Synthesizing Cluster Management Code for Distributed Systems

Lalith Suresh
VMware

João Loff
INESC-ID, Instituto Superior Técnico, U. Lisboa

Nina Narodytska
VMware

Leonid Ryzhyk
VMware

Mooly Sagiv
VMware

Brian Oki
VMware

Abstract

Management planes for data-center systems are complicated to develop, test, maintain, and evolve. They routinely grapple with hard combinatorial optimization problems like load balancing, placement, scheduling, rolling upgrades and configuration management. To tackle these problems, developers are left with two bad choices: (i) develop ad-hoc mechanisms for systems to solve these optimization problems, or (ii) use specialized solvers that require steep engineering effort.

We propose Weave, a tool that enables programmers to specify cluster management policies in a high-level declarative language, and compute policy-compliant configurations automatically and efficiently. Weave allows constraints and policies, the essence of a management plane, to be easily added, removed and modified over time, using a language familiar to developers (SQL). In this paper, we discuss our approach of management plane synthesis, its benefits, and present preliminary results from implementing a Kubernetes scheduler and a CorfuDB management plane using Weave.

CCS Concepts • **General and reference** → **Design**; • **Computer systems organization** → **Maintainability and maintenance**; *Reliability*; • **Software and its engineering** → *Cloud computing*.

Keywords declarative cluster management, constraint programming, SQL, distributed systems

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321444

ACM Reference Format:
Lalith Suresh, João Loff, Nina Narodytska, Leonid Ryzhyk, Mooly Sagiv, and Brian Oki. 2019. Synthesizing Cluster Management Code for Distributed Systems. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3317550.3321444

1 Introduction

Production-grade distributed systems execute a diverse range of management tasks, such as host lifecycle management, cluster resizing, failure recovery, load balancing, membership management, placement, and rolling upgrades. These tasks fall under the purview of a *management plane* (MP) that continuously tracks the state of the cluster and triggers management tasks in response to state changes.

At a high level, cluster management is guided by a set of *policies* that represent both hard constraints ("ensure a VM requesting a GPU is always placed on a host with a GPU") and soft constraints ("scatter replicas across as many data-centers as possible"). Given a policy, the MP must find an optimal cluster configuration satisfying all hard constraints while minimizing violations of soft constraints.

MP developers usually tackle the configuration problem by designing custom application-specific algorithms—*an approach that often leads to a software engineering deadend.* As new types of constraints are introduced, the developers get overwhelmed by having to solve arbitrary combinations of increasingly complex constraints. This is not surprising given that, beyond the simplest cases, cluster management policies amount to *NP-hard combinatorial optimization problems* that cannot be efficiently solved via naive search. In addition to the algorithmic complexity, the lack of separation between the application, the policy, and the constraint solving algorithm leads to unmaintainable code.

As a result, cluster management systems simply disallow many useful types of policies, requiring the end user to manually translate their high-level requirements into low-level configuration decisions.

A promising approach to overcoming these issues is using specialized tools, such as constraint solvers, integer linear programming (ILP) toolkits, and SMT solvers. These tools incorporate decades of progress on computational learning, constraint propagation, symmetry breaking, local search and many other techniques to solve real-world combinatorial optimization problems efficiently. Subsequently, researchers have used them to tackle instances of cluster management problems like scheduling [11, 16], server provisioning [8], traffic engineering [12] and configuration management [15]. They are thus applicable to a wide range of problems, eliminating the need for ad hoc algorithms. In addition, the solver-based approach enforces clean separation of concerns between MP components by interposing a layer of abstraction between the system and the solver: the developer expresses system configuration and policy using the solver's modeling language, without having to worry about the details of the search algorithm.

Why is this principled approach hardly ever used in production systems? The answer is two-fold. First, integrating optimization tools in a production system requires a steep engineering effort: one must map application-level constraints into an appropriate mathematical formalism, encode these constraints in the modeling language of the solver, translate cluster state into model input variables and translate solver output back to a new cluster configuration, generate failure explanations whenever the solver is unable to find a solution, etc. These tasks require specialized skills that system developers do not possess. For instance, coming up with an efficient encoding of a problem requires an in-depth understanding of solver internals.

Second, management planes typically evolve from simple designs, initially only supporting few constraints that can be readily solved using hand-crafted algorithms. At this stage, the complexity of integrating a solver into the system is hard to justify. By the time the developers hit a brick wall, where adding new features in an ad hoc manner becomes infeasible, the system has accumulated too much technical debt, with constraint solving logic being deeply intertwined with all parts of the system. At this point, integrating a solver in the system requires its major overhaul, which rarely happens in practice (see Section 2 for an example).

We present Weave, a tool that eliminates the barrier to solver-based MP programming by synthesizing optimization code automatically. We exploit the fact that in modern distributed systems the information required to formulate and solve the configuration problem is readily available in the *cluster state database*. This database maintains an up-to-date view of cluster configuration, including its physical inventory, workload, user preferences, etc. Instead of requiring the developer to build a separate system model for use by the solver, Weave extracts this model from the database schema.

Weave programmers express hard and soft constraints (or the policy) as a collection of views over the state database. Thus, they do not need to learn the low-level language of the solver, and instead work with familiar SQL.

Given the state database schema extended with views, Weave generates an efficient encoding of the configuration problem in the constraint solver's language. It uses the high-level information extracted from SQL to generate optimized encodings that would be hard for non-expert users to hand-craft.

At runtime, Weave automates the communication between the MP and the solver by extracting current cluster configuration from the state database and converting it into an input to the solver. It parses the solution computed by the solver and outputs the requisite modifications (deltas) to be made to the database state to satisfy all the policies. A controller, written by the developer, observes these deltas and issues lower-level control plane commands to the underlying system.

2 Motivating example

We use the *Kubernetes scheduler* [1] to illustrate the limitations of the ad-hoc approach to cluster management. However, the challenges associated with the ad-hoc approach are general and recur across different MPs, not just Kubernetes.

The scheduler is responsible for assigning collections of containers, called *pods*, to cluster nodes. In the current implementation it works as follows: For each pending pod placement request, the scheduler first iterates through each node in the system, and filters out the set of valid nodes using a list of *predicates* (analogous to hard constraints). The scheduler ranks all the valid nodes using some priority functions (analogous to soft constraints). The pod is finally placed on the highest ranked node.

This design is simple, flexible (allowing developers to contribute new predicates and priority functions), and efficient. Unfortunately, it also proved fundamentally limited when it comes to expressing more complex constraints due to three main challenges:

Dependencies among constraints (C1) Some time after the basic scheduling algorithm was implemented, Kubernetes introduced a new *eviction* policy: if the scheduler cannot place a pod, it checks whether it can preempt existing lower priority pods. This policy interacts with the basic node placement policy: the preemption code verifies whether affinity/anti-affinity predicates for the original pod were in use, and if they will be violated if that lower priority pod is evicted (causing those predicates to be re-executed).

As new, more complex predicates are added, the preemption code needs to remain in sync with the rest of the scheduler, making it challenging to evolve the code over time. The complexity accumulates making entire classes of policies difficult to implement in the scheduler. For example, cross-node preemption, where a pod on one node is preempted to allow a new pod to be placed on another node (because of affinity/anti-affinity requirements) is currently not possible [13].

Global constraints (C2) Consider Kubernetes' *service affinity* predicate. The predicate ensures that all pods from the same service (e.g., a fleet of load balancers) are assigned to nodes that have identical values for some configured labels (e.g., an availability zone label). This constraint holds over *groups* of pods and nodes, but it is hard to implement this in the current scheduler architecture that can only reason about placing pods, *one at a time*. Therefore, the scheduler, when placing each pod from a group, has to scan some data-structures to make sure its next decision will be consistent with past decisions for pods from the same service. The scheduler therefore contains additional code to pre-compute some of these results to avoid the inefficient scans.

Not surprisingly, the code is commented with a large warning that indicates that the predicate is not guaranteed to work without the precomputed metadata, and there is a discussion to ax the feature because of the accumulating technical debt [14].

Coupling between policies and state (C3) Finally, all the scheduler code is tightly coupled with the data-structures that hold the cluster state. For example, it is not possible to re-use the code for defining constraints and solving them in Kubernetes to be used in a different system. This is inherent in not just the interfaces to add new predicates and priorities, but also in all the associated code of the scheduler around preemption.

Specialized solvers While notoriously hard to use for the reasons stated in §1, specialized solvers overcome the above challenges. Solvers support input specification languages that capture complex constraints in a modular way, even though *internally*, the constraints may interact in complex ways during the search for a solution. Therefore, developers can easily add and modify constraints (**C1**).

The tools are capable of efficiently solving complex aggregate and global constraints (**C2**) using techniques like bounds propagation and learning. Lastly, these tools implement general-purpose algorithms, applicable to a wide range of optimization problems, eliminating the need to hand-craft new algorithms for each distributed system (**C3**). The combinatorial optimization problems that recur across MPs are therefore best left to specialized solvers.

3 Design

Weave enables programmers to specify cluster management policies in a high-level declarative language, and compute policy-compliant configurations automatically and efficiently. To achieve this, we have to address two design challenges. First, we need a specification language that is sufficiently expressive, yet easy to use. We chose SQL for three reasons. First, relational databases are already widely used to store system configuration in existing cluster management software. Second, unlike most formal specification languages, SQL is familiar to most developers. Indeed, its power has been harnessed in a wide-range of data-centric systems [2, 4, 7]. Finally, as we discuss in this and the following section, SQL allows expressing complex MP policies in a clear and concise fashion.

The second challenge is to bridge the gap between the SQL representation of the problem and formal modeling languages used by optimization tools. To this end, we implemented a compiler that synthesizes the necessary inputs to the solver from the SQL schema, and a runtime engine that feeds the current state of the cluster stored in the state database to the solver and reflects the solution returned by the solver back in the database.

As a result, Weave users do not have to learn a new language, and, more importantly, do not have to master the subtle art of writing efficient optimization models. In the coming sections, we discuss each component in more detail.

3.1 Configuration database

At development time, the programmer specifies an SQL schema that represents the cluster's state, annotating some of its columns as *decision variables*, i.e., variables to be assigned automatically by Weave. For example, a placement decision of a pod on a node in Kubernetes can be represented by the following:

```
create table pod_info
(
  pod_name varchar(100) not null primary key,
  ....
  variable__node_name varchar(36) not null,
);
```

Here, the `variable__` prefix indicates that the `variable__node_name` column should be treated as decision variables. Other columns become *input variables*, whose values are supplied by the database.

3.2 Policies

The next step is to specify constraints over decision variables.

Hard constraints Hard constraints are specified as SQL views with the prefix `constraint_`. For example, consider the following constraint for Kubernetes:

```
create view constraint_node_predicates as
select * from pod_info
join node_info
  on pod_info.variable__node_name = node_info.name
where not(pod_info.status = 'Pending') or
   (node_info.unschedulable = false and
    node_info.memory_pressure = false and
    node_info.disk_pressure = false and
    node_info.pid_pressure = false and
    node_info.network_unavailable = false and
    node_info.ready = true);
```

Here, Weave must ensure that for all records returned by joining the `pod_info` and `node_info` tables, the predicate in the `where` clause holds true. This constraint therefore ensures that pods that are pending placement are never assigned to nodes that have been marked unschedulable by the operator, are under resource pressure, or are not ready to accept new pod requests.

Soft constraints Like hard constraints, soft constraints are also specified as SQL views. However, soft constraint views need to be scalar expressions that return a single value. Weave ensures that the computed solution *maximizes* the sum of every specified soft constraint. All views with the prefix `objective_` are treated as soft constraints.

As an example, consider a simple load balancing policy to balance the CPU utilization of nodes in a cluster. To do so, we first write a convenience view (`demand_per_node`) that computes the spare CPU capacity per node. We then describe a soft constraint view (`objective_load_balance_cpu`) to compute the minimum spare capacity in the cluster. By simply declaring such a view, Weave computes solutions that maximize the minimum CPU utilization of nodes in the cluster.

```
create view demand_per_node as
select (node_info.cpu_capacity
        - sum(pod_info.cpu_request)) as cpu_spare
from node_info
join pod_info
  on pod_info.variable__node_name = node_info.name
where status = 'Pending'
group by node_info.name;

create view objective_least_requested_cpu as
select min(cpu_util) from demand_per_node;
```

3.3 Synthesizing optimization models

An optimization model expresses the optimization problem using the mathematical formalism supported by a specific solver, e.g., a set of linear inequalities (for an ILP solver), predicates in first-order logic (for an SMT solver), or a constraint modeling language (for a constraint solver).

By representing constraints in SQL, Weave is able to support multiple backend languages, allowing system builders to easily try different types of solvers. Currently, we support MiniZinc (which in turn supports a variety of solvers) and are also implementing a backend for Google's OR-tools.

Furthermore, our design enables different optimizations at various stages of the synthesis process. For example, we use backend-specific optimizations during code generation like rewriting expressions to use *global constraints*. Global constraints are constraints over *groups* of variables for which solvers implement specialized and efficient propagator algorithms that can dramatically reduce the search space of the problem. For example, when generating MiniZinc code, we rewrite usages of the `IN` operator in SQL and its arguments to instead use a suite of membership global constraints for which solvers typically implement fast propagators. Weave also allows developers to directly use global constraints in SQL predicates. For instance, a common global constraint we use is the `all_different` constraint, which enforces that a set of variables all take different values (which we use when we need to place replicas in different failure domains, for example).

3.4 Runtime

At runtime, Weave is responsible for communication between the solver and the cluster state database (Figure 1). At startup, the solver is initialized with the optimization model synthesized at the previous step. Weave populates the model by extracting the values for the input variables from the database. Whenever the state database changes (Step 1 in the Figure), e.g., a new pod is added to the system, Weave invokes the solver to compute a configuration update (Step 2), e.g., assigning the new pod to a suitable host while satisfying affinity constraints and optimally balancing load across hosts. Weave converts the solution returned by the solver into a database update (Steps 3 and 4). The final step is to apply the new configuration to the system, which is done by the distributed system controller and is not part of Weave.

3.5 Debugging

A common theme in debugging MPs is to understand why the MP could not identify a valid solution (e.g., a pod placement decision) in the face of a web of constraints.

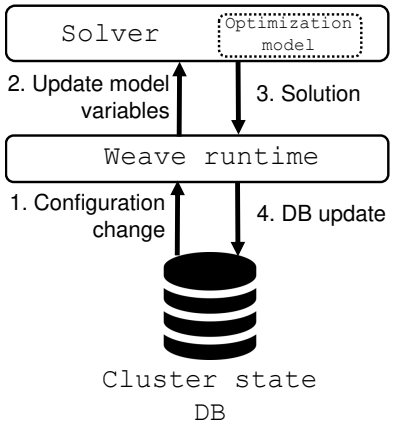

Figure 1. Weave runtime.

Weave improves debuggability by taking advantage of a common solver capability: that of identifying *unsatisfiable cores*. An unsatisfiable core is a minimal subset of model constraints and inputs that suffices to make the overall problem unsatisfiable (for example, a core might involve a single input variable that cannot simultaneously satisfy two contradicting constraints in the model). We leverage this capability by providing a translation layer that extracts an unsatisfiable core from the solver, and identifies corresponding SQL constraints and records in the tables that lead to a contradiction. We found this feature invaluable when debugging placement scenarios involving complex affinity constraints.

4 Evaluation

Our implementation of Weave is about 2K LoC in Java for the library and an additional 1.5K LoC for tests. We first discuss our implementation of a Kubernetes scheduler using Weave. Our scheduler works as a drop-in replacement for the current Kubernetes scheduler (written in Go). We evaluate both qualitative and quantitative gains in using Weave to implement the policies in the scheduler.

The Kubernetes scheduler has 20 configurable 'predicates', which behave as hard constraints, and 11 configurable 'priorities', which are equivalent to soft constraints. We implemented 15 of 20 predicates, where the predicates we skipped were those that were specific to GCE, AWS and Azure. For priorities, we implemented 4 out of 11. We are still working on implementing other constraints, but have no reason to believe that they are any more complicated. In general, we found ourselves spending more time studying the existing Kubernetes code and understanding the intent behind their policies than we did writing the equivalent code in Weave using SQL.

We were able to implement each of the constraints we considered using < 10 lines of SQL. We count every SQL clause (select, from, where etc.) and every predicate separated by and/or as a new line. For example, the *service affinity* policy, discussed in §2-C2, which is challenging to get right in Kubernetes, was implemented in only 6 lines of SQL. It merely involves joining one view (which itself is an additional 6 lines) and a table, and posting a constraint using a `having` clause. As another example, a commonly used soft constraint in Kubernetes is the *least requested* policy, which load balances based on CPU and memory utilization. This policy was implemented in 4 lines of SQL – we simply ask Weave to maximize the minimum sum of the CPU and memory utilization per node.

The number of joins is a commonly reported complexity metric for SQL. Except for one constraint that involved two joins, all hard and soft constraints were expressible using zero or one joins each.

In contrast, the existing Kubernetes scheduler consists of 10s of thousands of lines of Go code that is deeply intertwined with the rest of the system, and is challenging to maintain and evolve.

The resulting scheduler has a performance that is competitive with the Kubernetes scheduler. We attempt to place 400 pods on a 50 node Kubernetes cluster. The baseline Kubernetes scheduler takes roughly 80 seconds to place all pods. Weave on the other hand takes 35s to find an optimal solution and outputs a near-optimal solution in only 15s.

We also implemented a management plane from scratch for CorfuDB [3], a distributed transactional database. As with Kubernetes, we found it easy to add complex features like rack-aware load balancing and flexibly partitioning responsibilities among nodes in the cluster to be straightforward using SQL, all of which used fewer than 10 LoC in SQL.

5 Conclusions

Large distributed system management planes are hard to develop and evolve over time [5, 6, 9, 10]. We therefore believe that there are ample untapped opportunities for using automated tools in the development process. With Weave, we target the hard combinatorial optimization problems that management planes routinely encounter. By synthesizing the required optimization code from a high-level specification written in SQL, Weave allows developers to easily integrate specialized solvers in their MPs, that otherwise involve steep engineering efforts to use. Our preliminary results are promising: we were able to use Weave to power two distributed systems, Kubernetes and CorfuDB.

Acknowledgments

We thank our colleagues at VMware Research and the anonymous reviewers for their valuable feedback. João's research is funded by FCT grant UID/CEC/50021/2019.

References

[1] Kubernetes. http://github.com/kubernetes/kubernetes.

[2] Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J. M., and Sears, R. Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European conference on Computer systems* (2010), ACM, pp. 223–236.

[3] Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobbler, T., Wei, M., and Davis, J. D. Corfu: A shared log design for flash clusters. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 1–14.

[4] Cadoli, M., and Mancini, T. Combining relational algebra, sql, constraint modelling, and local search. *Theory and Practice of Logic Programming 7*, 1-2 (2007), 37–65.

[5] Candea, G., and Fox, A. Crash-only software. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9* (Berkeley, CA, USA, 2003), HOTOS'03, USENIX Association, pp. 12–12.

[6] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An empirical study of operating systems errors. In *ACM Symposium on Operating systems principles (SOSP)* (New York, NY, USA, 2001), SOSP '01, ACM, pp. 73–88.

[7] Delimitrou, C. *Improving resource efficiency in cloud computing*. PhD thesis, Stanford University, 2015.

[8] Guenter, B., Jain, N., and Williams, C. Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning. In *2011 Proceedings IEEE INFOCOM* (April 2011), pp. 1332–1340.

[9] Gunawi, H. S., Do, T., Joshi, P., Alvaro, P., Hellerstein, J. M., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Sen, K., and Borthakur, D. Fate and destini: A framework for cloud recovery testing. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2011), p. 239.

[10] Guo, Z., McDirmid, S., Yang, M., Zhuang, L., Zhang, P., Luo, Y., Bergan, T., Musuvathi, M., Zhang, Z., and Zhou, L. Failure recovery: When the cure is worse than the disease. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems* (Berkeley, CA, 2013), USENIX.

[11] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 261–276.

[12] Koster, A. M., Kutschka, M., and Raack, C. Towards robust network design using integer linear programming techniques. In *Next Generation Internet (NGI), 2010 6th EURO-NF Conference on* (2010), IEEE, pp. 1–8.

[13] Kubernetes. Pod priorities and preemption. https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/, 2018.

[14] Kubernetes mailing list. Let's remove ServiceAffinity . https://groups.google.com/forum/#!topic/kubernetes-sig-scheduling/ewz4TYJgL0M, 2018.

[15] Narain, S., et al. Network configuration management via model finding. In *LISA* (2005), vol. 5, pp. 15–15.

[16] Tumanov, A., Zhu, T., Park, J. W., Kozuch, M. A., Harchol-Balter, M., and Ganger, G. R. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 35:1–35:16.