IEEE Copyright Notice

This work has been published in “2017 Fifth International Symposium on Computing and Networking (CANDAR) “, 9 - 22 November, 2017, Aomori, Japan.
DOI: 10.1109/CANDAR.2017.55

http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=8345445

MemSpaces: Evaluating the Tuple Space Paradigm in the Context of Memory-Centric Architectures

Andreas Grapentin, Max Plauth and Andreas Polze
Operating Systems and Middleware Group
Hasso Plattner Institute for Digital Engineering
University of Potsdam
Potsdam, Germany
{firstname.lastname}@hpi.uni-potsdam.de

***Abstract*—With *memory-centric architectures* appearing on the horizon as potential candidates for future computer architectures, we propose that the *tuple space* paradigm is well suited for the task of managing the large shared memory pools that are a central concept of these new architectures. We support this hypothesis by presenting *MemSpaces*, an implementation of the *tuple space* paradigm based on POSIX shared memory objects. To demonstrate both efficacy and efficiency of the approach, we provide a performance evaluation that compares *MemSpaces* to message-based implementations of the *tuple space* paradigm. Due to the lack of commercial availability of adequate hardware, we perform the evaluation inside an emulated environment that mimics the general characteristics of *memory-centric architectures*. For many operations, *MemSpaces* performs an order of magnitude faster compared to state of the art implementations.**

I. Introduction

Memory-centric architectures propose a radical change in the design of *rack-scale* server systems [1]. Where traditional large server systems distribute the total main memory as *node-accessable* memory, typically fragmented across the sockets of the node, *memory-centric archictectures* instead propose to have the majority of the byte-addressable main memory of the system *globally shared* across all nodes in the system (see Figure 1). First prototypes following this architecture have been manufactured and emulators have already been made available [2].

While this novel memory architecture promises great improvements in performance and scalability of distributed applications, it also introduces a number of challenges, such as ensuring memory protection, maintaining cache coherence across separate nodes in the system and managing efficient access to shared and distributed data structures in the context of concurrent access. Where traditional architectures solved most of these issues on the operating system or the hardware level, they now have to be solved in the application. Consequently, to ensure correct execution of applications on such architectures, new resource management and communication approaches need to be created that efficiently manage the state of resources in the shared memory and that allow application developers to focus on the application logic.

This work introduces *MemSpaces*, an implementation of the *tuple spaces* communication paradigm in Python using the *POSIX shared memory interface*. *Tuple spaces* are an intuitive and powerful paradigm for sharing objects across distributed tasks. While traditional implementations of the *tuple spaces* paradigm adhere to a client/server model, which introduces scalability and consistency issues, *MemSpaces* stores tuples in the shared memory region in order to achieve greatly improved scalability and throughput on *rack-scale* and *memory-centric architectures*, while still maintaining consistency. We compare the performance of *MemSpaces* to an established implementation of *tuple spaces* and show that utilizing shared memory for the communication provides performance benefits of an order of magnitude. *MemSpaces* is free software, which is available online[1].

This work is structured as follows. Section II introduces key aspects of *memory-centric architectures* and the *tuple spaces* programming paradigm; Section III outlines previous work on these topics and its relevance for this paper. Sections IV describes relevant parts of the implementation of *MemSpaces* and Sections V and VI contain the findings of this work.

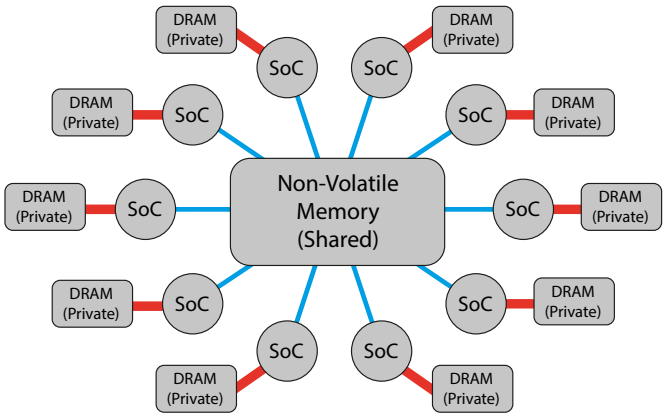


Fig. 1: A structure overview of a *memory-centric architecture*. The main memory of the system is in its logical center, possibly heterogeneous SoCs are connected to the byte-addressable main memory via a shared interconnect, but can also contain dedicated private DRAM.

[1] https://github.com/oaken-source/memspaces

II. Background

This section outlines the properties of the *memory-centric architectures*, and introduces the concept of *tuple spaces*, as defined and later refind by Gelernter et al. in the context of the programming language *Linda*.

A. Memory-Centric Architectures

Traditional large-scale server systems are facing difficult scalability problems, that have been researched extensively [3]–[6]. The dominant Non-Uniform Memory Access (NUMA) architecture, that virtually all of these systems adhere to, mandates that the main memory of the system is fragmented across all compute nodes. Groups of these compute nodes are interconnected, allowing each node of the group to access the main memory of the entire group on a CPU instruction level, with the caveat of varying latency to different areas of this group-local main memory, depending on the number of hops between the nodes. Additionally, the communication overhead to enforce cache coherence across the compute nodes is detrimental for the performance of the system. Consequently, large-scale server systems, such as *rack-scale* systems, generally need to contain multiples of these groups, in order to keep the overhead of the cache coherence protocols manageable. This introduces a *segmentation* of the systems resources, which makes it more difficult to write applications utilizing the entire system and introduces a performance overhead when sharing resources across these groups.

Recently, a new possible architecture for large scale server systems has started to emerge, the *memory-centric architecture*, which promises to alleviate some of these problems. The *memory-centric architecture*, as illustrated in Figure 1, collates the bulk of the main memory of the entire system in the logical center of the system, as opposed to fragmenting it across groups of compute nodes. This large memory pool is shared globally and byte-addressable on a CPU instruction level by all the compute nodes without large differences in latency. To remove the need for a segmentation of the system, the *memory-centric architectures* generally do not dictate the maintenance of cache-coherence across the compute nodes. Furthermore, the compute nodes of the system can be heterogeneous, allowing accelerators and custom hardware based on FPGA chips to directly access the memory, resulting in *rack-scale* systems tailored to the workload they are intended to run.

This architecture design promises significant scalability and performance benefits when compared to traditional large-scale systems, due to the reduced resource sharing overhead in distributed applications provided by the byte-addressable shared memory regions. Additionally, removing the cache coherence protocol can benefit the performance and scalability of the system because the load placed on the compute nodes and interconnect is reduced, allowing such systems to truly grow to *rack-scale*.

First prototype systems adhering to the *memory-centric architecture* have been developed, such as the *The Machine* prototype by HPE [1], and consortia around the concept have started to form, such as the *GenZ Consortium* [7], with the goal of standardizing the system structure and programming interfaces. A first draft of the GenZ Specification has been made available in July 2017.

B. Tuple Spaces

Tuple spaces are a model of *generative communication* defined by Gelernter et al. for the programming language *Linda* [8], [9]. The concept of *tuple spaces* encompasses an implementation defined storage of n-tuples of dynamically typed data entries with a well-defined API for storing and retrieving tuples. The API generally consists of the four functions `in`, `rd`, `out` and `eval`. While the names of these functions vary across implementations, their functionality is consistent with the descriptions outlined below.

Several mature implementations of the *tuple spaces* model exist beyond its original integration in Linda, with some of the more notable ones being *JavaSpaces* [10] for the Java programming language and *LinuxTuples* [11] for the C and Python programming languages, both of which adhere to a Client/Server paradigm. Aspects of those implementations have influenced the development of *MemSpaces* to some degree.

Following are the descriptions for the API of *tuple spaces* based on the original nomenclature introduced by Gelernter et al. in *Linda*. For reasons of practicality, we introduce new names for these functions as outlined below and use these names to refer to these functions in the remainder of this work.

a) `in(tuple):` The `in` function passes the given tuple to the space to be stored. Since `in` is a reserved keyword in python, this function is named `put` in *MemSpaces*.

b) `rd(template):` The `rd` function searches the *tuple space* for a tuple matching the given template and returns a copy of this matching tuple. A match is defined as a tuple of equal number of elements to the template, where all values that are not set to a null- or undefined value in the template must be equal to their counterparts in the matching tuple. If several tuples of the space match, it is unspecified which of these is returned. If no matching tuple is found, `rd` should block until a matching tuple can be produced. This function is more verbosely named `read` in *MemSpaces*. The current prototypical implementation of *MemSpaces* deviates from the specification in that `read` returns an error and terminates if no match can be found.

c) `out(template):` Similarily to `rd`, the `out` function produces a tuple matching the given template. If a match is found, it is removed from the space before returning. An implementation must guarantee that a tuple must only be produced by `out` from the space once. The `out` function is named `get` in *MemSpaces*.

d) `eval(tuple, f):` The `eval` function designates that the given function f should be executed concurrently to the calling process, receiving the given tuple as an argument. The return value of f should be a tuple which is then stored in the *tuple space* as if `in` had been called. While this function

specifies an elegant way of concurrent data processing, its implementation does not lend itself well to many programming languages, and it is consequently missing from many *tuple spaces* implementations. The current versions of *JavaSpaces* and *MemSpaces* do not implement `eval` yet.

III. Related Work

In this section, we identify different approaches for shared resources developed for the *memory-centric architectures*, provide an overview on the current landscape of *tuple spaces* research, and outline relevant techniques of performance evaluation we used to benchmark *MemSpaces*.

A. Programming Models for Memory-Centric Architectures

The problem of efficient memory management on *memory-centric architectures* has already been investigated by Spence et al., who have developed *mds*, a Java and C++ library for managed data structures in shared memory at Hewlett Packard Enterprises [12]. *mds* is designed to provide a high-level API to manage shared memory objects from an application library, with focus on fault-tolerance and object persistence in non-volatile memory environments. The development of *mds* is still in a very early stage and an alpha release of the library exists.

An orthogonal approach to managing large shared memory regions is described in the work by El Hajj et al. on *SpaceJMP* [13]. The authors describe a technique of nested virtual addresses, which allows for the utilization of physical memory larger than the virtual address space size, and for sharing entire address spaces between processes or even nodes. As a consequence, pointers and data structures inside of these address spaces remain consistent even when being shared. The authors describe a prototypical implementation of *SpaceJMP* integrated into the *Barrelfish* operating system, and evaluate this implementation against established benchmarks.

B. Tuple Spaces

Tuple spaces is a mature paradigm [14], and some older implementations have begun to degrade, as libraries and runtime environments have continued to advance. In particular, the *JavaSpaces* implementation, as described by Freeman et al. [10], which was based on the now-defunct *jini* architecture and was later migrated into the *Apache River* project, experiences difficulties in setup and deployment. In comparison, *LinuxTuples*, which was introduced by Will et al. [11], is a younger technology and is consequently easier to consistently build and deploy, but also less mature.

Both *JavaSpaces* and the C API of *LinuxTuples* are strongly typed and do not contain a dedicated tuple type. To compensate for this, *JavaSpaces* utilizes inspection features of the java virtual machine and the `Serializable` interface to send and retrieve tuples in the form of classes, while *LinuxTuples* operates on argument lists of values and a format string to produce memory regions which can be interpreted as tuples, and which require utility functions to extract the values.

C. Performance Evaluation

The *tuple space* paradigm never reached a distinct prevalence rate outside of the academic world. Even within academia, the importance *tuple spaces* has declined. As a result thereof, the *tuple space* community is lacking the wide availability of common benchmark setups. With JavaSpaces being one of the most popular implementations available, Noble et al. [15] have published a performance evaluation studying the feasibility of JavaSpaces for scientific computation workloads. In order to assess the performance of the JavaSpaces implementation itself, the authors measure the throughput of `put` and `get` operations per second. As a payload, `NullEntry` tuples are utilized to measure the maximum throughput of the message based implementation, whereas `DoubleEntry` tuples are used to quantify the performance for scientific payloads. In addition to these tests, scientific workloads are implemented and benchmarked on top of JavaSpaces. The authors draw the verdict that JavaSpaces inflicts many limitations, which is why many scientific problems did not perform well using the *tuple space* implementation.

Fiedler et al. [16] extend the performance assessment methods presented by Noble et al. and propose the SETTLE benchmark framework. Based on the assumption that the payload of a tuple has significant impact on the performance of `put` and `get` operations, the tuple types `IntEntry` and `StringEntry` are introduced in addition to the `NullEntry` and `DoubleEntry` tuple types. Another central assumption is that the performance of `put` and `get` operations heavily depends on the fill level of the *tuple space*. Therefore, an additional benchmark condition is introduced where the *tuple space* is pre-filled with a certain amount of tuples before the benchmark procedure is initiated. The evaluation procedures conducted in Section V are modeled after the core aspects of the SETTLE benchmark framework.

IV. Approach

The goal of *MemSpaces* is to provide a scalable, simple and coherent memory management paradigm for *memory-centric architectures*. One of the challenges of this endeavor was the commercial unavailability of actual hardware to evaluate our approaches on. The following subsections outline how we approximate the properties of the *memory-centric architectures* to be able to produce meaningful evaluation results. Additionally, we provide details of the implementation and identify strengths and weaknesses thereof.

A. Approximating Memory-Centric Architectures

An emulation environment for a system adhering to the *memory-centric architecture* has been made available by Hewlett Packard Enterprise for the *The Machine* prototype [17]. This emulation environment consists of a configurable number of *kvm* based virtual machines, where each guest is given direct access to a specified *POSIX shared memory object* on the host through an *inter-VM shared memory device* (ivshmem) functionality in the hypervisor. However, for the purpose of evaluating shared memory management

approaches, this setup can be simplified to running several processes on the host, accessing the shared memory object directly, thus removing the need for running virtual machines and trading the virtual machine isolation provided by the hypervisor for the process isolation provided by the operating system. We argue that the memory latency and bandwidth, as well as the concurrency aspects of processes on the host system is similar enough to the behavior of processes on the *memory-centric architectures* to allow conclusions on the performance of shared memory based resource management approaches on these architectures.

B. Tuple Space Initialization

The shared memory object underlying a *tuple space* is a shared resource across processes. We decided that, to make the *MemSpaces* API intuitive and useful, it should resemble the file access API specified by POSIX. In particular, we introduce a `memspace_open` function, which takes a path to a shared memory object and returns a pointer to an allocated, opaque `SPACE` structure, used to identify the *tuple space* in subsequent calls to the API, similarily to the `FILE` pointer returned by a call to `fopen`. If the path given to `memspace_open` does not identify an already existing shared memory object, it needs to be created and initialized. However, this introduces a concurrency problem, because only one process can be allowed to create and initialize the shared memory object. The process of providing this exclusive creation is illustrated in Figure 2.

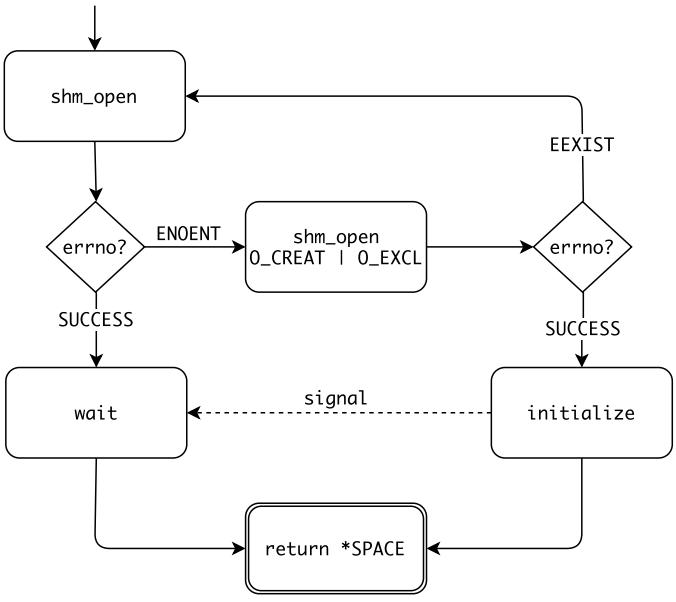


Fig. 2: The process of opening a *tuple space* with *MemSpaces* using `shm_open`. Exclusive initialization is achieved using the `O_EXCL` flag upon creation of the shared memory object.

When entering `memspace_open`, *MemSpaces* attempts to open the shared memory object using `shm_open`. If this succeeds, it tests whether the state is initialized, and enters a waiting state until initialization of the space is finished, if necessary. If opening the shared memory object fails, `memspace_open` attempts to create the shared memory object exclusively, by executing `shm_open` with a combination of the `O_CREAT` and `O_EXCL` flags set. POSIX specifies, that this combination of flags makes creating the shared memory object atomic, ensuring that only one process can succeed in creating it. If this operation succeeds, the process can continue with the initialization of the *tuple space*, whereas any other process concurrently attempting to creat the shared memory object using `shm_open` will fail and have `errno` set to `EEXIST`. In this case, `memspace_open` will know that another process is responsible for initializing the space and will revert to opening the shared memory object regularily and waiting for its initialization to finish.

This process ensures that only one process can initialize the shared memory data when a space needs to be created, using only POSIX-specified functionality. In the context of real *memory-centric architectures*, it needs to be evaluated how well those systems implement this particular section of POSIX, and whether alternative approaches need to be found.

The initialization of a *MemSpace* consists of truncating the file to a minimum length using `ftruncate` and writing *tuple space* metadata to the first page of the shared memory object, which is called the *superpage*. In addition to metadata of the space, the superpage also contains a POSIX *unnamed semaphore* used to lock the space upon operations that require exclusive access to the space metadata.

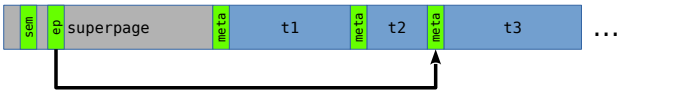
After opening the shared memory object, `memspace_open` will make the corresponding shared memory accessible to the current process using `mmap` on the file descriptor returned by `shm_open`. Subsequent `put` and `get` operations on the space will then operate on this memory.

C. Tuple Storage and Retrieval

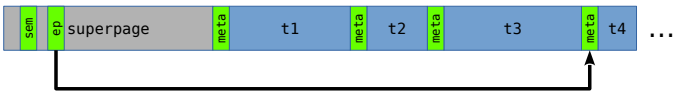
The `put`, `get` and `read` operations of *MemSpaces* are implemented naïvely and without performance optimizations. Tuples are stored in a serialized form, and are immutable. The *tuple spaces* API does not specify an update function, and removing tuples from the space is implemented by invalidation. This introduces holes in the shared memory object, where obsolete tuples are still residing. To solve this, we introduce a bookkeeping operation, which collapses the space and removes all invalidated tuples. Ideally, this bookkeeping operation would be invoked periodically or controlled by a heuristic, but in the current implementation it needs to be invoked manually.

The following paragraphs provide details on the implementation of the separate functions of *MemSpaces*, as further illustrated in Figure 3.

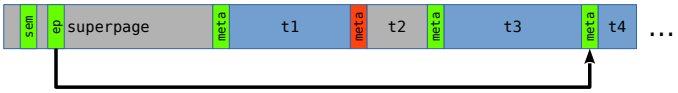
a) `put(tuple)`: The `put` function serializes the given tuple data into a byte array. This byte array is then appended to the tuple data already residing in the shared memory object. Additional metadata is added specifiying the length of the tuple data, the number of elements in the tuple and a flag indicating that the added tuple is the last tuple in the space.



(a) *Tuple space* memory layout before applying operations.



(b) *Tuple space* memory layout after applying `put(t4)`.



(c) *Tuple space* memory layout after applying `get(t2)`.

Fig. 3: An overview of the *MemSpaces* memory layout and how operations on the space can change it. (a) The superpage contains metadata of the space, such as a semaphore (`sem`) and a pointer to the end of the space (`ep`). (b) The `put` operation appends a tuple to the space, (c) the `get` operation invalidates a tuple in the space, leaving a gap.

The *tuple space* superpage contains the offset of the last tuple in the space, which needs to be updated upon inserting a new tuple, but which removes the need to seek the end of the space upon insertion. The insert operation needs to be guarded with a lock on the shared semaphore in the shared memory object to maintain consistency of the space.

b) `get(template)`: The `get` function iterates over the tuples stored in the space, matching them against the given template. During this process, the metadata of the tuples is checked, and tuples are discarded if they have already been invalidated, or their number of elements does not match. If the number of elements does match, the tuple is unserialized from the space, and the values of the template are compared to the values of the stored tuple. If a true match is found, it is returned and the matching tuple in the space is invalidated.

While the removal operation from the space needs to be guarded by locking the shared semaphore, the iterating and matching process can run unlocked, since no changes are performed on the actual data. If a match is found, the lock needs to be acquired. After acquiring the lock, the implementation checks whether the tuple is still valid. If the tuple has been invalidated by a concurrent process, the lock is released and the iteration needs to continue. If the tuple is still valid, it is invalidated and the lock is released.

If the iteration reaches the end of the space without finding a matching tuple, an undefined value is returned and an error is indicated.

c) `read(template)`: Similarily to the `get` function, the `read` function iterates over the tuples of the space, looking for valid tuples of matching size and data. However, since read does not invalidate a matching tuple, no locking is required at all. This may lead to `read` producing a tuple which is concurrently invalidated by a call to `get`. However, since `get` does not destroy the tuple, the actual tuple data is still intact and a concurrent access of `read` and `get` invocations will err slightly in favour of the `read` function.

D. Maintaining Consistency on Concurrent Access

To provide necessary locking for concurrent `put` and `get` operations on the space, *MemSpaces* uses an *unnamed semaphore*, which resides in the superpage of the *tuple space* in the shared memory region. This semaphore is created during initialization of the space using the POSIX-specified function `sem_init`.

MemSpaces can leak these semaphores if a shared memory object is unlinked directly, in which case they will linger unusable in the system until a reboot.

It needs to be evaluated, whether POSIX-specified unnamed semaphores will be functional on real systems adhering to the *memory-centric architecture*. However, the available specifications and architecture visions consistently promise a method of cache-bypassing atomic memory access, which trivially allows to implement spinlocks in the shared memory region, which in turn can be used to implement semaphores.

V. Evaluation

In this section, we provide a description of the benchmark methodologies applied for the evaluation of *MemSpaces* and elaborate on the measurements we gathered.

A. Method of Measurement

The general benchmark procedure is modeled after the core aspects of the SETTLE benchmark framework [16] and all tests are conducted on an HPE ProLiant m710p server cartridge [18] with the detailed specifications denoted in Table I. To make sure that the employed payloads provoke distinct performance impacts on modern hardware, which is several orders of magnitudes faster than the hardware employed by Fiedler et al., we extended the range of suggested payloads with larger double array configurations. Hence, we evaluate the performance of the of `put` and `get` operations for each *tuple space* implementation using the tuple entry types specified in Table II as respective payloads. All *tuple space* implementations are benchmarked within one single machine in order to exclude any impact of the networking performance for the message based *tuple space* implementations and to enable shared memory characteristics for *MemSpaces*.

TABLE I: Specifications of the test systems.

	HPE ProLiant m710p Server Cartridge [18]
Processor	Intel Xeon E3-1284L v4 (Broadwell)
Memory	4 × 8GB PC3L-12800 (SODIMM)
Disk	120GB HP 765479-B21 SSD (M.2 2280)
NIC	Mellanox Connect-X3 Pro (Dual 10GbE)
Operating system	Ubuntu Linux 16.04.1 LTS

With operation latencies to be expected in the sub-millisecond range, we evade timing inaccuracies by measuring the average duration of an operation from 100 successive invocations of the same operation. To avoid further confounding

effects from varying fill-levels of the *tuple space*, 100 `put` operations are always succeeded by the same number of `get` operations, forming a secluded `put-get` measurement cycle. In order to retrieve a sufficiently meaningful dataset, each measurement cycle is repeated for another 100 times [19]. Furthermore, we mitigate the impact of caching effects and dynamic recompilation techniques of the language runtime environments by a foregoing warm-up cycle. Lastly, we perform all benchmarks on both newly initialized *tuple space* instances and pre-filled instances that already contain 10000 tuples of the tuple entry type under investigation.

TABLE II: Tuple entry types employed for the benchmarks. The payload size of successive tuple types increases geometrically by a factor of 8.

Data Type	No. of Elements	Payload in Bytes
Null	0	0
Integer	1	4
Char	32	32
Double	32	256
Double	256	2048
Double	2048	16384

B. Results

The performance measurements yielded by this evaluation are illustrated in Figure 4 and Figure 5. When compared to the best-case performance of *JavaSpaces*, *MemSpaces* is consistently an order of magnitude faster for `put` operations. Especially for small tuple sizes, the advantage of using shared memory for tuple storage is signifant, likely to be caused by the overhead of messaging. For larger tuple sizes, the speedup is not as drastic, but still notable. Considering that the majority of tuples generated in a typical application scenario for *tuple spaces* are small, *MemSpaces* has a large performance advantage to *JavaSpaces*, despite being unoptimized. The `get` operation of *MemSpaces* requires some more optimization, but is also consistently faster than the *JavaSpaces* counterpart.

MemSpaces is not yet optimized for performance. When compared to an equally unoptimized naïve implementation in python using an xmlrpc client/server model, it becomes apparent that the shared memory has a consistent performance advantage of two orders of magnitude. We argue that future performance optimizations of *MemSpaces* have the potential of increasing the performance gap to *JavaSpaces* even further.

The performance difference between operations on empty and pre-filled spaces are statistically insignificant for all tested implementations. This shows that, contrary to the findings of Fiedler et al., the performance of operations on the space does not degrade significantly when the space has been in use for a while which makes *tuple spaces* suitable for long-running applications.

VI. Conclusions

With *memory-centric architectures* appearing on the horizon as potential candidates for future computer architectures, this paper proposed that the *tuple space* paradigm is well suited for representing the characteristics of large shared memory pools that are a central concept of these new architectures. We verified this hypothesis by presenting *MemSpaces*, an implementation of the *tuple space* paradigm based on POSIX shared memory objects, and released the implementation as free software on github. Since actual hardware was not available yet, we imitated the expected behavior of memory centric systems by letting independent processes communicate through a common shared memory region.

In a comparative performance evaluation, we were able to demonstrate both efficacy and efficiency of the *MemSpaces* approach. In a direct comparison with *JavaSpaces*, a state-of-the-art implementation of the *tuple space* paradigm, our implementation performed at least one order of magnitude faster for `put` operations.

At its current state, *MemSpaces* does not make use of any optimization techniques and yet it provides very good performance. This fact demonstrates the immense potential of the approach and encourages us to pursue our efforts in improving *MemSpaces*. Promising optimization strategies include caching strategies, indexing structures and SIMD optimizations to improve the overall performance. As *memory-centric architectures* may employ heterogenous processor architectures and accelerators, we are planning to extend *MemSpaces* to support heterogenous environments as well.

With this work having established *MemSpaces* as a viable strategy for leveraging the potential of *memory-centric architectures*, the vital next step will be to evaluate the performance of our approach using real-world workloads in addition to the synthetic micro-benchmarks presented in this paper.

Acknowledgement & Disclaimer

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866. This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

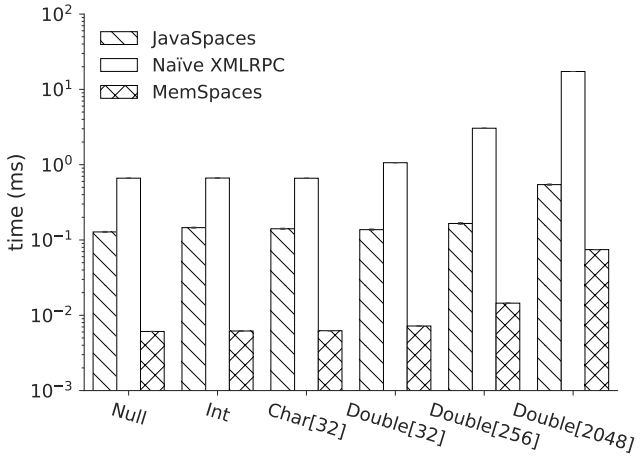
[1] K. Keeton, "The Machine: An Architecture for Memory-centric Computing," in *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '15. New York, NY, USA: ACM, 2015, p. 1.

[2] Hewlett Packard Enterprise, "Fabric-Attached Memory Repository," https://github.com/FabricAttachedMemory.

[3] L. Ma, K. Agrawal, and R. D. Chamberlain, "A Memory Access Model for Highly-threaded Many-core Architectures," *Future Generation Computer Systems*, vol. 30, pp. 202–215, 2014.

[4] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Scaling Up Concurrent Main-memory Column-store Scans: Towards Adaptive NUMA-aware Data and Task Placement," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1442–1453, Aug. 2015.

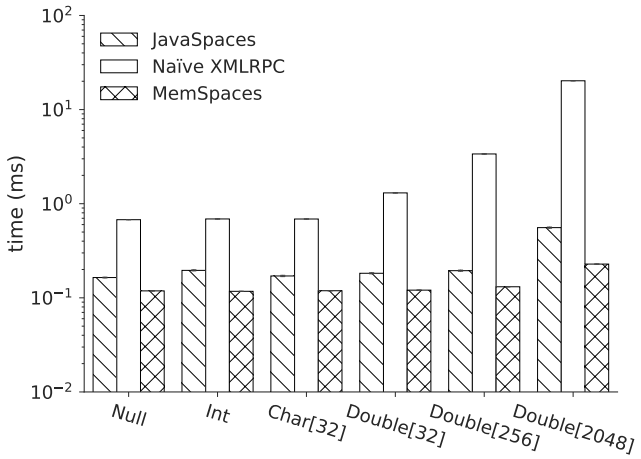
[5] M. Plauth, W. Hagen, F. Feinbube, F. Eberhardt, L. Feinbube, and A. Polze, "Parallel Implementation Strategies for Hierarchical Non-uniform Memory Access Systems by Example of the Scale-Invariant Feature Transform Algorithm," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2016, pp. 1351–1359.



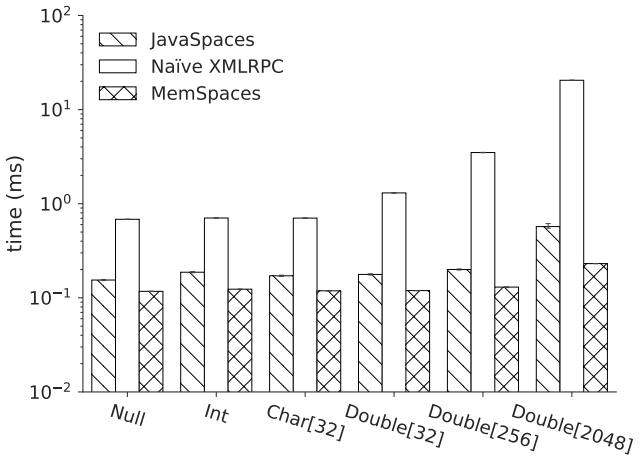
(a) Empty *tuple space* instance

(b) Pre-filled *tuple space* instance

Fig. 4: Execution time per `put` operation on (a) empty and (b) pre-filled *tuple space* instances, for varying payloads. (SD±)



(a) Empty *tuple space* instance



(b) Pre-filled *tuple space* instance

Fig. 5: Execution time per `get` operation on (a) empty and (b) pre-filled *tuple space* instances, for varying payloads. (SD±)

[6] W. Hagen, M. Plauth, F. Eberhardt, F. Feinbube, and A. Polze, "PGASUS: A Framework for C++ Application Development on NUMA Architectures," in *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, Nov. 2016, pp. 368–374.

[7] Gen-Z Consortium. (2017) Gen-Z Draft Core Specification. http://genzconsortium.org/specifications/draft-core-specification-july-2017/.

[8] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 80–112, 1985.

[9] N. Carriero and D. Gelernter, "Linda in context," *Communications of the ACM*, vol. 32, no. 4, pp. 444–458, 1989.

[10] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Professional, 1999.

[11] W. Will, "LinuxTuples," http://linuxtuples.sourceforge.net, 2011.

[12] S. Spence, "Managed Data Structures," https://github.com/HewlettPackard/mds, 2017.

[13] I. El Hajj, A. Merritt, G. Zellweger, D. Milojicic, R. Achermann, P. Faraboschi, W.-m. Hwu, T. Roscoe, and K. Schwan, "Spacejmp: Programming with multiple virtual address spaces," *SIGARCH Comput. Archit. News*, vol. 44, no. 2, pp. 353–368, Mar. 2016.

[14] J. Beilharz, F. Feinbube, F. Eberhardt, M. Plauth, and A. Polze, "Claud: Coordination, Locality And Universal Distribution," in *Parallel Computing: On the Road to Exascale*, G. R. Joubert, H. Leather, M. Parsons, F. Peters, and M. Sawyer, Eds. Amsterdam, Netherlands: IOS Press, 2016, pp. 605–614.

[15] M. S. Noble and S. Zlateva, *Scientific Computation with JavaSpaces*. Berlin, Heidelberg: Springer, Jun. 2001, pp. 657–666.

[16] D. Fiedler, K. Walcott, T. Richardson, G. M. Kapfhammer, A. Amer, and P. K. Chrysanthis, "Towards the Measurement of Tuple Space Performance," *SIGMETRICS Performance Evaluation Review*, vol. 33, no. 3, pp. 51–62, Dec. 2005.

[17] R. Craig, "Emulation of Fabric-Attached Memory for The Machine," https://github.com/FabricAttachedMemory/Emulation, 2017.

[18] Hewlett Packard Enterprise, "HPE ProLiant m710p Server Cartridge QuickSpecs," https://goo.gl/0dV579, 2015.

[19] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07*, vol. 42, no. 10. New York, New York, USA: ACM Press, Oct. 2007, p. 57.