FACULTY OF ENGINEERING AND ARCHITECTURE

Using computation graphs for improved network performance in serverless applications

Jonas Vander Vennet
Student number: 01605412

Supervisors: Prof. dr. Wouter Tavernier, Prof. dr. ir. Mario Pickavet
Counsellor: Gourav Prateek Sharma

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021

GHENT UNIVERSITY

Acknowledgements

In the past year, I have received a great deal of support and assistance from many different people. This dissertation would not have been what it is today without you.

First and foremost, I want to thank my supervisors Prof. dr. Wouter Tavernier and Prof. dr. ir. Mario Pickavet for the guidance and helpful insights throughout the year. I also want to mention Gourav Prateek Sharma, who stepped in to provide additional aid. I appreciate it highly. You all helped me stay on course even though I sometimes lost my way.

I want to give special thanks to Amoghvarsha Suresh and Anshul Gandhi from Stony Brook University. Without our small bit of correspondence, the chapter discussing Apache Openwhisk would never have been possible.

In addition, I would like to thank my parents for their wise counsel and sympathetic ear. You were always there for me. There were a lot of difficult moments this year, but you helped me through them. I also thank my brother Simon for the ever enjoyable pizza Sundays. Last but not least, a special mention to my friends, especially Boris, who helped with proofreading, and my team at KAAG for providing the best support and pep talks I could ever wish for.

"If someone asks me what cloud computing is, I try not to get bogged down with definitions. I tell them that, simply put, cloud computing is a better way to run your business."

~Marc Benioff, Founder, CEO, and Chairman of Salesforce.

Abstract

Operating a data centre is no easy feat. Especially not given the enormous scale at which modern cloud services are running at. A rising star inside the cloud computing world is "serverless" computing. With more focus on productivity for the end-user while reducing server management, serverless computing absolutely might become one of the more popular cloud services. One of its major challenges for the future lies in efficient networking. Using the information embedded in the computation graph that models a complex workload allows the cloud provider to manage container allocations more efficiently such that a minimal burden is placed on the data centre network.

This dissertation starts with a thorough introduction to the topic of cloud computing and serverless platforms in particular. Additionally, the core problem statement of this dissertation is outlined: a network-optimised scheduling approach for serverless functions. Afterwards, the second chapter is dedicated to introducing the main research domains that overlap with this research. For each domain, the context detailing its connection to the global problem statement is highlighted, followed by a review of the recent developments in the relevant literature. Chapter three is a deep dive into the intricacies surrounding serverless scheduling optimisation. In it, I propose using graph partitioning to split a computation graph into parts that can each be mapped to a server node in a data centre. The chapter concludes with the discussion of a simulation of a real-world workload. The fourth chapter takes a more practical approach. It contains the review of one of the most well-known open-source serverless platforms, Apache Openwhisk, and investigates the feasibility of implementing custom scheduling logic by adapting the source code. This chapter highlights the most important design decisions that decide whether a platform is suited for thorough experimentation or not. Finally, the main takeaways from this research are presented together with a brief glance towards future endeavours that may be interesting to explore.

keywords: serverless | function scheduling | graph partitioning | Apache Openwhisk

Using computation graphs for improved network performance in serverless applications

Jonas Vander Vennet

Supervisor(s): Prof. dr. Wouter Tavernier, Prof. dr. ir. Mario Pickavet

Abstract— **Operating a data centre is no easy feat. Especially not given the enormous scale at which modern cloud services are running at. A rising star inside the cloud computing world is "serverless" computing. With more focus on productivity for the end-user while reducing server management, serverless computing absolutely might become one of the more popular cloud services. One of its major challenges for the future lies in efficient networking. Using the information embedded in the computation graph that models a complex workload allows the cloud provider to manage container allocations more efficiently such that a minimal burden is placed on the data centre network.**

This dissertation discusses the intricacies of serverless scheduling and the simulation of a network-optimising scheduler based on a graph partitioning heuristic. Additionally, the feasibility of customising the scheduling logic in a well-known open-source serverless platform, Apache Openwhisk, is analysed in depth.

Keywords— **serverless, function scheduling, graph partitioning, Apache Openwhisk**

I. Introduction

"Just use the cloud, that's much easier to handle" is a phrase that is quite common in a lot of companies. Indeed, cloud services can alleviate the effort required for the upkeep and management of physical devices that you need to serve a product to customers around the world. This has been the main selling point for transferring from on-premise to cloud-hosted services [1]. Traditional cloud services provide the end-user with a remote virtual machine that is as close to the bare metal alternative as possible, without the worries that come with managing a physical device (AWS EC2 [2], Google Compute Engine [3], Azure Virtual Machines [4], IBM Cloud Virtual Servers [5]). The cloud provides the environment, and the operational focus is more and more on usefully leveraging the service rather than the small details behind the scenes [6].

This sparked the introduction of serverless cloud services, where the smallest unit of measurement is no longer a VM but a *cloud function*. Cloud functions are small, stateless, high-level programs that execute in a cloud environment, usually as part of larger compositions of functions. Traditional cloud services will be called *serverful* to distinguish them from the serverless computing paradigm.

A serverless application can be seen as a generic arrangement of inter-dependent functions [7], a function composition. A composition can be defined using its computation graph, indicating which functions depend on each other and how much data needs to be passed from a parent's output to its children's input. Depending on the specific data centre node each function is executed on, this data is either localised to a server or requires a transfer across the data centre network. Such a network transfer takes time, especially in the case of, for example, large matrices. Time spent transferring data is time not spent performing useful calculations. The information embedded in a composition's computation graph will provide additional insights that help more optimally schedule functions to servers, minimising the incurred delay due to network transfers. The real-world effects of this approach are projected to be (1) faster overall execution time per function composition (2) less load on the backbone network, resulting in a high capacity for more function compositions to be executed.

Throughout this dissertation, several aspects surrounding the intricacies of serverless scheduling are discussed, both theoretically by virtue of a proposed heuristic and practically using a well-known open-source serverless platform, Apache Openwhisk.

First, a formal description of the network-optimised scheduling approach is introduced in Section II. Section III contains a more detailed view of the scheduling parameters and different circumstances that influence the potential impact of the scheduler. A feasibility study of customising the scheduler logic in Apache Openwhisk is outlined in Section IV. Finally, Section V concludes the work, including some notes on future work.

II. Problem formulation

Formally, the problem for optimising network performance in the scheduling of serverless applications can be stated as follows. Given a directed graph $G = (V_G, A_G)$ with V_G and A_G sets of vertices and arcs, respectively, interpret the vertices as small task units to be executed and the arcs as communication between the vertices. Arcs are weighted by the size of the data that needs to be communicated. This graph represents a single function composition to be scheduled. Additionally, an undirected graph $H = (V_H, E_H)$ defines the connectivity of the servers in the underlying data centre where G will be scheduled onto. The edges E_H are weighted by the inverse of the physical bandwidth of the link between each edge's endpoints, constituting the time delay incurred for transferring a certain amount of data. The nodes in V_H consist of both server nodes (V_{Se}) and network switches (V_{Sw}), without overlap between the two categories:

$$V_H = V_{Se} \cup V_{Sw} \tag{1}$$

$$V_{Se} \cap V_{Sw} = \emptyset \tag{2}$$

Finding a network optimised placement for each function in V_G corresponds to finding the map $f : V_G \rightarrow V_{Se}$ that minimises the following expression

$$\sum_{i,j \in V_G} \omega_{ij} \times SP_H(f(i), f(j)) \tag{3}$$

with ω_{ij} the weight of arc $e \in A_G$ that connects nodes i and $j \in V_G$, and $SP_H(x, y)$ the weight of the shortest path between nodes x and $y \in H$.

III. Serverless scheduling

Mahmoudi et al. [8] have performed empirical research into which approaches are preferred by the major cloud providers. Their results provide another point of view on this matter. They argue that it is likely Amazon "treats instance placement as a bin packing algorithm", prioritising re-use of active VMs and utilise their memory over starting a new VM to balance out the load. In contrast, they also mention that Microsoft Azure tends to use a spreading algorithm to minimise the average load on each of their servers. This means that two of the largest cloud providers have an almost opposite approach when it comes to scheduling function invocations. It can reasonably be assumed that these two cloud providers experience a similar type of traffic and a comparable load factor. Therefore, it can be argued that this difference in scheduling approach indicates sub-optimal behaviour of the algorithms that are currently in use. The results of Mahmoudi et al. [8] are supported by Lloyd et al. [9] and McGrath et al. [10] as they also indicate sub-optimality in the resource management protocols leveraged by cloud providers.

Over the last two years, many papers have been published to discuss widely different views and interpretations on how to design a more optimal scheduling protocol. First, FnSched [11] attempts to minimise resources occupied while still achieving similar service quality by leveraging CPU-shares to regulate CPU usage and to reduce contention. Second, the Skedulix framework [12] exploits the interaction of two classes of programs with the CPU: IO-bound and CPU-bound. This distinction allows it to nearly double the system's capacity with little penalty, assuming a somewhat equal representation of both classes. Lastly, Mahmoudi et al. [8] leverage machine learning with similar baseline parameters as found in Skedulix: CPU time, CPU idle time, context switches, disk read time, ... Additionally, parameters measuring load averages and percentage of memory in use are employed to create a neural network capable of context-aware global decision making over the entire service. There does not seem to be a significant research effort directly into optimising for network usage in the current literature.

A. Scheduling parameters

Serverless function compositions can be depicted in the form of a Directed Acyclic Graph (DAG). The graph visualises the dependencies between the nodes and provides hints for the parallelisability and potential bottlenecks. Dependencies do not paint the whole picture, of course. There are many different parameters that influence the overall timing of the execution process:

- The time at which a node's parents have finished executing. The moment in time at which a node i finishes is denoted by $t_{end}(i)$.

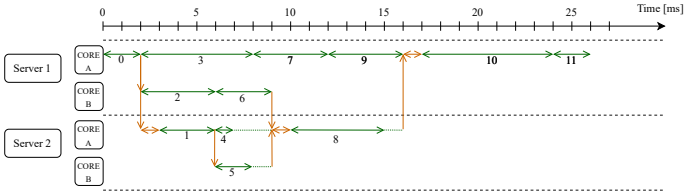


Fig. 1. Placement of an example composition according to the network-optimised approach. Green intervals resemble the execution of nodes in the composition, orange intervals resemble delays due to data transfers.

- The time it takes for each parent's output data to transfer to node i. Transfer time itself is dependent on the amount of output data (in MB) to transfer and the properties of the links connecting the source and destination servers. Assuming node j is a parent of node i, this timespan is denoted by $\Delta t_{trans}(j, i)$.
- The possible occurrence of a cold start can add an additional waiting period $\Delta t_{cs}(i)$ before actual code execution can begin. The value of $\Delta t_{cs}(i)$ will be equal to 0 if there is no cold start delay. Even though warm starts also introduce some delay, this delay can be assumed to be included in the node's execution time by default without loss of generality.
- The actual execution time $\Delta t_{exec}(i)$ of a node's source code. With the environment and parameters all present, the high-level source code usually needs somewhere in the order of milliseconds to finish execution.

These parameters, when combined with dependency information from the DAG, define the starting time at which a function can start executing:

$$t_{start}(i) = \max_{j \in parents} (t_{end}(j) + \Delta t_{trans}(j, i)) + \Delta t_{cs}(i) \tag{4}$$

$$t_{start}(0) = t_{begin} \tag{5}$$

$$t_{end,i} = t_{start,i} + \Delta t_{exec,i} \tag{6}$$

An example scheduling result is shown in Figure 1.

B. Graph partitioning and simulation metrics

Each node of a composition graph will be assigned a specific server to execute on. After execution, output data will transfer from parent nodes to their children. These transfers will introduce extra load on the network if the child node executes on a different server than its parent node. As a result, the goal behind scheduling a composition is to *group nodes together such that the time spent on transfers between the different groups is minimised*. These transfers are embedded in the composition graph as arc weights. As was also proposed by Verbelen et al. [13], wording the problem in this way (minimising arc weights between groups of nodes in a graph) immediately puts forward the study of *graph partitioning* as an interesting field of heuristics to explore. It is a fundamental problem in applications such as multiprocessor task allocation, circuit design and solving linear systems [14], [13]. For practical use, there exist well-known software that is designed to perform graph partitioning like METIS [15], [16], KaHIP [17], [18], PaToH [19] and SCOTCH [20].

Finding the candidate partitionings can be done fast and efficiently using one of the algorithms and software packages described by Verbelen et al. [13] and Li et al. [21]. The computationally most intensive part of this heuristic scheduling approach concerns the mapping of partitions to actual servers. Given a partitioning into n parts, a set of n servers needs to be selected such that there is a one-to-one mapping from partition to server. Ideally, these n servers are chosen such that they are "close" to one another in the underlying data centre. Close in this context means that the time to transfer data $\Delta t_{trans}(i, j)$ between two servers i and j is as low as possible. Fast transfers can be achieved through low hop count between servers (perhaps a direct connection), high bandwidth links on the path from i to j, or a combination of both. In order to be sure an optimal choice of servers is made, the procedure has to look at all n-tuples in the n-fold Cartesian product over the set of all servers.For a data centre consisting of K servers the number of n-tuples becomes $\binom{K}{n} = \frac{K!}{n!(K-n!)}$. Checking all n-tuples in the n-fold Cartesian product becomes infeasible for even slightly larger data centres. However, his computationally expensive check can be reduced by only considering n-tuples of close servers. Additionally, greedily opting for a first-fit search instead of checking all relevant n-tuples can speed up the search.

The ideal way to evaluate a heuristic like graph partitioning for this particular use case would be by integrating the necessary algorithms into real-world usage. The methodology behind this is discussed in the next section. Here, the focus will be on the alternative that is best suited for a heuristic methodology in development: simulation. By reducing the problem down to its most basic form and stripping it of unnecessary complications, a very simple Python program can simulate the scheduling process and evaluate the interesting metrics concerning the heuristic in play. Although simple to understand, the design and implementation of such a simulation environment can be tricky to get right. For example, deciding on what is relevant to the simulation and what is not, or addressing the integration of a metric that was originally not intended.

The most relevant metrics to extract from a simulation of a network-optimising scheduler are (1) execution time (of the total composition), (2) amount of data transferred between different servers and (3) resource usage. However, the raw value (in ms) of total execution time is not immensely useful. This is due to cold start delays being several orders of magnitude higher than most functions' execution time. For example, cold starts can range from 100ms to more than a second to complete while functions typically span a couple of milliseconds [22]. It is more valuable to report the execution time as fine-grained as possible, with its contributing factors (cold starts, transfer delays, setup times, raw execution time) separated. Resource usage can be interpreted in many different ways. As the number of servers used, cores used, memory used all either as a total value or as a graph over time. The behaviour over time shows how spiky or constant this load is. This metric will not make or break a scheduling approach but can be used to make more contextual judgement calls when evaluating all metrics together across different approaches.

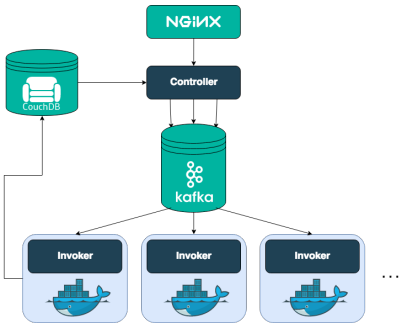


Fig. 2. High-level overview of the Openwhisk architecture. It visualises the different interactions between the most important components during a request's lifetime, starting from the initial HTTP request through nginx until the actual code execution in a Docker container and result storage for future reference [26].

IV. Feasibility study: Apache Openwhisk

Aside from the theoretical insights that can be gained from studying the properties and intricacies surrounding a newly proposed scheduling algorithm, it can be equally valuable to look into a practical approach. For example, the adaptation of a commercially well-known product and investigating whether it provides the necessary features to change the scheduling approach. This chapter dives deeper into the inner workings of one of the most popular open-source serverless frameworks available: Apache Openwhisk, due to its usage in IBM's FaaS platform IBM Cloud Functions.

The current literature mainly focuses on the discussion of theoretical aspects while skipping over the details of modifying existing platforms for testing purposes. Architecturally these platforms are usually well-documented, but the process of modifying internal algorithms can be somewhat obscure. This chapter will make it easier for new researchers to deploy their custom scheduling behaviour quickly.

During this section, the Openwhisk-specific terminology of invoker is used to indicate a server in a data centre. An invoker is an Openwhisk abstraction of a server node. The invoker is responsible for managing the execution of a particular function that is assigned to it, as well as preparing the environment required for execution.

The core of Openwhisk is an event-driven architecture [23], [24] with lots of components all designed to each serve one particular purpose, such as routing, controllers, invokers, authentication, messaging or logging. A high-level overview of the architecture is shown in Figure 2. Every component is a complete, fully customisable software package on its own. It can perfectly be swapped by any other customised service as long as the original interface contract remains intact. It is also noteworthy that, in a production environment, many components like the controller will have several replicas deployed at the same time [25]. Only a select number of components do not support multiple replicas, such as CouchDB and Kafka.

A. High-level communication chain

This section describes each of the different high-level steps involved in processing a request to execute an Openwhisk function based on the official code repository [26]. The official doc-

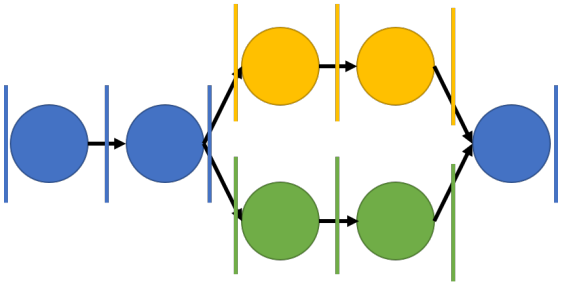


Fig. 3. Example composition with three subcompositions in total, each marked with a different colour. Secondary activations are marked as vertical lines and follow the colour scheme. Each subcomposition of n actions requires $n + 1$ secondary activations.

umentation on the "System Overview" contains substantially more detailed information that is out of scope for this section concerning rules, triggers, actions and integrations with additional services.

The reverse proxy (nginx) passes the request to one of the available controller instances. The controller is the central caretaker of this request: it will start gathering the extra information from inside the system. It is the controller's responsibility to make sure its decisions satisfy the request's requirements. A request could involve uploading the code for a new function, a change in event triggers or the start of (a chain of) function invocations. The first step a controller performs is to authenticate the request and verify the privileges associated with the user's role against the CouchDB database. After verifying any claims, CouchDB is contacted again to retrieve the relevant *action* that contains all of the metadata like function code, parameters, required available memory to execute and rate-limiting. The next step is combining all this metadata into a final decision which invoker the request will be assigned to.

Finally, the actual code execution occurs in an encapsulated, controlled and safely configured Docker environment. Here, potential cold start penalties occur, waiting for the correct runtime to initialise. After completion, the result of the function code is retrieved from the container and stored back into CouchDB. Openwhisk invocations are asynchronous by design, which means the result is not sent to the end-user directly. Therefore, storing the result in CouchDB marks the end of the request life cycle.

B. Under the hood: subcompositions and secondary activations

Contrary to the function compositions from before, an Openwhisk composition consists of a starting action followed by all simple actions that will be executed as one sequence with 100% certainty. Actions that cannot satisfy this requirement are grouped into subcompositions. Figure 3 shows a simple composition with each (sub)composition in a different colour. The split into two separate parallel sequences causes each branch to become a proper subcomposition. These subcompositions are treated just like a regular composition from the perspective of Openwhisk, except that they are kickstarted from inside another composition rather than by an external trigger.

Additionally, Openwhisk adds a second layer of functions to each composition. These *secondary activations* serve an administrative purpose and are not inserted into the composition by the end-user. They are responsible for orchestrating the different invocations in the correct order: deciding on the next action to execute, early stopping in case of an error. Consequently, secondary activations run before and after each action. Whenever possible, the before and after secondaries are combined into one to reduce overhead. As a result, a composition of n actions requires an additional $n + 1$ secondary activations. Figure 3 shows the interleaving of actions and secondary activations. As mentioned before, subcompositions are treated as fully separate, autonomous compositions with their own entry point and finalisation.

C. Openwhisk scheduling logic

The default implementation of the controller logic can be found in `core/loadBalancer/ShardingContainerPoolBalancer.scala` inside the official Openwhisk repository [27]. Customisation of the logic can be achieved by either editing this file, or by creating another LoadBalancer class and reconfiguring the loadbalancer spi source during Openwhisk setup in `ansible/group_vars/all`. The two main functions in `ShardingContainerPoolBalancer.scala` that need to be adapted are *publish* and *schedule*. The former is used to provide the necessary parameters to the latter, which is where the scheduling logic to pick a well-suited invoker for the current action is located.

The default load balancing algorithm used in Openwhisk is a simple, stateless sharding algorithm. At its core, there are many similarities with how a hash table determines the placement of elements. The load balancer calculates a hash of the namespace-action pair to derive the index of an invoker. This first invoker is called the *home-invoker* and is the same for every invocation of a specific serverless function. Several constraints are checked before assigning the function to the home-invoker, such as memory capacity. If any constraint cannot be satisfied, a step increment is added to the index of the current invoker, followed by a new attempt to find a destination for the invoked function. In order to reduce the number of possible collisions, this increment is picked out of the set of coprime numbers smaller than the number of invokers available.

D. Roadblocks

Adapting this stateless default algorithm into a smarter, stateful approach requires the controller to keep track of specific events and parameters during its execution. Adding state is not a problem, as there is already some state present by default, such as the number of invokers. Consequently, the core issue in the modification process will be getting hold of the right information and passing it to the controller logic. Once the controller has access to information, it is trivial to add this information to its existing state.

The introduction of state into one of Openwhisk's core components breaks some assumptions about the Openwhisk architecture. Stateless controller instances can be replicated indefinitely, but if the controllers start accumulating state on which they will base future decisions, this can become an issue. Either all actions dependent on a subset of information for their

scheduling have to be passed to the one controller instance managing it, or controllers have to communicate their state. A third option could be to disallow controller replication, similar to how it is discouraged for the gateway and CouchDB. In a document on design considerations [28], it is proposed that, due to the short-lived nature of cloud functions and a resulting fast-changing state, the time to aggregate the required information would become too large. Therefore, it is from now on assumed that there is only one controller instance that handles all requests meaning it does not have to communicate its internal state with others. Handling the increase in state is the first major issue in the practical feasibility of the proposed stateful algorithm.

Another major roadblock that prevents a straightforward modification process is the action-centric design of Openwhisk. Even though Openwhisk supports the invocation of chained actions as larger compositions, each action in a composition is still handled individually when its preconditions are met and is required to execute. If an action is part of a composition, additional metadata and parameters are provided. The question now becomes: "given the action-centric design of Openwhisk, is it possible for a stateful controller to still gather enough information to understand the required context for scheduling every action in a generic composition?" The answer to this question is *almost, but no*.

Openwhisk provides one piece of composition metadata by default: *cause* information. This optional field contains the ID of the root secondary activation that started the composition. Linking back to Figure 3, the colours indicate a common cause field. Notice that this means the different coloured parts of this composition have no link between them from the perspective of the available metadata. One way we can attempt to tackle the issue of fully separated subcompositions is by thoroughly investigating the internal data structures of Openwhisk. One should assume the Openwhisk architecture can be lightly modified to funnel more information towards the controller. Workarounds that require drastic changes to the Openwhisk internals will not be explored. This would deviate from the initial goal of studying the feasibility of a more intelligent scheduling approach *in Openwhisk*.

The solution to circumventing the issue of fully separated subcompositions is to look at the hierarchical transaction IDs that are used to provide clear and insightful application logs to developers. The Openwhisk team may not have designed their transaction IDs for this use case, but there is an intriguing property that follows from the way the creation of these values is structured. The inner transaction ID is tied to one specific action instance, and **the outer ID is equal to the inner ID of the action that started the current (sub)composition.** Additionally, it is trivial to add these transaction IDs as input to the scheduler: the inner ID is already part of the interface, and the outer ID is used to create the inner ID immediately before the scheduler invocation. Therefore it can easily be added as an optional metadata field to the existing parameters passed to the scheduler. Therefore, this property is ideal for providing crucial insight across subcomposition boundaries required for an intelligent scheduling algorithm. The new possibilities this provides to a composition to query historical information are visualised in Figure 4.

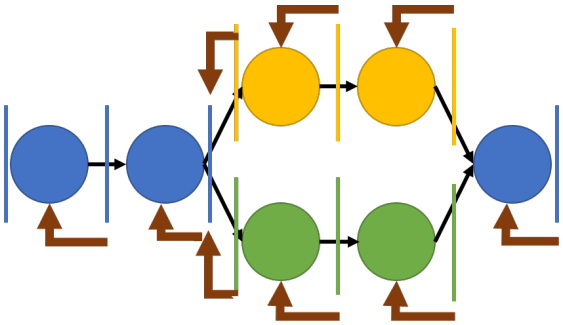


Fig. 4. Additional relationships that can be inferred by keeping track of inner and outer transaction IDs during scheduling passes. At the point where the blue chain branches into a yellow and green chain, there is now a newly formed information link across the subcomposition boundaries.

However, a multi fan-in parent-child dynamic still does not contain the necessary information for complete inference. This can be seen at the tail end of the composition in Figure 4, where there is no link between the final blue action and the yellow and green secondary activations preceding it. This is a parent-child dynamic that cannot be circumvented in the current Openwhisk architecture due to its action-centric nature.

In 2018, the Openwhisk designers noticed a discrepancy between the current architecture and the two ways of deploying production-ready Openwhisk services [29]. Openwhisk is either deployed on low-level machines (bare-metal or VMs) or a high-level container orchestrator (e.g. Kubernetes) [24]. The latter is widely used but makes the invoker design from Figure 2 feel rather foreign due to the container-in-container execution that ensues. The proposal that started back in 2018 has seen regular design updates but has not yet made its way into the core of Openwhisk due to the drastic changes it puts forward.

The focus of the revamped architecture seems to be around low-level container control mechanisms. Explicitly removing state from controllers and load balancers means most of the features that are required to make the previous modification fully functional do not seem likely. Openwhisk is doubling down on simple, stateless placement algorithms. There is no mention of compositions in the current proposals, indicating they represent only a small share of real-world Openwhisk usage and are not actively being optimised for.

V. Conclusion and Future Work

The main takeaways from this research are:

- The proposed network-optimised scheduler requires a lot of additional computational effort compared to the default schedulers found with most major cloud providers. Additional improvements are encouraged to further improve efficiency and effectiveness.
- The demand in serverless platforms for more complex scheduling algorithms is limited.
- Unless additional workarounds are discovered, Openwhisk does not seem suitable for detailed research experimentation.

Future work in this area may want to focus on the difference between graph partitioning and community detection for more imbalanced function-to-server mapping. Furthermore, introducing "slack time" where a function t_{start} can be delayed some

variable amount for additional degrees of freedom, potentially improving the optimality of the resulting placements at the cost of increased complexity.

REFERENCES

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia, "A view of cloud computing," 2010.

[2] Amazon Web Services, "Amazon EC2 Features - Amazon Web Services," 2020, Accessed on: 2021-05-23.

[3] Google, "Compute Engine: Virtual Machines (VMs)," Accessed on: 2021-05-23.

[4] Microsoft, "Explore Azure Virtual Machines - Learn — Microsoft Docs," Accessed on: 2021-05-23.

[5] IBM, "IBM Cloud Virtual Servers - Overview - United Kingdom — IBM," Accessed on: 2021-05-23.

[6] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson, "Cloud programming simplified: A Berkeley view on serverless computing," feb 2019.

[7] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu, "Serverless computing: One step forward, two steps back," in *CIDR 2019 - 9th Biennial Conference on Innovative Data Systems Research*, 2019.

[8] Nima Mahmoudi, Hamzeh Khazaei, Changyuan Lin, and Marin Litoiu, "Optimizing serverless computing: Introducing an adaptive function placement algorithm," in *CASCON 2019 Proceedings - Conference of the Centre for Advanced Studies on Collaborative Research - Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, USA, 2020, CASCON '19, pp. 203–213, IBM Corp.

[9] Wes Lloyd, Shrideep Pallickara, Olaf David, Jim Lyon, Mazdak Arabi, and Ken Rojas, "Performance modeling to support multi-tier application deployment to infrastructure-as-a-service clouds," in *Proceedings - 2012 IEEE/ACM 5th International Conference on Utility and Cloud Computing, UCC 2012*, 2012, pp. 73–80.

[10] Garrett McGrath and Paul R. Brenner, "Serverless Computing: Design, Implementation, and Performance," in *Proceedings - IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW 2017*, 2017, pp. 405–410.

[11] Amoghvarsha Suresh and Anshul Gandhi, "FNSched: An efficient scheduler for serverless functions," in *WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019*, New York, New York, USA, 2019, pp. 19–24, ACM Press.

[12] Anirban Das, Andrew Leaf, Carlos A. Varela, and Stacy Patterson, "Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications," in *IEEE International Conference on Cloud Computing, CLOUD*, jun 2020, vol. 2020-Octob, pp. 609–618.

[13] Tim Verbelen, Tim Stevens, Filip De Turck, and Bart Dhoedt, "Graph partitioning algorithms for optimizing software deployment in mobile cloud computing," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 451–459, 2013.

[14] Konstantin Andreev and Harald Räcke, "Balanced graph partitioning," in *Annual ACM Symposium on Parallel Algorithms and Architectures*, New York, New York, USA, 2004, vol. 16, pp. 120–124, ACM Press.

[15] George Karypis and Vipin Kumar, "METIS* A Software Package for Partitioning Unstructured Graphs , Partitioning Meshes , and Computing Fill-Reducing Orderings of Sparse Matrices," *Manual*, pp. 1–44, 1998.

[16] Ananth Kalyanaraman, Kevin Hammond, Jarek Nieplocha †, Manojkumar Krishnan, Bruce Palmer, Vinod Tipparaju, Robert Harrison, Daniel Chavarrıa-Miranda, Junichiro Makino, David Bader, Guojing Cong, Bruce Hendrickson, John Shalf, David Donofrio, Chris Rowen, Leonid Oliker, Michael Wehner, and John L. Gustafson, "Graph Partitioning Software," in *Encyclopedia of Parallel Computing*, pp. 808–808. Springer US, Boston, MA, 2011.

[17] KAHIP, "Karlsruhe High Quality Partitioning," Accessed on: 2021-05-03.

[18] Peter Sanders and Christian Schulz, "Think locally, act globally: Highly balanced graph partitioning," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013, vol. 7933 LNCS, pp. 164–175.

[19] Bruce Leasure, David J. Kuck, Sergei Gorlatch, Murray Cole, Gregory R. Watson, Alain Darte, David Padua, Utpal Banerjee, Olaf Schenk, Klaus Gärtner, David Padua, Howard Jay Siegel, Bobby Dalton Young, Roy H. Campbell, Ümit Çatalyürek, Cevdet Aykanat, Jasmin Ajanovic, Stefan Schmid, Roger Wattenhofer, E. N. Mootaz Elnozahy, Evan W. Speight, Jian Li, Ram Rajamony, Lixin Zhang, Baba Arimilli, David Padua, Michael Gerndt, Michael Gerndt, David Padua, Jack B. Dennis, Barry Smith, George Almasi, Alexandros Stamatakis, Davide Sangiorgi, Davide Sangiorgi, David Padua, John A. Gunnels, Jack Dongarra, Piotr Luszczek, Bernd Mohr, Rudolf Eigenmann, Paul Feautrier, Christian Lengauer, David Padua, Pradip Bose, Joseph F. JaJa, Anshul Gupta, Rocco De Nicola, David Padua, David Padua, David Padua, Roger S. Armen, Eric R. May, Michela Taufer, and Al Geist, "PaToH (Partitioning Tool for Hypergraphs)," in *Encyclopedia of Parallel Computing*, pp. 1479–1487. Springer US, Boston, MA, 2011.

[20] C. Chevalier and F. Pellegrini, "PT-Scotch: A tool for efficient parallel graph ordering," *Parallel Computing*, vol. 34, no. 6-8, pp. 318–331, 2008.

[21] Qi Li, Jiang Zhong, Zehong Cao, and Xue Li, "Optimizing streaming graph partitioning via a heuristic greedy method and caching strategy," *Optimization Methods and Software*, vol. 35, no. 6, 2020.

[22] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky, "Putting the "micro" back in microservice," in *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018*, 2020, pp. 645–650.

[23] Karim Djemame, Matthew Parker, and Daniel Datsev, "Open-source Serverless Architectures: An Evaluation of Apache OpenWhisk," in *Proceedings - 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing, UCC 2020*, 2020, pp. 329–335.

[24] Apache, "Apache Openwhisk Documentation," Accessed on: 2020-11-05.

[25] Apache, "Openwhisk Configuration Choices," Accessed on: 2021-05-04.

[26] Apache, "Openwhisk System Overview," Accessed on: 2020-11-06.

[27] The Apache Foundation, "Apache OpenWhisk is a serverless, open source cloud platform," 2018, Accessed on: 2020-10-15.

[28] Apache, "Difficulties for optimal scheduling," Accessed on: 2020-11-30.

[29] Markus Thömmes, "OpenWhisk future architecture," Accessed on: 2021-05-11.

Contents

List of Figures

List of Tables

"If you think you've seen this movie before, you are right. Cloud computing is based on the time-sharing model we leveraged years ago before we could afford our own computers. The idea is to share computing power among many companies and people, thereby reducing the cost of that computing power to those who leverage it. The value of time share and the core value of cloud computing are pretty much the same, only the resources these days are much better and more cost effective."

~David Linthicum, author of Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide

1

Introduction

This dissertation titled "Using computation graphs for improved network performance in serverless applications" tackles many different fields in computer science. Both traditional fields, such as graph theory and program scheduling are combined with the modern context of cloud computing and serverless architectures. Reinterpreting traditional theory in a modern context can prove immensely intriguing as the development of a new situation causes the need to reassess old assumptions, yielding valuable case studies. Cloud computing drives the need for a proper reinterpretation of how servers should be managed and how we think about applications both as an end-user, developer or system administrator.

Microsoft Azure states that, at its core, cloud computing is "the delivery of computing services — including servers, storage, databases, networking, software, analytics, and intelligence — over the Internet ('the cloud') to offer faster innovation, flexible resources, and economies of scale" [6]. The flexibility of the cloud stems from the many layers of abstraction that are created by cloud service providers such as Microsoft Azure, Amazon or Google. The most recent and extreme progression of these abstractions is serverless computing. Serverless applications are split up into small chunks of application logic called functions. In serverless computing, every part of the system administration effort up until this application-specific logic is handled by the cloud service provider. As is illustrated in Figure 1.1, the cloud service provider handles everything concerning storage, networking, VM management, runtime environment initialisation, authorisation and

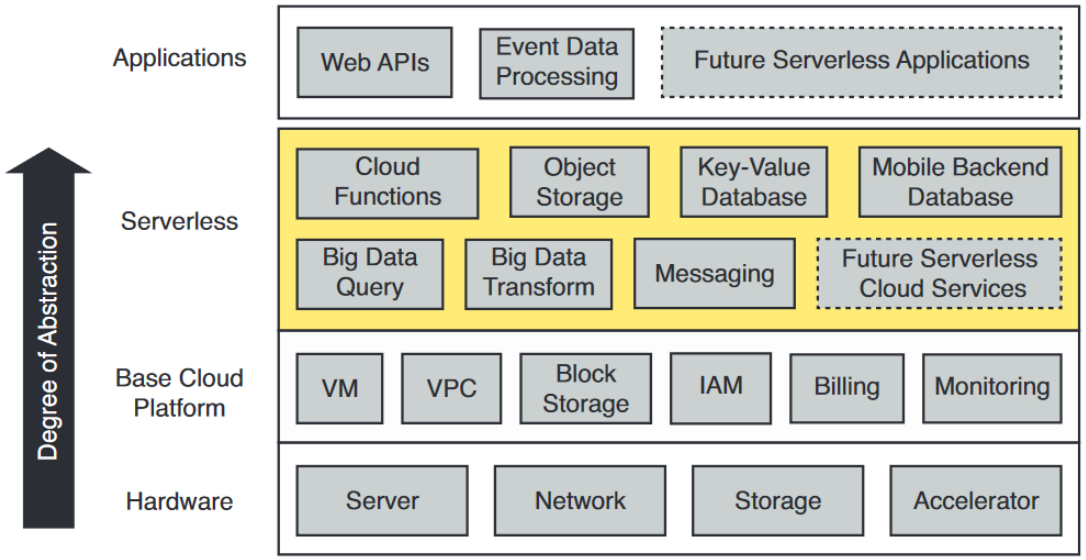


Figure 1.1: Architecture of the serverless cloud. Cloud service providers manage server resources, storage, networking and much more while application developers can focus on functionality. [1]

much more in a way that is invisible from the end-user. In the next chapter, the design behind serverless computing is discussed in more detail.

As more and more of the system administration becomes part of the automated background tasks handled by the cloud service provider, the algorithms behind these tasks gain significant importance. Every slightest improvement in efficiency in the management of resources is no longer bound to a few servers managed by a single client but rather felt globally in the underlying data centre. One of the main opportunities for the future of serverless computing lies with efficient networking. Improving network performance is the core research problem behind this dissertation.

A serverless application can be seen as a generic arrangement of interdependent functions, a function composition. A composition can be defined using its computation graph, indicating which functions depend on each other and how much data must be passed from a parent's output to its children's input. This graph will henceforth be referred to as the composition graph. Depending on the specific data centre node each function is executed on, this data is either localised to a server or requires a transfer across the data centre network. Such a network transfer takes time, especially in the case of, for example, large matrices. Time spent transferring data is time not spent performing useful calculations. The information embedded in a composition graph will provide additional insights that help more optimally schedule functions

to servers, minimising the incurred delay due to network transfers. The real-world effects of this approach are projected to be (1) faster overall execution time per function composition (2) less load on the backbone network, resulting in a high capacity for more function compositions to be executed.

Formally, the problem for optimising network performance in the scheduling of serverless applications can be stated as follows. Given a directed graph $G = (V_G, A_G)$ with V_G and A_G sets of vertices and arcs, respectively, interpret the vertices as small task units to be executed and the arcs as communication between the vertices. Arcs are weighted by the size of the data that needs to be communicated. This graph represents a single function composition to be scheduled. Additionally, an undirected graph $H = (V_H, E_H)$ defines the connectivity of the servers in the underlying data centre where G will be scheduled onto. The edges E_H are weighted by the inverse of the physical bandwidth of the link between each edge's endpoints, constituting the time delay incurred for transferring a certain amount of data. The nodes in V_H consist of both server nodes (V_{Se}) and network switches (V_{Sw}), without overlap between the two categories:

$$V_H = V_{Se} \cup V_{Sw}$$

$$V_{Se} \cap V_{Sw} = \emptyset$$

Finding an network optimised placement for each function in V_G corresponds to finding the map $f : V_G \to V_{Se}$ that minimises

$$\sum_{i,j \in V_G} \omega_{ij} \times SP_H(f(i), f(j))$$

with ω_{ij} the weight of arc $e \in A_G$ that connects nodes i and $j \in V_G$,
and $SP_H(x, y)$ the weight of the shortest path between nodes x and $y \in H$.

The research performed in this dissertation is twofold. First is a deep dive into the intricacies surrounding serverless scheduling optimisation, detailing other kinds of delays that can be incurred next to the transfer delays laid out above. Second is a discussion on the feasibility of implementing custom scheduling logic into one of the most well-known open-source serverless platforms, Apache Openwhisk. The following chapter provides a more detailed background on the various academic fields that overlap with the conducted research, as well as a review of their recent literature.

Relevant source code used throughout this document are all gathered here:

`https://github.com/jonasvandervennet/master-thesis-code`

"There was a time when every household, town, farm or village had its own water well. Today, shared public utilities give us access to clean water by simply turning on the tap; cloud computing works in a similar fashion. Just like water from the tap in your kitchen, cloud computing services can be turned on or off quickly as needed. Like at the water company, there is a team of dedicated professionals making sure the service provided is safe, secure and available on a 24/7 basis. When the tap isn't on, not only are you saving water, but you aren't paying for resources you don't currently need."

~Vivek Kundra, Federal CIO, United States Government, 2010.

2

Background and current literature

This dissertation spans several different research fields, each of which requires proper context and introduction. This chapter covers every topic that will be touched upon in later argumentation and analyses. These topics include cloud service providers, data centre layout, scheduling algorithms, graph theory and ILP problems.

2.1 Serverless cloud computing

"Just use the cloud, that's much easier to handle" is a phrase that is quite common in a lot of companies. Indeed, cloud services can alleviate the effort required for the upkeep and management of physical devices that you need to serve a product to customers around the world. This has been the main selling point for transferring from on-premise to cloud-hosted services [7]. Traditional cloud services provide the end-user with a remote virtual machine that is as close to the bare metal alternative as possible, without the worries that come with managing a physical device (AWS EC2 [8], Google Compute Engine [9], Azure Virtual Machines [10], IBM Cloud Virtual Servers [11]). The cloud provides the environment, and the operational focus is more and more on usefully leveraging the service rather than the small details behind the scenes [1]. The trend of offloading predominantly repetitive tasks to the cloud providers continued after the introduction of traditional cloud services [1]. A developer that wanted to host a reasonably

straightforward application consisting of about 50 lines of high-level code would spend most of their time setting up environments for the application to work in. This sparked the introduction of serverless cloud services, where the smallest unit of measurement is no longer a VM but a *cloud function*. Cloud functions are small, stateless, high-level programs that execute in a cloud environment, usually as part of larger compositions of functions. Traditional cloud services will be called *serverful* to distinguish them from the serverless computing paradigm.

Serverless can be a misleading term, as there are still servers involved. The servers are abstracted away from the perspective of the end-user, who only has to provide high-level code for their functions to execute [12]. As shown in Figure 2.1, everything from server maintenance to language runtime environment is managed by the cloud provider in a data centre. Jonas et al. [1] introduce an excellent analogy with programming languages. An assembly programmer has to select registers, perform simple arithmetic steps and interact with memory for loading and storing data for even the smallest set of instructions. Meanwhile, a programmer using a high-level programming language such as Python or Java does not have to worry about these menial tasks. The same idea holds when comparing serverful and serverless cloud services. Another critical distinction between serverless and serverful computing is the billing approach. Serverless bills in increments of 100ms while other services charge by the hour [7] and only for the resources used rather than resources allocated. This encourages the design of efficient functions by the end-user. For example, AWS Lambda calculates billing details based on the compute time (in "GB-seconds"[1]) a function uses [13].

A serverless function can be invoked by different types of triggers, both internal and external [14]. External triggers can be in the form of HTTP requests to an API gateway. Internal triggers can be based on events on other resources or direct triggers from other functions. Current state-of-the-art serverless architectures start to show their flaws once functions become part of larger compositions. Hellerstein et al. [15] split serverless applications into three categories:

- **Embarrassingly parallel functions:** Many different functions are executed but with no dependence on each other. Applications in this category can benefit from the auto-scaling handled by the cloud provider. However, these applications are not straightforward to get working, as shown by the research projects PyWren [16] and ExCamera [17].
- **Orchestration functions:** Functions in this category are used as an intermediary between the end-user and other cloud services. They preprocess relevant data and orchestrate the calls to heavy-duty services inside the same data centre.

[1]For example, if you allocated 512MB of memory to your function, executed it 3 million times, and it ran for 1 second each time, your charged compute would be calculated as follows: 3,000,000 * 512MB/1024 = 1,500,000 GB-s [13]

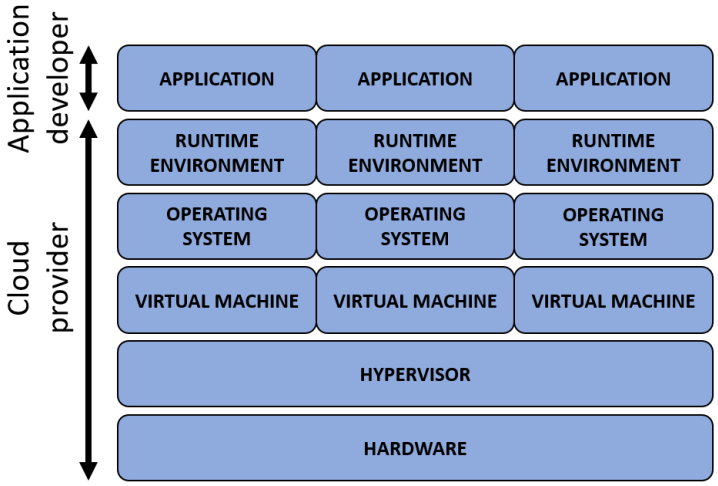


Figure 2.1: Responsibilities in terms of server management for cloud providers and application developers.

- **Function compositions:** In the third category, a sequence of functions is invoked with a data dependency between them. Usually, these compositions carry and mutate some state that is carried over from function to function. These compositions can be arbitrarily complex but suffer from high latency due to cumulative delays and overhead throughout the entire execution. One particular case study reports a composition handling account creation logic with an average end-to-end execution time of *ten minutes* [15]. In general, function compositions can be also used in every context the two previous categories are used. For example, sequential steps in account creation or scientific calculations such as matrix manipulations for QR decomposition and the linear least squares problem.

There is a lot of room for improvement pertaining to the third category of serverless applications. Therefore, function compositions will be the main focus of this dissertation going forward. The next part of this section will discuss in more detail the process of defining a composition and general considerations about execution from the viewpoint of a cloud provider.

2.1.1 Serverless compositions

Serverless compositions are applications that are made up of many small functional blocks. These functions are passed on to the cloud provider as high-level source code, frequently used languages include Python, Java, Go, C# [18]. When a composition is started, either by an external request or an internal event, it is the cloud provider's responsibility to orchestrate the entire chain of function invocations as efficient as possible. Efficient can mean different things here. Potential contributing factors include end-to-end completion time, staying within the end-

user's allocated budget, communication overhead, security and energy efficiency [19]. Scheduling these compositions efficiently and at a large scale is a challenge for any cloud provider. There can be millions of invocations passing through a data centre at the same time [20]. Many possible scheduling approaches exist, each with its respective assumptions and trade-offs. These approaches will be discussed in the remainder of this section, as well as in Section 2.3.

The structure of workflows for applications in the serverless paradigm often resembles that of a directed acyclic graph (DAG) [21]. This is especially true for data-intensive scientific use cases [22]. Functions are executed in short bursts, one after the other, with some pre-known logic determining the sequence of execution. This implies that insights gathered from constructing a DAG for a given workflow could be useful in optimising the scheduling and allocation of resources during execution.

Scheduling a series of function invocations can be split up into three separate tasks. First, knowing the order in which functions will be required to execute. Second, allocating a compute node in a data centre where the function can be invoked. Third, deciding which of the available nodes you want to invoke the function on.

When preparing a node to execute a function's source code, timing is essential. Invoking a function whose compute node is not ready introduces a significant delay in the workflow execution. Such an unprepared invocation is called a cold start. These can range from 100ms to more than a second to complete [23]. Cold start delays are experienced as pure overhead by the end-user and are generally detrimental to the user experience [13]. An efficient scheduler should aim to reduce the impact cold starts have on the end-user when a composition is being executed. There has been promising research into reducing the cold start duration down to the order of microseconds, which can also aid in lowering their negative impact [23]. The details of what constitutes these delays are outside the scope of this dissertation.

Picking the destination node for a function in a smart way requires knowledge about which nodes are available and how they interact with each other. Nodes operate in a data centre and are interconnected with switches and other servers in different ways depending on the layout of the data centre. The following section will discuss different layouts that are frequently used in current data centres.

2.2 Data centre layout

At the core of cloud computing services lies the data centre. The physical layout of a data centre determines a lot of its most relevant properties: scalability, throughput, latency, fault tolerance and environmental impact [2, 24, 25]. Scalability, throughput and latency are important metrics to evaluate the data centre network architecture with almost 70% of network communication taking place within a data centre [2]. Additionally, many researchers are currently looking into possible ways to operate a data centre in a sustainable way to prevent unnecessary damage to the environment [25, 26, 27, 28]. Heat flow, cooling infrastructure and energy consumption are the major factors in pollution caused by day-to-day data centre operations. While highly important, sustainability research is out of scope for this dissertation.

Current data centres house many thousands of servers, which all need to be connected to a core network [29]. Three main approaches are prevalent in the current literature: tree-based topologies [24], recursive topologies [30, 31] and flexible (optical switching) topologies [32]. These approaches are mentioned in chronological order, with as the oldest the traditional switch-centric three-tier tree-based architecture that is still widely in use today [2]. This legacy approach has the advantage of being very simple and intuitive but suffers from a lack of scalability, server-to-server throughput, and expensive equipment [30]. High-end hardware has to be used at the aggregate and core layers to prevent bottlenecking the bandwidth, as shown in Figure 2.2. Other tree-based variations have been proposed, such as the fat-tree by Al-Fares et al., to address some of the issues of the traditional approach. Still, tree-based methods continuously struggle with scalability and fault-tolerance [24]. Fat-tree layouts can still deliver high throughput and low latency for their fixed sizes, according to a performance analysis by Bilal et al. [2].

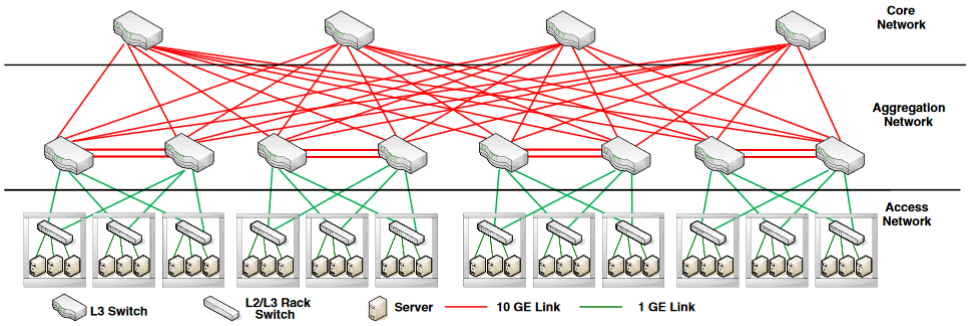


Figure 2.2: A three-tier data centre architecture. [2]

Guo et al. diverge from the switch-centric path and interconnect at the server level for their data centre network architecture proposals DCell [30] and BCube [31]. A DCell is recursively defined where a higher level DCell consists of some number of fully connected lower level DCells.

Figure 2.3 shows a level-two DCell consisting of seven fully connected level-one DCells, each containing three groups of two connected servers. In terms of throughput and latency over multiple types of traffic, DCell performs worse than tree-based alternatives for a large number of nodes in the network [2]. On the other hand, it does provide more scalability and fault tolerance than the tree-based networks due to the recursive definition and many available paths between any two servers. It is also less expensive due to it not requiring any enterprise-level hardware for the switches, instead relying on commodity hardware [30]. For massive data centres, this hardware cost is a non-negligible factor. Therefore, commodity hardware is often preferred [33]. To combat some of the problems with DCell, a modified proposal was published a year later about BCube [31]. BCube is targeted at modular data centres [34]. Performance analyses suggest that BCube is roughly equal in terms of throughput and latency to fat-trees. The improved path selection and routing allow for higher quality all-to-one and all-to-all communication than DCell with the same selection of inexpensive commodity hardware [31].

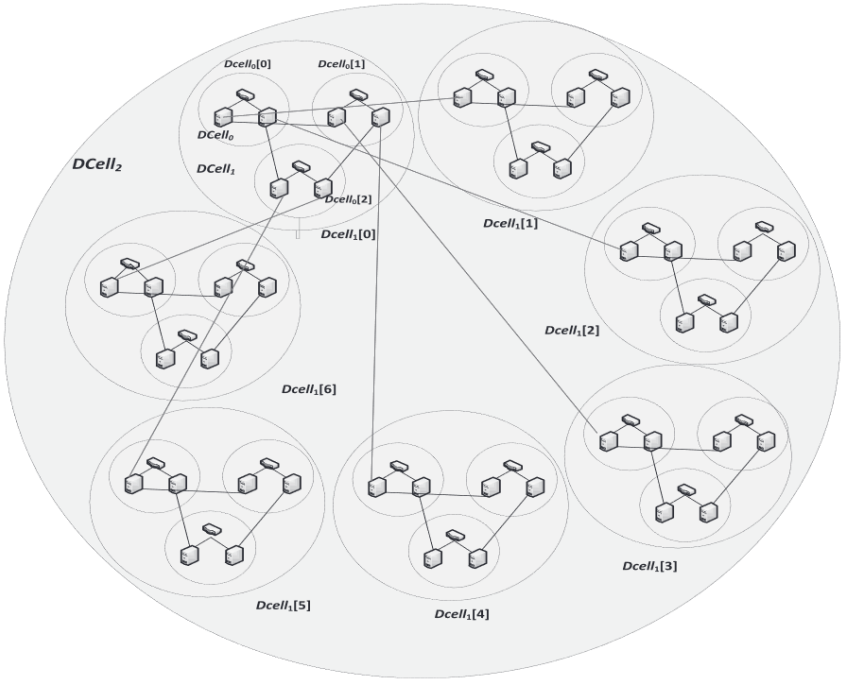


Figure 2.3: A level-two DCell architecture. Dcell_2 consists of seven DCell_1. Each DCell_1 contains three DCell_0 which ultimately group two connected services. Second-level connections are shown from $\text{DCell}_1[0]$ to all other level-one DCells. [2]

Lastly, there is a third alternative to data centre networking: flexible optical switching. Shukla et al. are not looking to influence the operations in existing data centres but are instead looking at the uses of optical communication with the next generation of data centres (NGDCs) in mind [32]. They expect NGDCs of "more than 10 million server cores" with major power reduction efforts in the near future. Many different types of optical switches with differing

underlying technologies, power efficiency and cost exist and are continuously being improved upon [32]. It can be assumed that the NGDCs will at least partially feature optical switches allowing for orders of magnitude higher capacities and drastically lowering power consumption compared to today's data centres [35].

There is a significant effort towards making data centres more scalable, maintainable and efficient. Along with the research and design efforts at the data centre level, the proposed scheduling approach can assist this development by managing the existing resources in a more scalable and resource-efficient way. Considering the network as a resource in itself that can be optimised for is therefore perfectly in line with the endeavours presented here.

2.3 Scheduling algorithms

This section aims to describe some of the most prevalent and interesting scheduling algorithms that have been proposed by researchers. Some of them are currently deployed by major cloud providers. The others serve a more academic purpose, inspiring future generations of algorithms that try to improve upon the flaws of the previous generation. First, the approaches that are in use by the major cloud providers will be discussed. Afterwards, a brief overview of state of the art research into this topic will be provided.

2.3.1 Cloud providers

Even though cloud providers do not publicly disclose how they operate their data centres in terms of scheduling practices, there are some parameters that are likely to influence their designs. A naive way to approach this is by looking at the way the cloud provider calculates the end-user cost for using their serverless services. Amazon, a serverless market leader [1], states in the pricing manual for their serverless function service AWS Lambda that the following three factors influence pricing: *(1)* the number of requests (expressed in millions), *(2)* the total compute in GB-seconds and *(3)* provisioned concurrency for an additional price per GB-second [13]. Listing better provisioning (fewer cold starts) as a premium feature reinforces the idea that end-users negatively experience cold starts and that the solution is not at all trivial. Next to that, Amazon's focus seems to be on the number of functions (load) and the total duration of each in relation to the required memory. This can be explained by assuming the data centre most likely has a limited amount of top-tier hardware with a large amount of RAM compared to the abundance of lower-tier hardware in the data centre.

Mahmoudi et al. [36] have performed empirical research into which approaches are preferred by the major cloud providers. Their results provide another point of view on this matter. They argue that it is likely Amazon "treats instance placement as a bin packing algorithm", prioritising re-use of active VMs and utilise their memory over starting a new VM to balance out the load. In contrast, they also mention that Microsoft Azure tends to use a spreading algorithm to minimise the average load on each of their servers. This means that two of the largest cloud providers have an almost opposite approach when it comes to scheduling function invocations. It can reasonably be assumed that these two cloud providers experience a similar type of traffic and a comparable load factor. Therefore, it can be argued that this difference in scheduling approach indicates sub-optimal behaviour of the algorithms that are currently in use. The results of Mahmoudi et al. [36] are supported by Lloyd et al. [37] and McGrath et al. [12] as they also indicate sub-optimality in the resource management protocols leveraged by cloud providers.

Optimising serverless scheduling seems to not be a high priority for the major cloud providers at the moment. Their focus is more towards language support and container-based optimisation for cold start delays [23, 38, 18]. Innovation may be slowing down due to a lack of external competition and serverless architectures not yet constituting a large portion of revenue. However, the demand for serverless computing is forecasted [39] to potentially triple over the coming five years. The academic community should perhaps strive to design a more optimal approach by then in order to facilitate the increase in traffic that is currently sub-optimally managed.

2.3.2 Academic improvements

Over the last two years, many papers have been published to discuss widely different views and interpretations on how to design a more optimal scheduling protocol. First, FnSched [40] attempts to minimise resources occupied while still achieving similar service quality by leveraging CPU-shares to regulate CPU usage and to reduce contention. Second, the Skedulix framework [41] exploits the interaction of two classes of programs with the CPU: IO-bound and CPU-bound. This distinction allows it to nearly double the system's capacity with little penalty, assuming a somewhat equal representation of both classes. Lastly, Mahmoudi et al. [36] leverage machine learning with similar baseline parameters as found in Skedulix: CPU time, CPU idle time, context switches, disk read time, … Additionally, parameters measuring load averages and percentage of memory in use are employed to create a neural network capable of context-aware global decision making over the entire service. There does not seem to be a significant research effort directly into optimising for network usage in the current literature.

To conclude, there are many possible approaches to improve the current state of the art in function placement. This is due to the wide variety in workloads a cloud provider can receive from its users, combined with the lack of initiative to improve for major cloud providers. Academic researchers have attempted to investigate many variations of possible improvements to the state-of-the-art algorithms, but there remains plenty of unexplored territory for further research. I intend to evaluate the influence of insights extracted from the workflow graphs and by prioritising the network as a parameter.

2.4 Integer linear programming problems

An integer linear programming (ILP) problem is an optimisation problem involving integer-valued decision variables. The values of these decision variables are tuned such that an objective function is either maximised or minimised. If some decision variables are real-valued ($\notin \mathbb{Z}$), the problem becomes a mixed-integer linear programming (MILP) problem. ILP optimisations find globally optimal solutions to the problem formulation [42]. The trade-off for this optimality is that integer programming is NP-complete and is therefore only practically usable for small input sizes [43]. An example ILP formulation in its canonical form is shown in Equation 2.1. Here, x is a vector of positive, integer-valued decision variables and $Ax \leq b$ signifies a set of constraints that are imposed upon x [44].

$$\begin{aligned} &\textbf{maximise} && c^T x \\ &\textbf{subject to} && Ax \leq b, \\ & && x \geq 0, \\ &\textbf{and} && x \in \mathbb{Z}^n \end{aligned} \tag{2.1}$$

The challenging aspect of leveraging the potential of integer programming lies in the precision of the formulation [45]. Any assumption, even implicit, can be exploited for a global optimum. Therefore, the formulation should be written with due deliberation. In the context of scheduling optimisation, the objective function should be concise and non-limiting as not to hinder potential solutions that are perhaps somewhat unintuitive [46]. Scheduling constraints usually include a spatial dimension concerning resource availability as well as a time dimension as a decided schedule at time t usually has an impact on other decisions that stretches Δt in time [45]. Gilpin et al. note that this can be covered in one or two constraints that check the global validity of interference between the different decision variables.

Although computationally inefficient for large inputs, formulating an optimisation problem with the ILP style can be used to formally explain what the intention behind the problem is.

This allows for a faster and less biased interpretation of the problem statement than can be achieved using a prosaic description. Aside from the formal definition explained in the previous chapter, the following chapter will introduce an ILP model for determining network-optimised placements.

2.5 Graph partitioning

Graphs are at the core of this dissertation's research. They are used to define the dependencies inside function compositions and the connectivity of servers inside a data centre. Features such as fan-in and fan-out for the input and output of a composition's functions are directly interpretable in the context of graph theory. They define the number of parents and children respectively for each node in the composition graph. Having constraints and layout specified as graphs means that it is possible to fall back onto well-known and well-documented graph algorithms for many aspects of composition analysis.

One graph algorithm that will be used in the following chapter is *graph partitioning*. Coming from the ILP formulation from the previous section, once problem size becomes reasonably large, the time and resources required to find an optimal solution scale uncontrollably. In these cases, a trade-off has to be made between resources and the optimality of the proposed solution. There are several heuristics available for the graph partitioning problem, as laid out by Verbelen et al. [47] and Li et al. [48]:

- Multilevel graph partitioning (MLKL)
- Simulated annealing
- Hybrid algorithm (KLSA)

Li et al. [48] describe multilevel graph partitioning as one of the most commonly used offline partitioning approaches. For practical use, there exist well-known software that is designed to perform graph partitioning like METIS [49, 50], KaHIP [51, 52], PaToH [53] and SCOTCH [54].

Practically, graph partitioning will be used in the next chapter to aid in setting up the network optimised scheduling approach that was formally described in the introduction. This scheduling approach fits extremely well with the concept of graph partitioning. If each partition is mapped to a single server, the cost that is optimised by graph partitioning algorithms resembles the cost of transferring data between these different servers. Therefore, it is helpful to have touched upon the state of the art graph partitioning solvers.

The specifics behind each of the proposed solution methods are not discussed in further detail, but it is now clear that there is a wide variety of efficient solution methods for partitioning graphs. However, the main issue in applying these implementations to this particular partitioning problem is that they focus on splitting the graph into relatively evenly sized partitions. However, when scheduling a composition onto a data centre, imbalanced partitions are not at all an issue but rather often the preferred solutions [55]. This is especially the case if the scheduler attempts to localise data as much as possible. Therefore, direct usage of the well-known software implementations listed above is not necessarily the best solution. A more customised approach that does allow for imbalanced partitions to occur would be beneficial for this particular use case. However, this is outside the scope of this dissertation.

"Where a calculator like ENIAC today is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1000 vacuum tubes and perhaps weigh only 1.5 tons."

~Popular Mechanics, March 1949

3

Conceptual study: scheduling approach

This chapter discusses the intricacies and considerations surrounding scheduling algorithms for serverless applications. An example composition is used to highlight the most important aspects of how placement decisions can influence the execution process of a composition. Additionally, it proposes the use of a graph partitioning heuristic to assist the scheduling process. Finally, a careful description of a simulation environment for scheduling approaches is discussed.

3.1 Parameters of the scheduling process

This section describes an example composition that should be scheduled. Doing so step by step allows the exploration of different options and considerations in a realistic environment. The composition in question can be depicted in the form of a DAG, as is shown in Figure 3.1. The graph visualises the dependencies between the nodes and provides hints for the parallelisability and potential bottlenecks without much effort. Over the course of this chapter, the term "composition graph" is used to reference the DAG description of a composition, more specifically its internal dependencies and general structure. The term "composition" on its own refers to a generic application, but not specifically to its structure. Initially, it is clear that there are multiple parallel paths in the middle of the graph and that nodes 8 and 10 are the most likely bottlenecks as they have a fan-in > 1 and therefore have to receive inputs from multiple other

nodes before they can execute.

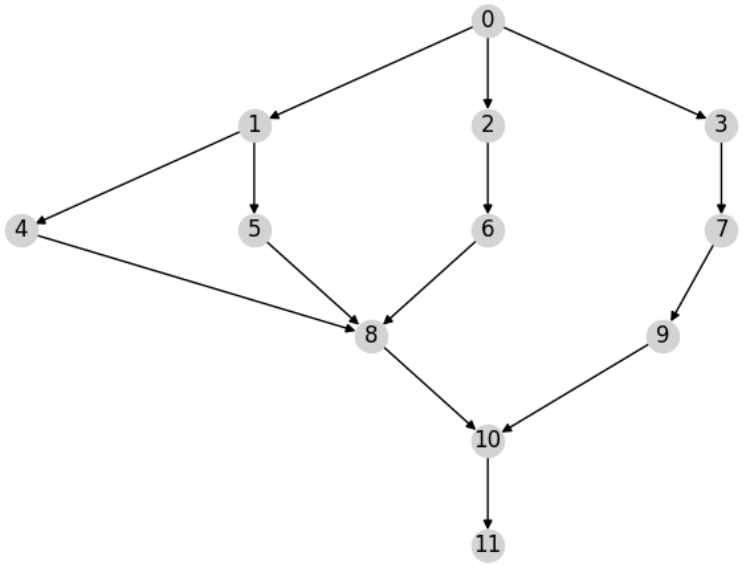


Figure 3.1: Example function composition graph to illustrate the reasoning and trade-offs that occur during scheduling. Numbers in each node are identifiers used to refer to a specific function in the graph.

Dependencies do not paint the whole picture, of course. Additional information is required to be able to schedule a graph. Each of these information sources will be explored in the following sections. Afterwards, they will be combined to illustrate the complete scheduling process.

- How long does each of the nodes in Figure 3.1 need to execute? Only with sufficient information about the timing of all the nodes in a composition graph can one make a well-informed and useful final scheduling decision.
- Next to the execution order dependency between nodes, there is also a data dependency: output from the parent node has to be transferred to its children. The size of these transfers combined with the network links they occur over provide another timing aspect necessary for proper scheduling. Additionally, it is also the main metric for evaluating the proposed heuristic for optimising the introduced network load.
- For which environment is the scheduling meant to be executed? What is the data centre layout? Depending on the number of servers, trade-offs in extra incurred delays need to be considered. The connectivity of servers (and switches) like discussed in Section 2.2 can vary significantly. The hop count between two communicating servers and the bandwidth of

the intermediate links directly influence the communication overhead encountered during execution.

3.1.1 Timing information

In order to decide upon the feasibility of a potential node-server allocation, server capacity needs to be able to fulfil the composition's needs during the entire execution. Checking this requires knowledge about the active time interval for each of the nodes in the underlying composition graph. The time t_{start} at which node i starts its execution is based on four contributing factors:

1. The time at which a node's parents have finished executing. The moment in time at which a node i finishes is denoted by $t_{end}(i)$.
2. The time it takes for each parent's output data to transfer to node i. Transfer time itself is dependent on the amount of output data (in MB) to transfer and the properties of the links connecting the source and destination servers. Assuming node j is a parent of node i, this timespan is denoted by $\Delta t_{trans}(j, i)$.
3. The possible occurrence of a cold start can add an additional waiting period $\Delta t_{cs}(i)$ before actual code execution can begin. The value of $\Delta t_{cs}(i)$ will be equal to 0 if there is no cold start delay. Even though warm starts also introduce some delay, this delay can be assumed to be included in the node's execution time by default without loss of generality.
4. The actual execution time $\Delta t_{exec}(i)$ of a node's source code. With the environment and parameters all present, the high-level source code usually needs somewhere in the order of milliseconds to finish execution.

Figure 3.2 displays the sizes of the data transfers and Table 3.1 contains the execution time in milliseconds for each node. All nodes have a single-digit millisecond execution time, ranging between 1ms and 7ms. The data transfers are split into two sizes, 5MB and 25MB, with the larger transfers all located on the rightmost path from top to bottom. These concrete values will further guide the concrete evaluation of different scheduling approaches.

Node i	0	1	2	3	4	5	6	7	8	9	10	11
$\Delta t_{exec}(i)$ [in ms]	2	3	4	6	1	2	3	4	5	3	7	2

Table 3.1: Table outlining the execution time $\Delta t_{exec}(i)$ (in milliseconds) for each node i in the example function composition graph from Figure 3.2

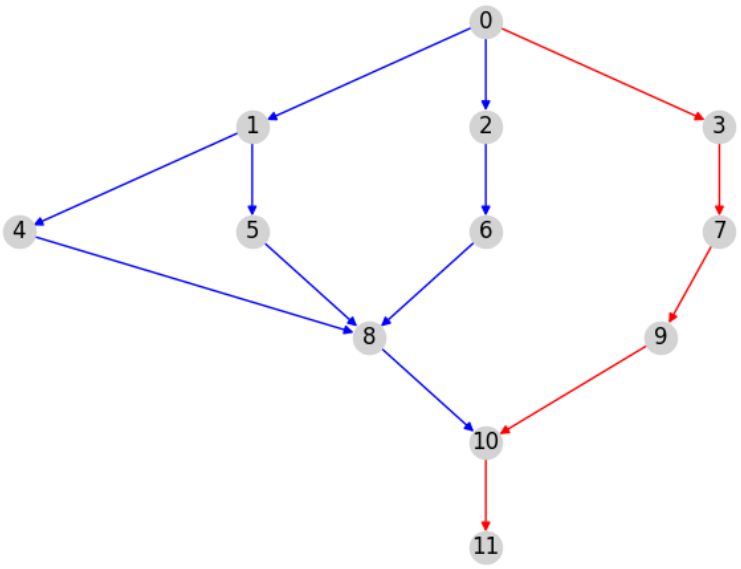


Figure 3.2: Example function composition graph with additional metadata for the sizes of transferred data between the different function. Red and blue arcs represent a transfer of 25MB and 5MB respectively.

As an example, when considering the available additional information, the start and end times for node 5 can be expressed using the following equations:

$$t_{start}(5) = t_{end}(1) + \Delta t_{trans}(1,5) + \Delta t_{cs}(5) \tag{3.1}$$

$$t_{end}(5) = t_{start}(5) + \Delta t_{exec}(5) \tag{3.2}$$

Given that the source node (node 0 in Figure 3.1) starts at a time t_{begin}, Equation 3.1 generalises to Equation 3.3 for any node i. The calculation of t_{end} does not change form.

$$\begin{cases} t_{start}(i) = \max\limits_{j \in parents} (t_{end}(j) + \Delta t_{trans}(j,i)) + \Delta t_{cs}(i) & \text{, if } i \neq 0 \\ t_{start}(0) = t_{begin} \end{cases} \tag{3.3}$$

$$t_{end,i} = t_{start,i} + \Delta t_{exec,i} \tag{3.4}$$

Before a concrete comparison between different scheduling approaches is possible, there is one more aspect that needs to be touched upon: the data centre layout. This will define the communication paths between the different servers and further concretise the timing value $\Delta t_{trans}(i,j)$ that was introduced earlier in this chapter.

3.1.2 Impact of server choice

There are several ways to model a data centre's layout for experimentation purposes. This topic was already discussed in Section 2.2. There are many possible types of underlying infrastructure such as fat-trees, DCells or optical solutions. This chapter will assume a data centre with a fixed, tree-based topology as these are the most prevalent [56]. A three-tier fat-tree layout will produce results that are significant for most of the current data centres [2].

The example data centre that will be used for the remainder of this explanation can be seen in Figure 3.3. It is a modest, highly simplified model of a data centre comprising of eight servers connected via two layers of switches. The switches are pairwise directly connected. For simplicity of the calculations, all links are modelled with a bandwidth of 80Gbps, which is on the high end compared to current data centres [2]. Each server can run two functions at the same time without interference. All eight servers are considered equal in terms of performance and computation power. This matches the uniformity of the functions' requirements in this composition. The execution times are, as laid out in Table 3.1 fixed for each function and not influenced by the choice of a particular server. This would needlessly complicate the guiding example.

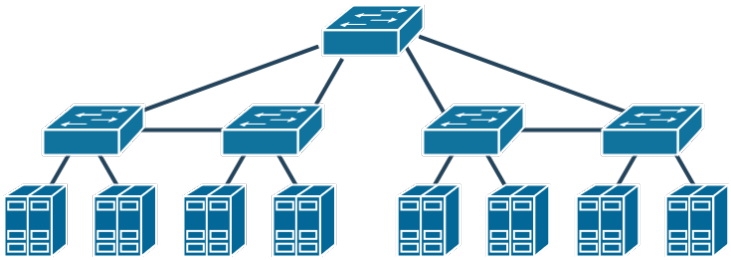


Figure 3.3: Simple data centre model for use in the example composition walk-through. A tiered, switch-centric data centre with eight servers. Level-1 switches are directly interconnected in pairs. All links have a bandwidth of 80Gbps.

3.2 Scheduling approach

With the required background information about the data centre layout and the nuances in timing for each node, the actual scheduling procedure can be explained in detail. The core optimisation objective used to pick a best-fitting schedule is still as was outlined on page 18: "*Minimise total time spent on data transfers over the data centre network during execution*".

To recap, each node of a function composition graph will be assigned a specific server to execute on. After execution, output data will transfer from parent nodes to their children. These transfers will introduce extra load on the network if the child node executes on a different server than its parent node. As a result, the goal behind scheduling a composition is to *group nodes together such that the time spent on transfers between the different groups is minimised.* These transfers are embedded in the function composition graph as arc weights. As was also proposed by Verbelen et al. [47], wording the problem in this way (minimising arc weights between groups of nodes in a graph) immediately puts forward the study of **graph partitioning** as an interesting field of heuristics to explore. Graph partitioning was already discussed in Section 2.5 on page 28. It is a fundamental problem in applications such as multiprocessor task allocation, circuit design and solving linear systems [57, 47]. Heuristics and approximations are often used to find a reasonable partitioning, as this is a known NP-hard problem [58].

In the following two sections, two solution methods will be discussed. First, an ILP formulation of the scheduling problem is presented and used to solve the example that was previously introduced. Afterwards, the process of using the graph partitioning heuristic and simulating an environment to evaluate this heuristic is discussed.

3.2.1 ILP-based scheduling

The topic of an ILP formulation as a means of finding an optimal solution bound by several constraints was already introduced in Section 2.4. This issue of optimising data transfers between nodes can be formulated as an ILP problem. However, it will not be a feasible approach for large-scale applications due to the exponential time complexity of ILP problems in terms of the graph size [43]. Problem sizes in the order of magnitude of the running example are small enough to be solved using an ILP solver such as IBM CPLEX [59]. Algorithm 1 describes the model in short. It is relatively straightforward to formulate the timing-based constraints to prevent server over-subscription for a composition consisting of functions with an execution time $\Delta t_{exec}(i)$ that is known ahead of time. These timings are used by the *overlap* method in Algorithm 1 to check if the execution intervals of two functions are overlapping or not. When combined with the $\gamma(i, j, k)$ to indicate a shared server k between functions i and j, invalid placement attempts can be filtered out. This ILP formulation is meant for deterministic compositions, real-life scenarios with nondeterministic delays and timings are outside its scope.

To illustrate the impact of input size on the time required to find the optimal solution, three variations of the example composition were optimised. The first variation looks at the scenario of scheduling a single composition. The second and third variations essentially do

Model 1: ILP model for network transfer optimisation

let F, S; // the number of functions and servers, respectively

let G; // directed composition graph, arc weight ω_{ij} indicates cost of data transfer

let *boolean* $x(i,k)$; // dec var: is function i mapped to server k?

let *boolean* $\gamma(i,j,k)$; // dec var: are functions (i,j) both mapped to same server k?

let *boolean* $z(i,j,k)$; // dec var: used for linearisation of x(i,k) * x(j,k)

maximise $\sum_{(i,j)\in G} \omega_{ij} * \sum_{k=1}^{S} \gamma(i,j,k)$; // maximise the reduced delay due to localised transfers

subject to

// Linearisation of x(i,k) * x(j,k) for next constraint

$\forall\ (i,j) \in [1,F]^2 : \forall k \in [1,S] : z(i,j,k) \leq x(i,k)$;

$\forall\ (i,j) \in [1,F]^2 : \forall k \in [1,S] : z(i,j,k) \leq x(j,k)$;

$\forall\ (i,j) \in [1,F]^2 : \forall k \in [1,S] : z(i,j,k) \geq x(i,k) + x(j,k) - 1$;

// $\gamma(i,j,k)$ is 0 unless both $x(i,k)$ and $x(j,k)$ are 1

$\forall\ (i,j) \in [1,F]^2 : \forall k \in [1,S] : \gamma(i,j,k) == z(i,j,k)$;

// each function is mapped to exactly one server

$\forall\ i \in [1,F] : \sum_{k=1}^{S} x(i,k) == 1$;

// Number of time-overlapping functions on a single server is limited

$\forall k \in [1,S] : \forall i \in [1,F] : \sum_{j=1}^{F} \text{overlap}(i,j) * \gamma(i,j,k) \leq \text{max\_concurency}(k)$;

and

// Boolean decision variables

$\forall\ i \in [1,F] : \forall k \in [1,S] : x(i,k) \in \{0,1\}$;

$\forall\ (i,j) \in [1,F]^2 : \forall k \in [1,S] : \gamma(i,j,k) \in \{0,1\}$;

$\forall\ (i,j) \in [1,F]^2 : \forall k \in [1,S] : z(i,j,k) \in \{0,1\}$;

Variation	Variables	Average time until completion
Single	1092	23.51s ±0.58s
Double	4200	1h42m16s ±7.94m
Triple	9324	> 6 hours

Table 3.2: Execution times measured for three different variations of Algorithm 1 in CPLEX Studio v20.1.0 with increasing numbers of decision variables.

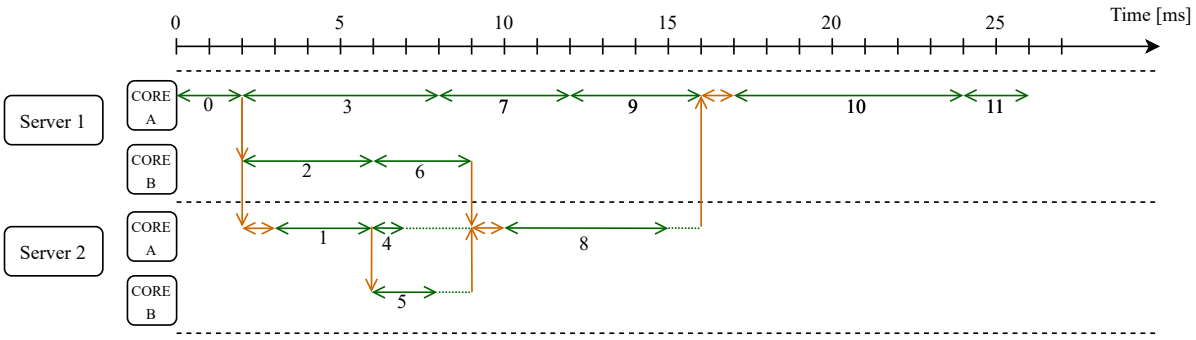


Figure 3.4: Placement of the composition in Figure 3.2 onto the data centre in Figure 3.3 according to the network-optimised approach. Green intervals resemble the execution of nodes in the composition, orange intervals resemble delays due to data transfers.

the same, but with two and three compositions simultaneously. Consequently, the composition graphs of the double and triple variant exist of two and three disconnected copies of the base composition graph. This small experiment was straightforward to set up using the model derived in Algorithm 1. It is not intended as a thorough benchmark of the solution time, but provides a simple indication of what is expected following the theory from Section 2.4. The results in Table 3.2 further illustrate the rapid increase in time that occurs for even slightly higher dimensions of decision variables. Therefore, it can be concluded that large-scale scheduling problems are not suited for ILP analysis in any case due to their more extensive scale.

For the default case of a single composition, the resulting optimisation solution is visualised in Figure 3.4. This figure illustrates the exchanges and successions between functions, each annotated with their ID, when scheduling on the data centre from Figure 3.3. In the same format, results for a spreading approach are placed in Figure 3.5. This is done to compare the network-optimised approach with a different approach that is easy to understand and concretely used (by Microsoft Azure [36]). For clarity purposes, cold starts have not been included in this solution model. Their impact will be discussed further in this section. The algorithm for spread-based scheduling is elaborated upon in Algorithm 2. The core idea is to greedily place all nodes such that the load on any given server is kept as low as possible. The tiebreaker mechanism that is used for multiple options is to prefer using a server that was already in use before to minimise

cold start delays, preferably the server on which the parent function was already executing.

Algorithm 2: Decision process behind a spreading scheduler

```
1  let f ← current_function;                    // function that has just finished executing
2  foreach child ∈ f.children do
3      if not prerequisites_completed(child) then
4          continue;
5      end
6      candidates ← get_servers_with_minimal_load();
7      function_placed ← false;
8      foreach candidate ∈ candidates do
9          if is_warm(candidate) then
               /* greedily pick first usable candidate server */
10             place_function(child, candidate);
11             function_placed ← true;
12             break;
13         end
14     end
15     if not function_placed then
16         leftover_pick ← candidates[0];   // grab first candidate if there are no warm servers
17         place_function(child, leftover_pick);
18     end
19 end
```

First, the prior expectations and final results for the network-optimised approach will be discussed. The trivial, yet ideal, scenario for deciding the final placement of nodes onto servers is to let one single server take care of the entire composition. This always reduces the transfer overhead to zero. The drawback of this trivial solution is that it deviates from the ideology behind serverless architectures, which is to fully embrace parallelism between small, independent tasks. If a server is capable of executing the different functions in parallel due to hardware support for concurrent execution, the available hardware limits the locality of the schedule.

In order to minimise transfer overhead, one would assume the red path from Figure 3.2 consisting of nodes 0, 3, 7, 9, 10 and 11 to be grouped together on a single server to avoid large back and forth data transfers. This projection is confirmed by the solution in Figure 3.4, as the first core of server 1 contains exactly this red path, forming the spine of the execution. As the blue part (nodes 1, 2, 4, 5, 6 and 8) of the composition graph is fully separated from the red path apart from the initial input and final output after node 8 it should be treated

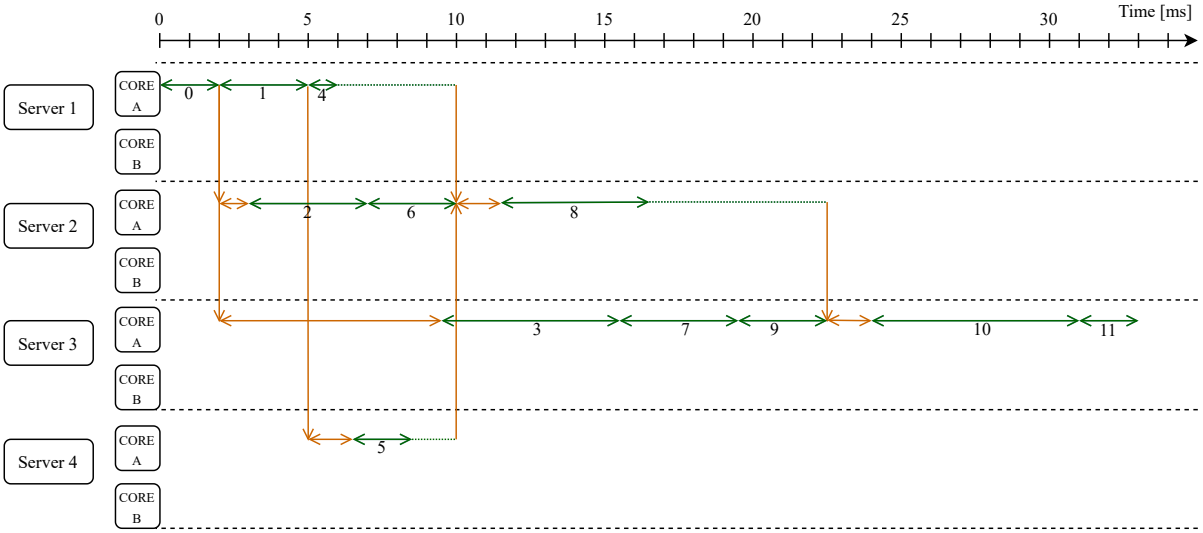


Figure 3.5: Placement of the composition in Figure 3.2 onto the data centre in Figure 3.3 according to a spreading approach. Green intervals resemble the execution of nodes in the composition, orange intervals resemble delays due to data transfers.

Scheduler	Number of transfers	Delay due to transfers [ms]	Transferred volume [MB]
network-optimised	4	3 ms	4*5MB = 20MB
spreading	6	13 ms	5*5MB + 1*25MB = 50MB

Table 3.3: Transfer delay and volume for two different scheduling approaches.

as a distinct subcomposition. The concurrency available on each server in this example data centre is two functions, which makes it impossible to place the entire composition on a single server. If it is impossible to have the entire composition on one server due to concurrency constraints, at least one additional servers should be employed. One additional server is enough is this particular case, but other composition whose graphs contain higher degrees of fan-out will require multiple additional servers. Additionally, when looking into the details for scheduling the blue subcomposition on the second server, it becomes clear that this single server cannot house the entire subcomposition either due to functions 2, 4 and 5 executing at the same time. This would cause a need for a third server, were it not for some leftover space on core B of the first server.

Figure 3.5 illustrates how the exact same scenario would have been scheduled under a spreading approach. There are clearly significant differences between this result and the one in Figure 3.4. First, the overall execution time has increased by 7ms compared to the network-optimised scheduling approach. Second, this scheduler could not prevent a large data transfer between server 1 and server 3. On top of it being one of the large transfers in the composition,

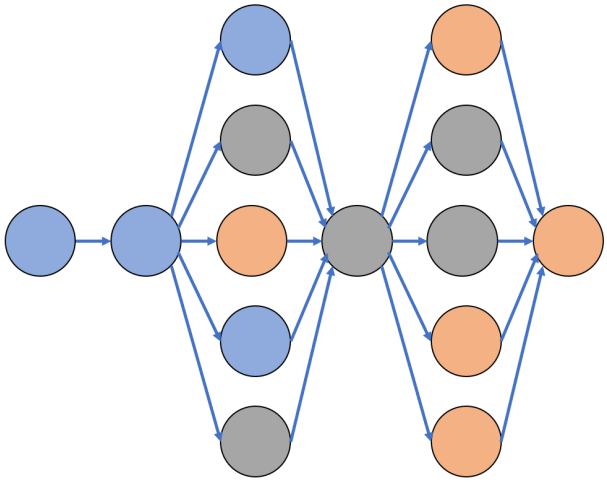


Figure 3.6: An edge case for balanced partitioning algorithms. Intuitively, it it clear that the majority of this composition should be allotted to one partition, with a significantly minor role for eventual extra partitions due to occasionally high fan-out.

the data is transferred between two servers that are not on an optimal transfer path. Without the intermediate pairwise linking of level 2 switches in the data centre, the impact would have been even more noticeable. Table 3.3 contains a comparison of network transfer-related metrics. It is clear that the network-optimised approach manages to reduce the incurred transfer delay significantly below the level of the spreading approach. This specific example composition was not designed to be a worst-case scenario for either approach. If the red path in Figure 3.2 was not a single chain but fanned out more, it is obvious that the spreading approach would have incurred much larger, more significant delays compared to a network-optimised scheduler. Due to the search for candidate servers with minimal load, a spreading scheduler also uses much more servers. This can cause issues with additional delays due to cold starts as well as with energy consumption. However, using more servers is not necessarily disastrous, as it could mean servers are pre-warmed for later invocations of other function compositions for example.

3.2.2 Partitioning-based scheduling

In the cases where ILP is too slow or cumbersome to use, we resort to heuristics to trade a degree of optimality for speed. Several algorithms and software packages that can be used for efficient graph partitioning were already mentioned in Section 2.5. Many widely different approaches are available, each with its own benefits and drawbacks. The main takeaway from the literature review is that all of these approaches look for a balanced partitioning, which is not necessarily desired in this specific case [55]. A chain of functions that occasionally fans out would ideally be scheduled mainly as one partition, with the occasional occurrence of a second or third partition at high fan-out. Such a scenario can be seen in Figure 3.6. If a balanced

partition is created, this edge case often produces where secondary and tertiary partitions are made quite large. In Figure 3.6, METIS was used to partition the illustrated graph for allocation on three servers. Each partition was assigned its own colour for visual clarity.

Sometimes, a partitioning can be very efficient in terms of data transfers between its different parts but not practically usable for scheduling. Each partition has to be mapped to one of many available servers, and an efficient partitioning that imposes a high load on a small number of servers may find the required capacity is not available. Partitions with fan-out that exceeds the concurrency limit of the target server will not be useful. Due to delays as a result of cold starts or data transfers, *many aspects of the final timing of the function composition graph are only determined after a suitable mapping has been picked.* The procedure for evaluating possible partitionings is described in Algorithm 3. This procedure uses the composition metadata together with a global context that contains the load on different servers from currently active compositions to decide on the ideal placement for each function in the composition using graph partitioning. It searches for the minimal number of partitions that yields a server mapping that is valid in the global context, i.e. there are enough servers with a sufficient amounts of resources available. The *GetOptimalParts* method on line 4 is a generic version of the implementations discussed in Section 2.5. The principle of the algorithm does not change depending on the choice of a particular partitioning method, as they all support a similar interface. As was mentioned previously, the choice of a particular implementation is worthy of an entire research effort on its own and is out of scope for this dissertation. Preferably, the chosen implementation provides multiple optimal partitionings, as not every optimally weighed partition yields a usable, globally valid server mapping. This is also indicated in Algorithm 3.

Algorithm 3: Choosing the most suitable graph partitioning

```
1  let compositionToSchedule;
2  let globalCtx;                    // global context that contains all current compositions
3  for nServers ← 2 to nTotalServers do
4      possiblePartitionings ← GetOptimalParts(compositionToSchedule, nServers);
5      foreach partitioning ∈ possiblePartitionings do
6          isValid ←process(partitioning, globalCtx);
7          if isValid then
8              return;
9          end
10     end
11 end
```

Algorithm 4: Detail view of the *process* method in Algorithm 3

```
1  let partitioning;        // Given a partitioning into n parts of a function composition graph
2  let allServers;                                  // Given the set of all available servers
3  let globalCtx;                                      // Given the global execution context
4  let currentOptimalOverhead;
5
6  foreach set_of_n_servers ∈ allServers^n do
7      overhead ← Σ_{i=0}^{n} Σ_{j=0,j≠i}^{n} transfer_time(server_i, server_j);   // inverse bandwidth
8      if viable choice of servers then
9          currentOptimalOverhead ← overhead;   // Only after formally deciding the mapping
           between partitions and servers and analysing timing details can the viability of the
           mapping be verified
10     end
11     if overhead < currentOptimalOverhead then
12         currentOptimalOverhead ← overhead;
13     end
14 end
15 return currentOptimalOverhead;
```

Finding the candidate partitionings can be done fast and efficiently using one of the algorithms and software packages discussed in Section 2.5. The computationally most intensive part of this heuristic scheduling approach is found in line 6 of Algorithm 3. The *process* method, which is expanded upon in Algorithm 4, handles the mapping of partitions to actual servers. Given a partitioning into n parts, a set of n servers needs to be selected such that there is a one-to-one mapping from partition to server. Ideally, these n servers are chosen such that they are "close" to one another in the underlying data centre. Close in this context means that the time to transfer data $\Delta t_{trans}(i, j)$ between two servers i and j is as low as possible. Fast transfers can be achieved through low hop count between servers (perhaps a direct connection), high bandwidth links on the path from i to j, or a combination of both. In order to be sure an optimal choice of servers is made, the procedure has to look at all n-tuples in the n-fold Cartesian product over the set of all servers. For each of these n-tuples, the transfer overhead is derived using the formula discussed at the end of Chapter 1 based on the inverse bandwidth of the links connecting the source and destination server. Algorithm 4 loops over all n-tuples to figure out the set of n servers that provides minimal transfer overhead.

For a data centre consisting of K servers the number of n-tuples becomes $\binom{K}{n} = \frac{K!}{n!(K-n!)}$. Checking all n-tuples in the n-fold Cartesian product becomes infeasible for even slightly larger

data centres. For example, in a data centre consisting of 1024 servers, $\binom{1024}{2} = 523776$ and $\binom{1024}{4} = 45545029376$. However, his computationally expensive check can be reduced by only considering n-tuples of close servers. Additionally, greedily opting for a first-fit search instead of checking all relevant n-tuples can speed up the search.

The ideal way to evaluate a heuristic like graph partitioning for this particular use case would be by integrating the necessary algorithms into real-world usage. The methodology behind this is discussed in the next chapter. Here, focus will be on the alternative that is best suited for a heuristic methodology in development: simulation. By reducing the problem down to its most basic form and stripping it of unnecessary complications, a very simple Python program can simulate the scheduling process and evaluate the interesting metrics concerning the heuristic in play. Although simple to understand, the design and implementation of such a simulation environment can be tricky to get right. For example, deciding on what is relevant to the simulation and what is not, or addressing the integration of a metric that was originally not intended.

The metrics that were used to compare the two solutions to the example composition above are the most interesting ones to extract from a simulation of a network-optimising scheduler. These metrics are (1) execution time (of the total composition), (2) amount of data transferred between different servers and (3) resource usage. However, the raw value (in ms) of total execution time is not immensely useful. This is due to cold start delays being several orders of magnitude higher can most functions' execution time. For example, cold starts can range between 100ms to more than a second to complete while functions typically span a couple of milliseconds [23]. It is more valuable to report the execution time as fine-grained as possible, with its contributing factors (cold starts, transfer delays, setup times, raw execution time) separated. Resource usage can be interpreted in many different ways. As the number of servers used, cores used, memory used all either as a total value or as a graph over time. The behaviour over time shows how spiky or constant this load is. This metric will not make or break a scheduling approach, but can be used to make more contextual judgement calls when evaluating all metrics together across different approaches. Additionally, the behaviour over time provides an indication about the scalability of a scheduler with regards to usage in a real-world setting.

Many parameters of the simulation environment are still to be tuned. Apart from the scheduling logic that is evaluated, there is the choice of data centre structure, server properties and the different workload that can be run. Concerning the simulated data centre layout, a 3-tier fat-tree data centre with 1024 servers is significantly sizeable to experiment with larger workloads and to provide varying levels of connectivity between the many different servers. The lowest tier switch in this data centre connects 16 servers. The two tiers upwards connect eight switches

into one, with a pairwise direct link like in the example data centre in Figure 3.3. Second, Server properties in the most basic sense include both a limit on usable memory, as well as an additional concurrency limit. A server with enough memory for 100 functions might not have the hardware to execute them all in parallel. Concurrency limits usually match the core count of the underlying CPU which can range from 4 - 6 up to 48 in current data centres. Each server holds a couple of GBs of memory, as memory is quite cheap to come by. Lastly, there are infinite possibilities of workloads to choose from. Depending on the scheduler being general-purpose or scientific-oriented, these can vary from many, very small compositions to a few highly complex compositions.

The example composition in Figure 3.2 is a good example of a small composition of which thousand can be run at the same time in a simulation. Scientific calculations are much more involved and complex to accurately describe as a sequence of individual tasks. Most algorithms are not designed and implemented specifically for serverless execution and have to be reinterpreted to fit the definition scheme of a generic composition. One such algorithm is QR decomposition. This algorithm splits a matrix A into the product of an orthogonal matrix and an upper-triangular matrix. Matrix manipulation is often used to illustrate efficient and parallel data usage, and this is no exception. Anderson et al. [3] created a reinterpretation of QR decomposition specifically for use on a GPU: communication-avoiding QR decomposition (CAQR). To accommodate for the highly parallelised nature of GPU computing, it they tried to limit the “memory traffic”. Their visual description of the different steps in the algorithm makes it possible to create a composition of functions that pass the matrix data in the same way as CAQR would. The different steps used in the CAQR paper are shown in Figure 3.7. Writing the logic for these functions to perform the actual algorithm requires a much deeper insight into the inner workings of CAQR, and is not required for simulation. In a simulation, the most important aspects of the workload is the function composition graph, the sizes of input and outputs and the execution time for each function. If a reasonable estimate of execution time per type of function can be made, this is sufficient for useful simulation results.

3.3 Final remarks

This concludes the chapter of this dissertation about the intricacies and considerations surrounding scheduling algorithms for serverless applications. I have presented a thorough walkthrough of an example composition, used to highlight the most important aspects of how placement decisions influence different aspects of the timing between the functions inside a compositions and vice versa. Additionally, a formal procedure was outlined for using graph partitioning algorithms to feed the decision making process of mapping each function to a respective server for

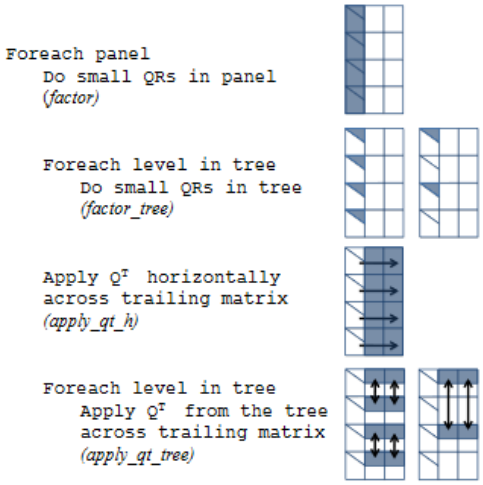


Figure 3.7: Data transfer patterns in communication-avoiding QR decomposition of a matrix [3].

execution. Finally, a careful description of a simulation environment for scheduling approaches was discussed.

There is a lot more that can be added to the research in this chapter. For example, it was mentioned that most popular graph partitioning algorithms currently in use try to find balanced solutions. For this particular use case this is not obligatory, and possibly even unwanted. Another heuristic approach that might be suited for additional research is *community detection*. Another addition that will likely prove interesting is to add the concept of "slack time" into the execution of a composition. Up until this point, functions were always assumed to start as soon as their prerequisites were fulfilled. However, in many situations waiting an additional 1-5ms can open the door for much more efficient scheduling. The major downside of introducing slack time is that is significantly complicates all solution and evaluation methods: many extra factors of decision variables in ILP formulations, an increased time spent verifying and evaluating possible server-to-partition mappings in an already large search space.

It has to always be taken into account that the more complex the scheduling logic becomes, the more difficult it is to use on a large scale. This is what will be further discussed in the next chapter about the feasibility of implementing custom scheduler logic in a widely used open-source serverless platform.

"Discontinued products and services are nothing new, of course, but what is new with the coming of the cloud is the discontinuation of services to which people have entrusted a lot of personal or otherwise important data — and in many cases devoted a lot of time to creating and organizing that data. As businesses ratchet up their use of cloud services, they're going to struggle with similar problems, sometimes on a much greater scale. I don't see any way around this — it's the price we pay for the convenience of centralized apps and databases — but it's worth keeping in mind that in the cloud we're all guinea pigs, and that means we're all dispensable. Caveat cloudster."

~Nicholas Carr, author of books about technology and culture

4

Practical feasibility study: Apache Openwhisk

Aside from the theoretical insights that can be gained from studying the properties and intricacies surrounding a newly proposed scheduling algorithm, it can be equally valuable to look into a practical approach. For example, the adaptation of a commercially well-known product and investigating whether it provides the necessary features to change the scheduling approach. This chapter dives deeper into the inner workings of one of the most popular open-source serverless frameworks available: Apache Openwhisk, due to its usage in IBM's FaaS platform IBM Cloud Functions.

The newly proposed scheduling algorithm is to minimise the delay incurred due to data transfer between different actions that make up a composition. Having a smaller transfer overhead lightens the load imposed on the back-end architecture and allows more users to use the provided service with the same hardware simultaneously.

The main goal of this chapter is to dive deep into the practical aspects of modifying the source code of the Openwhisk scheduling behaviour. Potential roadblocks encountered while modifying the relatively simple default behaviour will allow for a feasibility study of integrating quite complex logic in one of the market leaders [60] in open-source cloud function orchestration. Additionally, the current literature mainly focuses on the discussion of theoretical aspects while

skipping over the details of modifying existing platforms for testing purposes. Architecturally these platforms are usually well-documented, but the process of modifying internal algorithms can be somewhat obscure. This chapter will make it easier for new researchers to deploy their custom scheduling behaviour quickly.

It should also be noted that the behaviour in Openwhisk is subject to change in the future. Contributors of the project have recently published proposals that call for a redesign of (parts of) the architecture and scheduling process to increase performance in some of the current "worst-case" scenarios. These proposals are publicly available here: [61].

This chapter starts by describing the inner workings and design of Openwhisk as required to grasp the modification process fully. Afterwards, the reasoning behind a couple of workarounds and unorthodox tricks is illustrated. Additionally, there is a section dedicated to describing an ideal cloud platform, detailing the shortcomings of some Openwhisk design decisions that can make it less suitable for general experimentation. The end of this chapter presents the implications of the most recent efforts by the Openwhisk developers to upgrade their current algorithms and architecture.

4.1 Framework introduction

Openwhisk is an effort by the Apache Software Foundation to provide the world with a completely open-source serverless platform. It was created by IBM and subsequently donated to the Apache Software Foundation for future maintenance. The IBM Cloud Functions platform still runs on top of Openwhisk. All dependencies are also open-source projects such as Kafka, CouchDB, Consul, Docker, nginx, redis and Akka.

As is often the case with platforms and frameworks, Openwhisk uses specific terminology to describe certain features or collections of information that require the proper context in order to avoid confusion. Below is a list of terminology that occurs during this chapter.

- **function:** A cloud function is the smallest execution unit supported by Openwhisk. They do not occur on their own but always encapsulated in an action. A function should always be stateless as there is no guarantee that state will be maintained across invocations. Functions always execute in a sandboxed environment and are provided as source code in a high-level programming language.

- **action:** On top of wrapping the small, stateless cloud functions, an action keeps track of additional metadata corresponding to that function. This includes runtime environment information, version number, function parameters, and system limits that are enforced at the action level such as logging, timeout and memory requirements.
- **namespace:** A mechanism for grouping together actions and triggers belonging to particular a user. Subdivisions within a namespace are possible, for example, to distinguish production and development actions. Namespaces are mostly used for bookkeeping, such as imposing namespace-level rate-limiting and concurrency rules.
- **invoker:** Openwhisk abstraction of a server node. The invoker is responsible for managing the execution of a particular function that is assigned to it, as well as preparing the environment required for execution.
- **composition:** A sequence of functions with a data dependency between them. Output from parent functions is passed as input to its children, usually in the form of mutable state. Compositions are often modelled as a directed graph.

4.2 Openwhisk Architecture

The core of Openwhisk is an event-driven architecture [60, 62] with lots of components all designed to each serve one particular purpose, such as routing, controllers, invokers, authentication, messaging or logging. A high-level overview of the architecture is shown in Figure 4.1. Every component is a complete, fully customisable software package on its own. It can perfectly be swapped by any other customised service as long as the original interface contract remains intact. It is also noteworthy that, in a production environment, many components like the controller will have several replicas deployed at the same time [63]. Only a select number of components do not support multiple replicas: the API gateway, redis, CouchDB and Kafka.

4.2.1 High-level communication chain

This section describes each of the different high-level steps involved in processing a request to execute an Openwhisk function based on the official code repository [4]. The official documentation on the "System Overview" contains substantially more detailed information that is out of scope for this chapter concerning rules, triggers, actions and integrations with additional services.

The first point of contact the architecture has with an incoming request is the reverse proxy (nginx). A reverse proxy separates the communication inside Openwhisk from the public internet

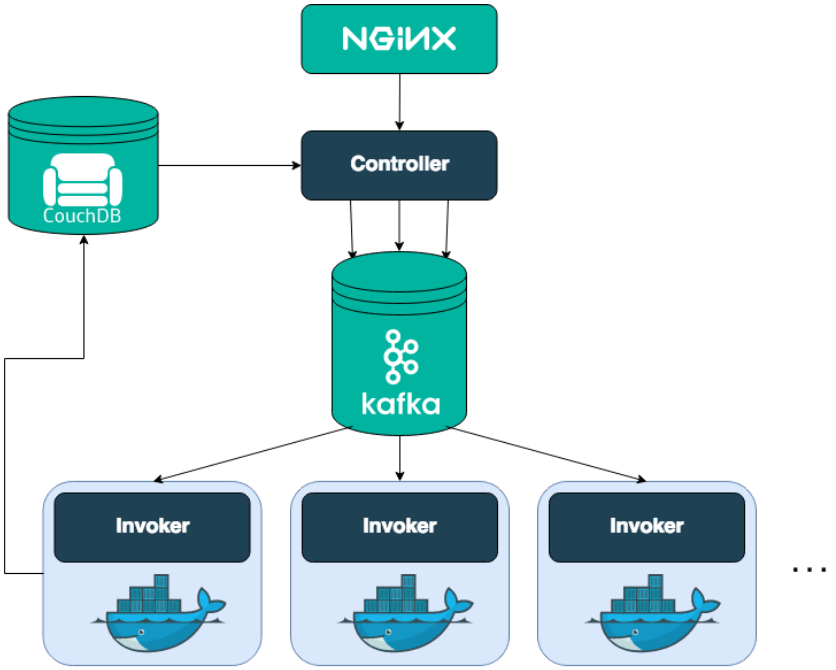


Figure 4.1: High-level overview of the Openwhisk architecture. It visualises the different interactions between the most important components during a request's lifetime, starting from the initial HTTP request through nginx until the actual code execution in a Docker container and result storage for future reference [4].

and forwards the incoming requests to the appropriate destination in the private Openwhisk network. As described by Cloudflare [64], a reverse proxy can serve as the endpoint for secure incoming connections (e.g. using TLS), perform load balancing and caching operations. No Openwhisk-specific actions occur at this point in the request life cycle.

Afterwards, the reverse proxy passes the request to one of the available controller instances. The controller is the central caretaker of this request: it will start gathering the extra information from inside the system. It is the controller's responsibility to make sure its decisions satisfy the request's requirements. A request could involve uploading the code for a new function, a change in event triggers or the start of (a chain of) function invocations. The first step a controller performs is to authenticate the request and verify the privileges associated with the user's role against the CouchDB database. After verifying any claims, CouchDB is contacted again to retrieve the relevant *action* that contains all of the metadata like function code, parameters, required available memory to execute and rate-limiting. The next step is combining all this metadata into a final decision which invoker the request will be assigned to. The details of this procedure are outlined in Section 4.3.

Now that a destination invoker has been decided, the controller will publish an event to Kafka, which acts as a buffer between the controller instances on one side and the invokers on the other side. The Kafka topic used is the one associated with the chosen invoker instance. This event contains the code that has to be executed together with the relevant parameters, which will be. Additionally, an ID is generated to associate this event with the future result such that the end-user can query the result from CouchDB.

Finally, the actual code execution occurs in an encapsulated, controlled and safely configured Docker environment. Here, potential cold start penalties occur, waiting for the correct runtime to initialise. After completion, the result of the function code is retrieved from the container and stored back into CouchDB together with the ID that the controller previously generated. Openwhisk invocations are asynchronous by design, which means the result is not sent to the end-user directly. Therefore, storing the result in CouchDB marks the end of the request life cycle.

The following section discusses in-depth the working and relevant intricacies of the controller component. This component houses the scheduling logic, and therefore the modifying efforts will take place here.

```
const composer = require('openwhisk-composer')

const l1 = composer.action(
    'l1', { action: () => { return {"left_state": 1} } }
);
const l2 = composer.action(
    'l2', { action:  (state) => { state["left_state"] += 7; return state } }
);
const left_branch = composer.sequence(l1, l2);

const right_branch = composer.action(
    'right_branch', { action: function () { return {"right_state": 2} } }
);

module.exports = composer.if(
    composer.action(
        'decision', { action: ({pw}) => { return { value: pw === 'abc123' } } }
    ),
    composer.parallel(left_branch, right_branch),
    composer.action(
        'failure', { action: function () { return { message: 'failure' } } })
    )
```

Listing 4.1: JavaScript composition description for Openwhisk. This composition validates a *password* parameter. If the right password is provided, two branches are executed in parallel. Afterwards, the output of both branches is combined into a single global output object. Adapted from the official demo samples [66].

4.2.2 Under the hood: subcompositions and secondary activations

Creating a composition of functions is supported by Openwhisk through the use of *conductor actions* [65]. There is also the option of simply sequencing actions together, but this limits the flexibility of the composition as there is no support for (1) parallel execution of different actions in the sequence and (2) control flow functions for loops or branches. Simple sequences can be useful in some scenarios where a predefined sequence of actions needs to be executed, but for a general workload, more flexibility is required.

Openwhisk provides a *composer* to describe compositions as code such as in Listing 4.1. These can then be translated to the less readable conductor description format and subsequently uploaded to a running Openwhisk instance. The main features of the composer are listed below. An Openwhisk composition consists of a starting action, followed by all simple actions

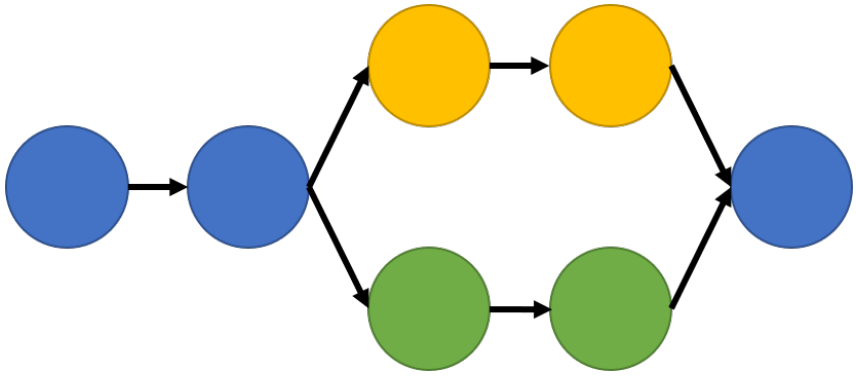


Figure 4.2: Example composition with three subcompositions in total, each marked with a different colour.

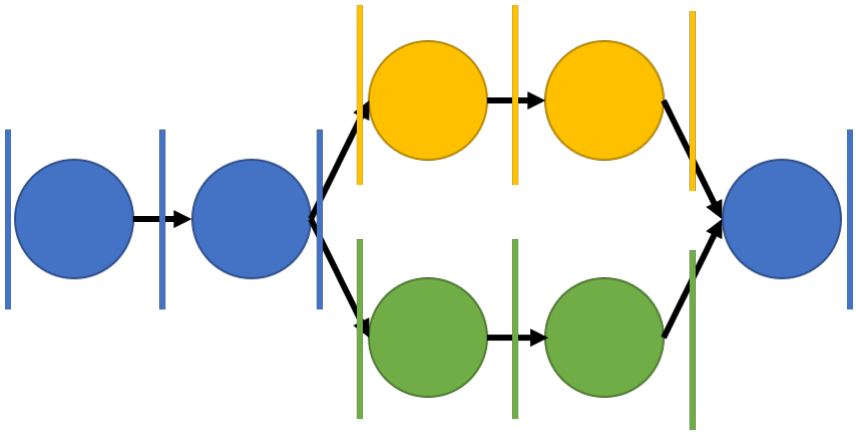


Figure 4.3: Ordering of actions and secondary activations for an example composition in Openwhisk. Each subcomposition of n actions requires $n + 1$ secondary activations.

that will be executed as one sequence with 100% certainty. Actions that cannot satisfy this requirement are grouped into subcompositions. Figure 4.2 shows a simple composition with each (sub)composition in a different colour. The split into two separate parallel sequences causes each branch to become a proper subcomposition. These subcompositions are treated just like a regular composition from the perspective of Openwhisk, except that they are kickstarted from inside another composition rather than by an external trigger.

- simple actions
- branching actions: if-then-else construct
- control flow actions: repeat and while
- parallel subcompositions of actions

Additionally, Openwhisk adds a second layer of functions to each composition. These *secondary activations* serve an administrative purpose and are not inserted into the composition by the end-user. They are responsible for orchestrating the different invocations in the correct order: deciding on the next action to execute, early stopping in case of an error. Consequently, secondary activations run before and after each action. Whenever possible, the before and after secondaries are combined into one to reduce overhead. As a result, a composition of n actions requires an additional $n + 1$ secondary activations. Figure 4.3 shows the interleaving of actions and secondary activations. As mentioned before, subcompositions are treated as fully separate, autonomous compositions with their own entry point and finalisation.

4.3 Dissection of the controller component

This section will dive deeper into the source code and the design features that need to be considered when modifying Openwhisk internals. The approaches presented here present my findings during my investigation into the possible ways to approach a scheduler rewrite. They highlight the major issues with Openwhisk in terms of modifiability and support for function compositions.

The default implementation of the controller logic can be found in `core/controller/src/main/scala/org/apache/openwhisk/core/loadBalancer/ShardingContainerPoolBalancer.scala` inside the official Openwhisk repository [67]. Customisation of the logic can be achieved by either editing this file, or by creating another LoadBalancer class and reconfiguring the loadbalancer spi source during Openwhisk setup in `ansible/group_vars/all`. The two main functions in `ShardingContainerPoolBalancer.scala` that need to be adapted are *publish* and *schedule*. The former is used to provide the necessary parameters to the latter, which is where the scheduling logic to pick a well-suited invoker for the current action is located.

The default load balancing algorithm used in Openwhisk is a simple, stateless sharding algorithm. At its core, there are many similarities with how a hash table determines the placement of elements. The load balancer calculates a hash of the namespace-action pair to derive the index of an invoker. This first invoker is called the *home-invoker* and is the same for every invocation of a specific serverless function. Several constraints are checked before assigning the function to the home-invoker, such as memory capacity. If any constraint cannot be satisfied, a step increment is added to the index of the current invoker, followed by a new attempt to find a destination for the invoked function. In order to reduce the number of possible collisions, this increment is picked out of the set of coprime numbers smaller than the number of invokers available.

4.3.1 Stateful controller

Adapting this stateless default algorithm into a smarter, stateful approach requires the controller to keep track of specific events and parameters during its execution. Adding state is not a problem, as there is already some state present by default, such as the number of invokers. Consequently, the core issue in the modification process will be getting hold of the right information and passing it to the controller logic. Once the controller has access to information, it is trivial to add this information to its existing state.

The introduction of state into one of Openwhisk's core components breaks some assumptions about the Openwhisk architecture. Stateless controller instances can be replicated indefinitely, but if the controllers start accumulating state on which they will base future decisions, this can become an issue. Either all actions dependent on a subset of information for their scheduling have to be passed to the one controller instance managing it, or controllers have to communicate their state. A third option could be to disallow controller replication, similar to how it is discouraged for the gateway and CouchDB. In a document on design considerations [68], it is proposed that, due to the short-lived nature of cloud functions and a resulting fast-changing state, the time to aggregate the required information would become too large. Therefore, it is from now on assumed that there is only one controller instance that handles all requests meaning it does not have to communicate its internal state with others. Handling the increase in state is the first major issue in the practical feasibility of the proposed stateful algorithm.

Another major roadblock that prevents a straightforward modification process is the action-centric design of Openwhisk. Even though Openwhisk supports the invocation of chained actions as larger compositions, each action in a composition is still handled individually when its pre-conditions are met and is required to execute. If an action is part of a composition, additional metadata and parameters are provided. The question now becomes: "given the action-centric design of Openwhisk, is it possible for a stateful controller to still gather enough information to understand the required context for scheduling every action in a generic composition?" The answer to this question is *almost, but no.*

4.3.2 Parent-child dynamic

Openwhisk provides one piece of composition metadata by default: *cause* information. This optional field contains the ID of the root secondary activation that started the composition. Linking back to Figure 4.3, the colours indicate a common cause field. Notice that this means the different coloured parts of this composition have no link between them from the perspective

of the available metadata. One way we can attempt to tackle the issue of fully separated subcompositions is by thoroughly investigating the internal data structures of Openwhisk. One should assume the Openwhisk architecture can be lightly modified to funnel more information towards the controller. Workarounds that require drastic changes to the Openwhisk internals will not be explored. This would deviate from the initial goal of studying the feasibility of a more intelligent scheduling approach *in Openwhisk*.

An initial approach would be to assume there is internal bookkeeping information readily available behind the scenes. However, looking into the parameters provided to the secondary activations, it seems the internal bookkeeping is not organised in a way that makes extracting parent-child relationships between concrete actions easy. Changing these structures is technically possible but would redefine several internal interfaces and be very error-prone. Deeper insights were required to assess the boundaries of default Openwhisk.

```
1 [<timestamp>] [INFO] [#tid_DuOhNqWQ...] [<component>] <message>
2 [<timestamp>] [INFO] [#tid_LMHgpKJG...][#tid_428pMJt7...] [<component>] <message>
```

Listing 4.2: Highlighted difference of Openwhisk log statements for an action that is not part of a composition versus an action that is.

The search for additional metadata involved a lot of debugging and reading logs. After a while, I noticed a potentially useful pattern in the logs. They are structured to clearly define the component that created the log statement using square brackets. The log contents are fairly straightforward entries such as timestamps, log severity level, and component name. One field in the logs called "*tid*" is not intuitively clear. This turns out to be a transaction ID, purely intended to make logs more useful. The original documentation defines it as "a transaction id intended for logging". Interestingly there is a small difference between the logs produced by an action that is not part of a composition compared to an action that is: a composition's actions produce logs with *two* transaction IDs. This difference is highlighted in Listing 4.2.

The Openwhisk team may not have designed their transaction IDs for this use case, but there is an intriguing property that follows from the way the creation of these values is structured. The inner (rightmost) transaction ID is tied to one specific action instance, and **the outer (leftmost) ID is equal to the inner ID of the action that started the current (sub)composition.** Additionally, it is trivial to add these transaction IDs as input to the scheduler: the inner ID is already part of the interface, and the outer ID is used to create the inner ID immediately before the scheduler invocation. Therefore it can easily be added as

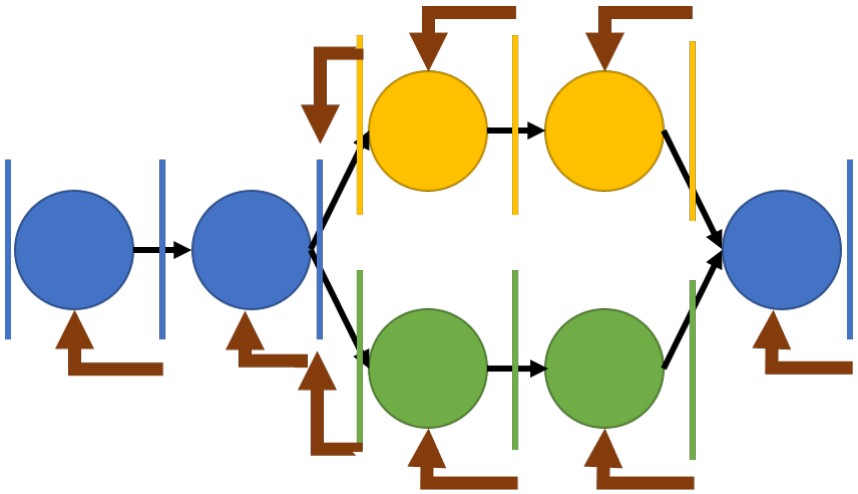
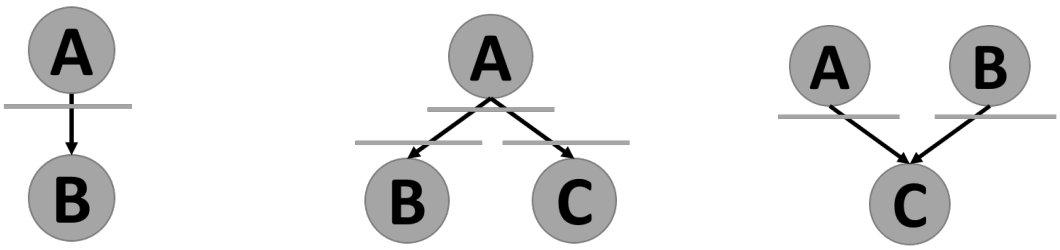


Figure 4.4: Additional relationships in an example composition that can be inferred by keeping track of inner and outer transaction IDs during scheduling passes. At the point where the blue chain branches into a yellow and green chain, there is now a newly formed information link across the subcomposition boundaries.

an optional metadata field to the existing parameters passed to the scheduler. Therefore, this property is ideal for providing crucial insight across subcomposition boundaries required for an intelligent scheduling algorithm. The new possibilities this provides to a composition to query historical information are visualised in Figure 4.4.

To fully understand the reason it is not possible to extract all necessary information from the available action metadata, a deeper understanding of the different possible parent-child dynamics is needed. There are three ways to depend on a parent as a child: either as an only child, as one child of multiple parents or as one of many children of a parent node. These options are visualised in Figure 4.5. In each of these cases, the view from the perspective of a controller



(a) One-to-one transition (b) Multi fan-out transition (c) Multi fan-in transition

Figure 4.5: Possible parent-child transitions with secondary activations between actions indicated as horizontal lines.

instance will be discussed.

- **One-to-one:** Actions A and B have a matching *cause* field. All information about A is available by looking up the previous schedule pass for this common cause field.
- **Multi fan-out:** Previously a roadblock, but as indicated in Figure 4.4, there is a new information link from B and C to A via their secondary invocations' outer transaction ID that crosses the subcomposition boundaries. By keeping track of the transaction IDs passing by during scheduling, the required information can easily be retrieved.
- **Multi fan-in:** This case remains unsolvable even with the transaction ID workaround. The issue holding back progress in this specific situation is the absence of any multi-parent information in Openwhisk. One would naively expect the fan-out and fan-in cases to be highly symmetrical. However, Figure 4.3 already illustrated that the secondary activations come just after (not before) an action.

One could naively assume that multi fan-in information propagation can be solved by implementing another layer of workarounds similar to leveraging the transaction ids. In fact, it remains possible to make an educated guess which historical information is relevant for some compositions. However, contrary to the workaround with transaction ids, but these guesses will not be applicable in every situation. Another solution to explore might be to uniquely name every occurrence of a particular action depending on its position in the composition. Even so, this will hinder the composition definition as many reused source code fragments will become duplicated in the naming process. This will negatively impact both storage of the definition as well as hinder a cached transfer system for initialising containers with the correct environment and source code.

To conclude this section, there are two issues with the current version of Openwhisk that hinder the implementation of more complex scheduling algorithms. First, these algorithms require an increasing amount of state in the controller component, which severely limits the parallelisability of the entire architecture. Second, a multi fan-in dependency between an action and its parents is not inversely resolvable without introducing other architectural trade-offs. The logical conclusion from these issues is that Openwhisk, in its current form, is not suitable for complex scheduling algorithms. However, thoughtfully crafted changes to the Openwhisk architecture might prove it more suitable in the future. As the development process is entirely public [69], it might be interesting to evaluate the direction the project is moving in before making any final conclusions. The following section will discuss the practical steps to recreate the network-optimised scheduling approach in Openwhisk, followed by a final review of the upcoming architectural design changes.

4.4 Network-optimised scheduling in Openwhisk

The previous section discussed adapting scheduling at a high level. Even though the context was mostly practical, the discussion involved only a few practical pointers to explicit Openwhisk code adaptations. First, this section examines structural changes to the original approach explained in Chapter 3 to fit into the Openwhisk model. Afterwards, the data structures that are required to keep track of relevant state are shown as a recap of the previous sections' findings.

Due to the action-centric scheduling design in Openwhisk, schedulers are pushed towards greedy algorithms. It is theoretically possible to implement the regular optimisation algorithm using the heuristics described in the previous chapter and run it once the first action of a composition is being scheduled. The preferred placements for each of the next actions could be stored in the controller state somewhere for future reference. However, this violates one of the initial principles of this chapter to not forcibly introduce a foreign design rather than focusing on slight adaptation. Restructuring the scheduling approach in a greedy way to more fit the Openwhisk mindset serves the proper purpose of studying Openwhisk's practical use.

Greedy placements are decided by looking back at the parent invoker. The controller tries to allocate space on the same invoker, thereby avoiding extra network load. Appendix A contains a complete implementation of the *schedule* function, which contains the relevant part of the scheduling chain. The most important input parameters for this function are *msg*, the ActivationMessage, and *schedulingState*, the LoadBalancerState. Additionally, the array of available invokers is passed as the *invokers* variable. The ActivationMessage contains a bunch of action metadata such as the *ptransid*, *transid* and *cause*. The inner and outer transaction IDs from the previous section are named *transid* and *ptransid* (parent transid) respectively. The value of these two transaction IDs and the cause field differentiate actions in five cases:

1. Action is not a part of a composition but a standalone invocation.
2. Action is the first invocation of a composition.
3. Action is the first invocation of a subcomposition that depends on another composition that was already started.
4. Action is part of a composition that was already started.
5. Action is a resumption of a composition that split into multiple subcompositions.

These cases are annotated as such in the complete code in Appendix A. Cases 1 and 2 are "fresh" action, and require looking for any invoker with a low load. Cases 3 and 4 resemble an

action that has a data dependency to a single parent action. The parent is determined by using cause and transaction ID information to backtrack up the composition graph as was visualised in Figure 4.4. A simple map can associate each subcomposition with the invoker that was most recently used for it, yielding the parent invoker. It is possible that this parent invoker has no space left for allocating the current action and a backup invoker needs to be found. Ideally, a backup invoker is "close" to the parent invoker from the perspective of the underlying network connectivity: high bandwidth and low hop count. However, due to the high-level abstraction provided by invokers in Openwhisk the underlying network is invisible from inside the controller. As a sub-optimal alternative the backup invoker can be decided in the same way as in cases 1 and 2. Finally, the resumption in case 5 boils down to the multi fan-in case from the previous section. It is not possible to infer which invokers the different parents of the current action resided in, as was already discussed. The logic falls short here. In cases 3 and 4, a backup strategy was used for a fully occupied parent invoker, but this situation is different because even the primary choice is not feasible. In the code, it is annotated as an incomplete case by virtue of a warning message if the controller reaches a situation that matches with case 5.

```
1 private var _causeToTransidHistory: Map[ActivationId, TransactionId],
2 private var _transidToInvokerMap: Map[TransactionId, InvokerInstanceId],
3 private var _pendingOutputTransfer: Map[TransactionId, (Number, InvokerInstanceId)],
4
5 // can add more metadata to transferlist if needed, potentially make it a case class
      on its own
6 private var _outputTransferHistoryPerComposition:
7     Map[ActivationId, ListBuffer[(Number, InvokerInstanceId, InvokerInstanceId)]],
8 private var _transidToCompositionIdentifier: Map[TransactionId, ActivationId],
9 private var _finishedCompositions: ListBuffer[ActivationId],
```

Listing 4.3: Data structures used to keep track of different aspects of active compositions during scheduling.

Aside from querying the *schedulingState* input parameter, the customised version of the *schedule* function contains plenty of bookkeeping actions before and after deciding the destination invoker on which the current action will be scheduled. The data structures in Listing 4.3 are added to the controller state in order to facilitate all the necessary bookkeeping. At its core, the additional state is mostly mapping different data sources to identifiers of compositions and recently used invokers.

The main issues encountered when implementing network-optimised scheduling in Openwhisk are (1) action-centric design pushing logic to be greedy, (2) high-level invoker abstractions ob-

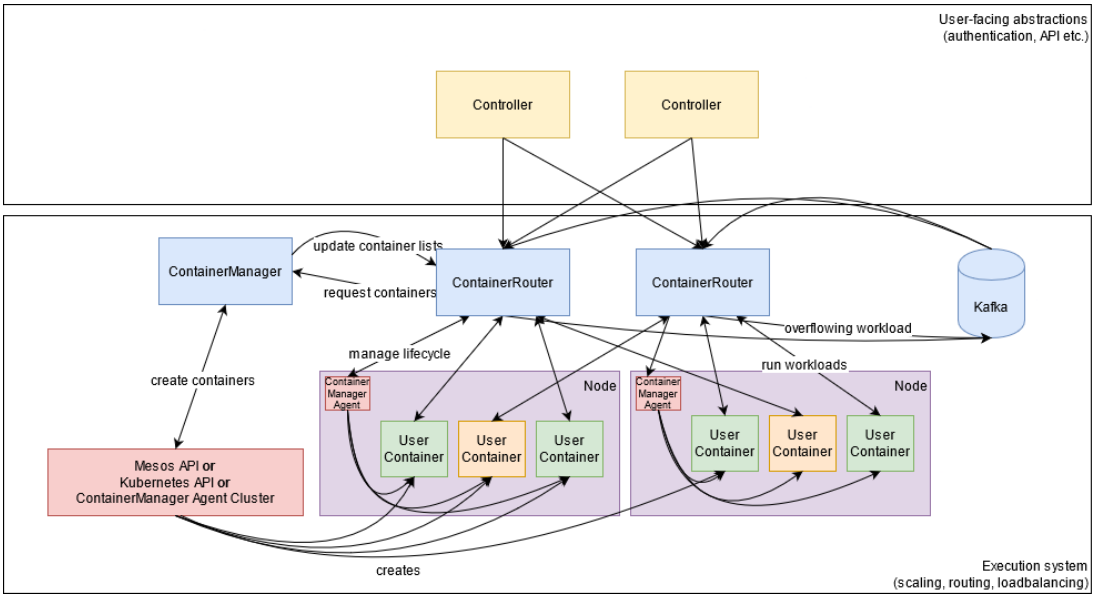


Figure 4.6: General dependencies between components in the newly proposed Openwhisk architecture [5].

scuring underlying server connectivity and (3) the multi fan-in case limiting the available information about parents' invokers. The following section investigates the most recently proposed adaptations to the scheduling logic and architecture in the development process of Openwhisk. The feasibility of Openwhisk from a research experimentation's point of view will increase if the issues listed above are addressed in these proposals.

4.5 Active proposals on the future of Openwhisk

In 2018, the Openwhisk designers noticed a discrepancy between the current architecture and the two ways of deploying production-ready Openwhisk services [5]. Openwhisk is either deployed on low-level machines (bare-metal or VMs) or a high-level container orchestrator (e.g. Kubernetes) [62]. The latter is widely used but makes the invoker design from Figure 4.1 feel somewhat foreign due to the container-in-container execution that ensues. The proposal that started back in 2018 has seen regular design updates but has not yet made its way into the core of Openwhisk due to the drastic changes it puts forward.

The core idea of the proposal is to manage container environments in an explicit manner [5]. Abstractions remain in place for the end-user, but internal operations will have limited abstraction levels. This means that first-level abstracted containers will replace the high-level invoker currently in use to reduce overhead. Kafka is also removed from the "hot path" as controllers will

now directly communicate with "container routers". These can pick existing warm containers or request the creation of additional containers to place the required actions. A visualisation of the various interactions in the new architecture can be found in Figure 4.6. Placement decisions have been extracted from the controller component and moved to the new container routers.

It should be noted that the current proposals do not address function compositions at all. With regards to the intermediate conclusion laid out above, there is nothing in these proposals that undermines the reasoning and findings listed there. In order to guarantee scalability of the design, the quote "shared state between the ContainerRouters is not feasible" [5] reinforces the infeasibility of adding more state to the core Openwhisk scheduler.

The focus of the revamped architecture seems to be around low-level container control mechanisms. Explicitly removing state from controllers and load balancers means most of the features that are required to make the previous modification fully functional do not seem likely. Openwhisk is doubling down on simple, stateless placement algorithms. There is no mention of compositions in the current proposals, indicating they represent only a small share of real-world Openwhisk usage and are not actively being optimised for.

4.6 Feasibility conclusion and potential improvements

Managing the necessary state across multiple instances of a scheduler remains a big issue. If actions take a couple of milliseconds to execute, it is unacceptable to spend the same order of magnitude updating state vectors between schedulers. Not allowing scheduler duplication will create a tight bottleneck. As was already proposed, a possible compromise could be to use a data partitioning scheme such that a single instance can perform scheduling of a particular subset of actions that require the same controller state information without increased communication overhead.

Even though the discovery of using transaction IDs to insert previously unavailable internal information into the scheduling logic was promising, it falls just short of ultimately enabling arbitrarily complex scheduling in Openwhisk. It can be assumed that a radically different information source is required to discover the parent information from previous scheduling passes that are appropriate for the current action in any situation. After conducting this investigation, I believe this information is not currently embedded inside the Openwhisk architecture in a useful way, mainly due to the action-centric design. A major architectural redesign with composition metadata in mind is required to utilise the potential of advanced scheduling approaches fully. The

trade-off for the designers is that there is no benefit in doing this for simple actions that do not belong to a composition, and there may not be enough demand for data-intensive composition optimisation to justify this redesign for Openwhisk. A 2018 survey estimated that around 21% of serverless applications involved data processing [1]. However, a re-estimation of this percentage might be of interest, as the nature of applications might have changed significantly in three years time.

Since the Openwhisk project was initially created as a commercial platform, it was not designed with extensive modifiability in mind. As illustrated above, this can make the platform somewhat unsuited for thorough experimentation of (radically) different custom logic due to design constraints. Below, I will describe some properties and design considerations that might be useful for serverless platforms to incorporate.

Decouple administrative actions: Currently, the secondary activations that execute administrative tasks are meshed into the composition uploaded by the end-user. However, if a secondary activation runs between two actions with a data dependency, the secondary activation shares this data dependency. It receives a set of parameters concerned with its current "position" in the composition and the parameters that should be passed to the following action. As far as I can tell, there is no explicit need in the administrative tasks for the following action's parameter set beyond simply being the middleman in communication. It would not be difficult to set up a dedicated key scheme to communicate these parameters separate from the current messages sent over Kafka. In a data-intensive composition, this could provide a significant decrease in network load as only half of the largest messages would be sent over the network.

Composition-first architecture: Build a platform focused around compositions of cloud functions and treat singular functions as a special case of a composition with one node. As illustrated by Openwhisk in this chapter, the other way around makes the architecture highly rigid and prone to feature lock-out in the future. Composition overhead can optionally be optimised away at the moment a singular function is uploaded to the platform, before it is ever executed such that the end-user does not experience unnecessary overhead and subsequent delays.

Event-driven architecture: Openwhisk leverages an event-driven architecture, which seems highly suited for the asynchronous nature of its use case. Credit where credit is due, this architecture is ideally suited for scalable, high-performance systems and should be used as creatively as possible. Other recent research endeavours such as Triggerflow [14] and Skedulix [41] exemplify the proper use of event-driven architecture for serverless platforms.

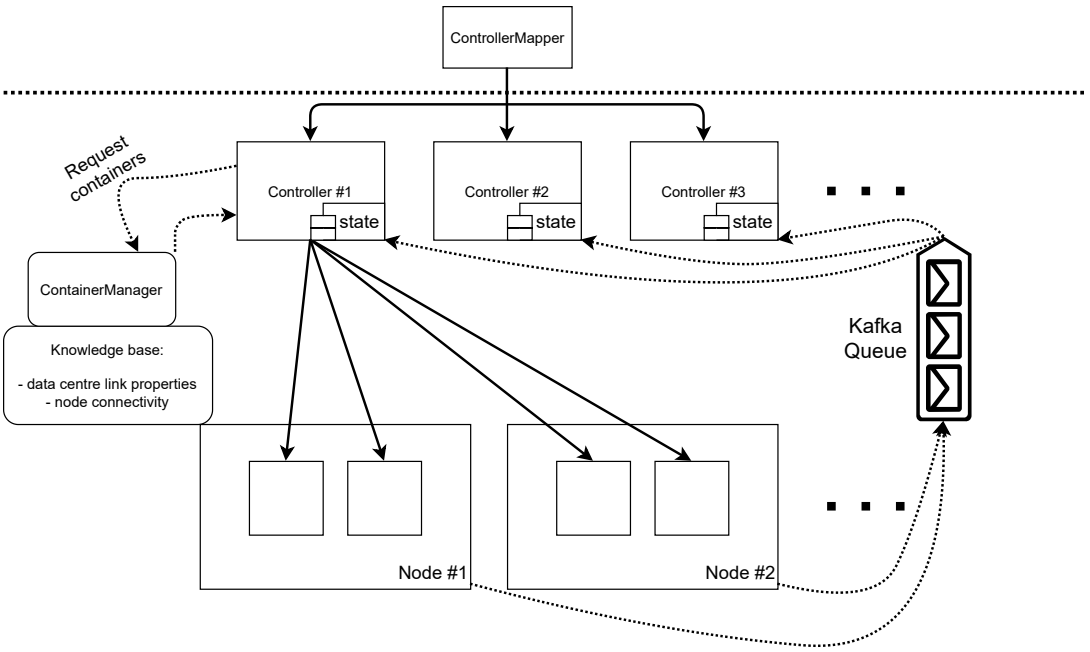


Figure 4.7: Proposed serverless platform architecture based on the future design of Openwhisk

Modifiability: Ideally, for research and development purposes, the event-driven architecture is used loosely. This means that it should be reasonably versatile in adapting to changes to the chain of communications. Consequently, a researcher or open-source contributor can experiment freely on an aspect of this chain they find interesting while still fully utilising the remainder of the platform's original functionality. As illustrated in detail in this feasibility study, insufficient modifiability can lead to alienation from research, inconclusive research or research based on workarounds that are difficult to translate into valuable real-world scenarios.

These properties and design considerations can be used to create a further adaptation to the proposal in Figure 4.6. My personal proposal can be found in Figure 4.7, and is structurally not much different from the Openwhisk proposal. The core idea is to have each controller hold the state concerning a set of compositions. Therefore, there is no need for additional data transfers to keep the state consistent across controllers. Each controller can query the ContainerManager to request the creation of a new container or to receive information about the connectivity of the different nodes and their link properties. The properties of physical nodes do not change regularly if at all, which means a query on startup can often suffice. Splitting the administrative actions from the useful execution tasks can be done by using clever Kafka key schemes. This can change the view of a composition from an alternating pattern between administrative and useful tasks to an interleaved occurrence of two compositions side-by-side. Both types of actions would still be handled by the same controller, but parent-child data transfers would not have their data passed on through these secondary activations like in Openwhisk.

"If we do it right, we might actually be able to evolve a form of work that taps into our uniquely human capabilities and restores our humanity. The ultimate paradox is that this technology may become the powerful catalyst that we need to reclaim our humanity."

~John Hagel, founder of Deloitte Center for the Edge

5 Conclusion

To conclude this dissertation, this chapter will list the main takeaways and pointers for future work that are relevant to the processes discussed over the previous chapters.

On the one hand is the academic effort of researching many different and unique angles from which this scheduling problem can be viewed. These often do not scale well due to their inherently more complex nature. This also applies to the network-optimised scheduler described throughout this dissertation. On the other hand are the cloud service providers, whose optimisation efforts have not been in the field of serverless scheduling in any significant way. This can be attributed to a low share of data-intensive serverless applications in the current userbase. Although the usage of serverless is forecasted to grow considerably, the future design proposals for Apache Openwhisk indicate that function compositions are not yet being promoted to first-class citizens in the serverless space.

The feasibility study of integrating a significantly more involved scheduling approach into Openwhisk than is used by default yielded some useful insights into the design decisions that influence the processing of information. The lessons that can be learned from that study are not limited to serverless platforms but apply to any cloud solution. Furthermore, I encourage similar studies for other open-source platforms such as Fission, OpenFaas and Kubeless. Thoroughly

dissecting the internals to extract the core issues causing roadblocks in implementation takes a significant amount of time. Still, I believe valuable lessons are to be learned from combining the experience in this dissertation surrounding Openwhisk with similar efforts in other platforms.

In terms of improvements that can be made to the simulation and modelling of the network-optimised scheduler, two main takeaways jump out. First, the introduction of so-called "slack time". Currently, a function tries to execute as soon as its prerequisites have finished. In the same way that a VLIW scheduler will hold off on placing operations that can wait, this scheduler could wait 1-5ms if appropriate to enable an even lower network load. Of course, this creates a much more extensive search space, but further analyses on the impact and particularities of adding slack time to a schedule could be interesting to follow up on. Second, for the discussion about graph partitioning solutions leaning towards balanced partitionings, a field that may prove suitable for additional research is "community detection", as that is tailored towards an unbalanced grouping of nodes in a graph.

Bibliography

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A Berkeley view on serverless computing," feb 2019. [Online]. Available: https://arxiv.org/abs/1902.03383

[2] K. Bilal, S. U. Khan, L. Zhang, H. Li, K. Hayat, S. A. Madani, N. Min-Allah, L. Wang, D. Chen, M. Iqbal, C. Z. Xu, and A. Y. Zomaya, "Quantitative comparisons of the state-of-the-art data center architectures," *Concurrency Computation Practice and Experience*, vol. 25, no. 12, pp. 1771–1783, 2013.

[3] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, "Communication-avoiding QR decomposition for GPUs," in *Proceedings - 25th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2011*, 2011.

[4] Apache, "Openwhisk System Overview," accessed on: 2020-11-06. [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/about.md#how-openWhisk-works

[5] Markus Thömmes, "OpenWhisk future architecture," accessed on: 2021-05-11. [Online]. Available: https://cwiki.apache.org/confluence/display/OPENWHISK/OpenWhisk+future+architecture

[6] Microsoft, "What is cloud computing? A beginner's guide," accessed on: 2021-05-28. [Online]. Available: https://azure.microsoft.com/en-in/overview/what-is-cloud-computing/

[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," pp. 50–58, 2010.

[8] Amazon Web Services, "Amazon EC2 Features - Amazon Web Services," 2020, accessed on: 2021-05-23. [Online]. Available: https://aws.amazon.com/ec2/features/

[9] Google, "Compute Engine: Virtual Machines (VMs)," accessed on: 2021-05-23. [Online]. Available: https://cloud.google.com/compute#section-9

[10] Microsoft, "Explore Azure Virtual Machines - Learn | Microsoft Docs," accessed on: 2021-05-23. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/

[11] IBM, "IBM Cloud Virtual Servers - Overview - United Kingdom | IBM," accessed on: 2021-05-23. [Online]. Available: https://www.ibm.com/be-en/cloud/virtual-servers

[12] G. McGrath and P. R. Brenner, "Serverless Computing: Design, Implementation, and Performance," in *Proceedings - IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW 2017*, 2017, pp. 405–410.

[13] Amazon, "AWS Lambda Limits - AWS Lambda," pp. 2–4, 2014, accessed on: 2020-12-07. [Online]. Available: https://aws.amazon.com/lambda/pricing/

[14] P. G. López, A. Arjona, J. Sampé, A. Slominski, and L. Villard, "Triggerflow: Trigger-based orchestration of serverless workflows," in *DEBS 2020 - Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, jun 2020, pp. 3–14. [Online]. Available: http://arxiv.org/abs/2006.08654http://dx.doi.org/10.1145/3401025.3401731

[15] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in *CIDR 2019 - 9th Biennial Conference on Innovative Data Systems Research*, 2019.

[16] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 992017 Symposium on Cloud Computing, 2017, pp. 445–451.

[17] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017*, 2017, pp. 363–376.

[18] Nitzan Shapira, "AWS Lambda Language Comparison: pros and cons," accessed on: 2021-05-27. [Online]. Available: https://epsagon.com/development/aws-lambda-programming-language-comparison/

[19] M. Adhikari, T. Amgoth, and S. N. Srirama, "A survey on scheduling strategies for workflows in cloud environment and emerging trends," *ACM Computing Surveys*, vol. 52, no. 4, 2019.

[20] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: Function composition for serverless computing," in *Onward! 2017 - Proceedings of the 2017 ACM SIGPLAN International Symposium on*

New Ideas, New Paradigms, and Reflections on Programming and Software, co-located with SPLASH 2017. New York, New York, USA: ACM Press, 2017, pp. 89–103.

[21] M. Cosnard and E. Jeannot, "Compact DAG representation and its dynamic scheduling," *Journal of Parallel and Distributed Computing*, vol. 58, no. 3, pp. 487–514, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731599915666

[22] K. Bessai, S. Youcef, A. Oulamara, C. Godart, and S. Nurcan, "Bi-criteria workflow tasks allocation and scheduling in cloud computing environments," in *Proceedings - 2012 IEEE 5th International Conference on Cloud Computing, CLOUD 2012*, 2012, pp. 638–645.

[23] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the "micro" back in microservice," in *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018*, 2020, pp. 645–650.

[24] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Computer Communication Review*, vol. 38, no. 4, 2008, pp. 63–74.

[25] S. Ramli and D. I. Jambari, "Capacity planning for green data center sustainability," *International Journal on Advanced Science, Engineering and Information Technology*, vol. 8, no. 4, pp. 1372–1380, 2018.

[26] M. T. Chaudhry, M. H. Jamal, Z. Gillani, W. Anwar, and M. S. Khan, "Thermal-benchmarking for cloud hosting green data centers," *Sustainable Computing: Informatics and Systems*, vol. 25, 2020.

[27] R. Rahmani, I. Moser, and A. L. Cricenti, "Modelling and optimisation of microgrid configuration for green data centres: A metaheuristic approach," *Future Generation Computer Systems*, vol. 108, pp. 742–750, 2020.

[28] Q. Zhou, J. Lou, and Y. Jiang, "Optimization of energy consumption of green data center in e-commerce," *Sustainable Computing: Informatics and Systems*, vol. 23, pp. 103–110, 2019.

[29] G. Qu, Z. Fang, J. Zhang, and S. Q. Zheng, "Switch-centric data center network structures based on hypergraphs and combinatorial block designs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 4, pp. 1154–1164, 2015.

[30] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: A scalable and fault-tolerant network structure for data centers," in *Computer Communication Review*, vol. 38, no. 4, 2008, pp. 75–86.

[31] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: A high performance, server-centric network architecture for modular data centers," in *Computer Communication Review*, vol. 39, no. 4, 2009, pp. 63–74.

[32] V. Shukla, R. Srivastava, and D. K. Choubey, "Optical Switching in Next-Generation Data Centers," 2019, pp. 164–193.

[33] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data center network virtualization: A survey," *IEEE Communications Surveys and Tutorials*, vol. 15, no. 2, pp. 909–928, 2013.

[34] J. Hamilton, "An architecture for modular data centers," in *CIDR 2007 - 3rd Biennial Conference on Innovative Data Systems Research*, 2007, pp. 306–313.

[35] K. Xi, Y.-H. Kao, and H. J. Chao, "A Petabit Bufferless Optical Switch for Data Center Networks," in *Technical Report, Polytechnic Institute of New NYU*, 2013, pp. 135–154.

[36] N. Mahmoudi, H. Khazaei, C. Lin, and M. Litoiu, "Optimizing serverless computing: Introducing an adaptive function placement algorithm," in *CASCON 2019 Proceedings - Conference of the Centre for Advanced Studies on Collaborative Research - Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '19. USA: IBM Corp., 2020, pp. 203–213.

[37] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. Rojas, "Performance modeling to support multi-tier application deployment to infrastructure-as-a-service clouds," in *Proceedings - 2012 IEEE/ACM 5th International Conference on Utility and Cloud Computing, UCC 2012*, 2012, pp. 73–80.

[38] M. Gabbrielli, S. Giallorenzo, I. Lanese, F. Montesi, M. Peressotti, and S. P. Zingaro, "No more, no less: A formal model for serverless computing," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11533 LNCS, 2019.

[39] Grand View Research, "Serverless Architecture Market Size, Share & Trends Report Serverless Architecture Market Size, Share & Trends Analysis Report By Organization (SME, Large Enterprises), By Vertical, By Service (Automation & Integration), And Segment Forecasts 2018 - 20," pp. 1–141, 2018, accessed on: 2020-12-07. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/serverless-architecture-market/toc

[40] A. Suresh and A. Gandhi, "FNSched: An efficient scheduler for serverless functions," in *WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019*. New York, New York, USA: ACM Press, 2019, pp. 19–24.

[41] A. Das, A. Leaf, C. A. Varela, and S. Patterson, "Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications," in *IEEE International Conference on Cloud Computing, CLOUD*, vol. 2020-Octob, jun 2020, pp. 609–618. [Online]. Available: http://arxiv.org/abs/2006.03720

[42] D. S. Johnson, C. H. Papadimitriou, and K. Steiglitz, "Combinatorial Optimization: Algorithms and Complexity." *The American Mathematical Monthly*, vol. 91, no. 3, p. 209, 1984.

[43] H. M. Verbeek and W. M. Van Der Aalst, "An experimental evaluation of passage-based process discovery," in *Lecture Notes in Business Information Processing*, vol. 132 LNBIP, 2013, pp. 205–210.

[44] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[45] S. Gilpin and I. Davidson, "A flexible ILP formulation for hierarchical clustering," *Artificial Intelligence*, vol. 244, pp. 95–109, 2017.

[46] X. Chen, L. Sun, T. Chen, Y. Sun, Rusli, K. J. Tseng, K. V. Ling, W. K. Ho, and G. A. Amaratunga, "Full coverage of optimal phasor measurement unit placement solutions in distribution systems using integer linear programming," *Energies*, vol. 12, no. 8, 2019.

[47] T. Verbelen, T. Stevens, F. De Turck, and B. Dhoedt, "Graph partitioning algorithms for optimizing software deployment in mobile cloud computing," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 451–459, 2013.

[48] Q. Li, J. Zhong, Z. Cao, and X. Li, "Optimizing streaming graph partitioning via a heuristic greedy method and caching strategy," *Optimization Methods and Software*, vol. 35, no. 6, 2020.

[49] G. Karypis and V. Kumar, "METIS* A Software Package for Partitioning Unstructured Graphs , Partitioning Meshes , and Computing Fill-Reducing Orderings of Sparse Matrices," *Manual*, pp. 1–44, 1998.

[50] A. Kalyanaraman, K. Hammond, J. Nieplocha †, M. Krishnan, B. Palmer, V. Tipparaju, R. Harrison, D. Chavarría-Miranda, J. Makino, D. Bader, G. Cong, B. Hendrickson, J. Shalf, D. Donofrio, C. Rowen, L. Oliker, M. Wehner, and J. L. Gustafson, "Graph Partitioning Software," in *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011, pp. 808–808.

[51] KAHIP, "Karlsruhe High Quality Partitioning," accessed on: 2021-05-03. [Online]. Available: https://kahip.github.io/

[52] P. Sanders and C. Schulz, "Think locally, act globally: Highly balanced graph partitioning," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7933 LNCS, 2013, pp. 164–175.

[53] B. Leasure, D. J. Kuck, S. Gorlatch, M. Cole, G. R. Watson, A. Darte, D. Padua, U. Banerjee, O. Schenk, K. Gärtner, D. Padua, H. Jay Siegel, B. Dalton Young, R. H. Campbell,

Ü. Çatalyürek, C. Aykanat, J. Ajanovic, S. Schmid, R. Wattenhofer, E. N. M. Elnozahy, E. W. Speight, J. Li, R. Rajamony, L. Zhang, B. Arimilli, D. Padua, M. Gerndt, M. Gerndt, D. Padua, J. B. Dennis, B. Smith, G. Almasi, A. Stamatakis, D. Sangiorgi, D. Sangiorgi, D. Padua, J. A. Gunnels, J. Dongarra, P. Luszczek, B. Mohr, R. Eigenmann, P. Feautrier, C. Lengauer, D. Padua, P. Bose, J. F. JaJa, A. Gupta, R. De Nicola, D. Padua, D. Padua, D. Padua, R. S. Armen, E. R. May, M. Taufer, and A. Geist, "PaToH (Partitioning Tool for Hypergraphs)," in *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011, pp. 1479–1487.

[54] C. Chevalier and F. Pellegrini, "PT-Scotch: A tool for efficient parallel graph ordering," *Parallel Computing*, vol. 34, no. 6-8, pp. 318–331, 2008.

[55] C. Aksoylar, J. Qian, and V. Saligrama, "Clustering and community detection with imbalanced clusters," *IEEE Transactions on Signal and Information Processing over Networks*, vol. 3, 2017.

[56] Z. Chkirbene, R. Hamila, and S. Foufou, "A survey on data center network topologies," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11277 LNCS, 2018, pp. 143–154.

[57] K. Andreev and H. Räcke, "Balanced graph partitioning," in *Annual ACM Symposium on Parallel Algorithms and Architectures*, vol. 16. New York, New York, USA: ACM Press, 2004, pp. 120–124.

[58] T. N. Bui and C. Jones, "Finding good approximate vertex and edge partitions is NP-hard," *Information Processing Letters*, vol. 42, no. 3, pp. 153–159, 1992.

[59] IBM Corporation, "IBM CPLEX Optimizer," pp. –undefined, may 2014. [Online]. Available: https://www.ibm.com/nl-en/analytics/cplex-optimizerhttp://ibm.com/software/integration/%5Cnoptimization/cplex-optimizer

[60] K. Djemame, M. Parker, and D. Datsev, "Open-source Serverless Architectures: An Evaluation of Apache OpenWhisk," in *Proceedings - 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing, UCC 2020*, 2020, pp. 329–335.

[61] Apache, "Openwhisk Proposals," accessed on: 2021-05-05. [Online]. Available: https://cwiki.apache.org/confluence/display/OPENWHISK/Proposals

[62] ——, "Apache Openwhisk Documentation," accessed on: 2020-11-05. [Online]. Available: https://openwhisk.apache.org/documentation.html

[63] ——, "Openwhisk Configuration Choices," accessed on: 2021-05-04. [Online]. Available: https://github.com/apache/openwhisk-deploy-kube/blob/master/docs/configurationChoices.md#replication-factor

[64] I. Cloudflare, "What Is A Reverse Proxy? | Proxy Servers Explained," pp. 1–6, 2020, accessed on: 2021-05-04. [Online]. Available: https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/

[65] Apache, "Openwhisk conductor actions," accessed on: 2021-05-09. [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/conductors.md#activations

[66] ——, "Openwhisk-composer demo sample," accessed on: 2021-05-11. [Online]. Available: https://github.com/apache/openwhisk-composer/blob/master/samples/demo.js

[67] The Apache Foundation, "Apache OpenWhisk is a serverless, open source cloud platform," 2018, accessed on: 2020-10-15. [Online]. Available: https://github.com/apache/openwhisk

[68] Apache, "Difficulties for optimal scheduling," accessed on: 2020-11-30. [Online]. Available: https://cwiki.apache.org/confluence/display/OPENWHISK/Design+consideration#Designconsideration-1.2Difficultiesfortheoptimalscheduling

[69] Matt Rutkowski, "Proposals - Openwhisk - Apache Software Foundation," accessed on: 2021-03-28. [Online]. Available: https://cwiki.apache.org/confluence/display/OPENWHISK/Proposals

Appendices

A

Custom Openwhisk scheduler logic

```
def schedule(
    msg: ActivationMessage,
    schedulingState: ThesisNetworkBalancerState,
    maxConcurrent: Int,
    fqn: FullyQualifiedEntityName,
    fqnName: EntityName,
    invokers: IndexedSeq[InvokerHealth],
    dispatched: IndexedSeq[NestedSemaphore[FullyQualifiedEntityName]],
    slots: Int)(implicit logging: Logging, transId: TransactionId): Option[(
    InvokerInstanceId, Boolean)] = {

    val healthyInvokers = invokers.filter(_.status.isUsable)
    if (!healthyInvokers.nonEmpty) {
        return None
    }

    var parentTransid = msg.ptransid
    val currentTransid = msg.transid

    var wasForceAcquisition = true
    var isComposition = true
    var isCompositionKickStart = false
```

```
22
23      // there should be a cause in the (recent) history of invocations
24      msg.cause match {
25          case Some(cause) => {
26              if (parentTransid == TransactionId.unknown) {
27                  parentTransid = schedulingState.causeToTransidHistory(cause) // TODO
    : watch out for multi-influx pattern!
28              }
29              if (schedulingState.knownComposition(cause)) {schedulingState.
    addChildToComposition(currentTransid, parentTransid)}
30              else {schedulingState.addComposition(currentTransid, cause)}
31          }
32          case None => {
33          // TODO: this is probably true for non-composition functions
34              logging.warn(this, s"[THESIS][SCHEDULER] This action is not part of a
    composition")
35              isComposition = false
36          }
37      }
38
39      val chosenInvokerId = if (parentTransid != TransactionId.unknown) {
40          // there is a parent transaction id
41          // options: - direct descendant of another action
42          //          - first scheduled action of this composition (or not a
    composition at all..)
43          if (schedulingState.knownTransid(parentTransid)) {
44              // CASE 3/4
45              // descendant of previous method (includes splits!)
46              // GOAL: try to acquire same invoker as parent
47              val parentInvokerId = schedulingState.transidToInvokerMap(parentTransid)
48              val parentInvoker = invokers(parentInvokerId.toInt)
49              if (parentInvoker.status.isUsable && dispatched(parentInvokerId.toInt).
    tryAcquireConcurrent(fqn, maxConcurrent, slots)) {
50                  logging.info(this, s"[THESIS][SCHEDULER] tryAcquire successful..")
51                  wasForceAcquisition = false
52                  parentInvokerId
53              } else {
54                  // CASE 3/4: backup
55                  // Cannot schedule to parent invoker
56                  // GOAL: choose an empty invoker (as empty as possible) to house
    potential descendants as well
57                  logging.info(this, s"[THESIS][SCHEDULER] parent invoker was not
    available, choosing invoker with most memory available")
58                  val chosenInvokerId = getHealthyInvokerWithMostMemoryAvailable(
    invokers, dispatched)
59                  if (!dispatched(chosenInvokerId.toInt).tryAcquireConcurrent(fqn,
    maxConcurrent, slots)) {
60                  dispatched(chosenInvokerId.toInt).forceAcquireConcurrent(fqn,
```

```
    maxConcurrent, slots)  // force acquire if necessary (should never be)
61                  }
62                  chosenInvokerId
63              }
64          } else {
65              // CASE 1/2
66              // first scheduled action of this composition (or not a composition at
    all..)
67              // GOAL: choose an empty invoker (as empty as possible) to house
    potential descendants as well
68              logging.info(this, s"[THESIS][SCHEDULER] first action or not composition
    , choosing invoker with most memory available")
69              if (isComposition) {isCompositionKickStart = true}
70              val chosenInvokerId = getHealthyInvokerWithMostMemoryAvailable(invokers,
     dispatched)
71              if (!dispatched(chosenInvokerId.toInt).tryAcquireConcurrent(fqn,
    maxConcurrent, slots)) {
72                  dispatched(chosenInvokerId.toInt).forceAcquireConcurrent(fqn,
    maxConcurrent, slots)  // force acquire if necessary (should never be)
73              }
74              chosenInvokerId
75          }
76      } else {
77          // CASE 5: unsolvable in current Openwhisk
78          logging.warn(this, s"[THESIS][SCHEDULER] Have to fall back due to lack of
    scheduling information: Scheduled randomly!")
79          val randomInvokerId = healthyInvokers(ThreadLocalRandom.current().nextInt(
    healthyInvokers.size)).id
80          dispatched(randomInvokerId.toInt).forceAcquireConcurrent(fqn, maxConcurrent,
     slots)
81          randomInvokerId
82      }
83      logging.info(
84          this,
85          s"[THESIS][SCHEDULER] Chosen invoker with ID ${chosenInvokerId.toInt} as
    target for execution"
86      )
87
88      // Register transid with invoker choice
89      schedulingState.registerInvokerAcquisition(currentTransid, chosenInvokerId)
90      // Register start of output transfer
91      schedulingState.registerPendingOutputTransfer(currentTransid, outputSize,
    chosenInvokerId)
92      // Resolve potential incoming data transfer to finish
93      if (schedulingState.resolveIncomingDataTransfer(parentTransid, chosenInvokerId))
     {
94          logging.info(this, s"[THESIS][SCHEDULER][BOOKKEEPING] resolved incoming data
     transfer!")
```

```
95      } else {
96          logging.info(this, s"[THESIS][SCHEDULER][BOOKKEEPING] no incoming data
    transfer to resolve..")
97      }
98      // Register cause-transid
99      msg.cause.foreach { c =>
100     schedulingState.updateCauseHistory(c, currentTransid)
101     }
102
103     logging.info(this, s"[THESIS][SCHEDULER] checking the function name for custom
    naming scheme")
104     if (fqnName.asString.contains("__")) {
105     if (fqnName.asString.split("__")(1) == "stop") {
106         schedulingState.scheduleCompositionForfinish(currentTransid)
107     }
108     } else {
109         // either management function or not a composition (or not following the
    naming convention)
110         // first function of a composition cannot be the end as well
111         if (isComposition){
112             if (!isCompositionKickStart && schedulingState.
    compositionScheduledForFinish(parentTransid)) {
113             // function was scheduled for termination, so terminate now:
114             // - stop registering for this composition,
115             // - log the internal bookkeeping structures
116             val transfers = schedulingState.finishComposition(currentTransid)
117             logging.info(
118                 this,
119                 s"[THESIS][SCHEDULER] Transfers [${transfers.length}]: ${transfers}"
120             )
121             }
122         }
123     }
124     Some(chosenInvokerId, wasForceAcquisition)
125 }
```