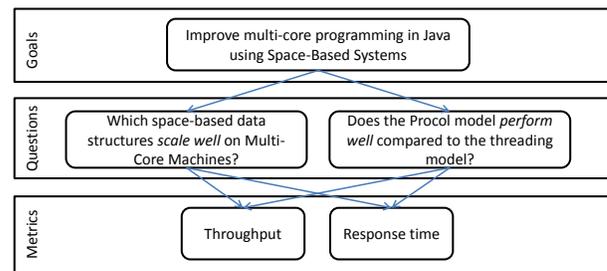


Figure 1. The quantitative questions and metrics considered in this paper according to the Goal Question Metric approach [5].

consider concurrency. The threading model can be regarded as the predominant model for general-purpose parallel programming. It is highly non-deterministic and requires software developers to cut away unwanted non-determinism by means of synchronisation [3]. The high number of possible interleaving thread instructions makes it very difficult to reason about the actual behaviour of a threaded application. This can easily render programming languages less appropriate for concurrent programming since it favours small syntactical modifications over semantic ones, as it is the case with the *synchronized* keyword in the Java language. However, concurrency now has to be exploited in applications of all kinds and domains. The challenge is not solely ultimate software performance, but to provide a convenient and scalable way to participate in the new performance drivers in general.

In this paper we analyse how space-based systems can mitigate multi-core programming in Java. We present a variant of space-based systems that differentiates between the *choreography* of multiple concurrent process components and the *orchestration* of fine-grained activities within a single process component. This variant, named Procol¹, fosters the separation of concurrency-related concerns from domain-specific computational activities to enable the model-driven software development of space-based concurrent systems [4]. We investigate two quantitative questions: (1) Which space-based data structures *scale well* on multi-core machines? (2) Does the Procol model show a *reasonable performance* compared to the threading model? These questions are answered by observing the throughput and the response time of space-based systems in several benchmarks, see Fig. 1.

In Sec. 2 we introduce space-based systems and argue about properties that encourage their use for multi-core programming.

¹PROcess COordination Language, <http://procol.sourceforge.net/>

Section 3 discusses how space-based systems and workflows can be combined to form a holistic model of coordination. In Sec. 4 we present the Java implementation of Procol as an example of such a holistic programming model: Section 4.1 presents our extensions to the third-party tuple space framework LighTS², and Sec. 4.2 presents a component orchestration layer on top of the extended LighTS framework. In Sec. 5 we investigate the quantitative questions shown in Fig. 1: First, we present the metrics and the experimental configuration of the benchmarks. Second, we evaluate the scalability of several space-based data structures that we have implemented (question 1). Third, we evaluate two equivalent applications that compute the Mandelbrot set concurrently and that show it on the complex plane. The first is implemented using Procol, and the second using Java’s standard threading model (question 2). Finally, we argue about the limitations of the conducted experiments and present future work regarding the performance analysis. Related work is discussed in Sec. 6. Section 7 concludes the paper.

2. Space-Based Systems for Multi-Core Programming

Space-based systems employ a data-sharing approach that is based upon object orientation and generative communication [6, 7]. While *object-orientation* provides composable, self-contained entities that are protected by their interfaces, *generative communication* allows communication partners to be uncoupled from each other. This idea was originally introduced by the Linda coordination model [8]. In space-based systems, spaces are shared by active components. These communicate by publishing data objects into the spaces by *out* operations and by reading or consuming data objects from them by *read* or *in* operations. Also, components can wait until a data object to be read or consumed actually has been inserted into the respective space. The decision which data object is to be read or consumed is made by a template specification on the side of the reader or consumer. Matching that template to data objects is performed by the space itself. The data objects are usually implemented in the form of tuples being ordered collections of data items. While tuples consist of *actual fields* (i. e., typed values), templates may consist of actual and *formal fields* (i. e., place-holders for typed values). A tuple is regarded as matching a given template if their arities are equal, and if each actual field in the tuple matches an actual field of the same type and value, or a formal field of the same type in the template. Although typically considered as a framework to coordinate components in open and distributed environments, space-based systems show appealing features for multi-core programming:

1. Space-based systems assume explicit indirect coordination amongst components. Instead of requiring software developers to cut away unwanted non-determinism by means of fine-grained synchronisation, they introduce non-determinism on a higher level of abstraction by the space operations.
2. While residing in spaces, data objects are immutable. To modify a data object, components must explicitly remove it from the space, modify it, and reinsert it. Data objects can never encounter conflicts or inconsistencies when multiple components attempt to modify them, thus eliminating undesirable situations such as lost updates.
3. Space-based systems ease concurrent programming by abstracting from the location of components in space and time. Components do not necessarily have to exist at the same time, and always remain anonymous to each other.

4. Space-based systems abstract from the computational model and are orthogonal to widespread general purpose programming languages such as C++, C#, and Java, since they are based on the computational model in principle.

We summarise the use of space-based systems for concurrent programming to the following formula:

$$\text{Shared spaces} + \text{Immutability of space-residing data objects} = \text{Manageable concurrency}$$

3. Combining Space-Based Systems and Workflows

A coordination model defines the interaction of active and independent components, whereas the interaction is subject to certain coordination laws that specify how the components coordinate themselves through the given coordination media [9]. The conceptual distinction between computation and coordination is shown by Wegner by discussing their different expressive power in terms of abstract machines: Computation and coordination are two orthogonal dimensions for programming languages [10]. His work suggests that models of coordination could have a significant impact on the engineering of complex systems, as it is shown today with coordination-based programming-in-the-large approaches that are employed in industry and academia, see for example [11]. Most modern coordination models coordinate fine-grained components denoted as *activities* within single composite components denoted as *workflows*. Coordination is thereby specified by control flows that define the order of activities by small pieces of data regarded as control information. This kind of coordination modelling is called *orchestration* [12]. Space-based systems, on the other side, represent a class of coordination models that emphasises the coordination between components using data flows: Component activity is defined by the availability of passive data structures as a pre- or postcondition. Consequently, we identify space-based coordination as a kind of *choreography*, since the interaction between black-box components is emphasised before the fine-grained coordination of activities in a single component [12]. Both, workflow models and space-based systems, represent an abstraction of complex system behaviour that is orthogonal to domain-specific computation. Both emphasise different aspects of coordination, namely orchestration and choreography. Therefore we suggest these models to be combined into one single coordination model. This combination has two fundamental advantages: (1) Space-based choreography decouples components in space and time: Components are anonymous to each other and do not have to exist at the same point of time in order to operate. (2) Orchestration separates fine-grained activities of domain-specific computation from the space-based coordination primitives (*out*, *in*, *read*, and their variants).

4. Implementation

In this section we present the implementation of the Procol programming model. This model is based on the combination of the space-based system model for the loosely coupled choreography of active components, and the workflow model for the orchestration of fine-grained activities within components.

4.1 Space-Based Data Structures in Java

There are several existing Java implementations of space-based systems which are mostly oriented towards distributed computing³, see Sec. 6. We selected LighTS as a basis to examine the scalability of space-based systems on multi-core machines since it pro-

²<http://lights.sourceforge.net/>

³<http://procol.sourceforge.net/dokuwiki/doku.php?id=list-tuplespaces> presents various space-based systems

```

public interface ITupleSpace {
    String getName();
    void out(ITuple tuple);
    // publishes a tuple
    void outg(ITuple[] tuples);
    // bulk publish
    ITuple in(ITuple template);
    // blocking consume
    ITuple inp(ITuple template);
    // non-blocking consume
    ITuple[] ing(ITuple template);
    // bulk consume (non-block)
    ITuple rd(ITuple template);
    // blocking read
    ITuple rdp(ITuple template);
    // non-blocking read
    ITuple[] rdg(ITuple template);
    // bulk read (non-block)
    int count(ITuple template);
    // num matching tuples
}

```

Listing 1. Interface of the tuple space framework.

vides appealing features: It provides a minimal, non-distributed (local) implementation and an adaptation layer that allows programmers to write applications using its tuple space interface, and to extend them by providing different tuple space implementations. We refer to the original authors for an extensive overview of the LighTS framework [13]. We extended the framework as follows but retained its original interface (see Listing 1). Tab. 1 summarises the synchronisation mechanisms we used in our tuple space extensions.

1. We exchanged the original synchronised Vector-based tuple space by a version based on `java.util.LinkedList`. Synchronising the tuple space operations is achieved using the `synchronized` keyword. This tuple space (**TS**) is regarded as a straight-forward reference implementation.
2. We added a *CopyOnWriteTupleSpace* (**CoWTS**) based on `java.util.concurrent.CopyOnWriteArrayList`. Synchronising the blocking tuple space operations is achieved using the `synchronized` keyword. The non-blocking read operation is not synchronised since it is allowed to return a null value and operates on a local copy of the underlying array of the `CopyOnWriteArrayList`⁴. `inp` must be synchronised. Otherwise, possibly interleaving `inp` operations can consume a tuple that conceptionally has already been consumed by another component (defective reduplication). The CoWTS is expected to perform well for read operations, but writes should add a performance penalty since mutative operations on the `CopyOnWriteArrayList` internally make a fresh copy of the underlying array.
3. We added a *ReadWriteLockedTupleSpace* (**RWLTS**) based on `LinkedList`, and `java.util.concurrent.locks.ReentrantReadWriteLock` and `java.util.concurrent.locks.Condition` for synchronisation. The RWLTS should show the effect of differentiating read locks from write locks in contrast to the TS, since `ReentrantReadWriteLock` allows multiple read locks to be held on the `LinkedList`.

⁴“The iterator provides a snapshot of the state of the list when the iterator was constructed. No synchronization is needed while traversing the iterator. The iterator does NOT support the remove method.” (Java SE 1.5 API)

4. We added a *ReadWriteLockedCopyOnWriteTupleSpace* (**RWL-CoWTS**) based on `CopyOnWriteArrayList`. Synchronisation is achieved using `ReentrantReadWriteLock` and `Condition`. The non-blocking read operations are not synchronised.
5. We added a *ConcurrentHashMapTupleSpace* (**CHMTS**) based on `java.util.concurrent.ConcurrentHashMap`. Synchronising the tuple space operations is achieved using `synchronized`. Tuples are stored as values and keys are generated upon insertion as simple unique identifiers (UUIDs). The count operation does not require synchronisation.
6. We added a *ReadWriteLockedConcurrentHashMapTupleSpace* (**RWLCHMTS**) based on `ConcurrentHashMap`. Synchronisation is achieved using `ReentrantReadWriteLock` and `Condition`. Tuples are stored as values and keys are generated upon insertion as simple unique identifiers (UUIDs).
7. We added a *PreselectingReadWriteLockedConcurrentHashMapTupleSpace* (**PRWLCHMTS**) based on `ConcurrentHashMap`. Synchronisation is achieved using `ReentrantReadWriteLock` and `Condition`. Tuples are stored as values and keys are generated upon insertion. A generated key consists of tuple arity (i. e., the number of fields), an encoded representation of field types, and a generated unique id (UUID). This enables to preselect tuple *candidates* before the actual matching by filtering with the first two matching rules *arity* and *type equivalence*. Filtering relies on splitting the keys to compute the candidates. Therefore, we suppose that this implementation can show a decreased performance in contrast to the other data structures. This is especially likely when there are many similar tuples in the space, since both the key set as well as the candidate set must be iterated and examined.
8. We added a benchmark suite whose design is based on a *BenchmarkDriver* that executes a set of *IBenchmark* implementations. These benchmarks instantiate a set of concurrent workload simulators that implement the `Callable` interface. The simulators either simulate work given a fixed run-time for throughput measurement (i. e., implement *AbstractFixedRuntimeWorkloadSimulator*) or given a fixed number of workload iterations to measure the response time (i. e., implement *AbstractFixedIterationCountWorkloadSimulator*). The operations to be simulated in each workload iteration are implemented in the method `simulateWorkloadIteration`, which is required by the abstract superclass of the respective simulator. The output of the benchmark suite is in XML format.
9. We added a tool to analyse the log files of the benchmark suite. The tool computes and plots the arithmetic mean and the 95% confidence interval for a measurements series that describes the behaviour of a tuple space. The confidence intervals are computed using the Central Limit Theorem and the sample standard deviation if the number of measurements is large ($n \geq 30$), and the Student’s t-distribution when the number of measurements is small ($n < 30$). We refer to the work of Georges et al. as an excellent reading on the importance of confidence intervals for statistically rigorous Java performance evaluation [14]. The analyser can either be used as a log appender for the benchmark suite at run-time or as a standalone tool to analyse the XML log files of the benchmark suite separately.
10. We restructured package organisation, and added Maven build integration⁵, additional tests, and documentation. We called the extended LighTS framework *procol-tuplespace*, the benchmark suite *procol-tuplespace-benchmark*, and the performance analysis tool *procol-benchmark-logging*.

⁵<http://maven.apache.org/>

Operation	TS	CoWTS	RWLTS	RWLCoWTS	CHMTS	RWLCHMTS	PRWLCHMTS
<i>out</i>	•	•	⊗ <i>w</i>	⊗ <i>w</i>	•	⊗ <i>w</i>	⊗ <i>w</i>
<i>outg</i>	•	•	⊗ <i>w</i>	⊗ <i>w</i>	•	⊗ <i>w</i>	⊗ <i>w</i>
<i>in</i>	•	•	⊗ <i>w</i>	⊗ <i>w</i>	•	⊗ <i>w</i>	⊗ <i>w</i>
<i>inp</i>	•	•	⊗ <i>w</i>	⊗ <i>w</i>	•	⊗ <i>w</i>	⊗ <i>w</i>
<i>ing</i>	•	•	⊗ <i>w</i>	⊗ <i>w</i>	•	⊗ <i>w</i>	⊗ <i>w</i>
<i>rd</i>	•	•	⊗ <i>r</i>	⊗ <i>r</i>	•	⊗ <i>r</i>	⊗ <i>r</i>
<i>rdp</i>	•	○	⊗ <i>r</i>	○	•	⊗ <i>r</i>	⊗ <i>r</i>
<i>rdg</i>	•	○	⊗ <i>r</i>	○	•	⊗ <i>r</i>	⊗ <i>r</i>
<i>count</i>	•	○	⊗ <i>r</i>	○	○	○	○

Table 1. Overview of space operation synchronisation used in the different space implementations: • denotes synchronisation with the synchronized keyword, ⊗ denotes the use of ReentrantReadWriteLock and Condition (*r* for readLock, *w* for writeLock), and ○ denotes that no synchronisation is used for the respective space operation.

```

public interface IComponent
  extends Callable<Boolean> {
  public abstract Boolean call()
  throws ComponentInvocationException; }

public interface ICompositeComponent
  extends IComponent {
  public void add(IComponent component);
  public void remove(IComponent component);
  public IComponent getComponent(int i); }

public interface IMultipleInstanceComponent
  extends IComponent {
  public void setInvokee(Class<? extends
  AbstractComponent> invokee, int invokeCount); }

public AbstractComponent(HashMap<String,
  ITupleSpace> spaces, boolean isReporting,
  String name)

```

Listing 2. Interfaces of the component orchestration layer.

4.2 Component Orchestration Layer

In Sec.3 we discussed the combination of space-based systems and workflows. We added a component orchestration layer *procol-components* to the extended LighTS framework to show the feasibility of this combination. This layer fosters orchestration by introducing the interfaces *IComponent*, *ICompositeComponent*, and *IMultipleInstanceComponent* (Listing 2). While the former extends `java.util.concurrent.Callable`, the latter two extend *IComponent*. By convention, *IComponents* are only allowed to (a) communicate with each other indirectly using spaces, (b) concurrency is implemented by `java.util.concurrent.ExecutorService`, and (c) domain-specific computational *activities* are implemented using parameter-less methods that use the private fields of the component.

We consider the following components: (1) *AbstractComponent* is a base class for the following components and for custom primitive component implementations. It requires components to expect a map of known tuple spaces upon instantiation, a name, and a decision if the component should report its execution time. (2) *ParallelInvocationCompositeComponent* and *SequentialInvocationCompositeComponent* implement *ICompositeComponent* and concurrently/sequentially invoke implementations of *IComponent*. The *IComponents* to be invoked are handled via the methods `add`, `remove`, and `getComponent`. (3) *MultipleInstanceParallelComponent* and *MultipleInstanceSequentialComponent* implement

MultipleInstanceComponent and concurrently/sequentially invoke a number of instances of a *single* *IComponent*. The *IComponent* to be invoked is set by the method `setInvokee`.

5. Performance Evaluation

In this section we present a comparative evaluation of two representative scenarios: the scalability of different tuple space data structures, and the scalability of a Mandelbrot application that is implemented with the Procol programming model compared to an equivalent application implemented with the standard Java threading model.

5.1 Performance Metrics and Experimental Configuration

The performance metrics used in this paper focus on throughput and response time. Following Fiedler et al. [15], we measured the scalability of a space-based system S_i in terms of the throughput $X_{bench_q}(S_i, O_{bench})$ in relation to the number of workload simulators q given a fixed response time T_{bench_q} , or as the response time T_{bench_q} given a fixed number of completed space operations $L_{bench_q}(S_i, O_{bench})$, see equation (1). $bench$ denotes the respective benchmark and O_{bench} denotes the (primitive or compound) operations to be completed. Subsequently, we define the scalability of a space-based data structure as presented in Definition 1.

$$X_{bench_q}(S_i, O_{bench}) = \frac{L_{bench_q}(S_i, O_{bench})}{T_{bench_q}} \quad (1)$$

To minimise interferences of individual benchmarks the benchmark driver pauses for 5,000 ms, then invokes the Java Garbage Collector, and pauses again for 5,000 ms between benchmark runs. We executed 12 runs for each benchmark, where the first two are excluded as warm-up runs to reduce the effects of bytecode interpretation and just-in-time compilation [16]. We use the arithmetic mean and the 95% confidence interval for the remaining ten results. The confidence intervals allows us to address two questions: Do different tuple space implementations show significantly different behaviour or not⁶? How large is the performance variation of the individual measurements for a single tuple space implementation? Benchmarks that implement *AbstractFixedRuntimeWorkload-Simulator* executed workload simulation iterations for 10,000 ms. The benchmarks were conducted on four machines M0 to M3. M1 and M2 are identical machines with different operating systems and Java environments. M0, M2 and M3 use the same operating system.

⁶ If the confidence intervals for two implementations do not overlap, the performance difference between the two is significant. Else, it is most likely caused by random performance variations in the system under measurement [14].

DEFINITION 1 (Scalability). A space-based system S_i is scalable, if

$$\left(R_{throughput_{bench}}(S_i, O_{bench}, n, m) = \frac{X_{bench_n}(S_i, O_{bench})}{X_{bench_m}(S_i, O_{bench})} \leq 1 \wedge T_{bench_n} = \dots = T_{bench_m} = const \right) \vee$$

$$\left(R_{resptime_{bench}}(S_i, O_{bench}, n, m) = \frac{T_{bench_n}}{T_{bench_m}} \geq 1 \wedge L_{bench_n}(S_i, O_{bench}) = \dots = L_{bench_m}(S_i, O_{bench}) = const \right)$$

M0 and M2 also use the same Java Version. This means that differences in the execution behaviour of M0 and M2 must be a matter of hardware, and differences between M1 and M2 must be a matter of different software stacks:

1. **(M0)** T6340, 2xUltraSparcT2+ 8core 1.4Ghz, 16x4GB FB DIMM, Solaris10, JAVA Version 1.6.0.21, Java(TM) SE Runtime Environment (build 1.6.0.21-b06), Java HotSpot(TM) Server VM (build 17.0-b16, mixed mode)
2. **(M1)**: X6270, 2xIntel Xeon E5540 4core 2.53GHz, 12x2GB DDR3-1066MHz ECC, Debian Lenny, Java Version 1.6.0.0, OpenJDK Runtime Environment (build 1.6.0.0-b11), OpenJDK 64-Bit Server VM (build 1.6.0.0-b11, mixed mode)
3. **(M2)** X6270, 2xIntel Xeon E5540 4core 2.53GHz, 12x2GB DDR3-1066MHz ECC, Solaris 10, Java Version 1.6.0.16, Java(TM) SE Runtime Environment (build 1.6.0.16-b01), Java HotSpot(TM) Server VM (build 14.2-b01, mixed mode)
4. **(M3)** X6240, 2xAMD Opteron 2384 4core 2.7GHz, 8x2GB DDR2-667, Solaris 10, Java Version 1.5.0.20, Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0.20-b02), Java HotSpot(TM) Server VM (build 1.5.0.20-b02, mixed mode)

5.2 Benchmark Results

Figures 2 and 3 show the results of the different benchmarks. These do not contain the measurements for the PRWLCHMTS since it showed catastrophic response time behaviour. For example, in previous test runs on M0 the PRWLCHMTS took an average of 2,013,632.33 milliseconds (33.56 minutes) to complete the *ConsumerProducerBenchmark* using 2 workload simulators. A possible reason is that there are very many similar tuples in the space so that both the key set as well as the candidate set contain all entries of the tuples so far produced. Both must be iterated and examined for matching. However, the response time of the PRWLCHMTS is about magnitudes worse than that of the RWLCHMTS so that we must consider the possibility that the implementation of the PRWLCHMTS contains defects that we have not yet identified. Deemed as impractical in its current implementation, we excluded the PRWLCHMTS from the discussion of benchmarks.

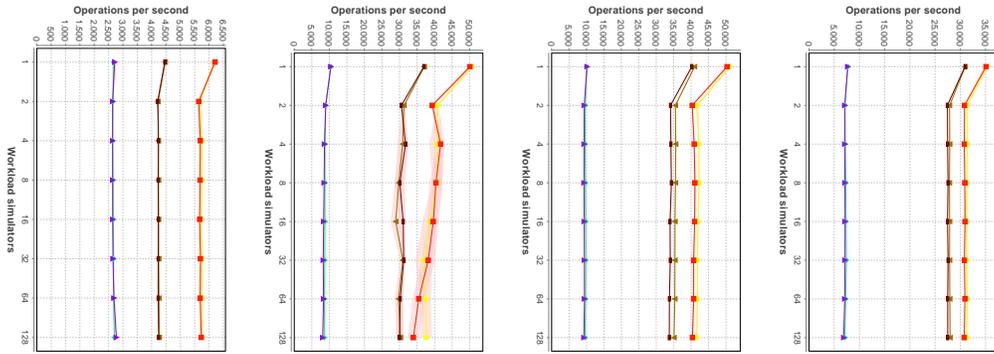
OutBenchmark Each workload iteration of the workload simulator publishes a simple tuple in the tuple space via the out operation. Regarding the *OutBenchmark*, we see a small drop of throughput between the use of 1 and 2 workload simulators on all machines. For example, $R_{throughput_{Out}}(S_{TS}, O_{Out}, 1, 2) = 1.25$ on M2. Except for that, all six tuple space implementations scale well on M0, M2 and M3. On M1, the throughput of the TS increases slightly from 2 to 4 workload simulators ($R_{throughput_{Out}}(S_{TS}, O_{Out}, 2, 4) = 0.94$) and then again continues to decrease slightly ($R_{throughput_{Out}}(S_{TS}, O_{Out}, 4, 128) = 1.23$). Additionally, we can see that the 95% confidence intervals of the TS and the RWLTS, and of the CHMTS and the RWLCHMTS mostly overlap. This means that the behaviour of the traditionally synchronised implementations is not significantly different from the read-write-locked versions. In general, the CoWTS and the RWLCoWTS show a much lower performance than the TS and the RWLTS due to the copy-on-write behaviour of the underlying CopyOnWriteAr-

rayList. Also, the CHMTS and the RWLCHMTS show a lower performance than the TS and the RWLTS.

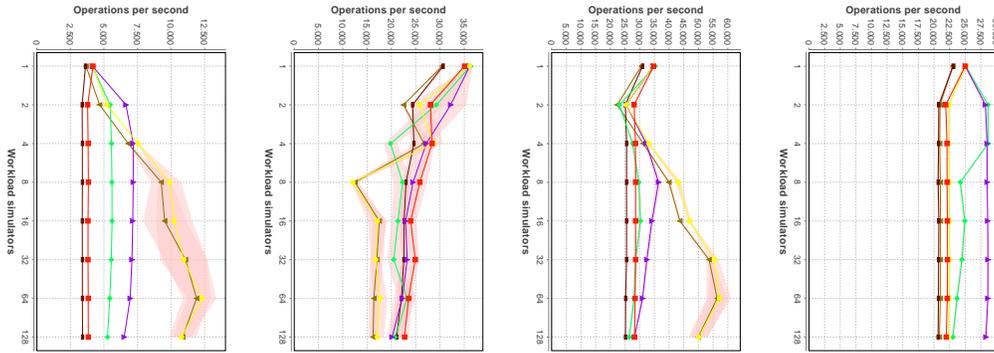
OutInBenchmark Each workload iteration checks the availability of a certain tuple with a non-blocking rdp, and if there is no result, it produces one with out. Else, it consumes the tuple with the non-blocking inp operation. The RWLTS and the RWLCHMTS perform surprisingly well on M0 and M2 with increasing performance until reaching an observable peak at 64 workload simulators, then levelling off. A possible reason is the differentiation of read and write locks that let both excel in reading tuples. The decreasing performance after the peak can be explained by effects of write locking when the number of concurrent simulators increases. We can also see a large performance variation for both implementations on M0, which starts when 4 workload simulators are used, as indicated by their confidence intervals. When 8 workload simulators or more are used both implementations show similar behaviour since their confidence intervals begin to overlap. On M1 we see a strong decrease in throughput and a fluctuating behaviour of the RWLTS and the RWLCHMTS. This is especially the case between 4 and 8 workload simulators. For example, $R_{throughput_{OutIn}}(S_{RWLTS}, O_{OutIn}, 4, 8) = 2.38$ and $R_{throughput_{OutIn}}(S_{RWLTS}, O_{OutIn}, 8, 16) = 0.70$. Using 16 simulators or more stabilises throughput. A possible explanation is contention with OS and VM threads. As their overlapping confidence intervals indicate, the RWLTS and the RWLCHMTS show similar behaviour. In contrast, the RWLCoWTS shows significantly different behaviour. This is likely due to the fact that the RWLCoWTS does not have to synchronise the rdp operation. All in all, throughput decreases from 1 to 128 workload simulators for all implementations on M1. On machine M2, the overall throughput of the RWLTS and the RWLCHMTS is much higher than on M1, and the other implementations scale better than on M1. Since M2 is hardware-wise similar to M1, we can identify the software stack of M1 as inferior compared to that of M2. On M3, the CoWTS and the RWLCoWTS scale well, with increasing throughput between 1 and 2 workload simulators. While the CoWTS scales constantly when using more than 2 simulators, the throughput of the RWLCoWTS decreases between 4 and 8 simulators before it stabilises. Again, a possible explanation is OS and VM thread contention. The other implementations, on the contrary, show a small drop of throughput between the use of 1 and 2 workload simulators ($R_{throughput_{OutIn}}(S_{RWLTS}, O_{OutIn}, 1, 2) = 1.12$) and a constant scalability from 2 to 128 simulators. Except for that, the TS and the CHMTS scale well on M0, M2, and M3. They show a worse performance than the CoWTS and the RWLCoWTS since the former employ conventional synchronisation for the space operations, and the latter do not employ any synchronisation for nonblocking read operations at all. Finally, the CHMTS and the RWLCHMTS show a slightly worse performance than the TS and the RWLTS. A possible reason is the additional synchronisation in the ConcurrentHashMap of the CHMTS and the RWLCHMTS in contrast to the TS and the RWLTS, which internally use the unsynchronised LinkedList.

OutReadBenchmark Each workload iteration checks the availability of a certain tuple with a non-blocking rdp and if there is no result, it produces one with out. Since the first workload iteration produces a tuple, this is a non-blocking read test. The

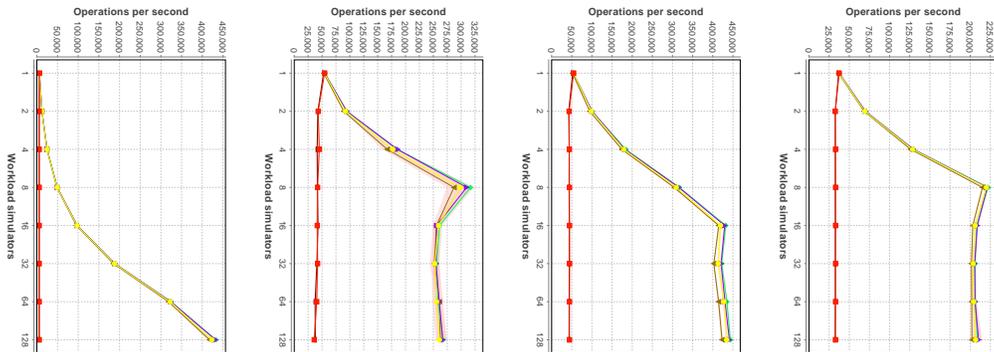
OutBenchmark @ 10,000 ms (M0 to M3 from left to right)



OutInBenchmark @ 10,000 ms (M0 to M3 from left to right)



OutReadBenchmark @ 10,000 ms (M0 to M3 from left to right)



ConsumerProducerBenchmark @ 10,000 operations per workload simulator (M0 to M3)

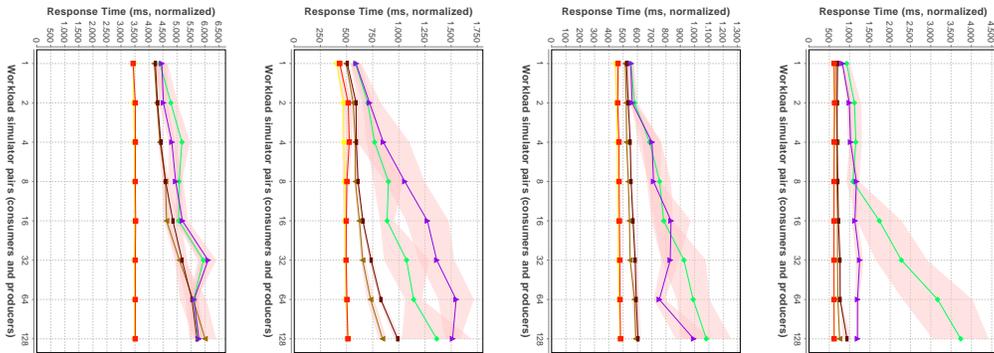


Figure 2. Results of the space operations benchmarks on machines M0 to M3 from left to right.

TS and the CHMTS scale constantly since their operations are synchronised with the `synchronized` keyword. The CoWTS, the RWLTS, the RWLCoWTS, and the RWLCHMTS perform exceptionally well, with increasing performance until 8 workload simulators on M1 and M3, and until 16 workload simulators on M2 ($R_{throughput_{OutRead}}(S_{RWLCHMTS}, O_{OutRead}, 1, 16) = 0.13$). Then their performance turns into a near-constant throughput on a high level of performance ($R_{throughput_{OutRead}}(S_{RWLCHMTS}, O_{OutRead}, 16, 128) = 0.98$). An obvious explanation is that concurrency can not further be exploited for real concurrency. On M1, the decrease of throughput of the CoWTS, the RWLTS, the RWLCoWTS, and the RWLCHMTS between 8 and 16 workload simulators is much greater than on M2. Again, we can identify the software stack of M1 as inferior compared to that of M2 since both are hardware-wise similar to each other. On M0, throughput continues to increase up to 128 workload simulators, the exact number of logical processing elements of M0 (2 processors with 8 cores each, which each support 8 logical processing elements). The reason for the overall behaviour of the implementations is that the OutReadBenchmark is essentially a non-blocking read benchmark. The CoWTS's and the RWLCoWTS's CopyOnWriteArrayList can excel since their `rdp` operation is not synchronised. The differentiation of read and write locks suggest that the RWLTS and the RWLCHMTS also excel at read benchmarks. The TS and the CHMTS can not speed up since their `rdp` operation is conventionally synchronised.

ConsumerProducerBenchmark The Benchmark starts a number of n consumer (blocking `in` per workload iteration) and n producer simulators (out per workload iteration), where n is the number of workload simulators specified by the respective command line parameter. We observe constant and almost identical response time behaviour for the TS and the RWLTS, and the CHMTS and the RWLCHMTS given a fixed number of operations to be performed. Thereby, the CHMTS and the RWLCHMTS show a lesser and less scaling performance than the TS and the RWLTS. Again, an explanation is the additional synchronisation in the ConcurrentHashMap of the CHMTS and the RWLCHMTS. The CoWTS and the RWLCoWTS show less scaling response time behaviour and noticeable performance variations in the measurements as indicated by their confidence intervals. Most striking, on M3 the response time of the RWLCoWTS starts to increase continuously from 8 to 128 workload simulators in contrast to the other implementations. Surprisingly, the conventionally synchronised CoWTS scales well on M3 although we would have expected a similar performance penalty for both implementations caused by the copy-on-write behaviour of the underlying CopyOnWriteArrayList.

AgeingBenchmark We exemplarily examined the effects of tuple space ageing on the results of the ConsumerProducerBenchmark, see Fig. 3. The benchmarks used 8 producer and 8 consumer workload simulators as the degree of concurrency. We pre-populated the tuple spaces with different numbers of tuples that contained a single string value each. We used different values for each string tuple. The benchmark indicates worst-case performance since the matching algorithm for tuple lookup iterates the tuple space in a straightforward manner. Performance increased by about 741.97% from a zero pre-population to a pre-population of 12,000 tuples regarding the RWLTS on M1, for instance. The reason is the primitive matching algorithm. It has to iterate over all the string tuples in the aged space each time a relevant tuple should be retrieved. A more sophisticated matching algorithm can certainly mitigate the effects of ageing up to a certain degree.

General observations We observed a lower overall performance of M0 compared to the other machines. For example, the number of completed space operations per second $X_{Out_1}(S_{TS}, O_{Out})$ is

only about 12.4% of that on M1, as shown in Tab. 2. Except for the AgeingBenchmark, the benchmarks were executed as single VM invocations. Benchmarking with multiple parallel VMs can show different results. Second, the reason for the different behaviour of the space implementations on M1 in contrast to M2 must be a matter of using different operating systems and Java VMs since M1 and M2 are hardware-wise similar. This is especially the case for the OutInBenchmark, in which the performance of the RWLTS and the RWLCHMTS on M1 differs significantly to that on M2. It seems that for non-block `inp` operations, the software stack of M2 provides better and more reliable results than that of M1. We suppose that the OpenJDK VM is not as optimised as its Java HotSpot counterpart. For most of the tuple spaces we observed a noticeable overhead for data structure management. As a consequence, further optimisations of the Procol framework should not only focus on improving the matching algorithm but on reducing space management overhead in general to improve scalability. However, in each of the benchmarks we found tuple space implementations for which our definition of scalability (Definition 1) holds or is only slightly violated. The conclusion drawn from the results is that tuple space implementations that scale *reasonably well* can be provided when the effects of ageing are not yet considered. This is at least the case for multi-core machines with a number of cores that is in the single-digit range, since the machines on which the benchmarks were conducted only have 4 and 8 core processors.

We give the following recommendations which tuple spaces should be used: (1) The CoWTS and the RWLCoWTS represent a reasonable choice when non-blocking read operations are used heavily since they do not need to synchronise these. On the other hand, they do not show a significantly better performance than the RWLTS, and are clearly not advisable for applications that favour blocking `in` and `out` operations over `read`. (2) As a general recommendation, we advise to use the RWLTS over the other implementations since its throughput and response time behaviour scale reliably in most benchmarks, and since it performs very well for applications that heavily depend on read operations. The RWLCHMTS can be regarded as an alternative to the RWLTS. It shows similar reliable behaviour as the RWLTS, but its performance is worse.

5.3 Mandelbrot Application Results

Using the extended LightS framework and the component orchestration layer, we created an application that computes the Mandelbrot set concurrently and shows it on the complex plane. The application represents an application benchmark for massive concurrency since each point of the complex plane can be computed independently. Figure 4 shows an informal overview on the coordination model of the application using the Business Process Model And Notation (BPMN). The activities in the upper lane represent IComponents except for the *configure activity* and the internal activities of the *Presenter* component (marked with *). Firstly, the Mandelbrot's main method executes the *configure activity*. It outs a configuration data object which is constructed from the application's command line parameters. Then, all application-specific components are started concurrently using a *ParallelInvocationCompositeComponent*. These components coordinate themselves along the data flows of the space operations. The *ImageProvider* provides a set of unrendered image slices that are consumed by as many concurrent *Renderer* instances (implemented as a *MultipleInstanceParallelComponent*) as there are image slices in order to produce rendered images. While *Renderer* encapsulates the domain knowledge to compute the Mandelbrot set on the plane, *Presenter* consumes rendered image slices, puts them together to a single image, and saves it into a file. As Fig. 5 shows, the application scales very well on the given machines using the RWLTS until 8 concurrent *Renderer* instances. The figure shows the mean

ConsumerProducerBenchmark @ 8 workflow simulator pairs (10,000 operations per workload simulator, M0 to M3)

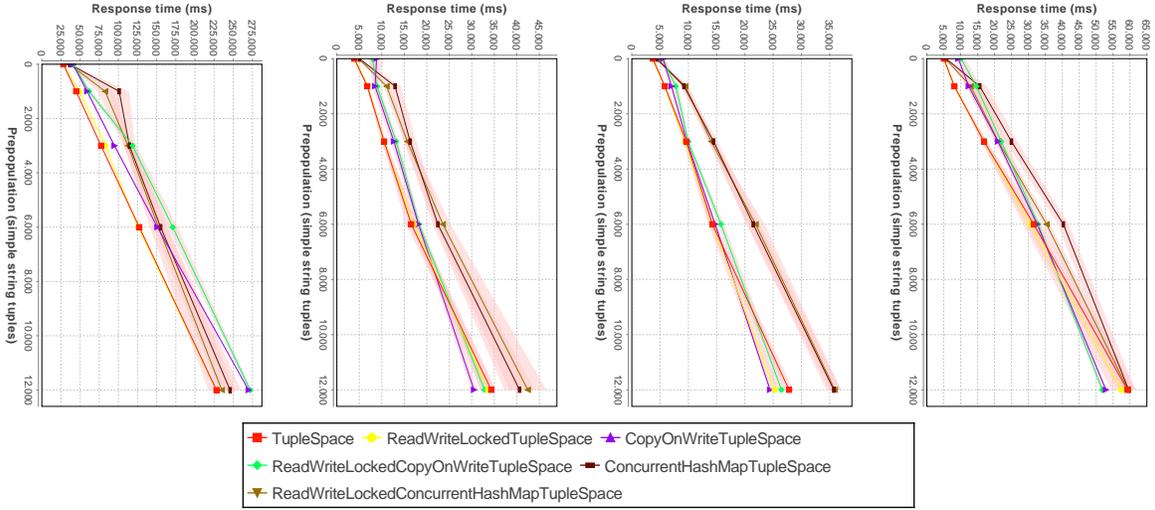


Figure 3. Effects of tuple space ageing on the ConsumerProducerBenchmark.

Performance of TS on M0...	Out $X_{bench_1}(S_{TS}, O_{bench})$	OutIn $(X_{bench_8}(S_{TS}, O_{bench}))$	OutRead	ConsumerProducer $T_{bench_1}(T_{bench_8})$
...compared to M1 :	12.40 (14.08)%	11.93 (14.86)%	11.39 (13.69)%	12.66 (14.32)%
...compared to M2 :	12.35 (13.84)%	12.04 (13.43)%	11.39 (12.96)%	13.54 (13.44)%
...compared to M3 :	17.69 (18.42)%	16.74 (17.39)%	16.63 (17.39)%	17.99 (17.55)%

Table 2. Percentaged results of benchmark runs on M0 compared to M1, M2, and M3 using the TS with 1 (8) concurrent workflow simulators.

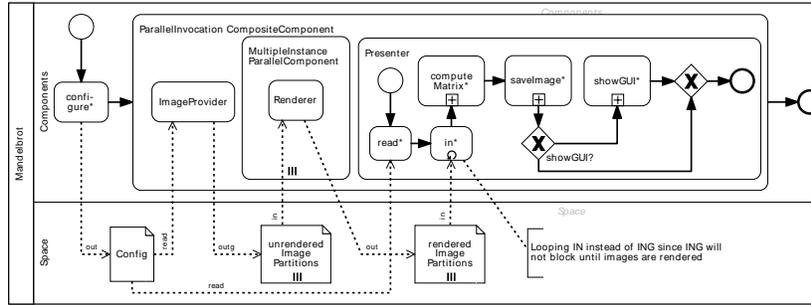


Figure 4. Coordination model of the space-based Mandelbrot application (illustrated using BPMN-like notation): The Presenter component is expanded so that inter-component orchestration is shown (methods computeMatrix, saveImage, showGUI).

of ten measurements after two warm-up runs. Regarding M1, we see that the response time decreased from 2,781.2 ms to 1,806.8 ms ($R_{resptime_{Mandel}}(S_{RWLTS}, O_{Mandel}, 1, 8) = 1.54$) until 8 concurrent *Renderer* instances were used, then began to increase back to 2,561.6 ms using 265 concurrent *Renderers* ($R_{resptime_{Mandel}}(S_{RWLTS}, O_{Mandel}, 16, 256) = 0.71$). Also, we can see that the application scales better on machines M0 and M3 than on M1 and M2 when more than 8 *Renderer* instances are used. A possible reason is that incipient effects of hyperthreading impose a performance penalty while up to using 8 renderers all renderers can be mapped directly to the cores of the two Xeon processors of M1 and M2. We also measured the execution times of each component individually to provide a hint for the increase in response time. As shown in Fig. 5, measuring and logging execution time on component level has virtually no effect on the overall response time

(Procol vs. Procol-Silent). The performance variation of Procol-Silent at 128 *Renderers* on M3 originates from a noticeable outlier in the measurements. The lower half of the figure illustrates that the *Presenter* component represents a constant serial fraction of the overall response time and that the execution time of the *Renderers* decrease as more *Renderer* instances are used. The response time of the *ImageProvider* increases slightly with an increasing number of *Renderer* instances. A possible reason is that it must prepare and publish more image partition tuples the more *Renderer* instances are used. Another explanation is that the *ExecutorService*'s thread management can show a noticeable effect on response time with an increasing number of *Renderer* instances although there is no data to be shared among threads. In each case, however, the response time using 256 *Renderer* instances never exceeded the time elapsed when only one *Renderer* instance was used.

As a reference application, we implemented a version of the Mandelbrot application that is solely based on the standard Java thread model. Its implementation follows the instructions of the project course “Mandelbrot Set Visualization” held by Daniel Tang at the Purdue University in 2009⁷. Since there is no data to be shared among threads, there is no synchronisation needed and the application measurements can be regarded as a best-case measurements. Compared to this reference application, we see that the space-based solution does not perform too bad. For example, when using 8 (256) concurrent `Renderer` instances, the response time of the Procol implementation takes 1.33 (1.81) times longer to execute on M0, 1.32 (1.85) times longer on M1, 1.32 (2.05) times longer on M2, and 1.31 (1.44) times longer on M3. In no case were the measurements about an entire order of magnitude different (which we actually regarded as a KO criterion). We conclude that the performance overhead that the Procol programming model imposes is at least for the Mandelbrot application a reasonable trade-off for the benefits that the Procol programming model provides.

5.4 Limitations and Future Work

Any empirical study like ours is vulnerable to certain threats to validity: (1) The benchmarks and the Mandelbrot applications have only been executed on multi-core machines and the performance on many-cores is yet to prove. As network-on-chip many-cores will be significantly different from multi-cores [17], new implementation strategies will be required, which likely will resemble implementation techniques for distributed tuple spaces. On the other hand, many-core chips based on Virtual Shared Memory (VSM) provide a likely target for space-based programming, see Sec. 6. (2) Only a primitive ageing technique has been applied to the tuple space implementations in order to yield worst-case performance measurements. (3) The Mandelbrot application represents a reference example only for application-specific best-case measurements. Reference examples for application-specific worst-case measurements, which require concurrent components to share data, have not yet been investigated. (4) The Procol-based Mandelbrot application is only compared to an equivalent version that is based on the threading model. Comparative analyses to other programming models, for example, light-weight publish-subscribe frameworks, are desirable. (5) It is likely that the PRWLCHMTS contains defects that we have not yet identified since the benchmark results for the PRWLCHMTS were so catastrophic in contrast to the RWLCHMTS. Limitations (1) can be addressed by providing additional tuple space data structure implementations for the extended LightTS framework and by using many-core machines if available. Addressing ageing (2) in more detail can be achieved by populating a tuple space with tuples of different size and type before benchmark execution [13]. We leave these topics together with limitation (3) and additional comparative analyses (4) for future work. The PRWLCHMTS (5) also requires further investigation.

6. Related Work

Balzarotti et al. present LightTS, a minimalistic **tuple space framework** originally developed as the core tuple space layer for the LIME middleware⁸ [13]. The framework can be extended in various ways, for example, by exchanging the back-end implementation. Also, it finds a reasonable compromise between object encapsulation stemming from object-orientation and the requirement to publish the internals of a data object stemming from generative communication: An interface `Tupleable` allows to flatten objects

into tuples with the method `toTuple`, while `setFromTuple` allows to recreate an object from a tuple. This is supported by the class `ObjectTuple`, which can remember the type of the object a tuple was created from. Although the framework emphasises its use for context-aware applications, we use LightTS as a basis to examine Java tuple space behaviour for multi-core programming. We modified the core tuple space data structure, and added alternative data structures, a benchmark suite, and a component orchestration layer. There are several other **Java implementations of space-based systems** that are mostly oriented towards distributed computing. Examples are Blitz JavaSpaces⁹, SemiSpace¹⁰, and the LightTS framework. Wells et al. present a survey of selected tuple space implementations in Java [18]. Their discussion focusses on the commercial offerings of the implementations and on the capability to provide alternative mechanism for tuple matching. They state that their performance measurements for a simple ray-tracing application indicate that none of the implementations considered were particularly suitable for fine-grained parallel processing. However, they state that the employed hardware was barely adequate to benchmark Java applications. It is also unclear how representative the results are, as the number of individual measurements is not stated. Heiser makes a case for Virtual Shared Memory (VSM) as an attractive model for **future many-core systems** [19]. VSM is a shared memory abstraction that is implemented over physically distributed memory by a hypervisor. It allows operating systems to directly access all memory of a many-core system. The model provides several benefits for hardware manufacturers since it integrates well with the use of virtualisation for resource management, and simplifies to handle message loss in the interconnect, faulty cores, and core heterogeneity [19]. Heiser also presents results from a cluster-based prototype that indicate that the VSM abstraction does not impose a significant overhead when shared memory is not required. We consider VSM-based many-cores as an attractive target architecture for space based programming since it provides a natural abstraction for the tuple space model. An overview of the architecture of future many-core systems in general is presented in condensed form by Borkar [17]. It discusses the topics power management, memory bandwidth, on-die-networks, and system resiliency. Our **benchmark suite** was inspired by the chapter on testing in the book “Java Concurrency in Practice” from Brian Goetz [16] and by the work of Fiedler et al. [15]. The latter present an approach to measure the throughput and the response time of a tuple space when it handles concurrent local space interactions [15]. They also present an ageing technique to populate a tuple space before the execution of a benchmark. The approach considers the phases *Ageing*, *Startup*, *Write* (equal to `out`), *Pause*, *Take*, *Shutdown*, *Ageing Cleanup*. It is used to measure the performance of a JavaSpace implementation. The presented benchmarks were focused on blocking operations to characterise the worst-case performance of the JavaSpace `take` operation (equal to `in`). Our tool for **performance evaluation** and the data analysis in this paper are motivated by the work of Georges et al. on statistical rigorous Java performance evaluation [14]. In general, we followed their advice on differentiating start-up from steady state execution, and on using the confidence intervals to discuss independent measurement series. We also followed their advice on using different techniques to compute the confidence interval based on whether the number of individual measurements for a series is large or not.

7. Conclusion

In this paper we investigated the use of space-based systems for multi-core programming in Java. Firstly, we argued about appealing

⁷<http://web.ics.purdue.edu/~cs180/Fall2009Web/projects/p3/>

⁸<http://lime.sourceforge.net/>

⁹<http://www.dancres.org/blitz/>

¹⁰<http://www.semispacespace.org/semispacespace/>

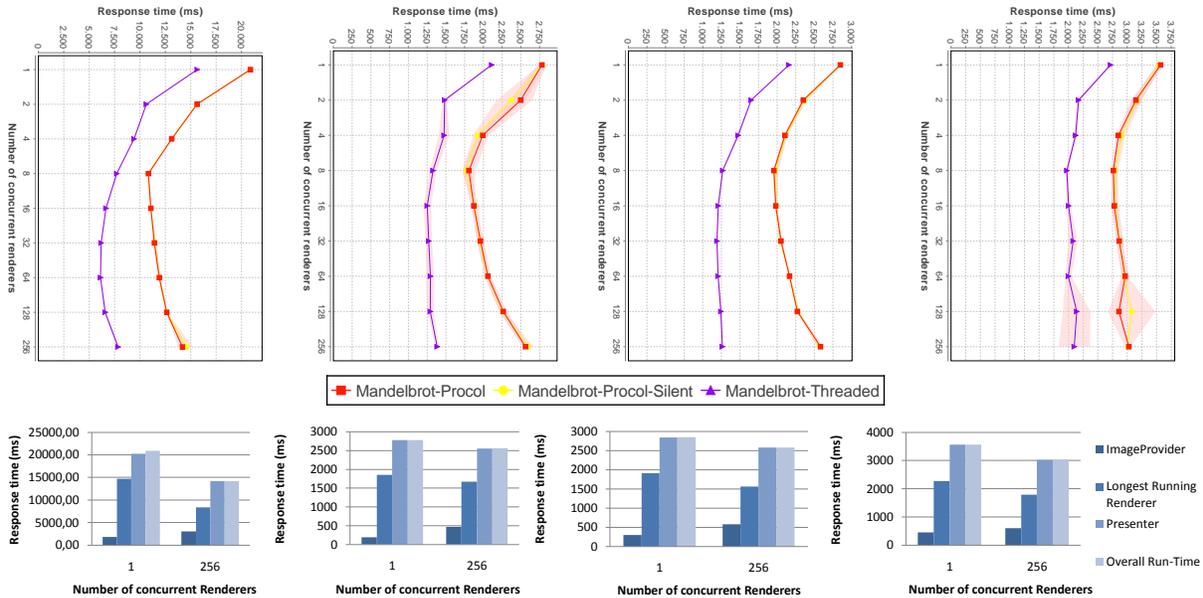


Figure 5. Upper half: Benchmarks of the Mandelbrot application on machines M0 to M3 using the RWLTS. The command line parameters were: Image width (-w): 1200 pixels, image height (-ht): 1024 pixels, number of image partitions and Renderer instances (-r), top left complex number on plane (-tl): $(-1) + 1i$, bottom right complex number on plane (-br): $1 + (-1)i$. **Lower half:** Response time of individual Mandelbrot application components at 1 and 256 concurrent Renderer instances.

properties of space-based programming for multi-core machines and presented the Procol programming model that introduces the space-based choreography of active components, which internally orchestrate fine-grained activities. This model is based on extensions to the third-party tuple space framework LighTS. Secondly, we evaluated the performance of different tuple space data structures for Procol and compared the performance of two equivalent Mandelbrot applications – one implemented with the standard Java thread model, one with our Procol programming model. Both the data structures and the Mandelbrot applications were evaluated on several machines. The conclusions drawn from these experiments are that in principle, tuple space implementations can be provided that scale reasonably well on multi-core architectures, and that the performance overhead that the Procol programming model impose is at least for the considered application a reasonable trade-off for the ease of programming that the model provides.

References

- [1] Marowka, A.: Parallel Computing on any Desktop. *Commun. ACM* 50, 74–78 (2007)
- [2] Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs's Journal* 30 (2005)
- [3] Lee, E.A.: The Problem with Threads. *IEEE Computer* 39, 33–42 (2006)
- [4] Gudenkauf, S.: A Coordination-Based Model-Driven Method for Parallel Application Development. *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers. LNCS 6002*, 21–35. Springer Heidelberg (2010)
- [5] Basili, V.R., Caldiera, G., Rombach, H.D.: The Goal Question Metric Approach. *Encyclopedia of Software Engineering*, 528–532. Wiley (1994)
- [6] Freisleben, B., Kielmann, T.: Object-Oriented Parallel Programming with Objective Linda. *Journal of Systems Architecture* (1997)
- [7] Hasselbring, W.: The ProSet-Linda Approach to Prototyping Parallel Systems. *Journal of Systems and Software* 43(3), 187–196 (1998)
- [8] Gelernter, D.: Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.* 7, 80–112 (1985)
- [9] Ciancarini, P.: Coordination Models and Languages as Software Integrators. *ACM Computing Surveys* 28, 300–302 (1996)
- [10] Wegner, P.: Why interaction is more powerful than algorithms. *Commun. ACM* 40(5), 80–91 (1997)
- [11] Scherp, G., Höing, A., Gudenkauf, S., Hasselbring, W., Kao, O.: Using UNICORE and WS-BPEL for Scientific Workflow Execution in Grid. *Euro-Par 2009 Workshops - Parallel Processing. LNCS 6043*, 335–344. Springer, Heidelberg (2009)
- [12] Melzer, I.: *Service-orientierte Architekturen mit Web Services*. 2 edn. Elsevier, München (2007)
- [13] Balzarotti, D., Costa, P., Picco, G. P.: The LighTS tuple space framework and its customization for context-aware applications. *Web Intelligence and Agent Systems: An International Journal* 5, 215–231. IOS Press (2007)
- [14] Georges, A., Buytaert, D., Eeckhout, L.: Statistically Rigorous Java Performance Evaluation. *ACM SIGPLAN Notices - Proceedings of the 2007 OOPSLA conference* 42(10), 57–76 (2007)
- [15] Fiedler, D., Walcott, K., Richardson, T., Kapfhammer, G. M., Amer, A.: Towards the Measurement of Tuple Space Performance. *ACM SIGMETRICS Performance Evaluation Review* 33(3) (2005)
- [16] Goetz, B.: *Java Concurrency in Practice*. 7th Printing edn. Addison-Wesley, Pearson Education, Inc. (2009)
- [17] Borkar, S.: Thousand Core Chips - A Technology Perspective. *44th ACM/IEEE Design Automation Conference*, 746–749 (2007)
- [18] Wells, G. C., Chalmers, a G., Clayton, P. G.: Linda implementations in Java for concurrent systems. *Concurrency and Computation: Practice and Experience* 16(10), 1005–1022 (2004)
- [19] Heiser, G.: Many-Core Chips A Case for Virtual Shared Memory. *Proceedings of the 2nd Workshop on Managed Many-Core Systems*. Washington, DC, USA (2009)