Cwmwl, a LINDA-based PaaS Fabric for the Cloud

Joerg Fritsch and Coral Walker
School of Computer Science and Informatics, Cardiff University, 5 The Parade, Roath, Cardiff, CF24 3AA, UK
Email: {coral.walker, j.fritsch}@cs.cardiff.ac.uk

Abstract — In this paper we introduce a new Platform-as-a-Service cloud environment that combines the LINDA coordination language, an in-memory key-value store, with functional programming to facilitate efficient execution of tenant plugins and applications. In the implementation a tuple space plays a central role in introducing deterministic services for basic parallel programming, including message passing, persistent infinite message pools and transactions. Redis, a key-value store, serves as the in-memory tuple space that glues together parallel constructs (i.e. skeletons) of formerly monolithic business applications to form an elastic distributed application. Although functional programming languages have adopted new runtime technology to achieve parallel execu- tion, which is mostly focused on threads, it rarely offers an obvious way to match functions to threads. We find that the LINDA tuple space and its coordination model offers a general purpose paradigm to tackle synchronisation issues that ties into both domains of computing clouds: computation through supporting common skeletons and big data (analytics) through serving as an in-memory data grid.

Index Terms—Tuple space, PaaS, cloud computing, coordination languages, LINDA, threads, multi-tenancy

I. Introduction

Corporations service developers and end users are not interested in information technology [1]. Their goals are, for example, to run a business process that requires certain resources (e.g. computation and storage), to use an existing application or service, or providing them for third parties. With the advent of virtualization technology and Infrastructures-as-a-Service (IaaS) corporations began to eliminate overcapacity in terms of available computing power. Platform-as-a-Service (PaaS) is the cloud computing layer that further reduces redundant functionality such as the expense for maintaining and administering operating system instances. In the PaaS model, the maintenance of the cloud computing platform, the underlying computing nodes and the operating system instances is done by the service provider. However, regarding the units of scale, there is a mismatch between the requirements of cloud computing platforms and parallel computing languages.

Current parallel computing languages are dominated by threads, a very small unit of scale that commonly has no mobility beyond the physical platform where it is started, and lacks crucial features that would enable true elasticity. Lee [2] is also skeptical about threads as an effective means to support parallelism and argues in favour of coordination languages instead.

Cloud computing platforms on the other hand work around this obstacle by focusing on very large units of scale such as the virtual machine (VM) or the con- tainer (normally consisting of JVM, Rails, or some other runtime system) in which the guest code is deployed [3], [4]. For example, cloud platforms scale guest code through replicating the container and subsequently create the illusion of seamless scaling by clustering the replicas with a load balancer. Considering that at least one instance of every guest application must be running and that frequently additional components such as the host operating system and the application execution engine need to be replicated as well, it becomes clear that modern PaaS clouds have a huge footprint and the size of the infrastructure is still dictated by the number of clients and their running instances rather than by the actual load [5].

Cwmwl (the Welsh word for “cloud”, pronounced “kumul”) is a LINDA-based coordination language that is presented here as the basis for a novel PaaS framework that provides a scalable platform for business applications and supports most of the common skeletons and units of scale. Kachele [6] has published eleven requirements of typical business applications that should be supported by cloud computing platforms but found that “none of the current platforms support a majority of these requested features”. Table I lists Kachele’s eleven requirements and how they are met by a platform based on the LINDA model.

Mirage [7], Erlang on Xen (Ling) [5] and the Haskell Lightweight Virtual Machine (HaLVM) [8] show that it is possible to convert high-level functional language source code into an (exo)kernel that runs directly on the XEN hypervisor that is used in most commercial public clouds such as Amazon EC2. Hence, it is possible to not only shift the responsibility for the PaaS host OS from the customer to the provider (who then hides the presence of the host OS by wrapping it into features of the PaaS framework), but also to completely do away with traditional host operating systems in PaaS frameworks.

Currently all three aforementioned cloud frameworks are single threaded that makes them good candidates for co- ordination models, and a stable coordination layer based on a tuple space could be an ideal basis to glue

Manuscript received April 24, 2013; revised April 21, 2014.
Corresponding author email: j.fritsch@cs.cardiff.ac.uk
doi:10.12720/jcm.9.4.286-298

both OS-less applications and OS-equipped legacy VMs to a hybrid elastic cloud platform.

TABLE I: ELEVEN REQUIREMENTS OF BUSINESS APPLICATIONS [6]

Application Requirement	Explanation	Tuple space
Application-centric approach	Developers only need to focus on core application development and functional aspects.	Simple, high level communications model.
Application-independent approach	Applications executed on PaaS must not be limited to web services.	Separation of concerns in coordination languages is achieved independent of how computation is performed. Associative memory and generative communication are interoperable.
Elasticity	Platform should be elastic and ideally preserve application state during scaling.	Application state is preserved in the central tuple store, elasticity depending on unit of scale. Tasks and Workers run decoupled, in parallel and at virtually any scale.
Virtual addressing	Location is irrelevant, customers must be able to reach their application from everywhere.	Associative addressing specifies what data, what message or what worker is requested rather than an address.
Cloud-independent programming-model	Software should be able to run on both cloud computing platforms and local systems.	Possibility to have a local tuple space.
Updating and bug fixing, Native support for modularity, adaptable design	On the fly updating to achieve high SLAs, preserve application state during updating.	Uncoupling of agents in space and time. No direct communication that could be broken, state is pre- served by tuple store.
Multi tenancy	Isolation and confidentiality inherent in the platform.	Multiple security architectures thinkable e.g. plugins for tenant code execution in Safe Haskell [9], ap- plication execution in light weight VMs, Rusello et al. show that confidentiality of tuples, data and state therein is possible [10].
Dynamic placement	Platform chooses where exactly to deploy a worker, re-balancing in case of changes in load.	
Consumption based cost tracking		

II. RELATED WORK

To start with, we take a look at the PaaS cloud technology and the interactions of this technology with coordination languages and tuple spaces before summarising the research on the use of tuple spaces in large- scale infrastructures, computing clouds and functional programming languages. At the end of this section we will look at possible units of scale and advanced formalisms, including the algebra of communicating processes (ACP) or constraint handling rules (CHR), that may be used to achieve complex coordination in peta-scale and exa-scale cloud computing platforms.

According to Gartner's technology forecast [11] the number of commercial and open source PaaS offerings has been growing steadily since the year 2011 and soon "*all major software vendors will have production offer-ings in the PaaS market*". Although PaaS is the fastest growing cloud computing market, the current scholarly research in this area is relatively limited. The AppScale project, presented in Chohan *et al.* [12], an open source version of the Google App Engine (GAE) PaaS frame-work [13], is one of the rare examples. Appscale is a PaaS framework that aims to scale tier-based Web2.0 architec- tures that in general consist of a web-tier, an application- tier and a data-tier (see Fig. 1). Vaquero [3] also looks at tier-based architectures and investigates a number of entrance points to dynamically scale applications tier by tier.

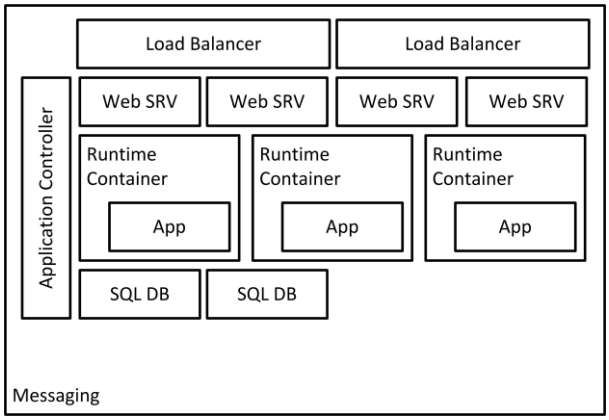

Fig. 1. Software stack of a conventional PaaS framework.

The Cwmwl PaaS framework is a flat fabric that unifies the traditional data-tier, messaging, and computation. A single unified communication protocol is used to store data, coordinate computation, and exchange information about state. Static (partially hierarchic) relationships be- tween web-frontend servers, application servers, message buses and database servers are removed (see Fig. 2). We share the views of Shalom [14] who states that the emulation of tier-based computing in the

cloud or in PaaS platforms does not scale effectively enough to the cloud scale. A unified PaaS fabric scales more predictably than traditional web applications (e.g. a LAMP stack) that need a mix of technologies and protocols to scale each tier separately.

The key differences between a conventional PaaS framework and the unified Cwmwl PaaS fabric can be observed from Fig. 1 and Fig. 2. A conventional PaaS framework scales horizontally through replication of the runtime containers for applications, while the Cwmwl PaaS fabric scales through distribution of the application instances.

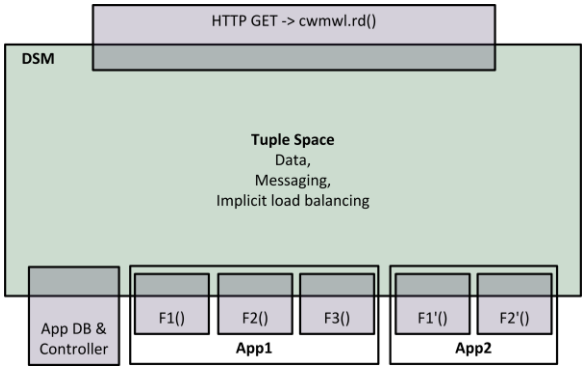

Fig. 2. Unified cwmwl PaaS fabric.

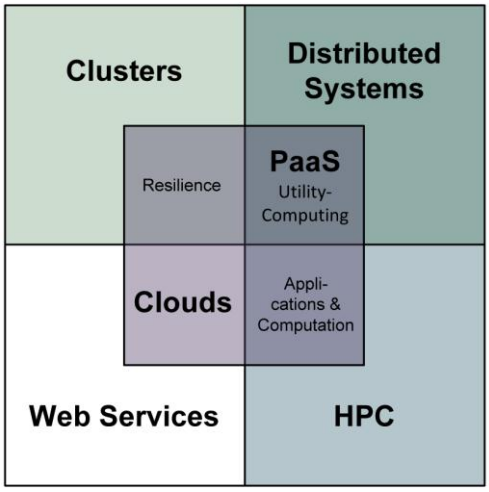

Fig. 3. Lewis Caroll Diagram showing how web services, clusters and HPC intersect with cloud computing (purple). The upper right field "Distributed Systems" is the intersection of clusters and HPC.

Fig. 3 illustrates the relation between cloud computing, web services, clusters and high performance computing (HPC).

We believe that the next generation of cloud computing platforms will embrace upcoming peta- and exa-scale mainstream systems and shift in focus from a technology that mainly delivers web services (Fig. 3 bottom left) to a more abstract service that leverages distributed computing to support non-web-service applications, or rather common business applications (Fig. 3 top right).

Cwmwl is thus developed to exploit distributed systems in cloud data centers by leveraging the strengths of coordination that we find in commodity cluster management tools (e.g. Clustrx [15], parallel Gaussian [16]) and commercial space-based PaaS frameworks (e.g. Gigaspaces XAP [17]).

A. Coordination Languages, Tuple Spaces, and Key-Value Stores

The term "coordination language" was coined in the year 1992 by Gelertner [18] to describe the LINDA programming language that he had proposed in the year

1985 [19]. Since then coordination languages have first influenced distributed computing and HPC, and later Jini/Apache River and web services. The LINDA coordination model consists of a tuple space and a library that implements four primitives, rd(), in(), out() and eval(), as extensions to virtually any (non parallel) programming language. The LINDA primitives manipulate and store tuples, which are key-value pairs, in the tuple space acting as distributed shared memory (DSM). The sender publishes a tuple to the DSM and the receiver queries the DSM without the need to maintain knowledge from where to receive or what process to send to.

Although the performance and scalability of the tuple space is crucial for the usefulness of LINDA-based coordination languages, for a long time the real performance of tuple spaces (and thus their suitability for HPC) remained doubtful. In 2005, Fiedler et al [20] presented SETTLE, an "approach for measuring the throughput and response time of a tuple space when it handles concurrent local space interactions".

Key-value stores, noSQL and BIG Data, all of which are strongly linked to cloud computing, have increasingly gained pace in recent years. Cwmwl suggests a promising cloud infrastructure through combining these new paradigms with a high performance tuple space, living outside the web services world, and obtaining its applicability by simply using tuple spaces and LINDA-based coordination languages. In the Cwmwl infrastructure a key-value store is used to serve application needs, to store BIG Data and as a tuple space, which, if employed wisely, can greatly reduce the software stack, complexity and footprint of applications in the cloud. The reduction of the software stack, complexity and footprint makes Cwmwl a flat PaaS fabric that differs from the more common hierarchical or tier-based PaaS paradigms.

Regarding the implementation of key-value stores, two seemingly opposing trends have been observed: in-memory data grids and distributed key-value stores. In-memory data grids, where all data is kept in memory, provide fast access to data with low latency and high performance. They are frequently used for near real time (BIG) Data analytics. In the Lewis Diagram shown in Fig. 3 in-memory data grids are part of the HPC sector of computing clouds. The corresponding commer- cial offerings for this market segment are "bare metal clouds" that allocate dedicated servers and offerings based on hardware-virtualization where one single instance can

have more than 200GB RAM. Distributed key-value stores that may involve map reducing and additional delays through vector clocks before a query is responded to. Dabek and Peng [23] introduce the Google development called Percolator, which is a combination of both types of key-value stores.

Tuple spaces are a lightweight means of memory virtualization and consequently very similar to persistent storage memory with the added value that tuple spaces can easily scale across the physical boundaries of nodes. An in-memory key-value store was chosen as the basis for the Cwmwl tuple space in order to make it suitable for the implementation of in-memory data grids as well as for the virtualization of (application) memory, data storage and interprocess communication (IPC) across computer or network architectures. The in-memory key-value store Redis [24] was chosen for the following reasons:

- It is very fast and lightweight, which makes it ideal for frequent random access with very low latency as required by tuple space implementations. The expected speed gain compared to a disk based key-value store is approximately 1:100000 [25].
- It supports persistent tuple spaces (if data and tuples are long lived) and preserves state across restarts if required.
- It supports atomic operations. Most key-value stores do not support transactions and use, for example, vector clock schemes [26] to detect and resolve conflicts.
- It supports a number of data structures (for example sets, lists, ordered sets) that can be used to support functions such as tuple matching and advanced co-ordination (see Section III-A of this document).
- It supports key-value pair expiry times which are used to release expired tuples.
- It supports persistence. Regarded as a less important feature of an in-memory tuple store, in practice, it cannot be underestimated as a source for debugging information.

B. *Message Passing Interface (MPI) and Barrier / Eureka Networks*

MPI [27] provides a message passing infrastructure that is, according to HPCWire [28], the de facto standard in parallel communications. Different to LINDA (and thus to the LINDA-based Cwmwl script language) as a higher-level coordination language, MPI is frequently seen as a low level message passing paradigm. Although there seems little in common between LINDA and MPI, in practice, they could very well be put together to provide communication paradigms for distributed parallel processing. For example, the integration of the MPI-based message passing style with coordination logic in implementing barrier/eureka networks is a well-known practice in HPC [29]. There are also a number of hardware solutions to solve coordination issues in barrier / eureka networks [30].

A barrier network is a set of (parallel) processes, all of which are required to signal a certain event in order for the main computation to continue, while a eureka network requires at least one of the participating processes to signal the desired event before the main process can move on. Barrier is supported in the MPI 3.0 standard (the MPI_barrier primitive) and is also implemented in Cwmwl. A commutative example that demonstrates a barrier is presented in Section III-B. The major difference between the Cwmwl implementation and the MPI imple- mentation is that Cwmwl is built on a LINDA tuple space (acting similar to shared memory) while MPI is not.

C. *Threads: Operating System Level Threads and OpenMP*

As already mentioned, threads are not portable and are not an efficient mechanism for distributed processing. A single thread can be seen as a sequential program with shared memory interactions used as an interface to the symmetric multiprocessing (SMP) functionalities of single computing nodes. Parallel programming languages provide relevant and powerful synchronization mecha-nisms, such as mutexes and semaphores in a single node environment where memory can be easily shared.

The main reason for the current domination of the threading model is that as hardware has advanced rapidly over time, the differences in speed between local shared memory (as used by the threading model) and shared distributed memory has kept growing, thereby making local shared memory much faster than any form of interaction with remote nodes.

However, the landscape is changing. Recent patent filings concerning (distributed) shared memory [31], [32] have been successfully implemented and commercialized.

With the widespread use of cloud techniques, not only inter-node communication but also the migration of virtual nodes and workload within a cloud computing platform are gaining importance. This leads back to the idea of better granularity (smaller computational units, single-threading), the realization of which may rely on message passing, asynchronous interaction, and, most importantly, serving as building blocks, the tuple space and its high-level coordination services. It is worth noting that, although the LINDA coordination language tackles synchronization issues at a much higher level of abstrac-tion [28], it is applicable for managing low-level threads as well as for inter-node coordination, which makes it a convenient candidate in building Cwmwl tuple space based coordination services. The outcome is a lightweight Cwmwl with finer granularity, and better portability.

III. Design and Implementation

Cwmwl is a PaaS fabric that consists of

- Messaging and serialization based on the UDP protocol.
- Secure and elastic workers.

- A means to identify and communicate with workers and processes.

Fig. 5 illustrates a high level overview of the Cwmwl structure. Cwmwl consists of a central tuple space (TS) and several application servers (AS). For a distributed application that uses the master-worker skeletons developers will need to write a worker process, upload the code package to an application server and start it. To interact with the worker the developer uses the Cwmwl primitives to query the TS for tuples that have been published by a worker process. The TS serves as DSM, IPC and in- memory data grid, thus collapsing traditional multi-tier applications. To scale the distributed application more workers must be deployed. This is merely a replication and does not require the implementation of an application load balancer that would represent additional overhead, to make use of the added computation power. The TS and the AS nodes are instances of guest virtual machines in commercial clouds. Cwmwl script is a means of coordi- nating and imposing constraints on a large pool of workers that, at large scale with hundreds or thousands of worker instances, will otherwise quickly become unmanageable. Whilst, for example, in commodity cluster management tools the coordinating middleware and the executed ap- plication are separate layers of software, Cwmwl script is part of the distributed application itself (see also Section III-C of this paper). Fig. 6 depicts a UML deployment diagram of the Cwmwl fabric. The upper part of the UML diagram illustrates a developer accessing the platform to upload the application (code) to the application database (AppDB). An application controller (AppController) is in charge of deploying the users' applications from the AppDB to the available guest virtual machines. The lower part of the UML diagram illustrates a client accessing the deployed application through an intermediary web server (here lighttpd) using HTTP GET requests to the tuple store as a replacement for the LINDA rd() primitive.

Cwmwl is written in Haskell, a functional programming language where the LINDA primitives are embedded into Haskell as Domain Specific programming Language (EDSL). The fact that Haskell allows EDSLs to use nearly arbitrary syntax via Quasi Quotes [34] and via the Haskell parsing library Parsec [35] makes it a good choice for embedding Cwmwl primitives. The reason that Haskell is used is that as a functional programming language it is inherently parallel, supporting asynchronous operations, and its code base is easy to maintain and test. Cwmwl is the first attempt to build a tuple space based cloud infrastructure using Haskell. Haskell has also been used to develop [36] a UDP-based message queue that is also part of the Cwmwl framework.

A. Tuples and Templates

The syntax of the LINDA primitives that are used in Cwmwl has been kept similar to the original formulation of tuple spaces. Although the host language Haskell is strongly typed, Cwmwl does not use Haskell data types and tuple matching is purely syntactic. As proposed by Wells [37], data-type conversions are handled by Cwmwl to create a truly heterogenous system that is not limited to any (type of) programming language.

A limited number of types (that could easily be extended) are currently inferred by the Cwmwl parser. The abstract syntax tree (AST) of the Cwmwl interpreter is built for the types string, integer and some other types (identifier, query, operation) that implement the domain abstractions that are required to model the LINDA functionalities. Strong typing can be achieved by segmenting the tuple space where every segment is created to store only a specific type [38]. A further option to preserve types is encapsulation, for example using JSON or ByteString.

Tuples stored in the key-value store (or in any tuple space) are accessed using a method that is called associative lookup, which matches tuples based on templates (also called anti-tuple). A template is similar to a tuple, except some of its fields may be replaced by a NULL value for wildcard matching. A template is said to match a tuple provided the following two conditions are met [39]:

- The template is the same length as the tuple.
- Any values specified in the template match the tuple's values in the corresponding fields. There are two kinds of matches between a template value and a tuple value: exact match where the two values are exactly the same, or wildcard match where a NULL template value matches any tuple value.

For example, the template (isFib, NULL) will match the tuples (isFib, 20365011074) and (isFib, 11021972), but not the tuple (isFib, 20365011074, ''true''). The example shows that a given template can match several tuples so the matching between template and tuple is not always unique. In Cwmwl, the interpreter uses the tuple template as a key to the Redis set data type, which is an unordered collection of strings. Since the interpreter cannot forecast if the key to a new key-value pair will always represent a unique mapping or if additional key-value pairs that reduce to the same key will follow, all new entries must start out as a Redis set with a single member. Tuples can be added to a set or removed from a set in O(1) constant time. Sorted sets are available and could for example be used to queue tuples that represent tasks or workers using a FIFO strategy. Identical work tasks need to be saved only once, and can be executed several times, either in parallel or sequentially.

B. Units of Scale

A tuple space matches the master-worker scheme quite naturally. Campbell [40] argues that the LINDA model is particularly natural for implementing some cases, such as task queues and recursive partitions. As for other skeleton he states that the segmentation of the tuple store (a technique that is frequently used in commercial tuple space implementations) and a degree of coordination

would be beneficial to harness the evolving complexity. For example, a sequential (pipelined) execution of processes with data flowing between them, featuring systolic access patterns (see Fig. 4), if implemented using a tuple space, would have each stage consuming the tuple from the previous stage and produces the tuple for the next stage. This is inefficient, considering that in order for tuples to pass between stages, the developer would have to index, track, and modify the tuples at each stage. An example of such tuple could be out (myWorker, prev_state, next_state, data). As a possible solution Campbell proposes to segment the tuple space and let the segments represent queues similar to pub/sub channels. For example, a sequential process may consist of the stages abcd where stage a produces a tuple in its result tuple space (segment) for stage b. Stage b reads the next tuple from there and writes its output to its own result tuple space. The result tuple space segments are comparable to pub/sub channels and reduce the amount of state and control information that needs to be passed around with every tuple, thus increasing the overall efficiency.

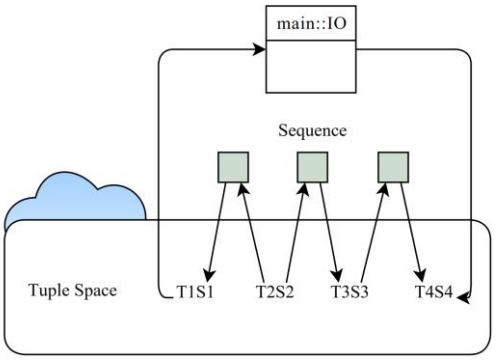

Fig. 4. Tuple space used with an algorithmic skeleton that requires sequential execution (pipelining) and results in a systolic access pattern to the tuple space.

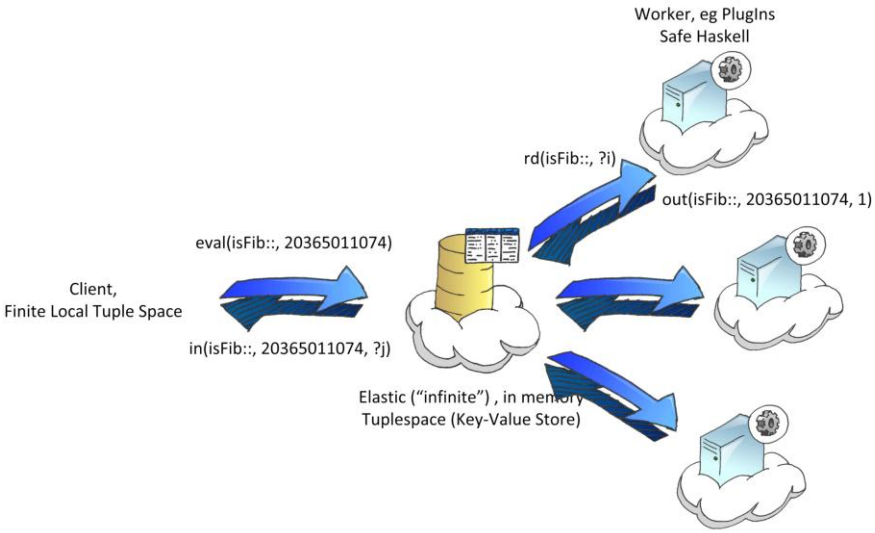

Fig. 5. High level architecture of the Cwmwl PaaS fabric. Workers can consist of any unit of scale: e.g. plugins, functions.

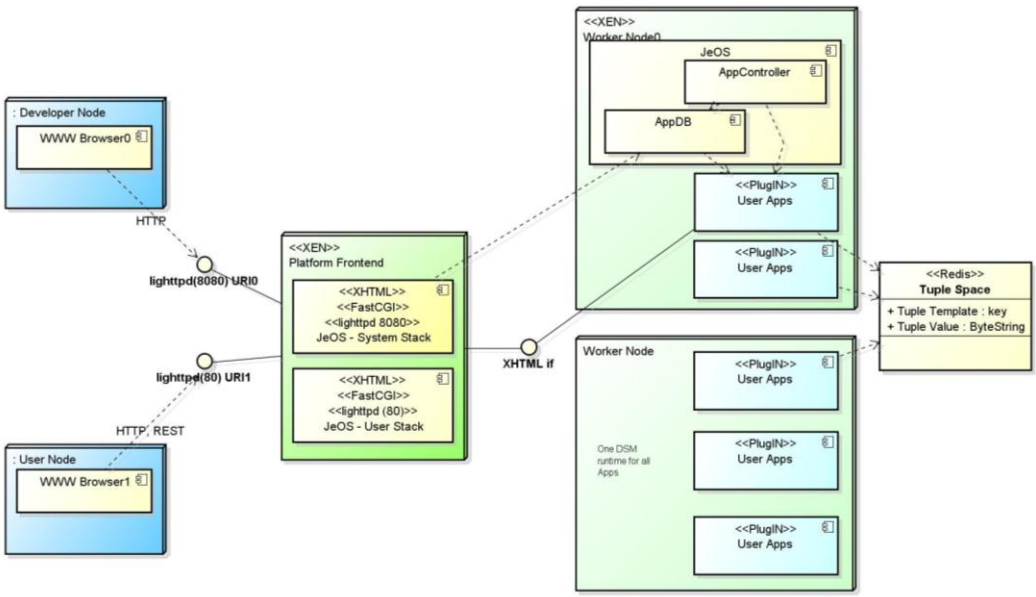

Fig. 6.UML diagram of the Cwmwl PaaS fabric.

In Cwmwl, three operators are available to express sequences and parallelism (see Table II for a description of the Cwmwl script operators) enabling the interpreter to automatically maintain the computation states and thus abstracting the need to manually maintain state and control information away from the developer. Cwmwl operators are aligned with the Algebra of Communication Processes (ACP) [41], among which there is a strong binding sequence operator ; that is used to express the sequential execution of processes, a non-binding sequence operator & that is used for barrier execution of processes and an exclusive choice operator | that performs eureka execution. Furthermore Cwmwl script adopts the axioms and the process algebra as defined by the ACP. Equation (1) reprints the ACP axioms using the Cwmwl script operators in the notation. ACP is defined by more axioms that are currently not relevant to Cwmwl and thus not reprinted in this paper.

TABLE II: CWMWL SCRIPT OPERATORS

Operator	Description
;	strongly binding sequence
&	non-binding sequence (commutative), and ''parallelism'', barrier network
|	exclusive choice, succeeds if any of the operands does so, heureka network

$$\begin{aligned} a\&b &= b\&a \\ (a\&b)\&c &= a\&(b\&c) \\ a\&a &= a \\ (a\&b);c &= a;c\&b;c \\ (a;b);c &= a;(b;c) \end{aligned} \quad (1)$$

multiple execution instances that must be executed within the constraints that are specified by the Cwmwl operators and the axioms of the ACP. The subset of the ACP that is illustrated in Table

An application that is run in Cwmwl may involve multiple execution instances that must be executed within the constraints that are specified by the Cwmwl operators and the axioms of the ACP. The subset of the ACP that is illustrated in Table II and Equation 1 is implemented in the Cwmwl interpreter through storing the running state of each sequential instance in a tuple template rather than in the tuple itself. A key is used for tracking a particular state and has the form of (myWorker, NULL):$id where $id is an integer that represents the current state of the instance (myWorker). It can be incremented and queried to make sure communication take place at the right state. Redis commands are used in manipulating, incrementing and tracking state keys.

A commutative (non-binding) sequencing operator & is implemented for all computations where there is a choice as to what operand gets evaluated first. The execution results of the instances involved are stored in the Redis set datatype. For commutative sequencing which task is to end when all instances have signaled a certain state, the Redis scard query is used to return the number of elements in the set that contains the results of the computations. Each instance must succeed (i.e., store a result in the set) to let the parallel commutative composition succeed. The number of elements in the set determines whether this is the case or not. This can be used for a barrier operation by requiring (ready) signals from all the involved processes before the next processing stage. In addition to the two sequencing operators Cwmwl supports a choice operator | that succeeds if any of the operands succeeds.

C. How to use the Cwmwl EDSL

Cwmwl primitives are embedded into Haskell as a Domain Specific Programming Language (DSL) that consists of a parser, an abstract syntax tree, algebraic operators, and a small interpreter. Three LINDA primitives are included in the Cwmwl primitives:

- Rd() retrieves a tuple that matches the given template
- In() retrieves a tuple that matches the given template and permanently removes the retrieved tuple from the tuple space
- Out() stores a tuple into a tuple space

Program 1 Pseudo Cwmwl implementation of a pipelined algorithm.

```
import language.CWMWL

main :: IO()

   runCWMWL[
   while (data) {
      in (myWorker, NULL) ;
     --reads a tuple that belongs to a
     --sequential computation

      out (myWorker, data) ;
     --writes back a tuple and updates
     --the state of the computation
   }
   ]
```

The LINDA eval() primitive, that spawns a new process, is currently not implemented in Cwmwl. In contrast to some LINDA implementations, such as C-Linda and JavaSpaces that support both blocking and non-blocking versions of rd() and in(), Cwmwl supports only non-blocking primitives. A blocking op- eration requires a worker to wait until its desired data is ready in the tuple store before its next stage of execution, contrary to the notion of uncoupling, and to a certain extent making scaling in space and time difficult. In the case that a blocking operation is unavoidable, it can be implemented through the blocking read primitive LPOP supported by the Redis list data type. Program 1 and 2 demonstrate how a Cwmwl script is executed. runCWMWL [] is the

Cwmwl interpreter function which has a Cwmwl script as its argument. It returns an instance of the Either monad (Either String) containing either an error or a result. In Cwmwl script, – is used for commenting, which is the same as in Haskell.

Program 1 shows the representation of a pipelined algorithm (illustrated in Fig. 4) using the strongly binding Cwmwl sequencing operator; that abstracts the underlying systolic access pattern away from the developer and leads to code with better expressivity. Program 2 shows the use of the non-binding Cwmwl sequencing operator & where both operands need to succeed, similar to a barrier network.

Program 2 Pseudo Cwmwl implementation of a two non-binding sequences.

```
import language.CWMWL

main :: IO()

   runCWMWL[
   in (myWorker, NULL, NULL) & 2
   --And parallelism coordinated with
   --a non-binding sequencing operator
   --valid if two tuples are present

   in (myWorker_a, NULL, NULL)
       & (myWorker_b, NULL, NULL)
   --And parallelism of two different
   --workers, valid if both
   --computations have finished
   ]
```

Cwmwl formalisms that are based on the ACP axioms increase the expressivity in the sense that, compared to plain vanilla LINDA, the code can better represent the state constraints of sequential, concurrent or (even) parallel skeletons, as often required in peta-scale clusters. Furthermore, ACP-based Cwmwl formalisms could make it possible to revive interest in structured programming by partially harnessing the non-determinism that is caused by the complete uncoupling between processes and the tuple store and tuples that are widely spread in time and space.

IV. Map Reduce: A Data-Oriented Example

Programs 3 and 4 give a data-oriented example of how the Cwmwl TS can be used to support distributed computations by providing DSM and a channel for interprocess communication. The programs implement a data intensive map-reduce algorithm for matrix multiplication [42] that partitions Matrix A and Matrix B into sub-matrices, and then performs the multiplications in parallel. Deploying this small distributed application involves the creation of the Redis Tuple Space and corresponding computing nodes - a mapper node and multiple reducer nodes.

A mapper node loads two sparse matrices from a csv file into the Cwmwl tuple space. The reducer nodes carry out the multiplication. Currently AWS EC2 (and many other large commercial clouds) does not support IP multicast, thus the tuple space must be registered to the computing nodes by means of a configuration file.

Program 3 The mapper code that casts matrices A and B from a csv file into Tuple Space.

```
import qualified language.CWMWL as cwmwl

mapper = do

     do csv <- getContents
          case parse csvFile "(stdin)" csv of
               Left e -> do putStrLn "Error in csv file import:"
                            print e
               Right r -> do
                            (key, value) <- castTpl r
                            --e.g. (a:i:k, value) ... (b:k:j, value)
                            cwmwl.out (key, value)
```

Program 4 The reducer code that executes matrix multiplication.

```
reducer = do
    let indices = [(i, j, k) | i <- [0..rowsA],
                               j <- [0..colsB],
                               k <- [0..colsA]]

    -- Since `cwmwl . in` returns a value in the IO monad,
    -- we can't just multiply the values returned.
    r1 <- mapM (\(i, j, k) -> (cwmwl . in) (a, i, k)) indices
    r2 <- mapM (\(i, j, k) -> (cwmwl . in) (b, k, j)) indices

    -- We can multiply r1 and r2 together though,
    -- since they are values extracted from the IO monad
    return $ sum $ zipWith (*) r1 r2
```

Depending on the available network bandwidth and speed, the performance of the mapper may benefit from the fact that loading data into key-value stores is much faster than casting data into a relational database [43]. Since the Cwmwl TS is based on a key-value store, the tuple (template) must contain all information required to address the matrices and their cells. In this example, we work with tuples of the format (a:i:k, value), in which a, i, k denote the matrix, the row, and the column, respectively.

The reducer function multiplies, depending on the configuration of the indices, one or more sub-matrices and sums the results. It would be possible to introduce an intermediary mapper by, for example, splitting the multiplication from the addition and have the reducer sum up the results only.

There are two problems involved in extending Haskell with the Cwmwl tuple space. Firstly, access to a tuple space involves network communication, and any communication over a network is placed in Haskell in the IO Monad, which makes all its subsequent computation (such as the multiplications in our example) impure. This is a problem unique to Haskell, while in some other functional languages, such as Erlang, communications are untyped. Secondly, Haskell uses a static type system which requires the two processes that use a tuple space as DSM to use the same data type. Cwmwl can work around this problem using JSON Frames (see also section III-A of this paper) or Scoped Type Variables [44].

The map reduce example shows that Cwmwl can easily connect multiple processes that reside on different physical nodes and provides a means for distributed programs to access data in a way similar to accessing data locally. Using the Cwmwl TS, interprocess communication and DSM is abstracted away from the developer. Accessing data, memory or inter-process communication are three distinct tasks in most state-of-the-art distributed applications that each require distinct efforts to implement, but in Cwmwl they are merged into one simple task - accessing tuples in Cwmwl TS.

V. Performance Evaluation

The SETTLE [20] framework is used to assess the performance of the Cwmwl tuple space. SETTLE assesses tuple space performance as execution time and throughput as a function of:

- Number of (concurrent) clients.
- LINDA operation: out() or in().
- Payload size of tuple (through changing the embedded data type).
- "Age" of the tuple space defined as the number of entries in the tuple space at the start of the test.

A. Experimental Design

The main goal of the experimental design was to approximate the environment as seen by real applications and to assess the impact of bandwidth and latency on the measured mean performance.

All of the experiments were conducted in the Amazon EC2 cloud. The central tuple store used a Cluster Compute instance so that throughput would not be limited by the available network bandwidth, application memory or computing capacity. Since Redis is single threaded and is best deployed on bare metal hardware without hypervisor [24], an AWS EC2 Cluster Compute instance is the best available match for these requirements. AWS EC2 Cluster Compute instances are based on Hardware Virtual Machines (HVM) where the guest VM runs as if it were on a native hardware platform [45].

AWS EC2 Cluster Compute instances have 60.5GB RAM, 8 physical cores and 10-Gigabit Ethernet connectivity. The clients C1 ...C20 were on separate instances of the type M1 Large. M1 Large instances have 7.5GB RAM, 2 virtual cores (4 EC2 compute units) and high I/O performance with unspecified network speed.

Before the benchmarks were executed, the systems were modified to reuse and recycle TCP connections. Additionally the default range for TCP source ports was changed to the maximum port range: 1024 - 65535 (see Program 5).

Program 5 Linux configuration file /etc/rc.local to modify all systems at boot time.

```
sudo bash -c 'echo 1 > \
 /proc/sys/net/ipv4/tcp_tw_reuse'

sudo bash -c 'echo 1 > \
 /proc/sys/net/ipv4/tcp_tw_recycle'

sudo bash -c 'echo 1024 65535 > \
 /proc/sys/net/ipv4/ip_local_port_range'
```

Two scenarios have been tested in our benchmarks: one with a simple tuple structure with changing payload, and the other with a large set of different tuple structures.

The first scenario involves a simple data tuple ("someData", PAYLOAD) with payload sizes ranging from three bytes to 12KB. Regardless of the payload size, all tuples map to the same template ("someData", NULL) which is used as the key in the Redis key- value store and consequently are stored in the same Redis set. Such a usage pattern eventually reduces to the Redis SADD command that works in O(N) time where N is the number of tuples to be added to the set [24]. However, in our test only one tuple is added at a time, and thus the time complexity for each addition is O(1).

In the second scenario, a series of tuples with different structures of the format ("somedata00001", PAYLOAD) are involved. "somedata00001" ranges from "somedata00001" to ''somedata10000", for the reason that the tuple space must easily fit into the memory of an EC2 Cluster Compute instance, even with the largest tuple payload. This usage pattern eventually reduces to the creation of a new Redis set with a single member.

To further simulate the environment as seen by real applications, prior to testing in each of the scenarios, the Cwmwl tuple space was aged with one million tuples with an automatically generated payload of the same payload size.

B. Tuple Space Performance

The initial state of the tuple space, which was simulated by tuple space ageing, had no impact on the performance results. The results confirm that both operations, inserting a new key-value pair and adding an additional value to an existing set, work in O (1) time. Fig. 7 shows a nearly logarithmic relation of the tuple work load size (measured in bytes) on the tuple space throughput (measured in operations per second) for 5, 10 and 15 concurrent clients. The tuple space throughput drops logarithmically with the workload size.

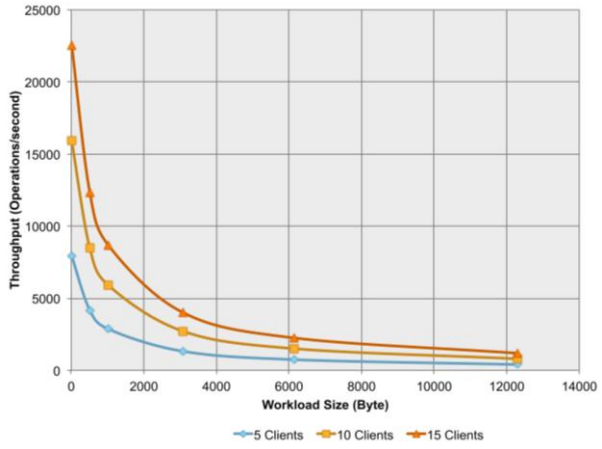

Fig. 7. Impact of workload size on throughput for 5, 10 and 15 clients.

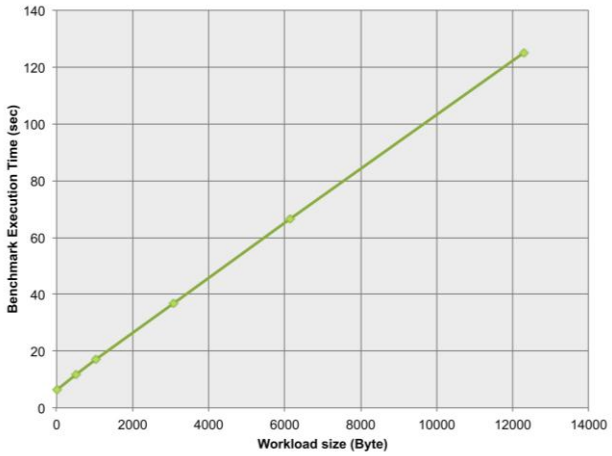

Fig. 8. Benchmark execution time for up to 15 clients.

The standard deviation of the measurements for the benchmark with 20 concurrent clients were very high (see also Fig. 10) and this benchmark is thus not included in this diagram.

Fig. 8 shows that the gross benchmark execution time for up to 15 clients is (within the standard deviation) constant and proportional to the work load size of the tuple. Fig. 9 shows that within the boundaries of our experimental design the tuple space throughput keeps increasing with the number of concurrent clients executing the same benchmark. Obviously the performance limits of our tuple space implementation could not be reached within our experimental design. On the other hand the large standard deviations (Fig. 10) of the benchmarks with 20 concurrent clients and the "knee" at 15 concurrent clients in Fig. 9 may imply that there is a performance boundary between 15 and 20 concurrent clients. Interestingly this is inline with Fiedler [20] who also finds a "knee" around 15 concurrent clients executing the same benchmark on JavaSpaces.

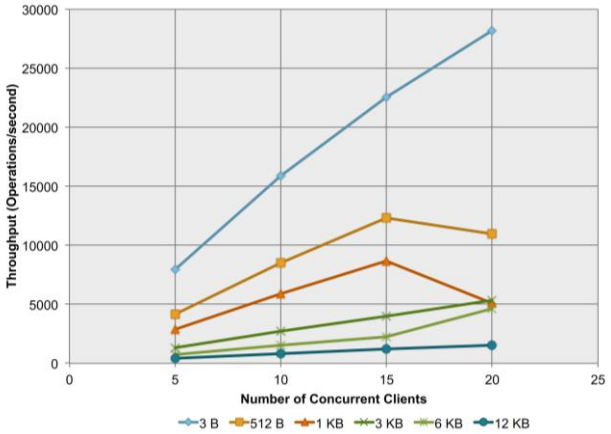

Fig. 9. Impact of the number of clients on throughput for 3B to 12KB.

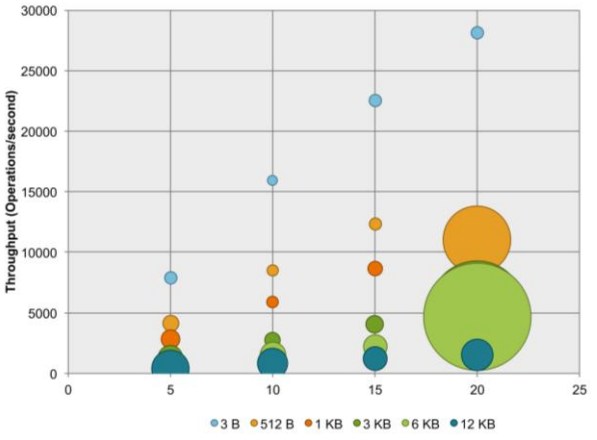

Fig. 10. Impact of the number of clients on throughput for 3B to 12KB.

The Cwmwl rd() primitive produced a constant benchmark execution time of around 6.5s that was not influenced by the size of the payload or by the number of concurrent clients. However, we had the impression that the rd() was altogether less scalable and locked the system network queues significantly longer than the out () primitive that in turn leads to undesirable exhaustions of the connection pools.

The overall results were very consistent and predictable giving a good basis to understand the impact of the Cwmwl tuple space on application performance and scalability. The repeatability of the benchmarks for up to 15 concurrent clients confirms the validity of our findings within the expressed range. Spot checks show

that the ab- solute values (operations / second) are approximately half the performance of the redis-benchmark tool that is included in a Redis installation. The difference could be caused by differences in serialization, sources and exploitation of randomness to generate the tuple payloads or the Redis bindings for Haskell.

The gross throughput with 15 clients was around 1.8 Gbps, slightly higher than the nominal throughput of SATA 1.0 (1.5Gbps).

It is natural to question the performance advantages of the Cwmwl Paas fabric over the current PaaS frameworks. However, most of the current PaaS frameworks are based on a complex software stack and their performance cannot be measured by computation speed or IO alone. According to Zhang *et al*. [46], the performance of current PaaS frameworks may be assessed tier by tier. In practice, PaaS performance is often discussed qualitatively in terms of the time and effort required to deploy a new (web) application or to do a major application upgrade. Also, it is often discussed to what extent, and with what effort, it is possible to elastically right-size (scale up and down) a deployed application. Frequently this is supported by additional middleware that must be subscribed (e.g. Rightscale [47]) and not by the cloud computing platform itself. Map Reduce is frequently sold outside PaaS frameworks as a separate capability that must be configured using work flows and storage, showing again the lack of harmonization of computation, data, applications and web applications in computing clouds.

Cwmwl is intended to rethink PaaS design and to merge brute replication and re-clustering (which is the current methodology to implement PaaS) with distributed computing to improve efficiency and cost. Our foremost design goal is simplicity by achieving a novel unified platform rather than virtualizing and replicating the implementation of a load balanced web application that has existed since the end of the 1990s. The performance figures of the Cwmwl TS that is accessed with the Cwmwl primitives support our claim that this can be done. After all, achieving a performance close to SATA 1.0 is a good start.

VI. Conclusions and Future Work

This paper has demonstrated domain abstractions to achieve a functional tuple space implementation based on an in-memory key-value store. We have introduced the EDSL Cwmwl script that virtualizes (application) memory, data storage and IPC, and detaches them from physical servers and operating systems. We have increased the expressive power of coordination languages by the use of ACP and demonstrated that undesirable tuple space access patterns, resulting for example from sequential algorithms, can be abstracted away from the resulting coordination language.

We will further develop Cwmwl to include the eval() primitive and the implementation will be based on Lua [48], a fast and lightweight functional script language, which is already built into Redis. Lua is very popular as a scripting language in massive multiplayer online games (MMOGs), which share many requirements similar to that of a computing cloud. Although Lua is dynamically typed, mechanisms are available that allow data exchange between Lua and Haskell, thereby supporting invocations between them.

In our current implementation, Cwmwl uses a centralized tuple space that on the one hand creates the problem of a single point of failure, but on the other hand makes the Cwmwl tuple space more capable in terms of migration than a distributed implementation. The elasticity of the Cwmwl tuple space itself is left up to the capabilities of the Redis key-value store. Unlike, for example RIAK, [21] or the Amazon Dynamo keyvalue store [49], the distributable version of Redis (named "Redis cluster") is currently under development, and distribution over multiple instances (e.g. via sharding) is left up to the developer. In an industrial-grade PaaS fabric, the question of in what use cases the benefits of a portable tuple store outweigh the benefits of a distributed tuple store will need to be investigated.

We also intend to investigate the idea of using Cwmwl as the basis of dynamic memory virtualization where an instance of the distributed tuple space is installed on distributed host nodes, which dynamically claims surplus local memory and makes it globally available.

Acknowledgment

The authors are grateful to Viktor Sovietov and to the anonymous referees for their valuable comments and suggestions to improve the presentation of this paper. Program 4 has been kindly contributed by Aaron Stevens.

References

[1] N. G. Carr, "The end of corporate computing," *MIT Sloan Management Review*, vol. 46, no. 3, pp. 67–73, 2005.

[2] E. Lee, "Are new languages necessary for multicore?" in *Proc. International Symposium on Code Generation and Optimization*, 2007.

[3] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynam- ically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 45–52, 2011.

[4] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski, "Appscale: Scalable and open appengine application development and deployment," *Cloud Computing*, pp. 57–70, 2010.

[5] V. Sovietov and M. Kharchenko. Erlang on XEN. [Online]. Available: http://www.erlangonxen.org

[6] S. Kächele, J. Domaschka, and F. J. Hauck, "COSCA: An easy-to-use component-based PaaS cloud system for common applications," in *Proc. First International Workshop on Cloud Computing Platforms*, April 2011, pp. 4.

[7] A. Madhavapeddy, *et al*., "Turning down the LAMP: software specialisation for the cloud," in *Proc. 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud*, vol. 10, 2010.

[8] The Haskell Lightweight Virtual Machine (halvm): Ghc Running on xen. [Online]. Available: http://halvm.org
[9] D. Terei, S. Marlow, S. Peyton Jones, and D. Mazières, "Safe haskell," in *Proc. Symposium on Haskell Symposium*. ACM, 2012, pp. 137–148.
[10] G. Russello, C. Dong, N. Dulay, M. Chaudron, and M. Van Steen, "Encrypted shared data spaces," in *Coordi-nation Models and Languages*, Springer, 2008, pp. 264–279.
[11] Y. V. Natis, "Paas 2012: Tactical risks and strategic re-wards," *Gartner Research Document*, 3 January 2012.
[12] N. Chohan, C. Bunch, S. Pang, C. Krintz, *et al.*, "Appscale: Scalable and open appengine application development and deployment," *Cloud Computing*, pp. 57–70, 2010.
[13] Google app engine - google developers. [Online]. Available: http://www.google.com
[14] N. Shalom. (February 14, 2013). Space-based Architecture and the End of Tier-based Computing. [Online]. Available: http://www.gigaspaces.com/WhitePapers
[15] Clustrx. [Online]. Available: http://massivesolutions.co.uk/ clustrx.html
[16] M. J. Frisch *et al.*, *Gaussian 03, Revision C.02*, Gaussian, Inc., Wallingford, CT, 2004.
[17] Gigaspaces XAP. [Online]. Available: http://www. gigaspaces.com
[18] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, no. 2, pp. 96, Feb 1992.
[19] D. Gelernter, "Generative communication in linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.
[20] D. Fiedler, K. Walcott, T. Richardson, G. M. Kapfhammer, A. Amer, and P. K. Chrysanthis, "Towards the measure-ment of tuple space performance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 3, pp. 51–62, 2005.
[21] Riak Homepage. [Online]. Available: http://docs.basho. com/
[22] DeCandia, Giuseppe, *et al.*, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, 2007.
[23] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.
[24] S. Sanfilippo and P. Noordhuis. Redis. [Online]. Available: http://redis.io
[25] T. Lossen, "Redis – memory as the new disk," in *Proc. NOSQL Europe Conference*, April 20-22 2010.
[26] R. Baldoni and M. Raynal, "Fundamentals of distributed computing: A practical tour of vector clock systems," *IEEE Distributed Systems Online*, vol. 3, no. 2, pp. 12, 2002.
[27] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference (Vol. 1): Vol- ume 1-The MPI Core*, MIT press, 1998, vol. 1.
[28] Programming clusters just got easier. [Online]. Available: http://archive.hpcwire.com/hpc/663546.html
[29] S. L. Scott, "Synchronization and communication in the t3e multiprocessor," *ACM SIGOPS Operating Systems Review*, vol. 30, no. 5, pp. 26–36, 1996.
[30] T. Hoefler, J. M. Squyres, T. Mehlan, F. Mietke, and W. Rehm, "Implementing a hardware-based barrier in open mpi," *Proceedings of KiCC*, vol. 5, 2006.
[31] K. Gopalan, M. Hines, and J. Wang, "Centralized adap- tive network memory engine," U.S. Patent 8,291,034, Oct. 16 2012.
[32] K Gopalan, M Hines, and J Wang, "Distributed adaptive network memory engine," U.S. Patent 8,280,976, Oct 2, 2012.
[33] N. M. Calcavecchia, B. A. Caprarescu, E. Di Nitto, D. J. Dubois, and D. Petcu, "Depas: A decentralized probabilistic algorithm for auto-scaling," *Computing*, vol. 94, no. 8-10, pp. 701–730, 2012.
[34] G. Mainland, "Why it's nice to be quoted: Quasiquoting for haskell," in *Proc. ACM SIGPLAN Workshop on Haskell Workshop*, ser. Haskell '07. New York, NY, USA: ACM, 2007, pp. 73–82.
[35] D. Leijen and E. Meijer, "Parsec: Direct style monadic parser combinators for the real world," 2001.
[36] J. Fritsch and C. Walker, "CMQ-A lightweight, asynchronous high-performance messaging queue for the cloud," *Journal of Cloud Computing*, pp. 1-13, 2012.
[37] G. Wells, "Coordination languages: Back to the future with linda," in *Proc. Second International Workshop on Coordination and Adaption Techniques for Software Entities*, 2005, pp. 87–98.
[38] R. van der Goot, J. Schaeffer, and G. V. Wilson, "Safer tuple spaces," in *Coordination Languages and Models*, Springer, 1997, pp. 289–301.
[39] A. Atkinson, "Tupleware: A distributed tuple space for cluster computing," in *Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2008, pp. 121–126.
[40] D. Capmbell, "Implementing algorithmic skeletons for generative communication with linda," Report- University of York Department of Computer Science Ycs, 1997.
[41] J. Bergstra and J. W. Klop, "Algebra of communicating processes," *CWI Monograph Series*, vol. 3, pp. 89–138, 1986.
[42] Norstad. A Mapreduce Algorithm for Matrix Multiplication. [Online]. Available: http://www.norstad. org/matrix-multiply/
[43] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. De- Witt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proc. ACM SIGMOD International Conference on Management of Data*, ACM, 2009, pp. 165–178.
[44] S. P. Jones and M. Shields, "Lexically scoped type vari- ables," *Submitted to ICFP*, 2004.
[45] I. Amazon Web Services. High Performance Computing on aws. [Online]. Available: http://aws.amazon.com/ hpc-applications/
[46] W. Zhang, X. Huang, N. Chen, W. Wang, and H. Zhong, "Paas-oriented performance modeling for cloud comput-ing," in *Proc. 36th Annual Computer Software and Applications Conference*, 2012, pp. 395–404.
[47] T. Clark, "Quantifying the benefits of the rightscale cloud management platform," Rightscale, 2010.
[48] R. Ierusalimschy, L. H. De Figueiredo, and W. C. Filho, "Lua-an extensible extension language," *Software Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996.
[49] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, *et al.*, "Dynamo: Amazon's highly available key- value store," in *Proc. Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, vol. 41, no. 6, 2007, pp. 205–220.

Joerg Fritsch is a Research Director in the Gartner for Technical Professionals Security and Risk Management Strategies Team. His specialties include information security, data center & cloud security, BIG data (analytics), cloud computing, PaaS, distributed systems, messaging and event-driven systems and very fast networks & servers. He is currently working towards his Ph.D. degree in computer science at Cardiff University, UK. His current research interest includes cloud computing, utility computing and network science.

Coral Walker is a lecturer in the School of Computer Science and Informatics, Cardiff University. She holds a Ph.D. and B.Sc. in computer science, and an M.Sc. in Computational Mathematics. She has been involved in a number of research projects involving program solving environments, Grid virtualization, and Web services based workflow engines, and her research interests include web services, web service based workflow, workflow virtualization, message passing and cloud computing.