Resource dependency processing in web scaling frameworks

Fankhauser, Thomas; Wang, Qi; Gerlicher, Ansgar; Grecos, Christos

Published in:
IEEE Transactions on Services Computing

DOI:
10.1109/TSC.2016.2561934

Published: 03/05/2016

Document Version
Peer reviewed version

Resource Dependency Processing in Web Scaling Frameworks

Thomas Fankhauser, *Student Member, IEEE,* Qi Wang, *Member, IEEE,*
Ansgar Gerlicher, *Member, IEEE,* and Christos Grecos, *Senior Member, IEEE*

Abstract—The upsurge of mobile devices paired with highly interactive social web applications generates enormous amounts of requests web services have to deal with. Consequently in our previous work, a novel request flow scheme with scalable components was proposed for storing interdependent, permanently updated resources in a database. The major challenge is to process dependencies in an optimal fashion while maintaining dependency constraints. In this work, three research objectives are evaluated by examining resource dependencies and their key graph measurements. An all-sources longest-path algorithm is presented for efficient processing and dependencies are analysed to find correlations between performance and graph measures. Two algorithms basing their parameters on six real-world web service structures, e.g. Facebook Graph API are developed to generate dependency graphs and a model is developed to estimate performance based on resource parameters. An evaluation of four graph series discusses performance effects of different graph structures. The results of an evaluation of 2000 web services with over 850 thousand resources and 6 million requests indicate that resource dependency processing can be up to a factor of two faster compared to a traditional processing approach while an average model fit of 97% allows an accurate prediction.

Index Terms—scalability, web service, cloud computing, graph processing, job scheduling, dynamic programming, reactive processing

1 Introduction

Modern web applications such as social networks and services for the Internet of Things are highly interactive. They provide a continuous experience across multiple mobile devices such as smart phones, smart watches, tablets and cars. The data sent and received by billions of devices is mined and analysed by big data algorithms and used for intelligent content creation. For all live traffic reports, pandemic and epidemic disease prognosis, live weather and global movie trends enormous amounts of requests are exchanged and processed. This puts high demands on modern web services. Requests have to be distributed to multiple servers, while the optimal number of necessary servers has to be adapted to the encountered load continuously by scaling up and down. Simultaneously, the high throughput of requests needs to be ensured with minimal processing delays in order to minimise waiting times for the users.

1.1 Background

As the creation of web services handling both logic and scaling is a very complex matter, we proposed Web Scaling Frameworks (WSFs) in our recent work [1], [2]. WSFs take over the responsibilities of scaling by embedding existing Web Application Frameworks (WAFs) in a larger system. By storing all requestable resources in a Resource Database (RDB), we were able to apply a novel request flow routing mechanism minimising the expensive processing of requests. To enable the novel request flow, workers processing requests need to ensure the RDB is kept up to date by processing the resources with their dependencies. A resource has a dependency on an other resource, if the other resource needs to be updated with the original resource. This declaration enables to calculate an optimal subset of resources that have to be updated when an other resource is changed. Our previous mathematical model allows calculating the maximum time available for the dependency processing where the novel request flow approach remains faster than the traditional WAF approach.

1.2 Motivation and Objectives

Finding an efficient resource dependency processing mechanism is a key requirement for building a scalable web service architecture with optimised request routing. With a slow processing performance, only read-driven applications can benefit from the RDB which limits the field of applications for WSFs. The first major objective of this work is to gain further knowledge of how dependencies between resources in web applications can be measured, stored and generated to fit existing application structures. Therefore, we identify a graph as the structure for expressing resource dependencies and relate known graph measures to describe significant dependency structures. To generate dependency graphs, we develop a service based graph algorithm using the APIs of six real-world applications and a fuzzy graph algorithm creating applications based on random graph measures. The second major objective is to find existing and novel algorithms that qualify for optimisation of processing performance and evaluate their performance compared to a typical traditional processing approach. Therefore, multiple graph algorithms in the area of job and workflow processing are evaluated. A novel algorithm using a topological

- *T. Fankhauser and Q. Wang are with the School of Engineering and Computing, University of the West of Scotland*
  E-mail: {Thomas.Fankhauser, Qi.Wang}@uws.ac.uk
- *C. Grecos is with the Sohar University*
  E-mail: grecoschristos@gmail.com
- *A. Gerlicher and T. Fankhauser are with the Stuttgart Media University*
  E-mail: {gerlicher, fankhauser}@hdm-stuttgart.de

sort with dynamic programming is proposed to efficiently compute a forest of valid processing trees alongside a model to calculate the tree processing durations. We conduct a comprehensive performance evaluation between a dependency processing approach and a traditional approach with over 2000 applications, 1 million resources and 18 million requests to find improved performances of up to factor two. The third and final objective is to model the dependency processing duration and analyse the effects of graph measures on the performance. Therefore, a simplified approximation model is developed and evaluated allowing for precise and cost-efficient computation of the processing duration, the duration delta and relative performance improvement compared to traditional methods. Further, a linear correlation of the processing duration with the dependency depth and the cluster size is found, evaluated and analysed for its influence on the processing performance.

The remainder of the paper is organised as follows: Section 2 summarises the results of the literature review of related work. Section 3 identifies the structure and measures of resources and dependencies. Section 4 evaluates suitable graph processing algorithms, whilst Section 5 finds correlations between the processing duration and other graph measures. Section 6 introduces dependency graph and traffic generation algorithms and Section 7 develops the models for the approximation model. Section 8 evaluates both the empirical performance and model fits with a cloud cluster and analyses different graph structures. Section 9 outlines the results and conclusions and gives a perspective on future research.

2 Related Work

We reviewed and classified related work with respect to our three major objectives as presented in Table 1. Work in the *Dependency Structure and Generation* category deals with caching policies and graph processing. In the *Algorithms and Evaluation* category we study work related to scheduling algorithms and reactive programming. Finally, work in the *Modelling and Analysis* category deals with measures and algorithms for self-similar traffic and web resource modelling.

TABLE 1
Categorisation of related work

Structure and Generation (2.1)	**References**
Caching Policies	[3], [4], [5]
Graph Processing	[6], [7], [8]
Algorithms and Evaluation (2.2)	**References**
Scheduling Algorithms	[9], [10], [11], [12], [13], [14], [15], [16], [17], [18]
Reactive Processing	[19], [20], [21]
Modelling and Analysis (2.3)	**References**
Service Measures	[22], [23], [24], [25], [26], [27]
Traffic Modelling	[28], [29], [30], [31], [32], [33], [34]

2.1 Dependency Structure and Generation

Traditional web services employ cache eviction approaches based on policies to select optimal cache contents. Evictions can be based on the distance between objects such as in the SACS system [3], recommendations systems analysing proxy access logs [4], or enhanced traditional eviction policies [5] such as Least Frequently Used (LFU) or the Weighting Replacement Policy (WRP). In contrast to the aforementioned approaches, we propose to explicitly declare all dependencies between service resources and store all requestable web objects in a persistent resource database instead of a volatile cache. Instead of cache eviction, our approach requires a precise update mechanism to keep the resource database in sync with the data. The work in [6], [7], [8] presents a comprehensive survey over implemented algorithms, usability, performance and scalability of multiple large scale graph processing platforms such as *GraphChi*, *Apache Giraph*, *GPS*, *GraphLab*, *GraphX*, *Neo4j*, *Apache Hadoop*, *YARN* and *Stratosphere*. We propose to store the dependency graph in one of the aforementioned large scale graph processing platforms and use the algorithms we develop in this work to extract a forest of individual dependency trees for each resource. Unfortunately, we can not use the datasets from [6], [7], [8] as they describe relationships between data entities, e.g. users but not service resources as required for this work. Hence we need to create our own service datasets.

2.2 Scheduling Algorithms and Evaluation

We identified the processing of dependencies as an optimisation problem in the domain of job and workflow scheduling. Job scheduling has a long history in the project management context [9], where critical path planning and scheduling [10] was developed to find a sequence of dependent jobs that determines the maximum duration of a project. Applications for job scheduling have been found and transferred to QoS-based scheduling in grid computing [11] and network activity times [12]. In the domain of workflow scheduling, the authors in [13] present a comprehensive survey and analysis of scheduling schemes in cloud computing, where a scheduling scheme tries to map the workflow tasks to multiple virtual machines based on different functional and non-functional requirements. The authors in [14] present a scheduling strategy that maps workflow tasks to multiple clusters and clouds with optimised balancing. In our work, we transfer and apply the critical path problem from the project management context to dependency processing of web resources. The processing of dependencies are the activities and the critical path duration is the maximum dependency processing duration. From workflow scheduling, we use basic structures such as a Directed Acyclic Graph (DAG) to declare dependent tasks. Our major objective however, is not to distribute work to multiple virtual machines as presented by [13], [14], but find an optimal forest of processing trees from a dependency graph. The trees are then used to process dependencies in an optimised fashion and calculate the maximum processing duration as critical path. In [15], [16], [17] the authors show that a topological sorting of jobs for correct orderly scheduling can be calculated in linear time. The dynamic programming method [18] solves complex problems by breaking them down into a collection of simpler subproblems if the

TABLE 2
Conceptual distinction of dependency related graph types

Resource Graph	All resources of a web service. Nesting of resources determines edges.
Dependency Graph	Dependencies between resources only. No resources that have no dependencies.
Structure Graph	Logical resource structure with unexpanded entities. Can be expanded to a resource graph.
Processing Tree	Tree that considers processing precedence constraints for a single resource as root.

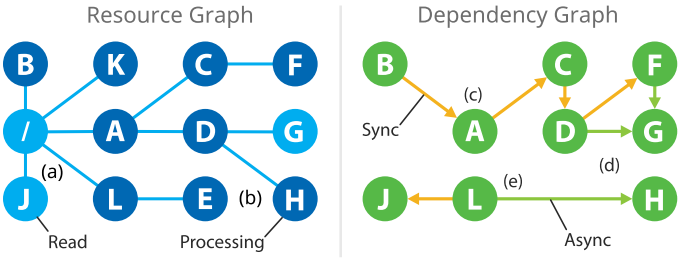

Fig. 1. A resource graph representing all service resources with an extracted dependency graph showing dependent resources. Resources in the resource graph are either read (a) or processing (b) resources. Dependencies in the dependency graph are either synchronous (c) or asynchronous (d). Clusters (e) highlight disconnected groups of resources.

subproblems have an optimal substructure and are overlapping. Our proposed approach combines both topological sorting with dynamic programming on a DAG structure to determine the critical paths of processing. The reactive programming paradigm [19], [20], [21] is oriented around data flows and the propagation of changes and thereby fits the requirements for dependency processing. In this work we use and apply reactive programming with eventual consistency to automatically update resource dependencies with a middleware approach observing incoming changes through requests.

2.3 Service Modelling and Analysis

The authors in [22], [23] improve the cache Hit-Miss Ratio (HMR) with machine learning approaches and define typical values between 0.35 and 0.75. In [24], [25] the processing time of web applications is characterised as a Hyper-Erlang, Weibull, generalised Pareto or Lomax distribution. Work in [26], [27] classifies the graph structure in the web to be heavy-tailed and studies the in-degree and out-degree distributions of a large web crawls. In this work we model the cache hit/miss ratio and processing duration according to [22], [23], [24], [25], [26], [27]. However, the work on graph structures on the web studies the dependencies between multiple web services, where our work considers the resource dependencies within a single web service. Hence, we analyse existing service structures in Section 6. The authors in [28], [29], [30], [31], [32] find that request arrival times exhibit self-similarity and use the following random processes for modelling: Fractionally Autoregressive Integrated Moving-Average (FARIMA), Fractional Brownian Motion (FBM), Poisson Pareto Burst (PPB), Poisson Lomax Burst (PLB) and Circulant Markov-Modulated Poisson (CMMP). For resource popularity [33], [34] the popularity distribution can be modelled using a Zipf distribution with parameter ranges between 0.64 and 0.84. In this work, we use all the proposed random processes (FARIMA, FBM, PPB, PLB, CMMP) for modelling the request arrival times and the Zipf distribution for modelling the selection of resources.

3 Resource Dependencies

In order to minimise the expensive processing of requests, we presuppose the declaration of resource dependencies that have to be processed for each request modifying data on the server. Table 2 presents the conceptual distinction of dependency related graph types, where the required dependency declaration extracts a dependency graph from a resource graph. Beforehand declaration of dependencies can be fully automated as dependency constraints are implicitly contained in the web service's view layer code. The code used to render a resource explicitly defines which data is used to create the view, e.g. an HTML page. Thus, all resources that use the same data have a dependency on each other. Resulting from the dependency processing, resources are stored in a distributed, cloud based resource database.

3.1 Resource Vertices and Dependency Edges

A web service exposes multiple routes delivering different resources such as markup, structured data, images or videos to the consumer. The resource graph in Fig. 1 presents all resources of a web service. Resources can be addressed by traversing through the graph, e.g. */A/D/H*. Each resource is either a read resource (HTTP GET or HEAD) or a processing resource (all other HTTP methods). This distinction is used to apply optimised request routing mechanisms, where read resources and processing resources can be handled by separate, individually scalable subsystems. In the dependency graph in Fig. 1, an edge expresses that after update of resource B, resource A has to be updated as well. Thus, following job and workflow scheduling methodology as in [9], [10], [13], [14], we can model our resource dependencies as DAG. The dependency graph does not contain resource contents, but only the edges between dependent resource URLs and thereby can efficiently be stored as sparse matrix. To store resource contents a distributed storage solution is needed as presented in [1], [2], where mechanisms to synchronise data between machines are reflected in an increased lookup delay. By design however, the lookup delay only influences the read subsystem. Resource dependencies are further categorised to be synchronous or asynchronous. For synchronous dependencies (Fig. 1 (c)), the response to the original request is delayed until all dependencies are finished processing. This guarantees that on response the resource database is updated with all changes triggered by the original request. For asynchronous dependencies (Fig. 1 (d)), the response returns immediately while the dependencies are processed in background. This guarantees that the resource database will be updated eventually. Web sites not caching state on the client mainly use synchronous dependencies as subsequent requests need to reflect all changes from previous requests. Web applications in contrast, cache state and present changes directly to the user which allows dependencies to be processed in the background.

TABLE 3
Interpretation of dependency related graph measures

Depth	Mean length of longest-paths from all vertices
Degree	Mean outgoing number of edges from all vertices
Cluster Count	Number of independent vertex clusters
Cluster Size	Mean number of connected vertices in a cluster
Sparsity	Number of edges divided by number of vertices

3.2 Interpretation of Dependency Graph Measures

In addition to simple graph measures such as vertex and edge counts, there exist more complex measures as presented in Table 3. A further interpretation in the context of dependency processing helps to classify their influence on the structure of web services and performance of processing. The mean dependency depth denotes the average number of steps required for dependency processing and is further detailed in Section 4. It is used in the approximation model in Section 7 to calculate the average processing steps needed. The mean dependency degree is used to estimate the portion of parallel processing, as all children of a vertex can be computed in parallel without collisions. The cluster count and size generally give an indication on the structure of a web service where a low number of clusters and a large cluster size indicate a deeply joint application structure. Based on data in Section 6, dependency graphs are sparse graphs. A low sparsity generally expresses a low amount of dependencies resulting in faster processing.

4 Processing Algorithms

For the efficient processing of dependencies we formulate the problem as follows: Given a vertex from a dependency graph, how long does it take to process all dependencies of the vertex while ensuring correct processing order. The key metric to optimise is the dependency processing duration, which refers to a makespan optimisation in [9], [10], [13], [14]. Consequently, the scheduling of dependencies is distinct from scheduling in an OS, where the major objective is to fairly balance computing resources between long-running processes that appear to be running in parallel. Additionally, it is distinct from scheduling jobs onto a cluster, where the major goals are an optimal distribution of the pieces of work with a high resource utilisation. The bottlenecks in scheduling resource dependencies are the processing durations of individual resource updates that need to be processed in the correct order. Thus, we transform the problem into a DAG structure as used in job and workflow scheduling [9], [10], [13], [14].

4.1 Evaluation

To find and evaluate suitable algorithms, we generate 1000 random dependency graphs with a custom created Incremental Edge Add (IEA) graph generation algorithm. The IEA generation algorithm starts with 1000 completely disconnected resource vertices in the first generation graph. In each next generation, it adds a random, directed edge to the previous generation graph while ensuring no cycles are created. We build 1000 generations of the graph with a maximum number of 1000 edges and give each resource vertex a processing delay of 100ms. The algorithms presented in the next sections are evaluated with a model of the applied algorithm and an empirical data collection from our Raspberry Pi computing cluster which consists of 42 machines. We implement the dependency processing algorithms using the Go programming language and create one million HTTP requests to measure the dependency processing duration.

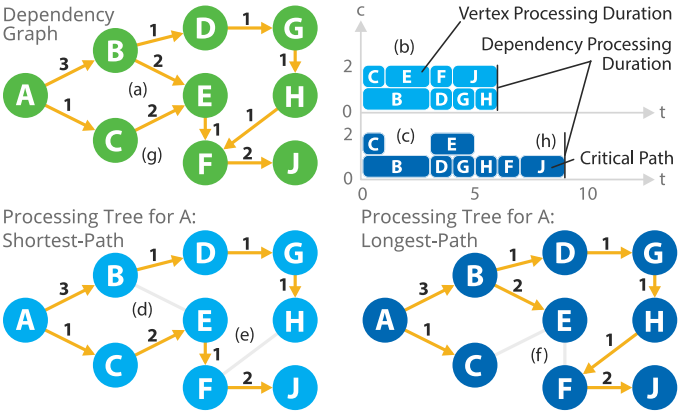

Fig. 2. Dependency processing scheduling problem presented with a shortest-path tree and a longest-path tree approach. (a) highlights multiple processing paths where the performance of (b) is superseded by the performance of (c). However, the shortest-path approach at (d) does not guarantee the correct job order enforced by the longest-path approach at (f). Processing trees depend on their root vertex, e.g. starting at C (g) leads to a different tree than starting at A (f).

4.2 Shortest and Longest-Path Approaches

As Fig. 2 (a) shows, a dependency graph defines multiple contingent processing paths. The resource vertex E can be reached via the path (A, B, E) and (A, C, E). Optional dependencies are not possible as E either contains content from B (B has a dependency on E), or it does not. The dependency graph states both B and C need to be finished processing before E can be processed as it contains changes from both B and C. Hence, an algorithm is needed to calculate the fastest and sequentially correct processing path. Our approach to calculate the processing path and thereby the duration is to weight the edges of the dependency graph with the mean processing delay introduced by the vertex the edge points to. From Fig. 2 (d-e), our collected test data and work in the job and workflow scheduling domain [13], [14], [35], [36], [37] follows that a shortest-path approach is not feasible for dependency processing as it violates precedence constraints. Longest-path algorithms guarantee the precedence constraints, however finding a longest-path in a directed graph is a NP-hard problem which can not be solved efficiently for large datasets. From the domain of workflow scheduling [13], [14] follows that constraining the problem to DAGs enables the application of efficient algorithms. Fig. 2 presents such a DAG, where the longest-path tree for A is shown at (f) and the processing duration is determined by the critical path at Fig. 2 (h).

4.3 A Forest of Processing Trees

Fig. 2 *(g)* additionally shows that processing trees are distinct for each resource vertex. The longest-path tree in Fig. 2

provides the solution for resource vertex A. Resource vertex C can not use the same solution as A, as it has no dependencies in the processing tree for A. The correct processing tree for C is C, E, F, J. In conclusion, for each resource vertex there exists an individual processing tree thus leading to a forest of processing trees. This forest has to be respected in the calculation of the algorithm time complexity. Using an adjacency matrix, a dependency graph with maximum edges can be constructed by filling either the upper or the lower triangular part of the matrix excluding the diagonal with edges where a 1 marks the presence of an edge E between two vertices V:

$$E_{\max}(V) = \frac{1}{2} \cdot (-1 + V) \cdot V \tag{1}$$

Using each vertex as a root for a subgraph of a DAG, the maximum number of vertices $V_{sub,\max}$ in each subgraph must decrease by at least one to ensure no cycles are present:

$$V_{sub,\max}(V) = \sum_{r=0}^{V} V - r = \sum_{r=1}^{V} r \tag{2}$$

4.4 Forest of Processing Trees Extraction Algorithms

We evaluate two algorithms to extract all longest-path trees efficiently: A version of Bellman-Ford that uses negated edge weights and an algorithm we propose that is based on a topological sort and uses dynamic programming. The authors in [35] prove the longest-path problem in edge-weighted directed acyclic graphs to be solvable by finding the shortest-paths in a graph where all edge weights are negated. The Bellman-Ford algorithm can deal with negative edge weights and takes time proportional to $E \cdot V$. The algorithm needs to be executed for every resource subgraph of the dependency graph, so from (1) and (2) follows that:

$$\sum_{v=1}^{V} E_{\max}(v) \cdot v \tag{3}$$

Albeit suitable, we do not use the Bellman-Ford algorithm to extract the forest of processing trees as a faster and more efficient algorithm exists. By topologically sorting a DAG, a linear ordering of vertices is generated guaranteeing a vertex v_1 to come before a vertex v_2 if an edge $v_1 \rightarrow v_2$ exists. The work in [35] proposes an algorithm able to find a longest-path tree from a root vertex in linear-time. The proposed algorithm however only calculates a single-source longest-path tree. For dependency processing a forest of processing trees needs to be extracted, so an all-sources longest-path tree algorithm is needed. For the further modelling of the processing duration, knowledge of the length of the critical path is required for all trees in the forest. Therefore, we extend the single-source algorithm in [35] with a dynamic programming approach so the computed result returns a forest of processing trees and the processing durations of all critical paths:

1: $delays \leftarrow$ processing delays of vertices of DG
2: $order \leftarrow$ topologically sorted vertices of DG
3: $forest \leftarrow$ subgraphs for all v out components
4: $durations \leftarrow 0$ for all vertices of DG
5: **for all** vertices v in DG **do**
6: $\quad suborder \leftarrow$ intersection of v in $forest$ and $order$
7: $\quad distances \leftarrow -\infty$ to each vertex t in $forest$
8: $\quad distances[v] = delays[v]$
9: $\quad$ **for all** vertices s in $suborder$ **do**
10: $\quad\quad$ **for all** children c of s **do**
11: $\quad\quad\quad d = distances[s] + delays[c]$
12: $\quad\quad\quad$ **if** $distances[c] < d$ **then**
13: $\quad\quad\quad\quad distances[c] = d$
14: $\quad\quad\quad\quad$ **if** $durations[v] < d$ **then**
15: $\quad\quad\quad\quad\quad durations[v] = d$
16: $\quad\quad\quad\quad$ **end if**
17: $\quad\quad\quad$ **end if**
18: $\quad\quad$ **end for**
19: $\quad\quad parents \leftarrow$ parents of vertex $forest[v][s]$
20: $\quad\quad$ **if** length of $parents > 1$ **then**
21: $\quad\quad\quad max \leftarrow$ maximum $distances$ of $parents$
22: $\quad\quad\quad$ **for all** vertices p in $parents$ **do**
23: $\quad\quad\quad\quad$ **if** $distances[p] < max$ **then**
24: $\quad\quad\quad\quad\quad$ delete edge $p \rightarrow s$ from $forest[v][s]$
25: $\quad\quad\quad\quad$ **end if**
26: $\quad\quad\quad$ **end for**
27: $\quad\quad$ **end if**
28: $\quad$ **end for**
29: **end for**
30: **return** $[forest, durations]$

Our proposed Forest of Processing Tree Extraction (FPTE) algorithm applies dynamic programming by calculating the topological order only once for the entire set of vertices. A subproblem is defined as finding a single-source longest-path tree based on the total order. The original algorithm [35] calculates only a single longest-path tree for a selected vertex. Further, in our algorithm the calculation of the critical path durations is injected into the original single-source algorithm's relaxation step to reduce the runtime (line 14-16). In detail, the FPTE algorithm initialises its data structures (line 1-4), calculates the topological order of all vertices and creates arrays for the distances and durations for each root vertex v. For each vertex, subgraphs are generated (line 3) from where the algorithm step-by-step removes the shortest-path edges. Using each vertex once as root vertex (line 6), the vertices of the subgraph are extracted in suborder. The distance to each vertex in the subgraph is set to $-\infty$ on line 7, where the distance of a node to itself is the processing delay as shown on line 8. Next, the vertices are traversed in this suborder and each child is expanded (line 9-10). As a next step, the edges are relaxed by checking if the currently stored distance to the child is smaller then the current path (line 11-12). If so, a new longest-path is found to the child and stored as new longest distance and duration (line 13-16). By line 18, the maximum distance to the vertex s is known. If multiple edges point to s, then the edges from the parents with the lowest distance can be removed to keep only longest-paths. At the end of the loop for v (line 28), the subgraph is fully converted to a longest-path tree. The single-source longest-path algorithm from [35] takes time proportional to $E + V$ as it visits each vertex and each node exactly once. Following (1) and (2), our all-sources and critical path durations extension to the

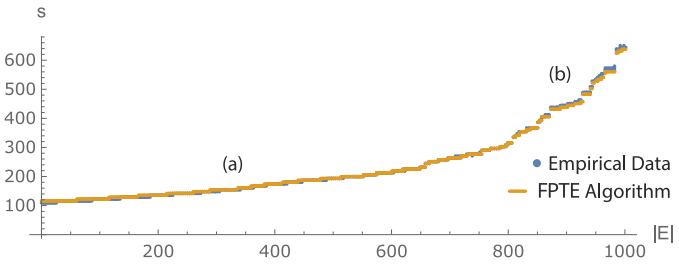

Fig. 3. Longest-path tree evaluation of 1000 dependency graphs with an increasing number of edges. Starting with a low slope at (a), with an increasing number of edges the slope increases as well at (b). The implemented model using the FPTE algorithm matches to 99.4% with the collected data.

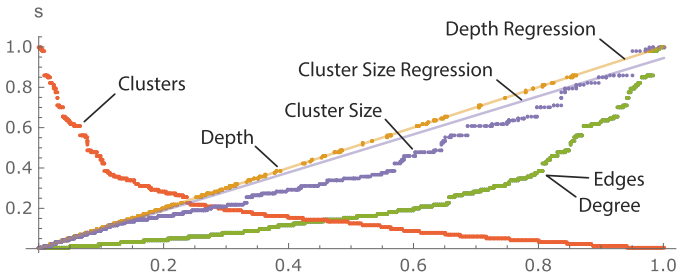

Fig. 4. Normalised correlations of the processing duration with the number of edges, mean dependency degree, dependency depth, number of clusters and mean cluster size. The mean dependency depth is linear correlated with the dependency processing duration. The FPTE regression determines the slope of this correlation and thereby allows to estimate the processing performance efficiently.

algorithm takes time proportional to:

$$\sum_{v=1}^{V} E_{\max}(v) + v \tag{4}$$

Hence, it is faster than the Bellman-Ford algorithm.

4.5 Results

Fig. 3 shows the results from the evaluation of our all-sources longest-path trees algorithm with empirical data. The processing duration starts with a low slope at Fig. 3 *(a)* and then increases in a non-linear fashion towards Fig. 3 *(b)* with the number of edges as paths get longer. The durations we calculate with the FPTE algorithm match to 99.4% with the empirical data collected on our computing cluster. To further understand the most influential graph measures of dependency processing, we analyse the processing duration correlations in detail in the following section.

5 Dependency Analysis

In this section, we analyse the correlations of the processing duration with the edge count, dependency depth, dependency degree, cluster count and cluster size in detail. Each generation of the analysed dependency graph contains one more edge than the previous one. The number of vertices is kept constant to prevent a skewing of the dependency analysis. If more vertices would be added throughout the evaluation, the correlation metrics would change based on the number of vertices, where we are interested in changes of edges.

5.1 Correlations with Processing Duration

Fig. 4 shows the normalised correlations of all dependency measures with the processing duration. For the analysis, we calculate the Pearson product-moment correlation coefficient R and the coefficient of determination R^2 to evaluate if a linear correlation between the measure and the processing duration exists. Additionally, we calculate a linear and nonlinear model for each measure and determine the Root-mean-square error $RMSE$ and Fit. Fit is determined using the normalised version of the $RMSE$ with $1 - NRMSE$ to denote the model fit. The best-fit functions along with the determined correlation metrics are shown in Table 4.

TABLE 4
Correlations of dependency measures with the processing duration

Measure	R	R^2	Function	$RMSE$	Fit
Edges	0.89	0.8	$102 + 0.000405x^2$	32.35	0.93
Degree	0.89	0.8	$102 + 405x^2$	32.35	0.93
Depth	1	1	$113 + 0.1x$	0	1
Clusters	-0.84	0.72	$25.12 + 83034/x$	20.25	0.96
Cluster Size	0.98	0.97	$25.12 + 83.034x$	20.25	0.96

5.1.1 Edge Count

From our model fit in Table 4, we find the number of edges to have a quadratic effect on the processing duration shown in Fig. 4. This stems from the fact that the probability to connect multiple vertices by adding a new edge is lower for initial generations of the graph, where many vertices are connected by a single edge only. With an increasing number of edges, the probability to connect multiple other edges with a new edge increases in a quadratic fashion. Thereby longer paths are created, thus increasing the processing duration.

5.1.2 Dependency Degree

In the same way as the number of edges, the dependency degree is a measure directly related to the probability of a vertex connecting to other vertices. The mean vertex degree increases along with the number of edges as the probability that a new edge connects a vertex to a longer path also increases in a quadratic fashion (Table 4). Hence, the normalised correlation of the dependency degree shown in Fig. 4 matches the normalised correlation of the number of edges.

5.1.3 Dependency Depth

Concluding from the correlation of the edge count and the vertex degree, the mean dependency depth has a linear influence on the processing duration as it directly reflects the average length of the longest-paths. Fig. 4 shows this linear correlation along with Table 4, where the slope of the function matches the mean processing delay for each vertex.

5.1.4 Cluster Count

The cluster count expresses how many unconnected clusters of vertices exist. Fig. 4 illustrates how more clusters lead to

shorter processing paths and thereby influence the processing duration inversely as shown in Table 4.

5.1.5 Cluster Size

The cluster size signifies the mean length of connected components and thereby dictates the maximum length for the longest-paths. As shown in Table 4, the processing duration increases linearly with the cluster size. In Fig. 4 however, the duration stays below a perfect linearity for the majority of the time and then jumps above the line close to the end. For a graph without edges, all vertices are disconnected from each other. Thus, the number of clusters equals the number of vertices with a cluster size of exactly 1. By adding some edges, small groups of vertices become connected forming clusters of small sizes, e.g. 2-3. With the further addition of more edges, small clusters become connected to other small clusters forming clusters of larger sizes, e.g. 30-40. This continues until eventually all clusters are connected to one single cluster. As a result, the cluster size jumps whenever multiple clusters join to a single cluster until finally the size equals the number of vertices.

5.2 Regressions for Processing Duration

Based on the results from the correlation analysis, we build two regression models in order to approximate the processing duration for a given dependency graph. Both models base their duration calculation on the mean processing delay d_p. Variable processing times for dynamic resources are implicitly hidden in the variance of the mean processing delay d_p. For example an *addToBasket* action in an e-commerce website will process longer if 50 items are put into the basket instead of just 1. However, the variance introduced by dynamic actions is not considered in our modelling as we found using the mean processing delay only provides sufficient accuracy. Two measures exhibit a linear correlation with the processing duration: the cluster size and the dependency depth.

5.2.1 Cluster Size Based

We find for sparse graphs with a sparsity $S \lessapprox 1$ the cluster size CS is steady and can be used to estimate the processing duration. Using the mean processing delay d_p and the network delay d_n from the previous work [2] we model:

$$d_{reg,CS} = CS \cdot d_p + d_n \quad \text{if } S \lessapprox 1 \tag{5}$$

Fig. 4 shows the cluster size regression with a model fit of 0.96. The cluster size can be calculated very efficiently for the whole dependency graph through its weakly connected components, however with an increasing sparsity S the approximation results deteriorate.

5.2.2 Depth Based

A more exact approximation of the processing duration can be performed using the processing depth if there are more edges than vertices in the graph. However, it is more expensive to calculate the processing depths using a depth-first search algorithm, as the depth has to be computed for every starting vertex. Equation (1) and (2) denote the maximum number of edges and vertices for a DAG, where the runtime for a depth-first search generally is limited to $V+E$. Using the depth $ddep$, the regression can be modelled as follows:

$$d_{reg,ddep} = d_p + d_p \cdot ddep + d_n \tag{6}$$

Fig. 4 shows the depth regression with a model fit of 1 as the processing delays for the evaluation are normally distributed around 0.1 (Table 4). The error of the regression is distributed exactly as the mean processing delay serving as regression slope.

TABLE 5
Graph and traffic parameters with distributions used to generate evaluation data for the performance comparison

Graph Based	**Param**	**Distribution**
Vertices	V	Uniform(100, 1000)
Edges	E	Based on clusters C and CS
Read/Processing	RPR	Uniform(0, 1)
Read Request	RR	Bernoulli(RPR)
Cache Hit/Miss	HMR	Uniform(0, 0.7)
Processing Delay	d_p	HyperErlang(Uniform(1,10)) Weibull(Uniform(0.1, 10), 1) Pareto(0.001, Uniform(1, 10)) Lomax(0.001, Uniform(1, 10, 0))
Clusters	CC	Uniform(10, 100)
Cluster Size	CS	Uniform(3, 10)
Traffic Based	**Param**	**Distribution**
Duration	D	Const(20)
Requests	R	Uniform(1000, 4000)
Path	P	Zipf(V, Uniform(10^{-6}, 0.1))
Offset	O	FARIMA(R, D) CMMPP(R, D) FractionalBrownianMotion(R, D) PoissonParetoBurstProcess(R, D)

6 Service Generation

To compare the performance of resource dependency processing with a traditional cache-eviction approach, we generate web services consisting of dependency graphs and traffic traces. Existing web services do not declare resource dependencies explicitly, hence no data is available and the graphs must be generated. For the generation, we develop two algorithms. The first algorithm bases its parameters on extracted values of six social network application APIs and the second algorithm selects its parameters at random.

6.1 Parameters

For the generation, we identify the parameters listed in Table 5.

6.1.1 Dependency Graph Based

The number of vertices V for each graph is distributed uniformly between 100 and 1000. The read/processing ratio RPR identifies the fraction of all resources that are read only. A $RPR = 0.3$ means that 30% of all vertices are read only. For each vertex in the graph, we determine whether it is a read or processing vertex using a Bernoulli distribution distributed by the read/processing ratio. The cache hit/miss ratio HMR determines how many vertices of the whole

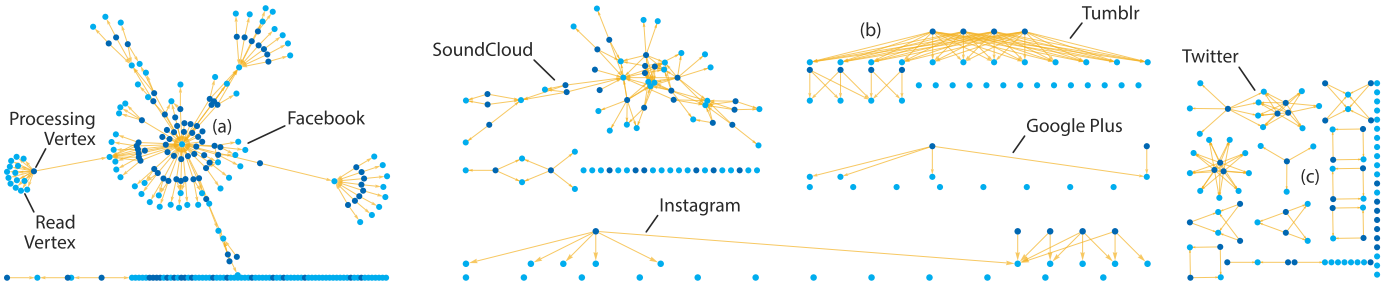

Fig. 5. API structure of the six inspected services with read and processing vertices. At (a) the Facebook structure graph is strongly connected with the central vertex representing a user's feed. (b) shows multiple sub-resources of a Tumblr post that are updated with the post and (c) highlights weakly connected clusters with sparse dependencies of the Twitter API.

graph are marked as cached. This parameter is only used by the traditional cache-eviction approach. Based on work in [24], [25] the processing delay d_p for each vertex is calculated by uniformly choosing one of the distributions listed in Table 5 for each graph. The HyperErlang(n) distribution uses a probability vector of length n and the other distributions follow the standard signatures Weibull(α, β), Pareto(k, α) and Lomax(k, α, μ) where α is shape, β is scale, μ is location and k is a minimum value parameter. The number of clusters and the cluster size is uniformly selected for each graph.

6.1.2 Traffic Based

For each graph, we generate traffic for the duration D with a uniformly selected number of requests R. Based on the work in [33], [34] the resource popularity P can be modelled using a Zipf(n, ρ) distribution where n is the range and ρ is the Zipf parameter. The offset O determines the arrival time of each request. We model the self-similar offset following work in [28], [29], [30], [31], [32] by using a Fractionally Autoregressive Integrated Moving-Average process FARIMA(R, D), a Circulant Markov-Modulated Poisson process CMMPP(R, D), FractionalBrownianMotion(R, D) and a PoissonParetoBurstProcess(R, D) where R is the number of arrivals and D is the arrival interval. For FARIMA we use 0.99 as AR coefficients, a random uniformly distributed MA coefficient between 0 and 0.49 and a white noise variance of 1. For the CMMPP we use the superposition of four two-state arrival rate vectors with a maximal arrival rate of 500. This rate is based on the maximum throughput of a single node in our evaluation cluster. The Fractional Brownian Motion uses a uniformly distributed hurst index between 0.5 and 0.99 in order to ensure self-similarity and the λ parameter of the Exponential distribution for the Poisson Pareto Burst process limits the maximum arrival rate to 500. We randomly select one of the models to generate the offset for each graph.

6.2 Service Based Graph Generation

In order to generate random dependency graphs exhibiting real-world properties, we develop an algorithm extracting parameters from six social network services: The Facebook Graph API v2.2, the Twitter API v1.1, the Tumblr API v1, the Instagram API v1, the Google Plus API v1 and the SoundCloud API v1.

TABLE 6
Key figures of the extracted service parameters

Measure	Param	Min	Max	Mean	Var
Read/Processing	RPR	0.58	0.85	0.71	0.014
Processing Delay	d_p	0.004	0.18	0.09	2.52
Cluster Size	CS	14	235	81	6642
Dependency Depth	$ddep$	0	4	0.38	0.71
Dependency Degree	$ddeg$	0	14	0.57	2.79

6.2.1 Service Structure Graphs

For each service, we extract all API resources as vertices and the dependencies of the resources as edges into an API structure graph. Dependencies are not declared in the API specifications, hence we analyse the effects of a request to a resource by comparing changes in all resources before and after a request. The changed vertices have a dependency on the initially requested vertex and need to be added as dependency edges. Fig. 5 illustrates all extracted service structure graphs. At Fig. 5 *(a)* the Facebook structure graph is strongly connected with the central vertex representing a user's feed. Fig. 5 *(b)* shows multiple sub-resources of a tumbler post that are updated with the post and Fig. 5 *(c)* highlights weakly connected clusters with sparse dependencies of the Twitter API. For detailed inspection of all graphs we provide the full evaluation dataset as download available at [38].

6.2.2 Parameter Extraction

From the service structure graphs we extract parameter ranges to be used for the generation of random dependency graphs. The results are presented in Table 6. Using a goodness-of-fit hypothesis test, our measured parameters do not follow a distribution. Hence, our algorithm selects random elements uniformly from all captured parameter data.

6.2.3 Algorithm

We develop the Service based dependency graph (SDG) algorithm by superposing and manipulating multiple adjacency matrices. The steps are as follows:

1) Create a sequencer function for the dependency depth that returns continuous sequences of 1s followed by a terminating 0, where the length of the

sequence is drawn randomly from all service structure graph depths, e.g. $111011011110\ldots$ for $3, 2, 4$.
2) Create a sequencer function for the dependency degree in the same fashion as the depth sequencer function.
3) Create a $N \times N$ matrix, where N is drawn randomly from all service structure cluster sizes and fill it with 0s.
4) Fill the matrix diagonal above or below the main diagonal with a sequence from the depth sequencer function.
5) Fill the matrix columns until the diagonal above or below the main diagonal with sequences from the degree sequencer function.
6) Create another matrix of size N and fill it with random samples of a Bernoulli distribution where the probability equals a random read/processing ratio.
7) Multiply the read/processing matrix with the depth/degree matrix.
8) Repeat 3-7 and concatenate the resulting cluster matrices until the desired number of vertices is reached.

By strictly manipulating either the upper or lower triangular portion of an adjacency matrix, the directed acyclic graph property of the resulting matrix is ensured. In addition, for each vertex a processing delay is drawn randomly from the service data. Furthermore, each vertex is marked as cache-hit or miss based on the distribution in Table 5. The only input parameter to the algorithm is the number of maximum vertices to be used as termination criteria.

6.3 Fuzzy Graph Generation

As the service based graph generator strictly uses parameter values drawn from the analysis of the service structure graphs, we develop a supplementary Fuzzy dependency graph (FDG) algorithm creating graphs with a wider range of parameters. This ensures the evaluation is not overfitted to the analysed service structure graphs. The steps are as follows:

1) Create C adjacency matrices with size V/C and randomly distribute a total of E edges in the upper or lower triangular portion.
2) Multiply the columns with a sequence of 1 and 0 distributed by the read processing ratio.
3) Concatenate all resulting cluster matrices.

The distributions used for the parameters are listed in Table 5. As the service based algorithm, the fuzzy generation algorithm additionally determines a processing delay and cache-hit for each vertex. The input parameters to the fuzzy generation algorithm are the number of vertices V, the number of edges E and the number of clusters C.

7 Performance Modelling

For the performance evaluation, we develop a model which calculates the processing duration for both a resource dependency processing and a traditional approach. The parameters serving as input to the model can be calculated from the structure of an application as shown in Section 3. Furthermore, the model allows to compare the performance of both approaches in order to find the approach best suited for a specific application.

7.1 Processing Duration

We extend the processing duration of our previous work by replacing the d_P constant with an equation explicitly calculating the processing duration based on the dependency graph.

7.1.1 Traditional Processing

We model d_P for the traditional processing (TP) approach where a web service directly receives every request but can only serve a fraction of the resources defined by the cache hit/miss ratio HMR from a cache as:

$$d_{P,TP} = HMR \cdot d_l + d_n + (1 - HMR) \cdot d_p \tag{7}$$

The lookup delay d_l describes the time it takes to lookup an item in the cache and the network delay d_n is modelled as linear or quadratic variable [2].

7.1.2 Resource Dependency Processing

The resource dependency processing (RDP) approach is based on the linear correlation between the cluster size CS and the processing depth $ddep$ as analysed in Section 5. If the sparsity is $S \lessapprox 1$ we model the processing duration as follows:

$$d_{P,RDP} = (d_n + CS \cdot d_p) \cdot (1 - RPR) \quad \text{if } S \lessapprox 1 \tag{8}$$

If the sparsity $S \gg 1$ we use the more expensive to determine dependency depth $ddep$:

$$d_{P,RDP} = (d_n + d_p + ddep \cdot d_p) \cdot (1 - RPR) \tag{9}$$

In contrast to the traditional processing approach, the resource dependency processing approach receives only a fraction of all requests defined by the read/processing ratio. All other resources are served directly from the resource database and do not influence the processing duration.

7.2 Processing Duration Delta

In order to compare both processing approaches, the processing duration delta can be modelled as:

$$\Delta d_P = d_{P,RDP} - d_{P,TP} \tag{10}$$

Fig. 6 illustrates the influence of all model parameters on the duration deltas where each parameter is plotted in the range of 0 to 1 while all other parameters remain constant with values given in the figure caption. For negative duration deltas the resource dependency processing approach is faster than a traditional processing approach. A greater absolute slope in Fig. 6 means that the analysed parameter has a greater influence on the processing duration. Consequently, the read/processing ratio has the greatest influence on the processing duration as it directly affects the number of requests that need to be processed. It is followed by the hit/miss ratio which determines the amount of cached resources in the traditional processing approach. The processing delay,

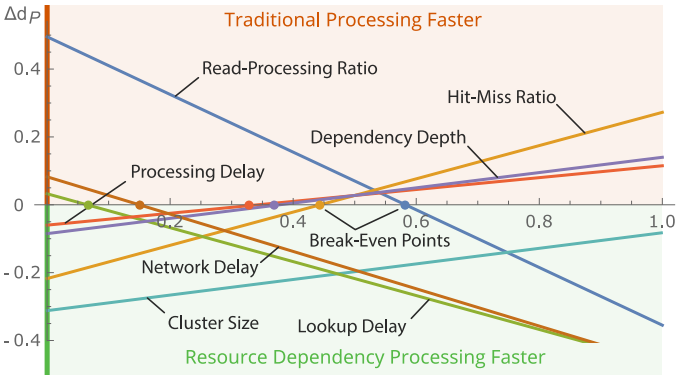

Fig. 6. Analysis of the influence and break-points of all model parameters to the duration deltas where each parameter is plotted in the x-axis range of 0 to 1, while all other parameters remain constant with default parameters $RPR = 0.45$, $HMR = 0.5$, $d_l = 0.01$, $d_p = 0.5$, $ddep = 0.5$, $CS = 0.5$ and $d_n = 0.1$. For negative duration deltas the resource dependency processing approach is faster than a traditional processing approach.

the dependency depth and the cluster size have equal influences on both approaches. In the traditional approach, a greater lookup delay negatively influences the caching performance and the network delay is applied to every request. The resource dependency processing approach uses the processing tree, thus the network delay only applies to the initial request and its response.

7.3 Relative Performance Improvement

To calculate the factor of improvement the resource dependency processing exhibits over the traditional processing, we define:

$$RPI = \frac{d_{P,TP}}{d_{P,RDP}} - 1 \tag{11}$$

For example, a positive RPI of 2.3 shows that the performance using resource dependency processing is 2.3 times better than the traditional processing performance. Similarly, a negative RPI means that traditional processing approach is faster.

7.4 Break-Even Points for Processing Duration

A break-even point calculation allows to determine the exact value of a parameter where the processing duration of both the traditional processing and the resource dependency processing are equal. The break-even point based on the dependency depth $ddep$ can be calculated as follows:

$$ddep = \frac{-d_l \cdot HMR + d_p \cdot HMR - d_n \cdot RPR - d_p \cdot RPR}{d_p \cdot (-1 + RPR)} \tag{12}$$

Based on the cluster size CS, we calculate the break-even point as:

$$CS = \frac{-d_p - d_l \cdot HMR + d_p \cdot HMR - d_n \cdot RPR}{d_p \cdot (-1 + RPR)} \tag{13}$$

Fig. 6 illustrates the break-even points for all model parameters where Δd_P is zero.

TABLE 7
Key figures of the generated evaluation data

Service Based	**Param**	**Min**	**Max**	**Mean**	**SD**
Vertices	V	100.	1000.	537.6	265.2
Edges	E	18.	2127.	408.82	332.45
Sparsity	S	0.15	2.89	0.77	0.48
Clusters	C	41.	807.	338.62	188.7
Cluster Size	CS	1.16	3.06	1.68	0.38
Dependency Depth	$ddep$	0.15	1.69	0.58	0.29
Dependency Degree	$ddeg$	0.14	0.93	0.43	0.17
Processing Delay	d_p	0.	0.13	0.04	0.03
Cache Hit/Miss	HMR	0.	0.7	0.35	0.2
Traffic	**Param**	**Min**	**Max**	**Mean**	**SD**
Requests	R	1002.	4000.	2496.89	867.48
Read/Processing	RPR	0.	1.	0.49	0.29
SD Path Popularity	P	22.56	229.64	121.19	57.66
SD Offset	O	4.2	7.34	5.83	0.34
Fuzzy	**Param**	**Min**	**Max**	**Mean**	**SD**
Vertices	V	30.	1000.	347.11	215.46
Edges	E	0.	4095.	320.45	488.07
Sparsity	S	0.	4.18	0.82	0.82
Clusters	C	10.	872.	188.84	153.95
Cluster Size	CS	1.	10.	2.49	1.78
Dependency Depth	$ddep$	0.	4.05	0.72	0.72
Dependency Degree	$ddeg$	0.	4.04	0.71	0.72
Processing Delay	d_p	0.	0.22	0.07	0.05
Cache Hit/Miss	HMR	0.	0.7	0.36	0.2
Traffic	**Param**	**Min**	**Max**	**Mean**	**SD**
Requests	R	1000.	3994.	2488.43	870.15
Read/Processing	RPR	0.	1.	0.5	0.3
SD Path Popularity	P	7.68	227.	79.86	47.61
SD Offset	O	4.35	7.32	5.82	0.34

8 Performance Evaluation

Finally, we use the service based and fuzzy graph generation algorithms to compare the performance of the resource dependency processing approach with a traditional processing approach in an aggregated combined case, best case, worst case and average case scenario and evaluate the fit of the performance models. We further create four series of graphs with increasing graph measures to evaluate the influence of different structures on the performance and map the results to structures observed in real-world APIs.

8.1 Aggregated Performance Results

For the aggregated evaluation, we generate 1000 web services using the service based graph algorithm and 1000 web services using the fuzzy graph algorithm. For each of the 2000 web services, we generate a distinct traffic trace following the distributions from Table 5. The full evaluation dataset is available at [38], where Table 7 lists the key figures. The evaluation is executed with machine pairs on our Raspberry Pi cluster, where one machine generates traffic and the other implements the web service. In addition to the traditional and resource dependency processing implementation, we evaluate a third asynchronous resource dependency processing (ARDP) implementation that processes all

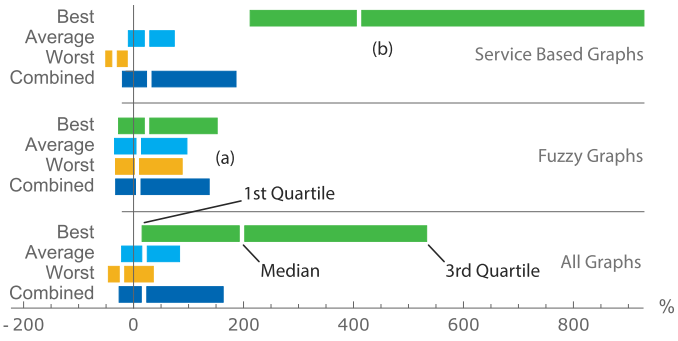

Fig. 7. Relative performance improvements for service based graphs, fuzzy graphs and all graphs in the combined case, best case, worst case, and average case when using resource dependency processing over traditional processing. Due to the greater range of fuzzy graph parameters, the quartiles at (a) do not vary as much with the cases as the quartiles in (b).

dependencies but the first level dependencies in the background. This is done to respect modern web applications with increased client-side logic. For this type of applications, state changes can instantly be presented to the user so a web service guaranteeing eventual consistency is sufficient. We further evaluate each graph in both requester mode (RM) using the generated traffic and sequencer mode (SM) requesting each resource exactly once. We do this in order to measure the influence of the traffic on the performance. Using all three implementations in both modes, we measure over 850 thousand resources with over 6 million requests in 2000 web services. We further calculate the processing durations and relative performance improvements with the models developed in this work. The results are presented in four aggregated cases, where each case individually limits the range of the read/processing ratio and hit/miss ratio.

8.1.1 Combined Case

In the combined case, we present the results of all aggregated graphs. Fig. 7 shows the quantiles of the relative performance improvement where the size of the boxes around the median divide the results in equal parts regarding the first quartile and the third quartile. As the minimum and maximum relative performance improvement factors have a high variance, we intentionally left out the whiskers in Fig. 7 to be able to visualise quantiles in a meaningful way. In total, 59% of all combined case services are faster using resource dependency processing rather than traditional processing.

8.1.2 Best Case

From Fig. 6 follows, that for all read/processing and hit/miss ratios between 0.0 and 0.3 the resource dependency processing is faster than the traditional processing. Consequently, for the best case aggregation both ratios are restricted to be within the 0.0-0.3 range. Fig. 7 illustrates median performance improvements of a factor higher than four for the service based graphs, 25% for the fuzzy based graphs and a factor of almost two for the combination of both. In total, 79% of all best case services are faster using resource dependency processing rather than traditional processing.

8.1.3 Worst Case

The worst case aggregates results in the ranges where in Fig. 6 the traditional processing is faster. Thus, the read/processing ratio is limited to the range between 0.6 and 1.0. The hit/miss ratio is limited to the range between 0.5 and 0.7 as this is the maximum evaluated hit/miss ratio (Table 7). The median performance improvements illustrated in Fig. 7 are -34% for the service based graphs, 5.4% for the fuzzy based graphs and -21% for the combination of both. In total, 37% of all worst case services are faster using resource dependency processing rather than traditional processing.

8.1.4 Average Case

For the average case aggregation we select results in the ranges where in Fig. 6 both approaches exhibit similar performance. Consequently, we use a read/processing ratio in the range between 0.3 and 0.6 and a hit/miss ratio in the range between 0.3 and 0.5. Further, we confirm the selected ranges to be within typical ranges as presented by [22], [23]. As shown in Fig. 7 the median performance improvements are 25% for the service based graphs, 9.4% for the fuzzy based graphs and 20% for the combination of both. In total, 62% of all average case services are faster using resource dependency processing rather than traditional processing.

8.2 Model Fits

We calculate the processing duration delta for all 8000 evaluations and compare the values to the empirical performance results. The residuals of all evaluations have a root-mean square error of 30.4. Using the normalised root-mean square error to put the errors in relation to the observed values (Section 5) we calculate the Fit as $1 - NRMSE$. The mean fit for the cluster size based model is $Fit_{CS} = 0.96$ and the mean fit for the dependency depth based model is $Fit_{ddep} = 0.98$. This implies that both duration delta models have very good fits. We can further observe that the cluster size based model is cheaper to compute while the dependency depth model is more accurate.

8.3 Structure Based Performance Results

The structure based evaluation is performed to analyse the effects of different graph structures on the performance. Therefore, four series of graphs with increasing graph measures are created and performance tested with our resource dependency processing and a traditional processing approach. As presented in Fig. 6, the major performance influencing parameters are the read/processing ratio and the cache hit/miss ratio. In order to analyse the effects of the graph structures only, for all series both the read/processing ratio and the hit/miss ratio are set to their calculated performance break-even points $RPR_{BEP} = 0.58$ and $HMR_{BEP} = 0.44$. A series of five graphs is created for an increasing dependency depth in Fig. 8 (a-e), dependency degree in Fig. 8 (f-j), cluster size in Fig. 8 (k-o) and number of clusters in Fig. 8 (p-t). The processing delay for each resource is set to 0.2 seconds and for presentation issues, all graphs have a total of 50 vertices. Each series starts with a low value of the measure that increases in five steps to a

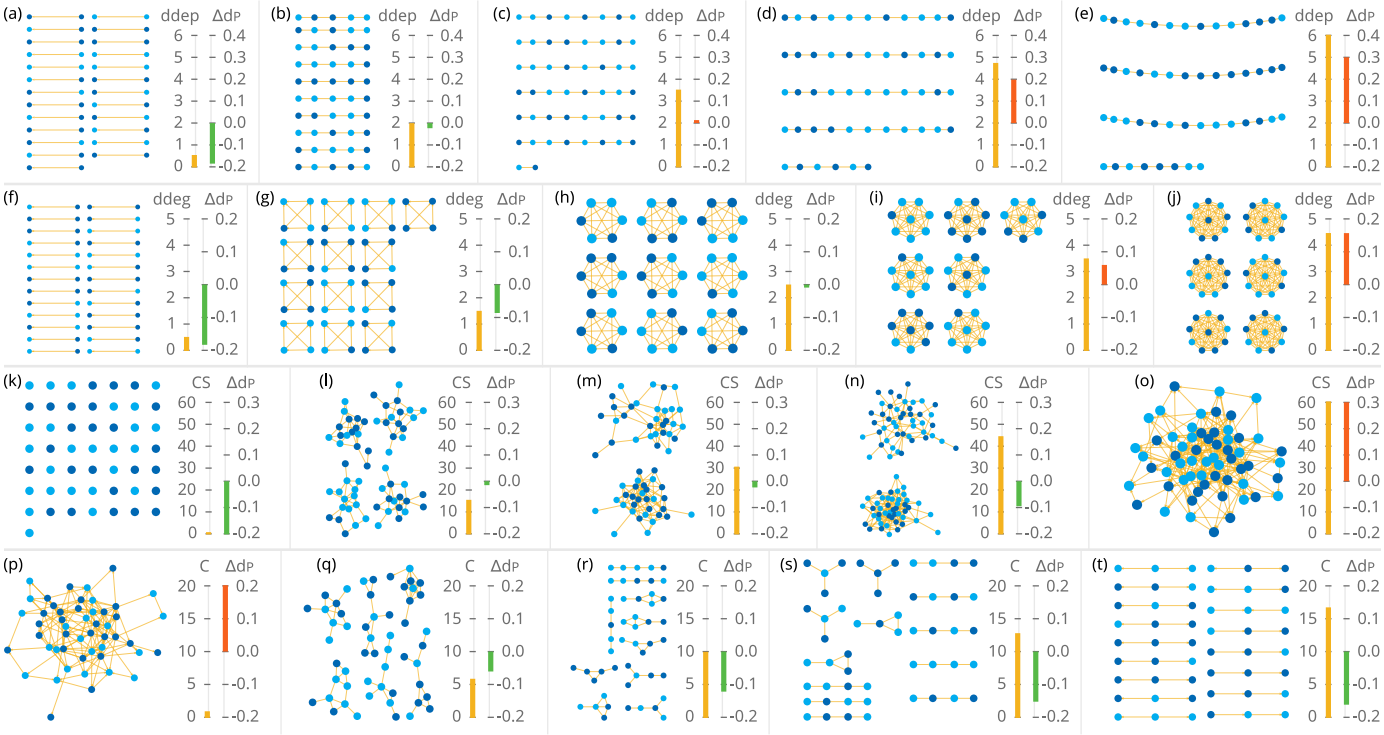

Fig. 8. Structure based results of four series of increasing graph measures: Dependency Depth (a-e), Dependency Degree (f-j), Cluster Size (k-o) and Clusters (p-t). A negative processing duration delta means our proposed RDP approach is faster than a traditional approach.

maximum measure as detailed in Table 8. The graphs are created using a special version of our fuzzy dependency graph generator, where we set the desired graph measure (depth, degree, cluster size and count) instead of letting the algorithm choose a random value.

8.3.1 Performance Results

The performance is calculated using the processing duration delta model from this work, where negative values indicate a better performance when using our proposed resource dependency processing approach. The results are presented in Fig. 8 and Table 8. From all 20 graphs, the performance using our proposed resource dependency approach is better for 13 structures (a-b,f-h,k-n,q-t). The increasing depth in the depth series (a-e) has a major influence on the performance as the length of the chains as seen in (d-e) massively increases the processing duration. An increasing degree (f-j) has a minor influence on the performance as most dependencies can be processed in parallel. The growing mean cluster size series in (k-o) has an effect on both the depth and the degree, where a lower cluster size results in lower maximal depths. Finally, the increasing total number of clusters series (p-t) is tightly inversely related to the mean cluster size. Thus, with many clusters the performance is better as the maximal depth is reduced.

8.3.2 Mapping to Real-World Structures

When searching for resemblance between graphs in Fig. 5 and Fig. 8, it is noteworthy that Fig. 5 presents structure graphs, where Fig. 8 presents full resource graphs (for distinction see Table 2). To extract resource graphs from structure graphs as presented in Fig. 5, variables for the number of users, posts, comments, photos etc. majorly influence the measures of the resulting resource graph. Thus, we suggest similarities must be compared with care.

TABLE 8
Structure based performance results for Fig. 8

Measure	**Param**	**Dependency Depth Series** (a-e)				
Depth	$ddep$	0.5	2.	3.4	4.7	6.
Degree	$ddeg$	0.5	0.8	0.86	0.9	0.92
Cluster Size	CS	2.	5.	7.1	10.	13.
Clusters	C	25	10	7	5	4
Proc. Delta	Δd_P	-0.16	-0.034	0.081	0.19	0.3
Measure	**Param**	**Dependency Degree Series** (f-j)				
Depth	$ddep$	0.5	1.5	2.5	3.5	4.5
Degree	$ddeg$	0.5	1.5	2.5	3.5	4.5
Cluster Size	CS	2.	4.	6.	8.	10.
Clusters	C	26	13	9	7	6
Proc. Delta	Δd_P	-0.17	-0.083	-0.0064	0.071	0.14
Measure	**Param**	**Cluster Size Series** (k-o)				
Depth	$ddep$	0.	2.6	2.9	3.	6.4
Degree	$ddeg$	0.	1.9	2.6	2.7	4.2
Cluster Size	CS	1.	16.	31.	46.	61.
Clusters	C	50	4	2	2	1
Proc. Delta	Δd_P	-0.2	-0.035	-0.0044	-0.099	0.3
Measure	**Param**	**Number of Clusters Series** (p-t)				
Depth	$ddep$	4.8	1.6	1.	0.81	0.55
Degree	$ddeg$	3.2	1.4	0.96	0.79	0.67
Cluster Size	CS	50.	9.	5.	4.	3.
Clusters	C	1	6	10	13	17
Proc. Delta	Δd_P	0.2	-0.078	-0.12	-0.14	-0.16

Facebook's rather centralised structure can be extracted to graphs similar to Fig. 8 (l-m) as the centralised structure is based around clusters of users. A single user in Instagram, Google Plus or Tumblr has a smaller cluster of resources leading to structures more similar to Fig. 8 (g-h,s). This leads to an increased processing duration with Facebook's structure compared to more decentralised structures such as Twitter's, Instagram's or Tumblr's. However, we suggest that this stems from the massively higher range of functions Facebook offers compared to the other platforms.

9 Conclusion and Future Work

In this work, we proposed an efficient resource database update mechanism that allows to build scalable web service architectures with optimised request routing, such as the novel request flow we presented in [1], [2]. For the first major research objective, we showed that resource dependencies can be stored as directed acyclic graphs, where vertices represent resources and edges dependencies. We further identified dependency depth, dependency degree, cluster count, cluster size and sparsity as the most influential graph measures and related them to existing graph measures. To generate random dependency graphs, we based the generation on service structures where the parameters were extracted from six real-world social applications and a fuzzy algorithm with random parameters. For the second major research objective, we found the optimisation problem to be in the domain of job and workflow scheduling, where the longest-path is the critical path for performance. As typical algorithms only compute a single-source longest-path tree, we extended an existing topological sort algorithm with a dynamic programming approach which is able to determine the processing order in linear time. Further, the evaluation of 2000 web services with 850 thousand resources and over 6 million requests showed the resource dependency processing approach to be up to a factor of two faster than a traditional processing approach. For the third and final objective, we found that the dependency depth and cluster size have a linear correlation with the processing duration. The evaluation of four series of different graph structures further highlighted the correlations with major graph measures. We found the processing duration to be adequately modelled based on both measures, where the accuracy depends on the sparsity of the graph. Both models allowed us to replace the constant post-processing delay from our previous work. The cluster size based model had an overall model fit of 96% and was cheap to compute, where the dependency depth based model had a model fit of 98%, thus being more accurate but also more expensive to determine.

The focus of our future work will be on further optimisation of the dependency processing algorithm. We will conduct research to find algorithms that update the dependency graph and forest of processing trees in an incremental fashion. Additionally, we will develop algorithms to automatically extract dependency graphs from web services. Furthermore, we will search for hot processing spots where a resource vertex is updated frequently and apply strategies to reduce and minimise the number of updates. We will also develop assistive systems that point out critical dependency depths, dependency degrees and clusters and provide an optimal solution that helps decoupling affected resources.

References

[1] T. Fankhauser, Q. Wang, A. Gerlicher, C. Grecos, and X. Wang, "Web scaling frameworks: A novel class of frameworks for scalable web services in cloud environments," in *Proc. IEEE Int. Conf. on Commun. (ICC14)*, June 2014, pp. 1414–1418.

[2] T. Fankhauser, Q. Wang, A. Gerlicher, C. Grecos, and Wang, X., "Web scaling frameworks for web services in the cloud," *Trans. on Serv. Comp., IEEE*, pp. 1–1, 2015.

[3] A. Negro, C. Roque, P. Ferreira, and L. Veiga, "An adaptive semantics-aware replacement algorithm for web caching," *Journal of Int. Serv. and App.*, 2015.

[4] P. Bangar and K. Singh, "Investigation and performance improvement of web cache recommender system," in *Proc. IEEE Int. Conf. on Fut. Trends on Comp. Anal. (ABLAZE15)*, Feb 2015, pp. 585–589.

[5] A. Sarhan, A. Elmogy, and S. Ali, "New web cache replacement approaches based on internal requests factor," in *Proc. IEEE Int. Conf. on Comp. Eng. Sys. (ICCES14)*, Dec 2014, pp. 383–389.

[6] O. Batarfi, R. Shawi, A. Fayoumi, R. Nouri, S.-M.-R. Beheshti, A. Barnawi, and S. Sakr, "Large scale graph processing systems: survey and an experimental evaluation," *Cluster Computing*, vol. 18, no. 3, pp. 1189–1213, 2015.

[7] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1804–1815, Aug. 2015.

[8] Y. Guo, M. Biczak, A. Varbanescu, A. Iosup, C. Martella, and T. Willke, "How well do graph-processing platforms perform? an empirical performance evaluation and analysis," in *Para. and Dist. Proc. Symp., IEEE*, May 2014, pp. 395–404.

[9] D. G. Malcolm, J. H. Roseboom, C. E. Clark, and W. Fazar, "Application of a technique for research and development program evaluation," *INFORMS Operations Research*, vol. 7, no. 5, pp. 646–669, 1959.

[10] J. E. Kelley, Jr and M. R. Walker, "Critical-path planning and scheduling," in *ACM IRE-AIEE Comp. Conf.* ACM, 1959, pp. 160–173.

[11] S. Abrishami, M. Naghibzadeh, and D. Epema, "Cost-driven scheduling of grid workflows using partial critical paths," *Trans. on Para. and Dist. Sys., IEEE*, vol. 23, no. 8, pp. 1400–1414, Aug 2012.

[12] S. Chanas and P. Zieliski, "Critical path analysis in the network with fuzzy activity times," *Fuzzy Sets and Systems*, vol. 122, no. 2, pp. 195 – 204, 2001.

[13] M. Masdari, S. ValiKardan, Z. Shahi, and S. I. Azar, "Towards workflow scheduling in cloud computing: A comprehensive analysis," *Journal of Net. and Comp. App.*, 2016.

[14] K. Maheshwari, E.-S. Jung, J. Meng, V. Morozov, V. Vishwanath, and R. Kettimuthu, "Workflow performance improvement using model-based scheduling over multiple clusters and clouds," *Fut. Gen. Comp. Sys.*, vol. 54, pp. 206–218, 2016.

[15] C. Pang, J. Wang, Y. Cheng, H. Zhang, and T. Li, "Topological sorts on {DAGs}," *Information Processing Letters*, vol. 115, no. 2, pp. 298 – 301, 2015.

[16] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan, "Incremental cycle detection, topological ordering, and strong component maintenance," *ACM Trans. Algorithms*, vol. 8, no. 1, pp. 3:1–3:33, Jan. 2012.

[17] D. Ajwani and T. Friedrich, "Average-case analysis of incremental topological ordering," *Discrete Applied Mathematics*, vol. 158, no. 4, pp. 240 – 250, 2010.

[18] R. Bellman, "The theory of dynamic programming," *RAND Corp, Santa Monica*, 1954.

[19] G. Salvaneschi and M. Mezini, "Towards reactive programming for object-oriented applications," in *Trans. on Asp.-Or. Soft. Dev. XI*. Springer Berlin Heidelberg, 2014, pp. 227–261.

[20] A. Margara and G. Salvaneschi, "We have a dream: Distributed reactive programming with consistency guarantees," in *ACM Proc. of Int. Conf. on Dist. Event-Based Sys., (DEBS14)*, 2014, pp. 142–153.

[21] G. Salvaneschi, A. Margara, and G. Tamburrelli, "Reactive programming: A walkthrough," in *IEEE Int. Conf. on Soft. Eng. (ICSE15)*, vol. 2, May 2015, pp. 953–954.

[22] C. Du and S. Wang, "Research on mobile web cache prefetching technology based on user interest degree," in *LISS 2013*. Springer Berlin Heidelberg, 2015, pp. 1253–1258.

[23] A. Songwattana, T. Theeramunkong, and P. C. Vinh, "A learning-based approach for web cache management," *Mobile Networks and Applications*, vol. 19, no. 2, pp. 258–271, 2014.

[24] A. Rajabi and J. W. Wong, "Provisioning of computing resources for web applications under time-varying traffic," in *IEEE Int. Sym. on. Mod., Anal. & Sim. of Comp. and Tel. Sys.* IEEE, 2014, pp. 152–157.

[25] N. Poggi, D. Carrera, R. Gavalda, E. Ayguadé, and J. Torres, "A methodology for the evaluation of high response time on e-commerce users and sales," *Information Systems Frontiers*, vol. 16, no. 5, pp. 867–885, 2014.

[26] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Graph structure in the web—revisited: a trick of the heavy tail," in *Proc. of the Int. Con. on World wide web Comp.*, 2014, pp. 427–432.

[27] A. Ramachandran, Y. Kim, and A. Chaintreau, "I knew they clicked when i saw them with their friends: identifying your silent web visitors on social media," in *ACM Proc. of the Conf. on Online social netw.*, 2014, pp. 239–246.

[28] S. Dick, O. Yazdanbaksh, X. Tang, T. Huynh, and J. Miller, "An empirical investigation of web session workloads: Can self-similarity be explained by deterministic chaos?" *Information Processing & Management*, vol. 50, no. 1, pp. 41–53, 2014.

[29] S. Chen, M. Ghorbani, Y. Wang, P. Bogdan, and M. Pedram, "Trace-based analysis and prediction of cloud computing user behavior using the fractal modeling technique," in *IEEE Int. Cong. on Big Data*. IEEE, 2014, pp. 733–739.

[30] M. Zukerman, T. D. Neame, and R. G. Addie, "Internet traffic modeling and future technology implications," in *IEEE Joint Conf. of. Comp. and Comm. Soc. (INFOCOM2003)*, vol. 1. IEEE, 2003, pp. 587–596.

[31] J. Chen, R. G. Addie, M. Zukerman, and T. D. Neame, "Performance evaluation of a queue fed by a poisson lomax burst process," *IEEE Comm. Lett.*, vol. 19, no. 3, pp. 367–370, 2015.

[32] R. Donthi, R. Renikunta, R. Dasari, and M. R. Perati, "Self-similar network traffic modeling using circulant markov modulated poisson process," in *Fractals, Wavelets, and their Applications*. Springer, 2014, pp. 437–444.

[33] K. V. Katsaros, G. Xylomenos, and G. C. Polyzos, "Globetraff: a traffic workload generator for the performance evaluation of future internet architectures," in *IEEE Int. Conf. on New Tech., Mob. and Secu. (NTMS12)*. IEEE, 2012, pp. 1–5.

[34] K. Visala, A. Keating, and R. H. Khan, "Models and tools for the high-level simulation of a name-based interdomain routing architecture," in *IEEE Conf. on Comp. Comm. Works. (INFOCOM WKSHPS14)*. IEEE, 2014, pp. 55–60.

[35] R. Sedgewick, *Algorithms II*. Addison-Wesley Professional, 2014.

[36] S. S. Ray, *Graph Theory with Algorithms and Its Applications*. Springer, India, 2013.

[37] T. H. Cormen, *Introduction to Algorithms*. MIT press, 2009.

[38] (2016) Resource Dependency Processing Dataset. [Online]. Available: http://webscalingframeworks.org/graphs

Thomas Fankhauser (SM13) received his Bachelor and Master degree in computer science from Stuttgart Media University, Germany. He is currently pursuing a PhD degree at the University of the West of Scotland while working as an academic research assistant at Stuttgart Media University. His research interests include the scalability of web services, software architectures and frameworks in cloud computing environments. Since 2013 he is a student member of the IEEE Consumer Electronics Society. He was a Best Poster Award Winner of UWS ICTAC 2013, has published at the IEEE flagship conference ICC 2014 in Sydney and the IEEE Transactions on Services Computing in 2015. He is author of a book on social phenomena in social networking services. Previously, he worked as a software architect and developer in social web and mobile advertising industries for several years.

Qi Wang (S02M06) is a Professor in Networks and Video Communications at UWS, UK, and a Board Member of EU 5GPPP (Public Private Partnership) Technology Board. He is the Technical CoManager of the EU Horizon 2020 5GPPP project SELFNET, and Principal Investigator of UK EPSRC project "Enabler for NextGeneration Mobile Video Applications" and a number of other funded projects. His current research interests include video processing and transmission, 5G wireless/mobile networks, cloud computing and softwaredefined networking. He has published over 80 peerreviewed papers in related areas. He was a Best Paper Award Winner of several international conferences. He received his PhD degree in Mobile Networking from the University of Plymouth, UK.

Ansgar Gerlicher (M06) is a Professor in Mobile Applications and Director of Research in Mobile Applications & Security with the Institute of Applied Science, Stuttgart Media University, Germany. Previously, he worked for several years as a Software Architect and Project Manager in the Telecommunication and Automotive Industries. He is on the programme committee of the Apps To Automotive Conference, Stuttgart, TCP member of multiple international conferences and guest editor of the Journal on Mobile Information Systems. He published several papers in international top-tier conference proceedings, journals and in the Springer Lecture Notes in Computer Science. He is co-author of several books on mobile software development and computer science and media in Germany. His research interests include integration of consumer electronic devices in vehicles and mobile and embedded software architectures, frameworks and mobile security. He received his PhD degree in real-time collaboration systems from the London College of Communication, UArts, UK. Since 2012 he is a member of the IEEE Consumer Electronics Society and the Communications Society.

Christos Grecos (SM IEEE 06, SM SPIE 2008) is the Dean of the Faculty of Computing and Information Technology in Sohar University, Oman. He worked as a Professor in Visual Communications Standards, and Head of School of Computing in the University of the West of Scotland and previously in the Universities of Central Lancashire and Loughborough, all in UK. He has also worked as an Independent Imaging Consultant for many years. His research interests include image/video compression standards, image/video processing and analysis, image/video networking and computer vision. He has published over 165 research papers in top-tier international publications including a number of IEEE transactions on these topics. He is on the editorial board or served as guest editor for many international journals, and he has been invited to give talks in various international conferences. He has obtained significant funding for his research as the Principal Investigator for several national or international projects funded by UK EPSRC or EU. He received his PhD degree in Image/Video Coding Algorithms from the University of Glamorgan, UK.