Towards the Measurement of Tuple Space Performance

Daniel Fiedler, Kristen Walcott,
Thomas Richardson,
Gregory M. Kapfhammer[1]
Department of Computer Science
Allegheny College

Ahmed Amer,
Panos K. Chrysanthis
Department of Computer Science
University of Pittsburgh

Abstract

Many applications rely upon a tuple space within distributed system middleware to provide loosely coupled communication and service coordination. This paper describes an approach for measuring the throughput and response time of a tuple space when it handles concurrent local space interactions. Furthermore, it discusses a technique that populates a tuple space with tuples before the execution of a benchmark in order to age the tuple space and provide a worst-case measurement of space performance. We apply the tuple space benchmarking and aging methods to the measurement of the performance of a JavaSpace, a current example of a tuple space that integrates with the Jini network technology. The experiment results indicate that: (i) the JavaSpace exhibits limited scalability as the number of concurrent interactions from local space clients increases, (ii) the aging technique can operate with acceptable time overhead, and (iii) the aging technique does ensure that the results from benchmarking capture the worst-case performance of a tuple space.

1 Introduction

A tuple space is a shared memory component of middleware that provides communication and coordination facilities to the services in a distributed system. Tuple spaces have been used to implement a wide variety of applications, including parallel UNIX utilities [6], parallel genetic algorithms (GAs) [27], distributed regression testing frameworks [12], large-scale mobile agent systems [17], "lifestream" information management applications [3], and scientific computations that support both astrophysics [16] and bioinformatics [22, 23] research. Even though tuple spaces have been used to implement a wide range of applications, there is a relative dearth of benchmarking frameworks that focus on the measurement of tuple space performance. This paper describes an approach to tuple space performance evaluation that supports the creation of concurrent local clients and the use of aging to populate the space with tuples before the execution of a benchmark. This paper also provides the results from experiments that characterize the performance of a specific type of tuple space and evaluate the efficiency and effectiveness of the proposed aging technique.

This paper applies the benchmarking framework to the measurement of the performance of a JavaSpace, an example of tuple space that integrates into the Jini network technology [7]. A JavaSpace can interact with clients that are either resident on the same network node or located on a remote node. In the context of applications like parallel GAs, a client's interaction with a JavaSpace frequently occurs when one or more local clients execute on the same machine as the space itself [27]. Recent research by Noble and Zlateva has also highlighted some concerns about the performance characteristics of a JavaSpace [16]. Furthermore, the experiences reported by Zorman et al. indicate that the performance of a JavaSpace decreases significantly when the number of concurrent space clients exceeds a certain threshold [27]. Thus, there is a need for an approach to determine the number of local clients that will cause tuple space throughput to "knee" as client response times continue to increase. To this end, this paper introduces a benchmarking framework that can measure tuple space throughput and response time when a pre-defined number of space clients simultaneously perform the same benchmark.

Due to the fact that the performance of an unused storage system differs from one that has been active, aging techniques have been constructed in order to make file system benchmarks more realistic [21]. A tuple space benchmarking technique that measures the performance of an empty space will not provide an accurate characterization of worst-case tuple space performance. This paper establishes and evaluates a novel approach to tuple space aging that yields worst-case performance measurements by populating a tuple space with tuples before a benchmark is executed. The presented aging technique uses automatically generated workloads to place different types of tuples into the space. During the benchmarking phase that removes tuples from the space, the space's template matching algorithm must examine the additional tuples that were placed within the space by the aging mechanism. Thus, aging supports the characterization of the worst-case performance of a tuple space.

[1]Kapfhammer is the correspondence author and can be contacted at gkapfham@allegheny.edu.

This paper also includes the description of a framework for tuple space benchmarking and aging, collectively referred to as the **S**pace b**E**nchmarking and **T**es**T**ing modu**LE**s (SETTLE), that currently supports the execution of micro benchmarks to measure the throughput and response time of a JavaSpace under a variety of different configurations. This paper describes the results from the use of SETTLE to conduct experiments that take the first step towards the measurement of the performance of a noteworthy component in distributed system middleware. In summary, the important contributions of this paper are as follows:

1. A benchmarking framework to support the collection of response time and throughput measurements for a tuple space that handles concurrent local client interactions.
2. The introduction of a tuple space aging technique that enables tuple space performance benchmarks to capture worst-case performance.
3. A detailed empirical study that investigates the following phenomena: (i) the throughput and response time characteristics of a JavaSpace, (ii) the overhead associated with the tuple space aging technique, and (iii) the impact that tuple space aging has on the benchmark results.

2 JavaSpaces Background

This paper focuses on the performance analysis of an implementation of the tuple space concept that is known as a JavaSpace. However, Section 5 explains that the benchmarking and aging techniques presented by this paper are also applicable to other examples of the tuple space concept. JavaSpaces rely upon the substrate provided by the Java programming language and the Jini network technology by taking the form of a Jini service that uses Java remote method invocation (RMI) and Java serialization. JavaSpaces were designed to support loosely coupled communication with an interface that is both simple and expressive. A few of the simple operations provided by the JavaSpace include `read`, `write`, and `take` [7] .

Even though the JavaSpace also provides the non-blocking operations of `readIfExists` and `takeIfExists`, this paper focuses on the blocking JavaSpace operations. A `read` returns an object from the JavaSpace that matches the template provided by the requesting client and a `take` does the same but also removes the object from the JavaSpace. A `write` simply places an object into the JavaSpace [7]. A JavaSpace can only store Java objects that implement the `Entry` marker interface. In order to support template matching, all of the fields within an `Entry` must be subclasses of `java.lang.Object` that are publicly visible. Finally, the JavaSpace can be configured to operate in either a transient or a persistent mode. A transient JavaSpace stores all of its `Entry` objects in memory while a persistent JavaSpace can utilize secondary storage to maintain state across restarts.

Figure 1: The Phases of a SETTLE Benchmark.

3 The SETTLE Approach

3.1 Overview

SETTLE is an extension of the JavaSpaces benchmarking tool created by Noble and Zlateva [16]. One of the key contributions of SETTLE is the inclusion of facilities to create q local space clients that each execute the specified benchmark. The space clients are configured to pause for a random amount of time during two different phases of the benchmark lifecycle. The random pause, denoted T_{delay}, is defined in Equation (1). In this equation, T_{random} is a randomly generated time period, V is a uniformly distributed variation, T_{min} is the user specified minimum time delay, $T_{random} \in [0.0, 1.0]$, and $T_{delay} \in [T_{min}, T_{min} + V]$.

$$T_{delay} = (T_{random} \times V) + T_{min} \tag{1}$$

Figure 1 describes the phases that exist within a SETTLE benchmark.[1] The aging technique, as described in Section 3.4, can be used in addition to the standard five phase benchmark. The phases whose names are bold and italic involve the use of Equation (1) in order to introduce a pause into the execution of the space client. In the startup phase, each of the SETTLE space clients is configured to wait for T_{delay} milliseconds before attempting to contact the JavaSpace. Upon space contact, each client executes three separate phases: the `write` phase, the pause phase, and the `take` phase. Finally, the space client enters the shutdown phase. For a specified benchmark, each client uses the same type of Java `Entry` object to `write` into the JavaSpace, waits for a random period of time, and then attempts to `take` the same `Entry` objects out of the space. If aging is not used to seed the space with `Entry` objects, a benchmark always leaves the space in the state that existed before the benchmark was executed.[2] If tuple space aging is used, an additional aging cleanup phase is required to remove the `Entry` objects that were initially placed into the space during aging.

[1]SETTLE also supports other orderings of the phases within a benchmark. For example, the pause phase could be removed or the `write` and `take` phases could occur in the opposite order for some of the space clients. However, the empirical results examined in Section 6 focus on the phase ordering depicted in Figure 1.

[2]This condition holds only if a benchmark always performs a `take` instead of a `read`. Thus, this paper does not focus on the benchmarking of the `read` operation.

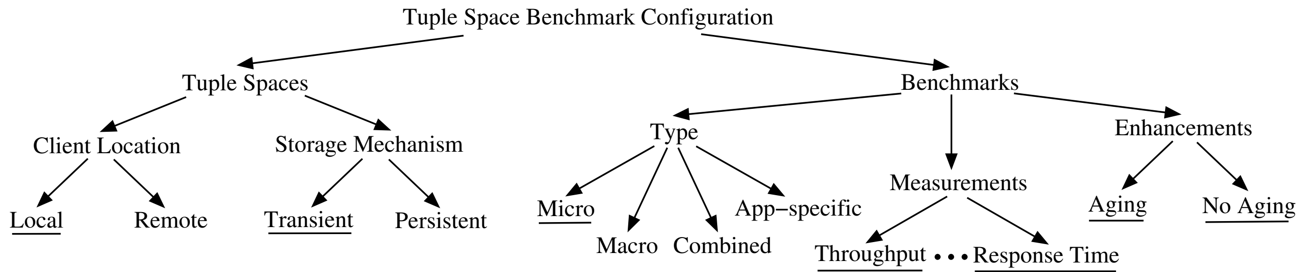

Figure 2: Classification Scheme for Tuple Space Benchmarking Techniques.

Figure 2 presents a classification scheme for tuple space benchmarking and shows the features that are supported by the SETTLE framework. The arrows in this scheme indicate that a benchmark configuration for a tuple space requires the selection of a tuple space and a benchmark. The tuple space must be configured by choosing a client location and a storage mechanism. The configuration of the benchmark requires the choice of a benchmark type and one or more performance measurements. Finally, a benchmark can be executed with or without the aging technique. Any underlined characteristic in Figure 2 (e.g., "Local" and "Transient") is described and evaluated in this paper. For example, the experiments discussed in Section 4 and Section 6 use micro benchmarks that perform local space interactions to measure the throughput and response time characteristics of a space. Furthermore, these experiments are conducted on aged and non-aged transient JavaSpaces.

This paper focuses on benchmarks that evaluate tuple space performance in a controlled environment in order to establish accurate and repeatable measurements of throughput and response time. To this end, this paper does not address tuple space benchmarking with: (i) remote clients, (ii) persistent tuple spaces, and (iii) macro, combined, and application-specific benchmarks. Yet, it is important to note that SETTLE does facilitate the benchmarking of tuple spaces when the client is remote and when the space uses a persistent storage mechanism. Finally, SETTLE's extensible design also supports the integration of other types of benchmarks.

This paper focuses on local clients rather than remote clients for two important reasons. First, benchmarking tuple spaces with local clients mirrors the behavior of real-world applications such as the mobile agents described by Picco et al. [17] and the parallel genetic algorithms created by Zorman et al. [27]. Second, focusing on local clients provides a more accurate performance estimate since the response time of a remote client is increased by network latencies and other remote communication overheads. This paper does not specifically address the benchmarking of persistent tuple spaces because it is expected that transient, in-memory, tuple spaces will normally provide better performance than persistent tuple spaces that must interact with secondary storage.[3] The use of macro, combined, and application-specific benchmarks is not addressed in this paper because these benchmarks often require detailed knowledge about the application domain. In contrast, our techniques can be used without having knowledge of the application domain and without the existence of a completely implemented application.

3.2 Benchmarking

Zhang and Seltzer observe that there are three main purposes for benchmarking: (i) the comparison of the performance of different systems, (ii) the guidance of performance optimizations, and (iii) the prediction of an application's performance in new software and hardware environments [26]. Furthermore, Zhang and Seltzer place software performance benchmarks into one of four categories: (i) micro benchmarks, (ii) macro benchmarks, (iii) combined benchmarks, and (iv) application-specific benchmarks [26]. This paper focuses on micro benchmarks that can serve as the first step towards realizing the stated purposes for the benchmarking of a tuple space. Micro benchmarks focus on the analysis of certain primitive operations that are used to create traditional applications [26]. A micro benchmark for JavaSpaces could measure the performance of basic space operations like `read`, `write`, and `take`.

Each SETTLE benchmark places a certain type of `Entry` object into the JavaSpace. The framework uses the same technique as Noble and Zlateva in order to approximate the size of the objects that are used in the benchmark [16]. The NullIO benchmark places a `NullEntry` object into a JavaSpace that is 356 bytes in size and the StringIO uses a `StringEntry` that is 503 bytes. The ArrayIO benchmark is configured by a user provided parameter that controls the size of the `double[]` array that is placed into the JavaSpace. For example, a `DoubleArrEntry` that contains a `double[]` array of size 100 consumes 1031 bytes of storage. The FileIO benchmark `write`s and `take`s a `FileEntry` that stores any type of binary

[3] However, our preliminary experiments revealed that this assumption does not hold when the transient tuple space exceeds the physical memory of its host node. We believe that this is due to the fact that the persistent JavaSpace is better able to manage the use of the file system than the virtual memory manager of the operating system.

or text file in a `byte[]` array. For example, when the FileIO benchmark stores an 86 line eXtensible Markup Language (XML) file in a `byte[]` array, the resulting `FileEntry` is 3493 bytes in size.

3.3 Performance Metrics

The tuple space benchmarking framework supports the measurement of: (i) the response time for each individual client's `write` and `take` operations, (ii) the time required for each client to contact the JavaSpace, (iii) the overall time used to complete all of the client interactions, (iv) the time required to age the tuple space, and (v) the time needed to remove the `Entry` objects that were placed within the space by the aging technique. Intuitively, the throughput of a tuple space is the number of operations that the space can complete within a specific duration of time. Furthermore, the response time for a specific space client's interaction with a tuple space is the time interval between the client's submission of the request and the space's completion of the computation required by the request.

The definitions of the performance metrics used in this paper adapt the notation established by Jain [11]. Equation (2) defines $X(S_i, O, q)$, the throughput of tuple space S_i during its servicing of the requests made by the set of clients $C = \{C_1, C_2, \ldots, C_q\}$ for operation O. The definition of $X(S_i, O, q)$ uses $L(S_i, O)$ to denote the number of completed space operations O that occurred at space S_i during the time required to execute the benchmark, denoted T_{bench}. In the context of the phases in Figure 1, T_{bench} refers to the duration of the time period needed to execute the phases within the bold vertical lines. The actual space operation O varies with respect to the granularity that is chosen to view the space interaction. For example, O might be a single `write` or `take` operation or some meaningful combination of the primitive tuple space operations.

$$X(S_i, O, q) = \frac{L(S_i, O)}{T_{bench}} \quad (2)$$

Equation (3) defines $R(S_i, C_j, O)$, the response time of tuple space S_i when it services C_j's request for operation O. The definition of $R(S_i, C_j, O)$ assumes that the space's response to the request for space operation O will always arrive after the request is sent. The measurement of response time uses the point at which C_j's interaction with S_i is complete. Finally, Equation (4) describes $R(S_i, C, O, q)$, a measure of the average response time for all of the client interactions that occur with space S_i.

$$R(S_i, C_j, O) = T_{complete}(S_i, O) - T_{submit}(S_i, O) \quad (3)$$

$$R(S_i, C, O, q) = \frac{\sum_{j=1}^{q} R(S_i, C_j, O)}{q} \quad (4)$$

It is also important to measure the percentage change in throughput and response time as the value of q changes.

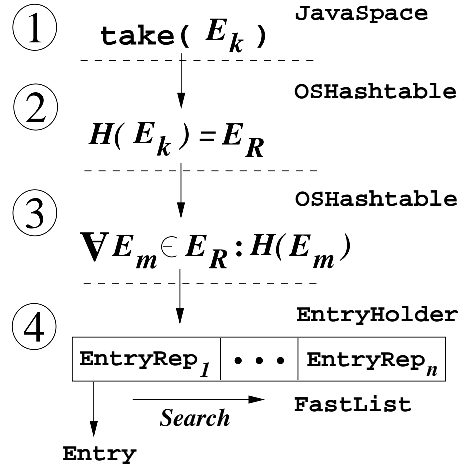

Figure 3: Template Matching.

Equation (5) defines $X^{\%}(S_i, O, q, q')$, the percent change in throughput as the number of clients is increased from q to q'. Equation (6) defines the percent change in response time, denoted $R^{\%}(S_i, C, O, q, q')$, in an analagous fashion. A positive value for the $X^{\%}(S_i, O, q, q')$ metric demonstrates a desirable increase in throughput while a negative value shows that throughput decreased as the number of clients increased. A positive value for $R^{\%}(S_i, C, O, q, q')$ shows an unwanted increase in response time while a negative value indicates a decrease in response time.

$$X^{\%}(S_i, O, q, q') = \frac{X(S_i, O, q') - X(S_i, O, q)}{X(S_i, O, q)} \times 100 \quad (5)$$

$$R^{\%}(S_i, C, O, q, q') = \frac{R(S_i, C, O, q') - R(S_i, C, O, q)}{R(S_i, C, O, q)} \times 100 \quad (6)$$

3.4 Tuple Space Aging

It is likely that the initial state of a tuple space will have a significant impact upon the performance results produced by benchmarking. The most straightforward solution to the problem of selecting an appropriate tuple space state is to simply mandate that the benchmark always interacts with a space that is initially empty. Yet, in the context of file systems, Smith and Seltzer have argued that while this approach might be simple, it does not accurately approximate the environment seen by users of real applications [21]. Even though Smith and Seltzer argue for aging in the context of file system benchmarks, Section 6 shows that aging impacts the performance of a tuple space.

Since it is not always possible to obtain information about the state of a tuple space that results from the execution of a real space-based application, this paper investigates the use of an aging technique to populate the tuple space with `Entry` objects before a benchmark is executed. Suppose that we desire to populate tuple space S_i through the execution of a space workload

Parameter	Value(s)
T_{min}	200 ms
V	50 ms
T_{delay}	$[200, 250]$ ms
# of `Entry` Objects	$\{1000\}$
Aging Workload Size ($\|W\|$)	$\{1000, 3000, 6000, 12000\}$
# of Clients (non-aged) (q)	$\{2, 8, 14, 22\}$
# of Clients (aged) (q)	$\{8, 14\}$
$\{r, t, w\}$-frequency	$\{0, 0, 100\}$
`Entry` Objects	{`Null`, `String`, `Array`, `File`}

Figure 4: Experiment Parameters Used for All Benchmarks.

$W = \langle w_1, \ldots, w_u \rangle$ with each space operation $w_x \in W$ being one of the primitive tuple space operations `read`, `take`, or `write`. The approach to tuple space aging must be able to select a workload W that can effectively age a space. Once a workload has been selected, the aging approach must introduce the result of workload execution into S_i before the execution of a benchmark.

For any approach to tuple space aging, there are two potential techniques for the creation of W: (i) automatically generated workloads and (ii) recorded/derived workloads. This paper focuses on automatically generated workloads and their impact upon the performance of tuple spaces because the impact of aging can be evaluated without relying upon prior workload studies. Yet, it is important to note that since the aging technique has already been implemented, the results of detailed tuple space workload studies can be used to create recorded/derived workloads that mimic real world tuple space usage. The automatic generation of workloads is governed by the $\{r, t, w\}$-frequency and the `Entry` object selection technique.

The $\{r, t, w\}$-frequency defines the fraction of the workload that will be associated with the tuple space operations of `read`, `take`, and `write` and must adhere to the restriction that the frequencies for each of the respective operations sums to one (i.e., $r + t + w = 1$). A random $\{r, t, w\}$-frequency selects a frequency each time the aging mechanism is executed. It is also possible to fix the $\{r, t, w\}$-frequency to a ratio of space operations that will be used to age the tuple space. Alternatively, an "all write" $\{r, t, w\}$-frequency would mandate that the aging workload only performs the `write` space operation (i.e., $w = 1$ and $r = t = 0$). It might also be useful to perform `read` and `take` operations during space aging if this accurately models a real world workload or performance measurement is focused on a tuple space implementation that caches frequently requested `Entry` objects.

As noted in Section 2, the `read` and `take` operations require an `Entry` template that specifies the desired type of object. Furthermore, the `write` operation must place a certain `Entry` object into the tuple space. To this end, the benchmarking framework selects from a set E of `Entry` objects when any space interaction occurs. Each `Entry` must be associated with a selection frequency so that all of the frequencies of the $E_k \in E$ adhere to the restriction stated in Equation (7). The type and number of objects within E is specified by the user of the benchmarking framework. For example, if E is defined as the set of `Entry` objects used by the benchmarks discussed in this paper, then it is evident that $E = \{\texttt{NullEntry}, \texttt{StringEntry}, \texttt{DoubleArrEntry}, \texttt{FileEntry}\}$.

$$\sum_{E_k \in E} freq(E_k) = 1 \tag{7}$$

An aging technique's impact upon tuple space performance will vary with respect to the space's rules for template matching, the implementation of the tuple space, and the `Entry` objects that are placed within the space by the aging mechanism. Since the `take` operation that performs the template matching is a core algorithm within a JavaSpace, this paper focuses on aging techniques that support the characterization of the worst-case performance of a `take`. For example, assume that operation $w_x \in W$ is a `take` and `Entry` $E_k \in E$ has been selected for the template. Freeman et al. state that the template E_k matches an `Entry` E_l within a JavaSpace if two rules hold: (i) the E_k's type is the same as E_l's type or E_k's type is a supertype of E_l's type and (ii) every field within E_k matches the corresponding field within E_l [7]. For rule (ii), a field f_k within E_k matches a field f_l within E_l if f_k is a "wildcard" or if the values of f_k and f_l are the same [7]. Many different data structures can be used to store `Entry` objects inside of the tuple space and implement these template matching rules. The Jini 1.2.1 implementation of the JavaSpace uses the following data structures, among many others, to perform template matching:

1. `COM.odi.util.OSHashtable`
2. `com.sun.jini.outrigger.SimpleEntryHolder` (an implementation of the `EntryHolder` interface)
3. `com.sun.jini.outrigger.FastList`
4. `com.sun.jini.outrigger.EntryRep`
5. `net.jini.core.entry.Entry`

Figure 3 summarizes the four steps that must be taken to determine which `Entry` objects within a tuple space match the template E_k that is provided to a `take` operation. [4] In step one, the JavaSpace receives a `take` operation for all of the `Entry` objects stored within the JavaSpace that match the `Entry` template E_k. In step two,

[4] This discussion uses a simplified model of the data structures used to store `Entry` objects and perform template matching. Without loss of generality, the model enables this paper to intuitively motivate the assertion that aging can support the characterization of worst-case space performance.

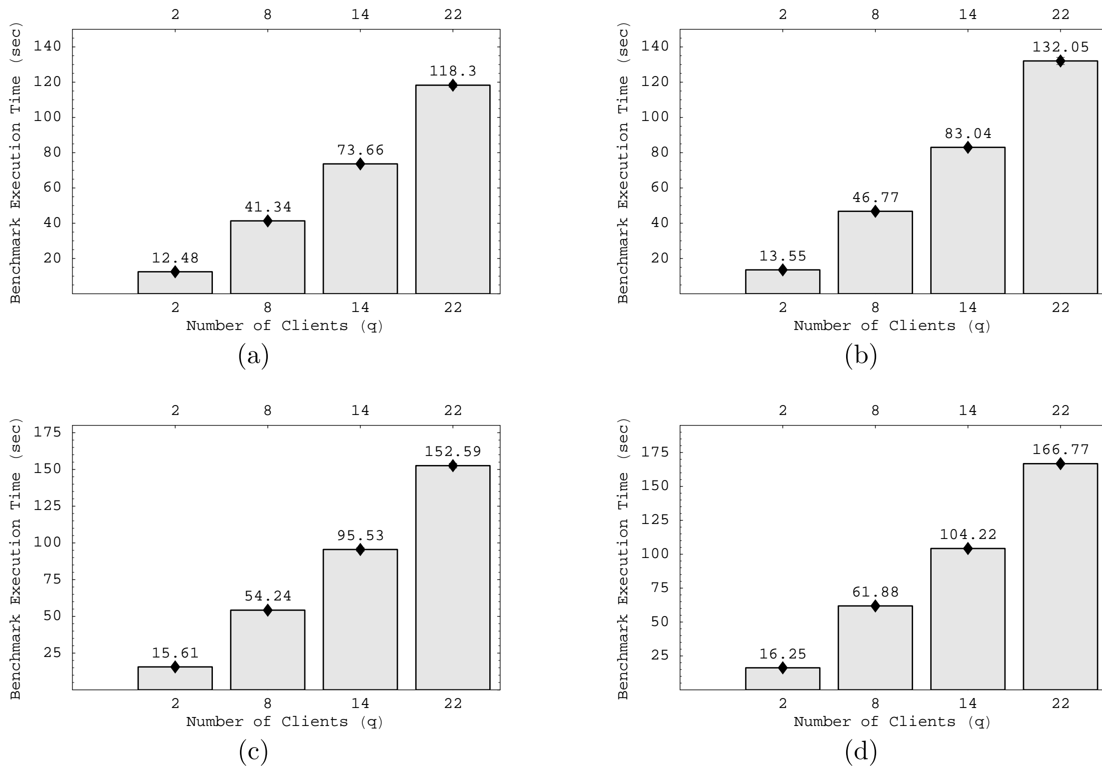

Figure 5: Benchmark Execution Time for (a) NullIO, (b) StringIO, (c) ArrayIO, (d) FileIO.

the template matching technique uses an `OSHashtable`'s hashing function H to determine the set of relevant types and subtypes of E_k, denoted E_R. For each specific type $E_m \in E_R$, an additional instance of `OSHashtable` is used in step three to identify the `SimpleEntryHolder` that contains, among other data fields, a `FastList` of all of the relevant `EntryRep` objects. Once one of the desired `EntryRep`s is identified by a search of the `FastList` in step four, the `Entry` object can be extracted and returned to the client that invoked the `take`. The `take` can return the first `Entry` within the `FastList` that matches the template E_k.

A tuple space aging technique can take two different approaches to impact the performance of the template matching algorithm that is intuitively depicted in Figure 3. For example, assume that the tuple space is initially populated with `Entry` objects that are related in a deep inheritance hierarchy. This initial configuration could increase the number of relevant types for a `take`'s template E_k, thus enlarging the set E_R, and subsequently requiring a search through the many `FastList`s that are stored in all of the relevant `SimpleEntryHolder`s. Alternatively, suppose that the tuple space is aged with the same type of `Entry` object while varying the values of the fields within the `Entry`. This could increase the number of the `EntryRep`s that must be searched in step four in order to identify an `Entry` object in the `FastList` that match the template E_k that was provided to the `take` operation.

As an example of the second aging technique, suppose that a tuple space is aged with `StringEntry` objects that contain the string s_a, the StringIO benchmark `write`s and `take`s `StringEntry` objects that contain the string s_b, and $s_a \neq s_b$. In this circumstance, the `FastList` will contain `EntryRep`s for the `Entry` objects with both the strings s_a and s_b. When a `take` is executed with a `StringEntry` template that wraps the string s_b, all of the `StringEntry` objects with the string field s_a, in the worst case, must be examined before finding an `Entry` that matches a template with the string s_b. This circumstance occurs when all of the `Entry` objects with string s_a are stored before all of the objects that contain the string s_b. If there are a total of n_a objects that contain the string s_a, the execution of the template matching algorithm requires a linear search of the `FastList` that has a worst-case time complexity of $O(n_a)$ in the aged configuration. If the tuple space is not aged and all `Entry` objects in the space contain the string s_b, the first `Entry` in the `FastList` will always match E_k and this results in a worst-case time complexity of $O(1)$ for the template matching algorithm.

If aging places n_a objects within the tuple space and n_a is large, this could change the results of a benchmark. In summary, this example intuitively demonstrates that tuple space aging has the potential to cause the execution of the tuple space matching rules during a `take` to deviate from the constant time complexity of $O(1)$ to a linear time complexity of $O(n_a)$. In light of the simplic-

ity and potential impact of the second technique, this paper evaluates how aging with the same `Entry` type will change the throughput and response time of a tuple space. Since aging is only useful if it does not introduce large time overheads into the benchmarking process, this paper also investigates the efficiency of aging by measuring T_{age} and T_{clean}, the time required to perform aging and cleaning, respectively. Equation (8) defines $T^{\%}_{age}$, the percentage of benchmark time that is consumed by the aging of a tuple space. In this equation and the analogous definition of $T^{\%}_{clean}$, T_{bench} is defined in the same manner as Equation (2) in Section 3.3.

$$T^{\%}_{age} = \frac{T_{age}}{T_{bench} + T_{age}} \times 100 \quad (8)$$

4 Experiment Design

The empirical study described by this paper focuses on the calculation of $X(S_i, O, q)$ and $R(S_i, C, O)$ in order to evaluate the performance of aged and non-aged transient JavaSpaces. This study also measured $T^{\%}_{age}$ and $T^{\%}_{clean}$ to characterize the efficiency of the tuple space aging and cleaning techniques. All of the experiments with SETTLE were conducted on a workstation with dual Intel Xeon Pentium III processors, 512 MB of main memory, and a SCSI 10,000 RPM disk subsystem. The workstation was running GNU/Linux with a 2.4.18-14smp kernel. The experiments used a Java 2 standard edition (J2SE) 1.4.2 compiler, a J2SE 1.4.2 virtual machine configured to operate in HotSpot Client mode, and the Jini 1.2.1 network technology. The Java virtual machine (JVM) was configured to use the LinuxThreads 0.10 thread library and a one-to-one mapping between Java threads and GNU/Linux kernel threads. Throughout the experiments, the GNU/Linux kernel was allowed to schedule the Java processes on the two available processors according to its internal scheduling algorithm.

We also configured the generation of the T_{delay} that is used during the startup and pause phases so that $V = 50$, $T_{min} = 200$, and thus $T_{delay} \in [200, 250]$. The value of T_{random} was generated with `java.util.Random`'s `nextDouble` method. Finally, all experiments were executed with each local client being responsible for writing 1000 Java `Entry` objects. In each experiment, we varied the number of tuple space clients so that $q \in \{2, 8, 14, 22\}$. For each benchmark and each value of q, we conducted the experiment five times in order to collect throughput and response time metrics and facilitate the computation of arithmetic means and standard deviations. Standard deviations are represented by error bars whenever the bar does not obscure the experiment results. In the bar charts of Figure 5, the diamonds at the top of the bars indicate that the standard deviations were insignificant.

We aged the tuple space with the type of `Entry` object that is used by each of the benchmarks. For example, when the ArrayIO benchmark is executed, we aged the tuple space with `DoubleArrEntry` objects that wrap `double[]` arrays. Other `Entry` selection techniques that age the tuple space with a variety of objects are also possible. However, we were interested in evaluating whether the impact of aging correctly corresponded to the one that was discussed in Section 3.4. Since the `NullEntry` does not contain any fields that would require inspection during the execution of the template matching algorithm, aging would not normally impact the performance of the NullIO benchmark. To this end, we modified the `NullEntry` object so that it contained a `boolean` field called `aged` and then executed the aging experiments. All of the aging experiments were conducted in three phases: (i) the space was aged using the same `Entry` type as the chosen benchmark, (ii) the benchmark was then executed, and (iii) the space was cleaned.

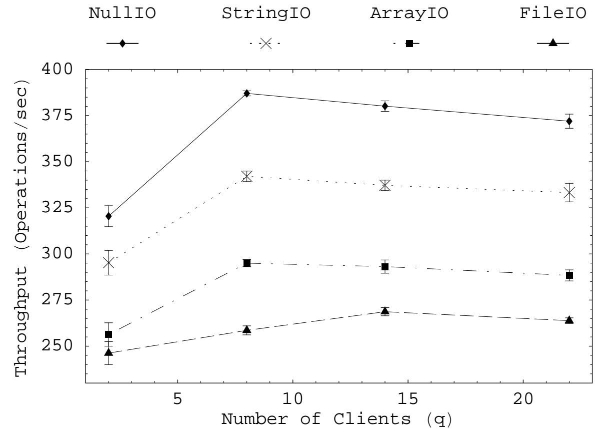

Figure 6: Throughput Measurements.

Aging used automatically generated workloads with a $\{r, t, w\}$-frequency of all writes and a workload size governed by $|W| \in \{1000, 3000, 6000, 12000\}$. Benchmarks were executed as previously discussed, except that the number of clients was restricted so that $q \in \{8, 14\}$. We limited the values for q because the experiments revealed that tuple space throughput normally exhibited a "knee" between eight and fourteen clients. For every benchmark and value of q and $|W|$, we conducted the experiment five times in order to collect average throughput, response time, and the aging/cleaning cost metrics. Figure 4 reviews the experiment parameters.

5 Threats to Validity

Any empirical study of the performance of a software application must confront certain threats to validity. During the analysis of a tuple space's performance and the impact of aging we were aware of potential threats to validity and took steps to control the impact of these threats. A threat to internal validity concerns factors that might have impacted the measured variables (e.g., the throughput and response time of the aged and

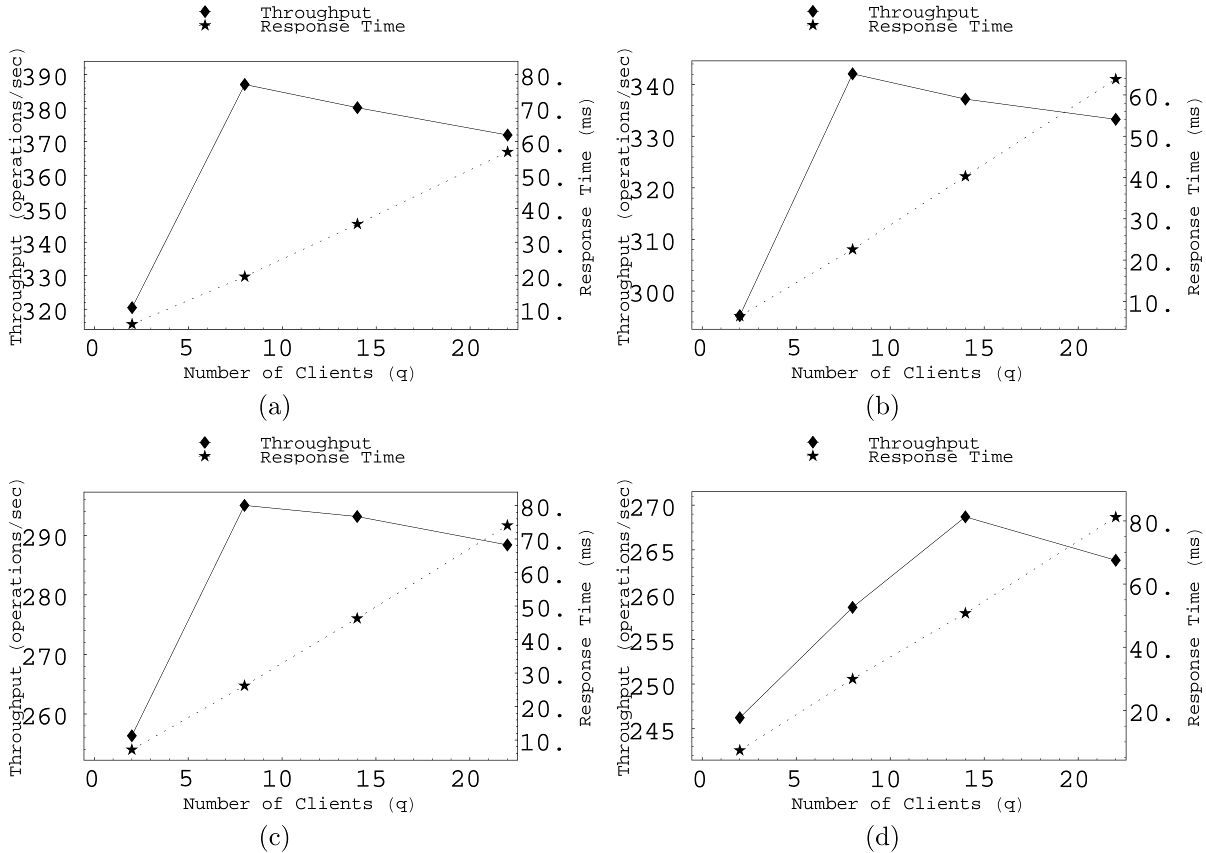

Figure 7: Scalability Measurements for (a) NullIO, (b) StringIO, (c) ArrayIO, (d) FileIO.

non-aged tuple spaces) without our knowledge. Defects within the SETTLE framework could cause benchmarks to be executed improperly or metrics to be calculated incorrectly. To this end, we used the benchmarking framework of Noble and Zlateva [16] to confirm that SETTLE calculated correct throughput and response time values when the selected benchmarks were executed with a single client. Furthermore, we measured the wall clock time required to complete the benchmark and ensured that this corresponded to the time calculated by SETTLE's instrumentation. We also used the same workstation for all experiments and we prevented external user logins to this workstation throughout experimentation.

A threat to external validity would limit the generalization of our approach and the experiment results. Our empirical study exhibits threats that are similar to other benchmarking efforts. First, this paper only focuses on the four micro benchmarks NullIO, StringIO, ArrayIO, and FileIO and the results from these benchmarks might be different than those produced by other micro, macro, and application-specific benchmarks. The SETTLE framework can support the integration of other benchmarks in order to control this threat and this paper does report on the results of previously used benchmarks (e.g., [16]). Second, the experiments only focused on a single implementation of the tuple space concept, the JavaSpace that is provided with the Jini 1.2.1 network technology. However, this is one of the most widely used tuple space implementations with an available source code and binary download. As long as the tuple space implementation adheres to the conventions established by the Jini network technology, SETTLE can measure the performance of this space. SETTLE can also benchmark tuple space implementations that do not integrate with Jini (e.g., [25]) by customizing the framework to use the space's own communication facilities. Finally, the experiments only measure tuple space performance in a single execution environment since the JVM, operating system kernel, thread library, and other environmental factors were not varied during experimentation. Since SETTLE is implemented in the Java programming language, it is possible to use the framework to measure tuple space performance in any execution environment for which a JVM is available.

6 Results Analysis

6.1 Benchmark Execution Time

Figure 5 shows the time that was required to execute each of the benchmarks for the four different values of q and a non-aged JavaSpace. These results indicate that for each q, NullIO always takes the least amount of execution time and FileIO always requires the greatest time to execute. The execution of FileIO with $q = 22$ require 41% more execution time than the similarly configured

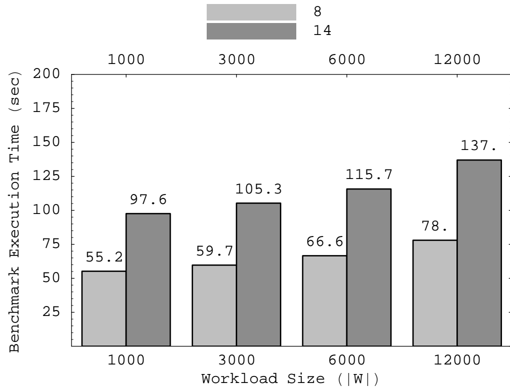

Figure 8: NullIO Execution Time with an Aged JavaSpace.

Object	#	Age (Std Dev) ms	Clean (Std Dev) ms
`Null`	1000	5233.2 (630.5)	3266.0 (72.8)
	3000	10527.8 (36.6)	6264.4 (239.3)
	6000	19098.4 (1285.9)	10925.2 (2150.9)
	12000	34838.6 (1393.8)	17468.6 (112.1)
`String`	1000	5670.4 (694.7)	3565.0 (15.0)
	3000	11655.0 (64.3)	6893.2 (47.0)
	6000	21446.2 (1056.0)	11335.6 (81.2)
	12000	39200.0 (1632.1)	20376.6 (177.4)
`Array`	1000	5601.2 (128.2)	3719.4 (77.1)
	3000	12584.4 (1146.4)	7399.8 (91.4)
	6000	22142.8 (1234.5)	12413.2 (114.9)
	12000	40381.2 (1136.2)	22531.6 (21.8)
`File`	1000	5901.6 (204.1)	3837.0 (21.7)
	3000	13567.4 (1280.3)	7747.2 (147.3)
	6000	23836.8 (1492.6)	13044.8 (95.8)
	12000	44198.8 (1848.4)	23589.6 (262.9)

Figure 9: Aging and Cleaning Overhead.

NullIO benchmark. Across all of the benchmarks, the execution with twenty-two clients uses 882% more time on average than the same benchmark that only uses two clients. These results also indicate that the complete execution of five trials of the four benchmarks for a non-aged tuple space and all values of q consumed approximately 100 minutes of execution time.

6.2 Throughput and Response Time

Figure 6 provides a graphical representation of the measurements of $X(S_i, O, q)$, the throughput of a JavaSpace.[5] These results confirm that the FileIO benchmark is the most "difficult" since it must `write` and `take` the largest `Entry` object. Each of the benchmark results also contain a "knee" where throughput levels off after the number of clients progresses beyond a certain threshold. The throughput of the NullIO, StringIO, and ArrayIO benchmarks levels off at eight clients and the throughput of the FileIO benchmark knees at fourteen concurrent clients.

The results show that $X^{\%}(S_i, O, 2, 8) = 20.8\%$ and $X^{\%}(S_i, O, 8, 14) = -1.8\%$ for the NullIO benchmark. The StringIO and ArrayIO benchmarks demonstrate similar throughput percent changes since the transition from two to eight clients respectively yields values of 15.9% and 15.1%. At the point where throughput knees, $X^{\%}(S_i, O, 8, 14) = -1.4\%$ for StringIO and $X^{\%}(S_i, O, 8, 14) = -.6\%$ for ArrayIO. Even though the FileIO benchmark knees at a different location in its throughput curve, the percent changes in throughput are like those of the other benchmarks because $X^{\%}(S_i, O, 2, 14) = 9.1\%$ and $X^{\%}(S_i, O, 14, 22) = -1.8\%$. Finally, the error bars that are used in Figure 6 to indicate standard deviations show that all throughput measurements show little dispersion.

It is also important to examine the relationship between the throughput and response time curves for a tuple space. Figure 7 provides a "two-axis" graphical depiction of the throughput and response time curves for each of the benchmarks. Every benchmark creates a response time that approximates a linear increase in $R(S_i, C, O, q)$ as the number of clients increases. These results indicate that $R^{\%}(S_i, C, O, 8, 14)$ is 78.2% on average for the NullIO, StringIO, and ArrayIO benchmarks and $R^{\%}(S_i, C, O, 14, 22) = 60\%$ for FileIO. The graphs in Figure 7 can also be used to show what the average client response time will be if a tuple space attempts to maintain a certain level of throughput or a specific number of concurrent clients. Using the results from the FileIO benchmark, it is evident that a tuple space-based file transfer program that attempts to perform twenty-two concurrent file transfers will create an average client response time of approximately 81 ms while maintaining a throughput of approximately 263 operations/sec.

6.3 Aging and Cleaning

Figure 8 presents the time that was required to execute the NullIO benchmark with an aged JavaSpace. When these results are combined with those provided by Figure 5, it is evident that when $|W| = 3000$ this results in approximately a 30% increase in execution time over the same benchmark that uses a non-aged tuple space. Figure 9 shows the mean and the standard deviation of the time required to age and clean a space. The experiments demonstrate that: (i) the mean time for aging and cleaning increases as the workload size increases, (ii) the mean time required for aging is always greater than that required for cleaning, and (iii) aging and cleaning time overheads exhibit relatively little dispersion. The first result confirmed our intuitive expectation and the second result is due to the fact that the aging technique must repeatedly accrue the costs of serializing the different `Entry` objects when it performs a `write`, while the space cleaner can avoid these costs by using a snapshot of a "wildcard" template when it performs a `take` [7].[6]

[5] In all graphs that depict "Operations/sec," we consider an operation to be either a `take` or a `write`. Thus, a benchmark executed with two clients that each `write` and `take` 1000 objects must perform a total of 4000 space operations.

[6] Intuitively, snapshotting replaces a template with a sparser representation that is only meaningful to the JavaSpace that produced the snapshot.

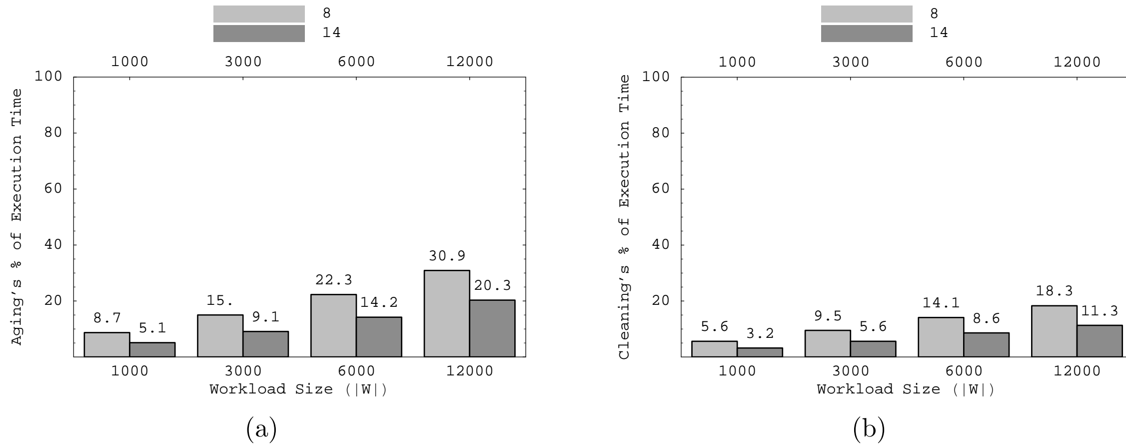

Figure 10: The Percentage of Execution Time for (a) Aging and (b) Cleaning During the NullIO Benchmark.

Figure 10 shows $T^{\%}_{age}$ and $T^{\%}_{clean}$, the percentage of NullIO benchmark execution time devoted to aging and cleaning, respectively. The trends depicted in Figure 10 also hold for the other benchmarks used in the experiment. These results reveal that aging with large workloads never consumes more than 31% of the entire time required to execute a benchmark. Furthermore, aging with a small or moderate sized workload (e.g., $|W| = 1000$ or $|W| = 3000$) always needs less than 15% of the benchmark execution time. As expected, aging has a smaller impact on execution time when q is larger and $T^{\%}_{clean}$ is always smaller than $T^{\%}_{age}$ for the same value of q. In summary, these results indicate that the time overhead associated with aging and cleaning is acceptable.

Figure 11 provides the results from the experiments to determine the impact of aging when eight clients perform the benchmarks (the results for fourteen clients were similar, yet more pronounced). In these graphs, a workload of size zero (e.g., $|W| = 0$) on the horizontal axis corresponds to a data point from a benchmark that did not use aging. These results clearly indicate that the aging technique causes an increase in the average response time and a decrease in the overall throughput as the aging workload size increases. However, certain benchmarks are more dramatically impacted by the use of aging. For example, when an aging workload of size 1000 is used, NullIO shows a decrease in throughput from 387 operations/sec to 289 operations/sec while ArrayIO only decreases from 298 operations/sec to 287 operations/sec.[7] Yet, as the size of the aging workload increases to 12000, all of the benchmarks demonstrate similarly high response times and low throughputs. This indicates that aging can impact the performance of an implementation of the tuple space concept.

We also investigated the impact of aging when an `Entry` type different from the one used by the benchmark is placed into the tuple space. For example, we placed `NullEntry` objects into the space when the FileIO benchmark was executed. For each benchmark and all non-benchmark `Entry` types, we found that aging with the all-write workload never had an impact on tuple space performance. In light of the intuitive discussion in Section 3.4, this is due to the fact that the benchmark and non-benchmark `Entry` objects are stored in different instances of `SimpleEntryHolder`. When a `take` is executed with a template of the benchmark type, the `FastList` within the `SimpleEntryHolder` for the non-benchmark type used during aging does not have to be searched. In summary, these results provide support for the assertion that aging with the same `Entry` type as the chosen benchmark can cause the `take` operation to exhibit $O(n_a)$ performance, as discussed in Section 3.4.

7 Related Work

To our knowledge, no prior work has specifically addressed the measurement of tuple space throughput and response time when an aging technique is applied before the execution of a benchmark that uses concurrent local clients. However, the research described in this paper is directly related to prior research in the areas of: (i) the benchmarking and behavior characterization of distributed system middleware, (ii) the benchmarking of tuple space implementations, (iii) the measurement and modeling of computer program performance, and (iv) the performance analysis of software applications implemented in the Java programming language.

Detailed empirical studies by Bulej et al. [2], Cecchet et al. [4], and Pugh and Spacco [18] indicate that the analysis and characterization of middleware performance is challenging. Bulej et al. detail a new middleware bench-

[7] This result is due to the fact that the initial NullIO experiments were conducted with a `NullEntry` configuration that did not contain any fields and the aging experiments used a `NullEntry` with the `boolean` field called `aged` (see Section 4 for more details about this design choice). In future experiments we will only use the modified version of `NullEntry`. Preliminary experiments indicate that when the modified `NullEntry` is used during all experiments, the reduction in throughput and the increase in response time is similar to those found in the other benchmarks.

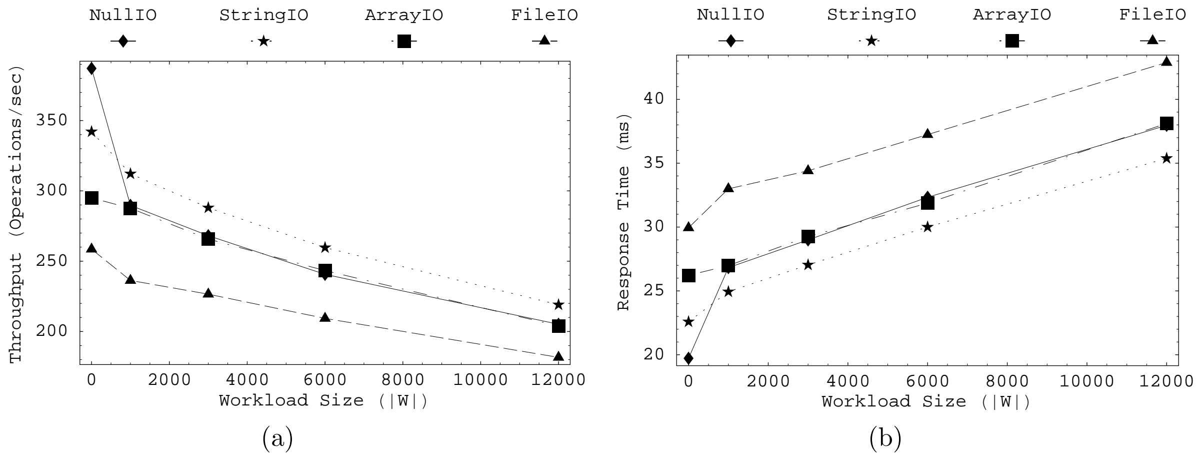

Figure 11: Impact of Aging on (a) Throughput and (b) Response Time for Eight Clients.

marking technique called regression benchmarking (similar to software regression testing [13, 19, 20]) that repeatedly executes performance benchmarks in an attempt to identify performance regressions in middleware [2]. Interestingly, Bulej et al. also reveal that certain complex benchmarks such as RUBiS and TPC-W are not applicable in the context of regression benchmarking because their execution incurs a significant time overhead and the results are subject to mis-interpretation [2]. Similar to the research described in this paper, Bulej et al. focus on the use of simple benchmarks that test an isolated feature of a distributed system middleware.

In the context of category (ii), Sterck et al. evaluate the performance of a tuple space that is a component within a framework for performing bioinformatics computations [22, 23]. However, the application-specific benchmarks in these papers do not produce results that are meaningful in other application domains. While Noble and Zlateva do report benchmark results for astrophysics computations, some of their micro benchmarks do produce results that are relevant to other domains [16]. Hancke et al. and Neve et al. specifically focus on measuring the performance of JavaSpaces through statistically guided experiments [8, 15]. However, Hancke et al.'s and Neve et al.'s current focus on remote workers could introduce unwanted experimental variability (neither paper clearly states whether the experiment network was isolated from contaminating traffic). Unlike this paper, neither Hancke et al. nor Neve et al. describe the initial state of the JavaSpace or a technique that can be used to populate the space before benchmark execution [8, 15].

In category (iii), Jain [11] and Lilja [14] both provide excellent introductions to the art and science of computer performance analysis. Furthermore, the file system aging technique proposed by Smith and Seltzer motivated the approach to tuple space aging described in this paper. Horgan et al.'s platform-independent analysis of Java program performance at the bytecode level [10] is relevant to category (iv). Alternatively, Zhang and Seltzer present an application-specific benchmarking framework that evaluates Java virtual machine performance in light of the behavior of a specific application. Dufour et al. propose a series of dynamic metrics that attempt to characterize the behavior of a Java application in a platform independent fashion [5] while Sweeney et al. use hardware performance monitors to shed light on the performance characteristics of Java programs [24]. Finally, Hauswirth et al. [9] present different techniques that can be used to profile and understand the behavior of Java software applications. Each of these techniques could complement the benchmarking and aging provided by the SETTLE framework.

8 Conclusions and Future Work

Since many distributed applications rely upon a tuple space to provide communication and coordination facilities, there is a clear need for benchmarking techniques that are customized for spaces. To this end, this paper describes the **S**pace b**E**nchmarking and **T**es**T**ing modu**LE**s (SETTLE), a framework that supports the execution of benchmarks that characterize the performance of an example of the tuple space concept known as the JavaSpace. This paper makes three important contributions. First, this paper describes a technique for the measurement of the throughput and response time characteristics of a tuple space that handles concurrent requests from local clients. Second, this paper explains a novel tuple space aging technique that seeds a space with `Entry` objects before a benchmark is executed. Third, the detailed empirical study in this paper demonstrates that: (i) the JavaSpace can support between eight and fourteen concurrent requests from local clients without suffering a reduction in average client response time, (ii) tuple space aging can be performed with acceptable time overhead and (iii) aging does support the characterization of the worst-case performance of a tuple space.

Future research can be divided into two categories: (i) extensions to the SETTLE framework and (ii) additional empirical studies of different tuple space performance characteristics. For example, the SETTLE framework could be extended by including new micro, macro, combined, and application-specific benchmarks. SETTLE can also be extended to sample T_{random} from probability distributions like Zipf and Gaussian. It would also be useful if the SETTLE framework provided testing techniques to isolate defects within and establish a confidence in the correctness of tuple space implementations and tuple space-based applications.

Future experiments could focus on one or more of the following: (i) a comparison of the baseline performance of transient and persistent tuple spaces, (ii) the evaluation of a tuple space aging technique that creates deep inheritance hierarchies for the `Entry` objects, (iii) the investigation of the impact that tuple space aging has on persistent tuple spaces, (iv) the benchmarking of JavaSpaces that handle remote client interactions, (v) the comparison of the performance of different tuple space implementations such as Arnold et al.'s RDBSpace [1] and Wyckoff et al.'s TSpaces [25], and (vi) an examination of the impact that operating system thread libraries have on the performance of tuple spaces. Finally, detailed workload studies for real world tuple space-based applications will enable the recording of tuple space aging workloads and the creation of new benchmarks. These workload studies must be supported by a tuple space monitoring infrastructure that can analyze the contents of a space and the interactions between a space and its clients.

References

[1] G. C. Arnold, G. M. Kapfhammer, and R. S. Roos. Implementation and analysis of a JavaSpace supported by a relational database. In *Proceedings of the 8th International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 2002.

[2] L. Bulej, T. Kalibera, and P. Tuma. Repeated results analysis for middleware regression benchmarking. *Performance Evaluation*, 60(1-4):345–358, 2005.

[3] N. Carriero and D. Gelernter. A computational model of everything. *Communications of the ACM*, 44(11):77–81, November 2001.

[4] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 246–261, 2002.

[5] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 149–168, 2003.

[6] C. J. Fleckenstein and D. Hemmendinger. Using a global name space for parallel execution of UNIX tools. *Communications of the ACM*, 32(9):1085–1090, 1989.

[7] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces: Principles, Patterns, and Practice.* Addison-Wesley, Reading, Massachusetts, 1999.

[8] F. Hancke, G. Stuer, D. Dewolfs, J. Broeckhove, F. Arickx, and T. Dhaene. Modelling overhead in JavaSpaces. In *Proceedings of EuroMedia*, pages 77–81, 2003.

[9] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-oriented applications. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 251–269, 2004.

[10] J. Horgan, J. Power, and J. Waldron. Measurement and analysis of runtime profiling data for Java programs. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 124–132, November, 2001.

[11] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley and Sons, Inc., New York, 1991.

[12] G. M. Kapfhammer. Automatically and transparently distributing the execution of regression test suites. In *Proceedings of the 18th International Conference on Testing Computer Software*, Washington, D.C., June 2001.

[13] G. M. Kapfhammer. *The Computer Science Handbook*, chapter 105: Software Testing. CRC Press, Boca Raton, FL, second edition, 2004.

[14] D. J. Lilja. *Measuring Computer Performance: A Practicioner's Guide.* Cambridge University Press, 2000.

[15] H. D. Neve, F. Hancke, T. Dhaene, J. Broeckhove, and F. Arickx. On the use of DOE for the characterization of JavaSpaces. In *Proceedings of the European Simulation and Modelling Conference*, pages 24–29, 2003.

[16] M. S. Noble and S. Zlateva. Scientific computation with JavaSpaces. In *Proceedings of the 9th European Conference on High Performance Computing and Networking*, June 2001.

[17] G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME: Linda meets mobility. In *Proceedings of the 21st International Conference on Software Engineering*, pages 368–377, 1999.

[18] B. Pugh and J. Spacco. RUBiS revisited: why J2EE benchmarking is hard. In *Companion to the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 204–205, 2004.

[19] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[20] M. Rummel, G. M. Kapfhammer, and A. Thall. Towards the prioritization of regression test suites with data flow information. In *Proceedings of the 20th Symposium on Applied Computing*, Santa Fe, New Mexico, March 2005.

[21] K. A. Smith and M. I. Seltzer. File system aging – increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 203–213, 1997.

[22] H. D. Sterck, R. S. Markel, and R. Knight. *Parallel Computing in Bioinformatics and Computational Biology*, chapter TaskSpaces: A Software Framework for Parallel Bioinformatics on Computational Grids. John Wiley and Sons, 2005.

[23] H. D. Sterck, R. S. Markel, T. Phol, and U. Rude. A lightweight Java TaskSpaces framework for scientific computing on computational grids. In *Proceedings of the ACM SIGAPP Symposium on Applied Computing*, pages 1024–1030, 2003.

[24] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, May 2004.

[25] P. Wyckoff, S. McLaughry, and T. Lehman. T Spaces. *IBM Systems Journal*, pages 454 – 474, 1998.

[26] X. Zhang and M. Seltzer. HBench:Java: an application-specific benchmarking framework for Java virtual machines. In *Proceedings of the ACM Conference on Java Grande*, pages 62–70, 2000.

[27] B. Zorman, G. M. Kapfhammer, and R. S. Roos. Creation and analysis of a JavaSpace-based genetic algorithm. In *Proceedings of the 8th International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, June 2002.