Minimizing Effort and Risk with Network Change Deployment Planning

Carlos de Andrade, Ajay Mahimkar, Rakesh Sinha, Weiyi Zhang, Andre Cire†,
Giritharan Rana, Zihui Ge, Sarat Puthenpura, Jennifer Yates, Robert Riding

AT&T † University of Toronto

Abstract

Networks undergo continuous changes to introduce new services and improve existing ones. Network change deployment involves carefully deciding when each change activity will be executed and who will be executing the change. This is a complex process because each service group has to plan its activities following a set of operational and technological constraints. Besides, multiple groups may be working on the same or dependent nodes at the same time, and they must co-ordinate their deployment plans. If they do not co-ordinate, conflicting change execution could result in unexpected impacts. Traditionally, change deployment has been a tedious and time-consuming task. To address this, we propose an innovative solution **Zapper** that aims for minimal human effort to coordinate the changes, minimal risk to service quality, and efficient plans to rapidly deploy the changes. Zapper maps change scheduling constraints into mathematical equations and then uses optimization algorithms to generate conflict-free change plans that satisfy all constraints across service groups. We have deployed Zapper at a large service provider and it is being used regularly by the network operations teams for more than two years to schedule over 4.5 million change activities.

I. Introduction

Internet today offers a dizzying variety of services ranging from mission-critical applications such as emergency and police communications to business transactions such as stock trading, as well as end-user entertainment such as video streaming and online gaming. This wealth of services introduced a massive number of devices such as network routers, switches, layer-1 equipment, base stations, web servers, multimedia gateways, set-top boxes, firewalls, and naming servers. The increasing scale and software complexity of these devices and their resulting interaction makes service management and operations extremely challenging.

In addition to day-to-day monitoring and resolution of network issues, operations teams are tasked with continual change deployment on large-scale networks. New network nodes are introduced, old ones removed, software and hardware are upgraded, and configurations and technologies are modified to introduce new network and service features or patch security vulnerabilities and bugs. A change deployment typically goes through multiple phases. First, the operations team designs the change to be deployed, and the workflow (also referred to as the Method of Procedure, MOP) required to execute the change. Second, they execute the change activity on a smaller scale in an operational setup referred to as the First Field Application (FFA), or the phased canary approach [1] and evaluate the impact of the FFA. Finally, once the FFA impact results are verified, they roll out the change activity across the whole network as "crawl–walk–run." People executing change activities are known as *loaders*. In the crawl phase, only a small number of changes are performed by the most experienced loaders. They may be further restricted to operate on a limited range of equipment types. Additionally, network operations teams perform check-ups to ensure that there are no unexpected service disruptions during the change deployment. If such disruptions are observed, they perform a roll-back of the change, or halt the future deployment based on the magnitude of the disruption. If the crawl phase is successful, some of the deployment restrictions are relaxed in the *walk* phase. Finally, full-scale deployment happens during the *run* phase, where the goal is to change the entire network quickly while ensuring co-ordination across multiple service groups to minimize the risk of service impact. At any point in the crawl–walk–run phases, if the teams observe any unexpected impact, then they can change the overall deployment plan e.g., slow down, or halt the deployment.

A. Change Deployment Planning Problem

The goal of the change deployment planning is to design a plan that avoids all conflicts, satisfies all operational and technological constraints, and still finishes all changes in a short amount of time. This problem becomes significantly harder in a heterogeneous network with complex cross-layer and service-layer dependencies. The speed of change deployment matters specifically during application of security patches, or rapid introduction of new service features.

Change conflicts. There exist changes that cannot be performed simultaneously because of *node availability*. We call these situations ***conflicts*** because executing one change activity blocks another change activity. For example, if one were to plan a concurrent change execution on a cloud hypervisor and the virtual network function (or tenant) hosted by the hypervisor, then it would be a conflict. This is because if the hypervisor is unavailable because of a reboot after the software upgrade, then the tenant cannot access the virtual network function. Similarly, a cellular base station cannot be upgraded while its associated switch is unavailable. While it is theoretically possible for the same person to work on a base station and a switch together, many times, these nodes are managed by different organizations (including 3rd party vendors), each with their own deployment plan constraints and it is hard for them to coordinate their work. We refer

ISBN ISBN 978-3-903176-39-3

to such nodes as ***dependent nodes***. Therefore, it is a standard operational practice to make sure that conflicting changes on the dependent nodes are scheduled at different times.

Service impact. There are a number of service impact constraints associated with network change deployments related to (a) capacity, (b) compatibility, and (c) post-deployment risk. (a) Capacity – changes that require node reboot result in a reduction of available network capacity, and thus have to be carefully planned to minimize the disruption to service quality. (b) Compatibility – underlying technology can place constraints to execute the change activities close in time on dependent nodes to ensure inter-operability. (c) Post-deployment risk – if the change deployment results in an unexpected impact to service quality [2], [3], [4], then one has to quickly make a decision to halt the deployment or roll it back. Thus, it is important to introduce changes in co-located parts of the network at a time so that any adverse effects can be quickly identified and mitigated.

Operational restrictions. Finally, there are a number of additional operational constraints associated with change deployments. There is a limit on how many change activities a loader can execute in any maintenance window and there should be some uniformity in the work assigned to a loader. A maintenance window is a time-slot for implementing the change activities. For example, as much as possible, they should be working on the same hardware version within a single maintenance window so that they are not forced to deal with multiple MOPs and thus reduce the risk of mistakes and mis-configurations. Loaders may also be unavailable on certain times (vacation or sick) and have preferences or restrictions regarding change activities that must be respected. For example, an equipment needing hardware upgrade may have to be scheduled during business hours when the facility is accessible.

B. Operational State of the Art

The network operations teams today proactively create change deployment plans for nodes and identify conflicts across different work groups. This action is performed well in advance of the actual change execution because it provides enough time for the operations teams to resolve their conflicts by re-scheduling the change activities. Their experiences have shown that proactive conflict discovery and resolution is more effective than conducting this only at the execution time due to the effective management of their time resources and minimizing the risk of service disruption during change execution. This is especially important when scheduling a wide variety of changes ranging from software to hardware to even construction work and 3rd party vendors, and involving different operations teams responsible for different layers of the network (core versus transport versus edge of the network) and managing multiple services (e.g., VoIP, streaming video) riding on top of these networks. Some of the operations teams use project planning tools (e.g., Gantt charts [5]) in planning their work deployments. However, these tools still require them to manually perform the conflict discovery and resolution and take a huge amount of time when dealing with a large number of nodes.

Challenges: Change deployment planning is challenging due to the following reasons:

1. **Large scale:** The large number of nodes (on the order of hundreds of thousands in large service provider networks) and the large number of work groups deploying change activities bring difficult challenges particularly in discovering conflict-free change plans. The search space is exponentially large and enumerating across them to find the optimal change plan is non-trivial.
2. **Complex service dependency:** Identifying and modeling the service dependency is important to accurately determine the change conflicts and potential service impact risks. Some dependencies are cross-layer (e.g., physical host to virtualized network functions) versus others are service-layer (e.g., a top-of-rack switch serving multiple application servers).
3. **Diverse network configuration:** Service providers have diverse configurations across network, specifically in the edge such as the cellular radio access. It is important to account for such diversity when planning the change deployment.
4. **Multiple deployment constraints:** The change deployment plan has to conform to multiple constraints that have to be applied across different network configurations. Also, different node types and services can require a different set of constraints.

C. Our Approach and Contributions

We propose a new solution called **Zapper** for change deployment planning by carefully modeling the deployment constraints and conflicts a priori and using a suite of optimization algorithms to automatically generate conflict-free change schedules. Such a schedule conforms to service impact and operational constraints. Zapper significantly reduces the human time and effort in discovering the change deployment plans, achieves efficient and timely deployment of changes across the network, and reduces the risk to service impact. Zapper is ***automated*** (minimal human time and effort for co-ordinating change deployment), ***scalable*** (accommodates a very large number of nodes: 100K+), ***effective*** (conflict-free, minimal service impact risk, and timely deployment of changes), ***adaptive*** (discovers change plans across a wide variety of deployment constraints and node configurations), and ***proactive*** (advance planning enables better packing of change plans and minimal completion times).

Key ideas:

1. **Abstracting the heterogeneity of constraints.** Each work group and each type of equipment introduces its own set of constraints. We provide a unified solution by abstracting change activities as a set of attributes and then designing "constraint templates" that can handle a large variety of constraints from different domains uniformly.
2. **Constrained model-driven optimization.** Depending on the constraints and objective function specified by the operations teams, we build on-the-fly sophisticated mathematical models. We then use mixed-integer linear programming or constraint programming solvers and custom heuristics to scale up to large number of nodes and constraints. Zapper then packs changes to minimize the total deployment times subject to constraints.
3. **Two-level formulation.** Zapper not only identifies the best time-slot when the change should be deployed at

the node, but also the loader responsible for the change execution (either executing manually, or watching the automated execution).

4. **Problem segmentation using diverse network configuration.** To address the scale challenge, we segment the problem into smaller instances based on network configuration attributes. We identify the attributes that lead to independent subsets of nodes which can be further considered for optimization independently.

Contributions. We have designed and implemented Zapper (§III) in automatically discovering conflict-free network change deployment plans with minimal risk to service impact. We successfully evaluated Zapper (§IV) using data collected from large operational networks and demonstrated its effectiveness in reducing the change deployment times[1] for zero conflicts and minimal service impact risks. In cases of tighter deployment time-lines (as specified by operations teams input), Zapper is designed to *minimize* the conflicts and service impacts. We have deployed Zapper at a large service provider and it is being used regularly by the operations teams for more than two years to schedule over 4.5 million change activities. We share our operational experiences in §V. Zapper has resulted in significant improvement in operational efficiencies (85% human time savings averaged across multiple teams).

II. Data Sets

Network inventory: The service provider collects inventory of live instances in the production network. The configuration snapshot from each node is collected via the Element Management System (EMS). The node-level configuration captures information such as the software version, hardware version, EMS it connects to, time-zone, and its manufacturer.

Network topology: The topological configuration captures information about the neighbors (e.g., X2 links in LTE network capture the logical neighbor), upstream and downstream nodes (e.g., eNodeBs[2] are downstream for MME[3]), or cross-layer (e.g., virtual machine or Virtual Network Function (VNF) hosted on a physical server). Configuration data plays a vital role in deriving the topological relationships. We derive end-to-end service path by stitching the pair-wise relationships. We show an example service path for LTE data and control planes in Fig. 1. This information is needed to check some of the constraints. For example, we may not bring down adjacent eNodeBs to avoid coverage holes, or schedule SIAD [6][4] and its attached eNodeB in the same time slot.

Planned change logs: The operations teams use a scheduling tool to create and track change requests in a central change repository. The tool enables them to input information about the planned start and end time for executing the change, the node information, change identifier, summary about the change, a MOP document to capture the change workflow,

[1]We define change deployment time as the total time taken to deploy the change across all the nodes in the network, also commonly referred to as makespan in the optimization community.

[2]An eNodeB is a LTE cellular base station in the radio access network. A gNodeB is an equivalent base station in the 5G network.

[3]MME is a Mobility Management Entity in the cellular core network.

[4]SIAD is a Smart Integrated Access Device (or, switch) that connects the eNodeB/gNodeB with the transport and the core network.

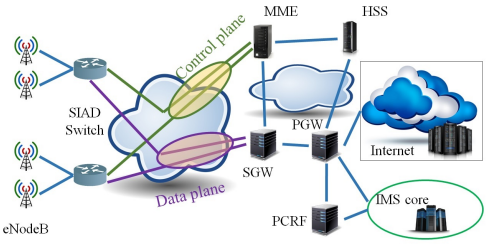

Fig. 1. Service path for LTE control and data planes.

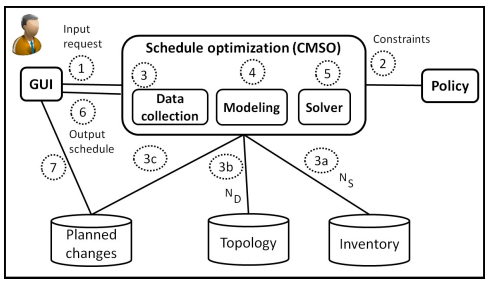

Fig. 2. Change schedule planning in Zapper.

the requester/executor names, the business risk associated with the change, and whether the execution is going to transiently disrupt the service. Changes can take the form of software upgrades, hardware changes, configuration changes, new feature activations, capacity improvements, or introduction of new technologies such as 5G, virtualization, containerization, and software defined networking. We analyzed network changes using data collected over two and a half years and observed that majority of the changes are executed in the edge of the network. This is expected because of the large number of nodes in the edge (∼100K+) as compared to transport and core (∼1K+). We also observed that each of the change activities is planned ahead with a typical advance time of 10 days. The advance planning enables operations teams to ensure their change schedules are conflict-free. Before the scheduled change is implemented, new conflicts can sometimes arise due to unplanned urgent changes or the lack of accounting of already planned changes. We compute the number of conflicts across all the node types and before Zapper had been deployed. We observe using our data that before Zapper, the conflicts were a significant fraction of the total number of change activities. A large number of conflicts implies the need for communication across teams to resolve them because if not resolved, it introduces a higher risk of service impact and increased time in troubleshooting. This highlights the importance of determining a conflict-free change schedule.

III. Zapper Design & Implementation

In this section, we describe the design and implementation of **Zapper** that accommodates a wide array of constraints, and provides a common, automated, and scalable solution to systematically create change deployment plans.

A. Solution Overview

Fig. 2 shows a flow diagram for network change schedule planning using Zapper. A user typically provides (a) list of nodes either in the form of instance IDs or selection criteria based on attribute definitions (e.g., all LTE eNodeBs with specific hardware version in New York region), (b) time interval to complete the change activity across those nodes, (c) expected

duration of the change activity for each node, and (d) list of scheduling constraints, including loader related constraints. Note that a loader can be an individual person or a team responsible for the change. Based on this input, our ***Change Management Schedule Optimization (CMSO)*** first queries the policy engine to obtain the planning constraints related to the input and then queries the topology and inventory service to identify the list of nodes to be scheduled (N_S) and dependent nodes (N_D). It then uses a central change repository to identify the pre-planned changes in the input time interval for N_S and N_D . These would be important either to de-conflict (i.e., avoid the time-slots) or, satisfy the planning constraints (e.g., schedule SIAD and its attached eNodeB on the same time-slot). CMSO then determines the change deployment plan that best conforms to the planning constraints. The constraints capture the risk of service impact and are derived using operational domain knowledge and experience. Some of the constraints can be violated before the change execution due to higher priority activities, alarms, unexpected node failures resulting in network capacity reduction, or changes in the availability of loaders. We execute a ***constraint verifier*** periodically as well as triggered by external events to discover any violations and then notify users accordingly and recommend re-scheduling.

B. Mathematical Formulation

In this section, we discuss the mathematical formulation for the time-slot assignment problem. The loader assignment problem is solved using a **mixed-integer linear program**. that is analogous to a transshipment problem with additional capacity constraints (see, e.g., Chapter 2 from [7]). We omit the models for the loader assignment problem due to space considerations.

We now present a mathematical formalization of the problem constraints for the time-slot assignment problem. Given a list of change activities on the list of nodes N_S to be scheduled, we wish to determine a discrete start time s_i for each $i \in N_S$ that encodes the time-slot where change i starts. The granularity of the schedule is defined by setting an appropriate time-slot. Because Zapper must handle a variety of devices and constraints, we have abstracted them to provide generalized solutions. For example, eNodeBs have a constraint that only a small number from any given region should be scheduled together to avoid coverage hole. A seemingly different constraint applies to servers where only a small number from a given pool can be scheduled together to avoid capacity hole. Instead of writing two different constraints, we provide general *constraint templates* that can capture both of them. For each $i \in N_S$ we associate a list of m attributes $a_i^{(1)}, a_i^{(2)}, \ldots, a^{(m)}$ encoding the properties of the change schedule; e.g., $a^{(1)}$ may represent the region associated with the network node for change i, its hardware/software type, or its EMS. The two example constraints above can be abstracted as a capacity bound on one of the attributes.

The network operations teams establish their desired granularity and select *high-level constraint templates* to impose constraints based on the attributes of the changes in N_S. In Zapper, we automatically map the high-level constraints to mathematical models. We use MiniZinc [8], an open-source constraint modeling language to construct the mathematical models for the following constraint types. Let p_i be the number of time-slots to perform change i.

General concurrency: A concurrency constraint [9], [10] imposes an upper bound on the number of change activities that can be performed per time-slot. Formally, let $a_i^{(\ell)}$ be the loader assigned to change $i \in N_S$ and define $N_S(\mathcal{L}) := \{i \in N_S \colon a_i^{(\ell)} \in \mathcal{L}\}$ the changes with loader in set $\mathcal{L}$. Then CONCURRENCY($\mathcal{L}$, C) is formulated as:

$$\sum_{i \in N_S(\mathcal{L})} \sum_{i' \in N_S(\mathcal{L}) \setminus \{i\}} \mathbb{I}\left([s_i, s_i + p_i) \cap [s_{i'}, s_{i'} + p_{i'}) \neq \emptyset\right) \leq C,$$

where $\mathbb{I}(C)$ is an indicator function that evaluates to 1 if the condition C is true and 0 otherwise.

We note that often finer concurrency control may be requested. For example, when upgrading eNodeBs, even though a loader can execute up to 100 upgrades in one time-slot, we may want to distribute them geographically so as not to create coverage holes. One way to assure this is to state that, among eNodeBs matching on attribute "city", we want to execute at most 30 in a time-slot. This is accomplished by defining a more general version ATTRIBUTE_CONCURRENCY($\mathcal{L}$, C, j, v) where, among all changes assigned to loader $\mathcal{L}$, at most C changes can be performed with $a^{(j)} = v$. The constraint above is written analogous but replacing $N_S(\mathcal{L})$ by $N_S(\mathcal{L}, j, v) := \{i \in N_S \colon a_i^{(\ell)} \in \mathcal{L}, a_i^{(j)} = v\}$ for the index sets. Another flavor is ATTRIBUTE_DISTINCT($\mathcal{L}$, C, j) that dictates that, among all changes assigned to loader $\mathcal{L}$, we have at most C distinct values on j-th attribute. E.g., on any given time-slot, we want to touch at most 5 markets.

Consistency constraint: Certain change activities need to be executed within a short time span between each other. For example, VNFs on the same host talking to each other may face software inconsistency if they are on different software versions. CONSISTENCY(j, m) ensures that any nodes which are within value m of each other on the j-th attribute should be scheduled at the same time. In particular, if the j-th attribute represents a location and $m = 0$, the constraint ensures that all changes on co-located nodes are scheduled at the same time. It is written as:

$$s_i = s_{i'}, \text{ for all } i, i' \in N_S \text{ s.t. } |a_i^{(j)} - a_{i'}^{(j)}| \leq m, i \neq i'.$$

Uniformity constraint: Executing changes is a tedious work that requires following a set of scripts. Thus, when loaders are performing changes, it is desirable that there is some uniformity in the work assigned to them. For example, it is best for loaders if changes are from similar hardware types or from the same time-zone. UNIFORMITY(j, m) states that all changes assigned to a given loader on the same time-slot should be within m of each other on the j-th attribute.

$$[s_i, s_i + p_i) \cap [s_{i'}, s_{i'} + p_{i'}) \neq \emptyset \Rightarrow |a_i^{(j)} - a_{i'}^{(j)}| \leq m, i \neq i'.$$

Load balancing constraint: While UNIFORMITY aspires to schedule similar changes together, LOAD_BALANCE spreads them out. One example relates to time-zones. For any given maintenance window, we prefer each loader to execute changes from one time-zone but, in aggregate (across all loaders), we do want to schedule changes that cover all possible time-zones to fully utilize periods of low traffic in all time-zones.

Localize constraint: Often teams wish to restrict changes. For instance, when upgrading to a new software release, teams

want to make sure that changes are introduced to geographic regions one at a time so that they have a chance to evaluate the outcomes of the updates and, if any problem occurs, they can locate the issue quickly. LOCALIZE(j) ensures that among nodes assigned to a given loader, we do not interleave changes that differ on values of attribute j.

Finally, we note that the objective can be easily written as a function of the variables s_i. For example, if the goal is to minimize the time the schedule finishes, the objective is $\min\{\max_{i \in N_S}(s_i + p_i)\}$. If the goal is to minimize the number of conflicts based on the dependent changes N_D, suppose $s_k,\ p_k$ represents the (fixed) start time and duration, respectively, of the dependent change $k \in N_D$. The objective is hence $\min \sum_{i \in N_S} \sum_{j \in N_D} \mathbb{I}([s_i, s_i+p_i) \cap [s_j, s_j+p_j) \neq \emptyset)$.

C. Optimization Solvers in Zapper

The mathematical formulation for the time-slot assignment problem can be directly addressed by a constraint programming solver (e.g., Gecode [11] or Google OR-Tools [12]) or, after an appropriate transformation, by a mixed-integer linear programming solver (e.g., COIN-OR Cbc [13]). We use the solvers available within MiniZinc to identify the change schedules.[5] Through experimentation, we observed that these solvers can effectively tackle small-scale networks (on the order of a few hundred nodes). This is typically the case for the core networks (hundreds of core routers). However, the edge networks of a large carrier typically has on the order of hundreds of thousands of nodes. The solvers within MiniZinc could not tackle the large scale of the edge network. So, we designed custom algorithms to address the problem.

Achieving scalable optimization. We implement a decomposition approach that partitions the problem into a *loader assignment* and a *time-slot assignment*, which are solved in sequence. Specifically, the loader assignment problem determines the maximum number of change activities at each time-slot so as not to violate loader-assignment constraints. This upper bound on the number of change activities per time-slot becomes input to the time-slot assignment problem that determines the exact start times of each change activity so as not to violate scheduling constraints. This decomposition greatly reduces the overall complexity of the problem and enables different solution methodologies to address each more efficiently. The time-slot assignment problem is further decomposed per attribute to create smaller instances of problems. Our approach works in two steps:

1. *Problem Segmentation:* Since the general problem can be difficult to handle, we create smaller instances based on network configuration attributes. In particular, we look for attributes that lead to independent subsets of nodes (in the sense that these subsets can be optimized independently); e.g., each EMS has independent capacity.
2. *Constrained model-driven optimization:* Based on the constraints and objective function specified by the network operations team, we implement a local search procedure [14] that evaluates permutations of N_S specifying the sequence of change activities to be performed in a deployment plan. For example, suppose that a team specifies that no two changes associated with distinct regions can be performed simultaneously (i.e., a localize constraint). This implies that, when grouping changes by regions, these groups must be performed in sequence. We search for distinct sequences that reduce the number of conflicting nodes. We then pack the changes as early as possible within the sequence specified during the search.

[5]We provide an example MiniZinc model in Appendix §IX.

We now present our custom algorithm for discovering the schedules for SIAD. SIAD operations teams incorporate (a) concurrency constraint at MTSO (A Mobile Telephone Switching Office consists of multiple SIADs), and group level (A SIAD group consists of multiple SIADs), (b) load balance constraint for the time-zone attribute, and (c) conflict constraints on the same SIAD as well as the connected eNodeBs and gNodeBs. Specifically, each MTSO consists of two groups of SIADs that can be scheduled in parallel. Each time-slot (typically, a maintenance window spanning hours), we are allowed to schedule up to M distinct MTSOs and within each scheduled MTSO, we can select up to G nodes from each of its groups. So altogether, we can schedule up to $2 \times M \times G$ nodes on a given time-slot. We have to pick MTSOs carefully because each of its groups may have different conflicts. If we exhaust (say) all nodes from the first group of a MTSO, we are losing some capacity because we can schedule at most G nodes (instead of $2G$ nodes) from this MTSO on all subsequent time-slots.

We propose a *probabilistic greedy* algorithm that makes decision for one time-slot at a time. On the j-th time-slot, we are trying to decide which MTSOs to schedule, and which nodes within those MTSOs should be picked up. We start by assigning a score to each unscheduled node with the idea that a lower score makes this node more attractive to schedule on the current time-slot. Score of node n on j-th time-slot is equal to (number of conflict free time-slots for n on time-slots $j+1, j+2, \ldots$) + NumberOfTimeSlotsLeft $\times$ conflict(n, j).

The first term counts how many more opportunities we have for scheduling this node in a conflict-free manner. A node that has very few conflict-free time-slots remaining should be scheduled with urgency. The second term adds an additional weight equal to number of time-slots remaining in the scheduling window if n has a conflict on j-th time-slot. This additional weight serves two purpose (a) ensures that any node without conflict has lower score than any conflicted node, (b) the scores for conflicted node are higher early on because we want to avoid scheduling a conflicted node in the hope that we may be able to find a conflict-free time-slot later on. Then for each group within a MTSO, we compute its score as the sum of lowest score of G (or leftover) nodes belonging to this group and the score of a MTSO as the sum of its two group scores. Finally, we pick M MTSOs of lowest scores.

One challenge we encountered with this algorithm is that it always picks G nodes from a group to be schedule, even when many of them may have conflicts. This creates a very tight schedule but may not provide the right trade-off in terms of minimizing conflicts by stretching the schedule a time-slot or two and by picking fewer than G nodes per group, we may have a much better schedule. So after computing scores of groups and MTSOs, we perform a probabilistic filtering by removing each conflicted node with a small probability that is proportional to the ratio of leftover scheduling capacity and the number of unscheduled nodes. E.g. if we can schedule

100 nodes each day and we have 5 days remaining and 300 unscheduled nodes, then this ratio is $500/300$ which suggests that we can be aggressive in dropping nodes in the hope that we can schedule them later.

IV. Zapper Evaluation

In this section, we present evaluation of Zapper using real-world data collected from a large cellular service provider. We used the following node types: (i) 76K eNodeB and 4K gNodeB (radio access network), (ii) 39K SIAD (transport), and (iii) 60 MME (core network). For all the figures, we present the normalized values for the deployment times.

A. Impact of Advance Planning on Change Deployment Time

We compare the change deployment time achieved using the advance planning in Zapper with the schedule discovery of nodes primarily at run-time (similar to Statesman [15]). We compute change deployment time as the difference between the first and the last change schedule time.

Statesman overview and adaptation. Statesman primarily considers conflict and capacity constraints, and not the other technological and operational constraints that our network has. In Statesman, each change request in run-time is executed if constraints are met. If constraints are not met, the change request is rejected and the request waits certain time before trying again. We adapted Statesman to incorporate the conflict and operational constraints required in our context and determined the total change deployment time if planning was conducted at run-time. We model all change requests coming at the same time, and the initial selection of nodes at run-time is **random** among nodes satisfying all the constraints. All subsequent selections conform to the set of constraints specified by the operations teams and remember what has been completed until previous iteration. Thus, think of this as a single permutation of the change schedule plans, which may or may not be optimal. In Zapper, we search over a large solution space of change schedule plans and pick the one that minimizes the utility (e.g., completion time). The key distinction is run-time would not know what is to be scheduled in the future, and thus cannot do a better packing of schedule to achieve the objective. For example, consider two work groups G_1,G_2 working on nodes A, B, and G_1 has precedence constraint A before B. Then with Zapper advance planning, we can have G_2 work on B, while G_1 works on A in the first slot, and then vice versa in second slot. The total completion time is 2 slots. However, with run-time, if it picks A for G_2 in first slot, then G_1 has to be idle in first slot to avoid conflict on A. Thus, the total completion time would be 3 slots. It is well-known that on-line algorithms almost always deploy worse solutions than off-line algorithms. For example, for simple scheduling problems such as job shop, the competitive factor is 2, which means that even the best known on-line algorithm can perform two times worse than an off-line algorithm [16].

Results. Fig. 3 compares the total deployment time for Zapper using advance planning versus run-time only planning (Statesman adaptation) across four node types. We observe that Zapper discovers the change deployment plan with shorter deployment time using advance planning. Moreover this difference gets more pronounced as the number of nodes or complexity of constraints go up, which suggests that, as networks

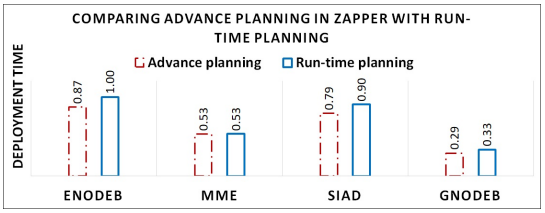

Fig. 3. Normalized deployment time using advance v/s. run-time planning.

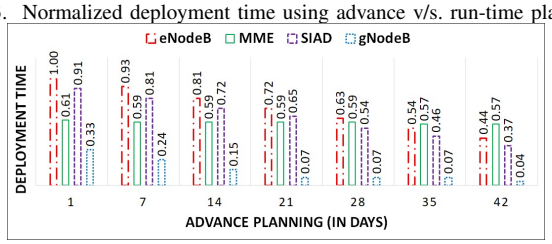

Fig. 4. Impact of advance planning on the normalized deployment time.

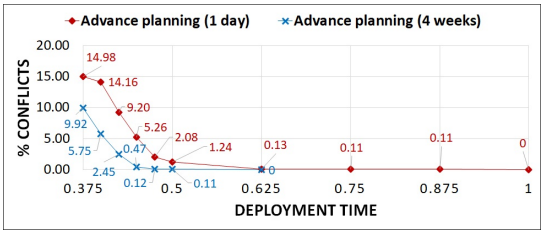

Fig. 5. Normalized deployment time increases as the conflicts reduce.

grow, simple schemes may not scale well and we will need more careful planning. The difference is higher specifically for eNodeB and SIAD that have many more instances than gNodeB and MME. The benefits with advance planning come from the early discovery of the change schedule and the higher likelihood of finding a compact schedule without conflicts.

Next, we quantify advance planning by the number of days in advance a network operations team member uses Zapper to request the change deployment plan. We observed that it was operational practice for teams responsible for major software releases to plan the network-wide deployment in advance, versus other teams handling configuration changes and hardware work would plan on a smaller scale and not much in advance. If these network changes are spread out, then multiple teams benefits from advance planning. Fig. 4 shows that our results using Zapper match our expectation. The total change deployment time decreases significantly for eNodeB, gNodeB, and SIAD. This is due to their large scale and frequent changes. However, for MME, the reduction is small because of the smaller scale and less frequent changes.

B. Impact of Conflicts on Change Deployment Time

We first ran Zapper with the goal of zero conflicts and identified the change deployment time. Next, we fixed different change deployment times and ran Zapper with the goal of minimizing conflicts. For each run, we identified the number of conflicting change activities and calculated the percentage of conflicts relative to the total number of planned changes. Fig. 5 shows the trade-off between deployment time and percentage of resulting conflicts for advance planning in Zapper with 1 day versus 4 weeks. We observed (a) an increase in the deployment time as the permissible conflicts decrease and (b) a better packing of schedule with an advance planning. The

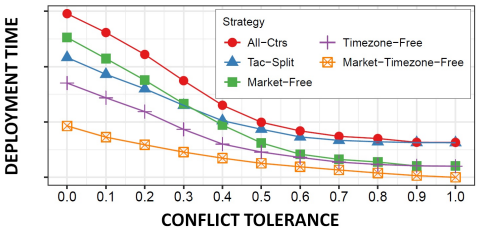

Fig. 6. Normalized deployment time increases with decrease in conflicts and increase in constraints.

reason is that scheduling of each change activity is critically dependent on scheduling of many other change activities. So finding a group of nodes to schedule on first time-slot is straight-forward, but gets increasingly harder towards the end.

C. Impact of Constraints on Change Deployment Time

Another aspect that we studied is how the number and types of scheduling constraints impact the quality of the change deployment plan. Usually, removing constraints makes it easier to find a good solution if we are finding exact solutions. Adding or removing constraints results in different problems, which may have different properties to be explored by the solver.

To illustrate this point, we consider the following (simplified) eNodeB scheduling problem: each eNodeB belongs to a Tracking Area Code (TAC), which belongs to a market, and multiple markets belong to a time-zone. We have the following constraints: 1) a CONCURRENCY constraint on how many eNodeBs can be scheduled within an EMS; 2) a CONSISTENCY constraint that eNodeBs from a TAC must be scheduled together; 3) a LOCALIZE constraint at the market level; and 4) a UNIFORMITY constraint to make sure that only eNodeB from adjacent time-zones can be scheduled together. We chose to relax some of these constraints (except the concurrency constraint), creating five different scenarios.

`All-Ctrs` is the regular optimization with all constraints activated. In `Tac-Split`, we allow splitting eNodeBs from a TAC (Tracking Area Code) on consecutive time-slots. In `Market-Free`, we remove the market localize constraint and allow to interleave eNodeBs from different markets. In `Timezone-Free`, we remove the timezone constraints, allowing eNodeBs from any timezone to be scheduled on the same time-slot. Finally, in `Market-Timezone-Free`, we remove all of TAC-Split, market, and timezone constraints, allowing great flexibility to the solver during the search. For most of our results, we have used the "deployment time" metric, which is the number of time-slots between start of schedule to when the last change gets scheduled. Sometimes, it is more interesting to look at the *average* deployment time where we measure the number of time-slots between start of schedule to when a change gets scheduled and then take an average over all change activities. Fig. 6 shows how average deployment time varies across different constraints types and for varying conflict tolerances. This result demonstrates the trade-off between reducing the change deployment time versus ensuring conformance to operational constraints and allowing conflicts. It is interesting that even without caring for conflicts, just the constraints put a lower bound on the deployment time and certain constraints make the scheduling harder than others.

D. Zapper Running Time

We compute the running time of Zapper from the time an operations team member submits a request and gets back change deployment plan that conforms to the input set of operational and conflict constraints. This time captures (i) inventory and topology lookup, (ii) change ticket management database lookup, and (iii) optimization time to discover the deployment plan. We deploy Zapper on Intel®Xeon®CPU E7- 4850 @ 2.00GHz with 40 cores and 512GB RAM. Since Zapper leverages problem segmentation and constrained model-driven optimization to scale up to a large number of nodes, we explored leveraging varying degrees of parallelism with multiple threads working in parallel for independent sets of network attributes. With x4 parallelism, Zapper was able to complete within three minutes even for 76K nodes. This demonstrates that Zapper is parallelizable and can discover change plans for a very large number of nodes in a matter of minutes which is orders of magnitude lower as compared to the total change deployment times (days to weeks) or the human times needed to discover the change plan without Zapper (days to weeks).

V. Operational Experiences

We now describe our experiences in applying Zapper for more than two years in operational cellular networks to schedule over 4.5 million change activities. The network operations teams use Zapper for planning change activities across the network for LTE eNodeB, 5G gNodeB, SIAD switch, and MME. We present results on operational efficiencies in terms of human time savings and reduction in the number of re-scheduling actions to co-ordinate the changes.

Human time savings. We interviewed the network operations teams (approximately 30 work groups) and identified the amount of time they had been spending on discovering the network change plans prior to Zapper. Before Zapper, an operations team member would manually identify a time slot for a subset of nodes that do not have any conflict and conform to constraints. We observed that different teams were spending different times for change deployment planning based on the complexity of the change activity, the number of nodes involved, and the type of nodes. With Zapper, the operations team member can now request the change plan across the whole network in a single request. For example, for a network size 100K, each group size (S_i) 1K, the number of iterations (N) would be 100. Each request of approximately 1K nodes without Zapper would take operations teams around an hour looking at databases or co-ordinating with other groups over chat windows to discover a schedule. With Zapper, they can generate the schedule for all 100K nodes in a few minutes. We used 24-months of Zapper usage to track S_i. Larger S_i results in higher time savings! We calculate the average human time savings as the percentage reduction in time as a result of using Zapper and taking into account the sample node size S_i in each request i ($i = 1..N$).

$$Time\ savings = \frac{\sum_{i=1}^{N} \left(\frac{T_{withoutZapper} \times S_i - T_{Zapper}}{T_{withoutZapper} \times S_i} \right)}{N} \quad (1)$$

Table I shows the average percentage human time savings realized after network operations teams started using Zapper across different types of change activities. Before Zapper, the

TABLE I
AVERAGE HUMAN TIME SAVED IN DISCOVERING CHANGE DEPLOYMENT PLANS USING ZAPPER.

Time savings	**# requests** (N)	**# changes** ($\sum_{i=1}^{N} S_i$)
85.59%	13,940	4,574,736

operations teams would take on the order of days or weeks to discover the change plan. This was reduced by several orders of magnitude to a few minutes or hours because of Zapper.

Reduction in re-scheduling actions. Before Zapper, the operations teams would manually discover the change conflicts and resolve them by co-ordinating and re-scheduling the conflicting change activities. With Zapper, they can discover the change time-slot that is conflict free and eliminate the need to manually discover and resolve the conflicts. We thus expect the number of re-scheduling actions for conflict resolution to decrease because of Zapper. We calculate the number of re-scheduling actions based on the changes in planned times recorded in the change ticket data for that change activity. We compare 24 months of Zapper with 24 months without Zapper by computing the percentage of change activities that require re-scheduling as compared to the total change scheduling requests. We observe that without Zapper, the re-scheduling actions were higher (19.2%) compared to with Zapper (13.83%). The re-scheduling actions did not go down to zero because some re-scheduling cannot be prevented due to emergency changes.

VI. PRACTICAL CONSIDERATIONS

Network dynamics. Network failures can reduce network capacity, and some nodes need to be re-scheduled. The constraint verifier in Zapper (§III-A) continuously looks for such changes in network that would result in constraint violations. The operations teams conduct pre-checks to proactively identify nodes that are impacted. They then use Zapper to re-schedule these nodes.

Roll-out co-ordination. At run-time, some changes can fail. The operations teams refer to them as *stragglers*, and conduct post-hoc analysis to understand their root-causes. Pre/post service impact analysis is performed using a separate capability [2], [3], [4] and if any unexpected impact is observed from changes, then the operations teams decide to halt the roll-out, or even roll-back changes. Zapper is used to determine the roll-back schedule.

Missing/delayed/incorrect data. Since we rely on configuration to derive inventory and topology, any missing, delayed or incorrect data can affect Zapper. We use a simple approach to fill any gaps in the current snapshot using information from previous snapshots as well as report any discrepancies to the operations teams for resolving any inconsistencies.

What if Zapper returns no schedule? We observed this early on in our Zapper deployment. We enhanced Zapper to relax some of the constraints by either returning a schedule with minimum conflicts (operations negotiate with other groups to reschedule), or adjusting their own schedule by softening some constraints (assign weights to different constraints). The "softening" of constraints was never done on the service risk related constraint.

A central repository for scheduled changes. This enables effective co-ordination across work groups. Fall-outs can occur if urgent schedule updates are not recorded and that can lead to last-minute unpleasant conflicts and in some cases negative service performance impacts. Zapper automatically populates this repository with computed schedule.

VII. RELATED WORK

Network change scheduling. Statesman [15] focuses on run-time conflict avoidance and network-wide safety and performance variants for network applications. Frenetic [17] and Pyretic [18] support modular composition and conflict resolution over specific OpenFlow rules. Athens [19] uses voting and non-rule based methods to resolve resource conflicts among SDN and cloud controller modules. Dionysus [20] uses node dependencies to automatically discover the schedule for network updates. Liu et al. [21] present linear programming based algorithm to discover the schedule for multi-flow updates in SDN networks. Concord [22] focuses on co-ordinating software upgrade roll-outs in cellular networks and aims to strike a balance between deployment time and service impact during the upgrade. Janus [23] is a change planner focusing on minimizing the service risk in data center networks. CloudCanary [24] focuses on real-time auditing to prevent correlated failures due to service updates and generating improved change plans. Zapper tackles a wide variety of constraints, uses a model-driven approach, and leverages advance time planning to do a better packing of change schedule as compared to run-time only planning.

Operations research. The optimization problem for loader and time-slot assignment is an extension of a classical scheduling problem [9]. Specifically, change activities and loaders can be perceived as tasks and machines, respectively. The schedule optimizer must hence assign tasks to machines within certain time periods to ensure operational restrictions are satisfied. Because a machine has a discrete limit on the number of simultaneous tasks (e.g., the *general concurrency* constraint), the problem is typically classified as a *cumulative* scheduling problem [10]. There are several optimization papers [25], [26], [27] to tackle constraint-based time-tabling. We differ from these works in applying optimization algorithms in Zapper for network change deployment planning by leveraging network configuration attributes and a wide variety of constraints.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a new solution Zapper for network change deployment planning that minimizes the service disruption and avoids conflicts across multiple work groups arising due to cross-layer and service-layer dependencies. We thoroughly evaluated Zapper using real-world network data collected from a large tier-1 service provider. We have deployed Zapper in a large service provider network and it is resulting in significant improvement in operational efficiency and change deployments. While our custom algorithms work well to address the large scale nature of the optimization, it currently assumes a *fixed* set of constraints. It remains to be a very interesting open research problem on making the constraint programming approach more *dynamic* so that we can automatically compose the set of constraints on the fly and more *versatile* so that we can apply our approach across a large number of network element types.

References

[1] D. Zhuo, Q. Zhang, X. Yang, and V. Liu, "Canaries in the network," in *ACM HotNets*, 2016, p. 36–42.

[2] A. Mahimkar, H. H. Song, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and J. Emmons, "Detecting the performance impact of upgrades in large operational networks," in *ACM SIGCOMM*, 2010.

[3] A. Mahimkar, Z. Ge, J. Wang, J. Yates, Y. Zhang, J. Emmons, B. Huntley, and M. Stockert, "Rapid detection of maintenance induced changes in service performance," in *ACM CoNEXT*, 2011.

[4] A. Mahimkar, Z. Ge, J. Yates, C. Hristov, V. Cordaro, S. Smith, J. Xu, and M. Stockert, "Robust assessment of changes in cellular networks," in *ACM CoNEXT*, 2013.

[5] H. Maylor, "Beyond the gantt chart:: Project management moving on," *European Management Journal*, vol. 19, no. 1, pp. 92–100, 2001.

[6] Wikipedia, "Siad - smart integrated access device," https://en.wikipedia.org/wiki/Integrated_access_device.

[7] M. Conforti, G. Cornuejols, and G. Zambelli, *Integer Programming*. Springer Publishing Company, Incorporated, 2014.

[8] MiniZinc, "MiniZinc - a free and open constraint modeling language," 2020, accessed on 2019-09-16. [Online]. Available: https://www.minizinc.org

[9] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 3rd ed. Springer Publishing Company, Incorporated, 2008.

[10] P. Baptiste, C. L. Pape, and W. Nuijten, *Constraint-Based Scheduling*. Norwell, MA, USA: Kluwer Academic Publishers, 2001.

[11] C. Schulte, G. Tack, and M. Z. Lagerkvist, "Modeling and programming with gecode," 2019. [Online]. Available: https://www.gecode.org/

[12] L. Perron and V. Furnon, "Or-tools," Google. [Online]. Available: https://developers.google.com/optimization

[13] COIN-OR, "Cbc: COIN-OR branch and cut," 2021. [Online]. Available: https://github.com/coin-or/Cbc

[14] F. Xhafa and A. Abraham, *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[15] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin, "A network-state management service," in *ACM SIGCOMM*, 2014.

[16] R. M. Karp, "Online algorithms versus off-line algorithms: How much is it worth to know the future?" 1992. [Online]. Available: http://www.icsi.berkeley.edu/pubs/techreports/TR-92-044.pdf

[17] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ACM SIGPLAN ICFP*, 2011.

[18] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *USENIX NSDI*. Berkeley, CA, USA: USENIX Association, 2013, pp. 1–14.

[19] A. AuYoung, Y. Ma, S. Banerjee, J. Lee, P. Sharma, Y. Turner, C. Liang, and J. C. Mogul, "Democratic resolution of resource conflicts between sdn control programs," in *ACM CoNEXT*, 2014, pp. 391–402.

[20] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *ACM SIGCOMM*, 2014, pp. 539–550.

[21] Y. Liu, Y. Li, Y. Wang, and J. Yuan, "Optimal scheduling for multi-flow update in software-defined networks," *J. Netw. Comput. Appl.*, vol. 54, no. C, pp. 11–19, Aug. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jnca.2015.04.009

[22] M. A. Qureshi, A. Mahimkar, L. Qiu, Z. Ge, M. Zhang, and I. Broustis, "Coordinating rolling software upgrades for cellular networks," in *IEEE ICNP 2017*, 2017, pp. 1–10.

[23] O. Alipourfard, J. Gao, J. Koenig, C. Harshaw, A. Vahdat, and M. Yu, "Risk based planning of network changes in evolving data centers," in *ACM SOSP*. Association for Computing Machinery, 2019, pp. 414–429.

[24] E. Zhai, A. Chen, R. Piskac, M. Balakrishnan, B. Tian, B. Song, and H. Zhang, "Check before you change: Preventing correlated failures in service updates," in *USENIX NSDI*, 2020.

[25] E. Demirovic and P. J. Stuckey, "Constraint programming for high school timetabling: A scheduling-based model with hot starts," in *CPAIOR*, 2018.

[26] S. Kristiansen, M. Sorensen, and T. R. Stidsen, "Integer programming for the generalized high school timetabling problem," in *Journal of Scheduling*, 2014.

[27] M. Banbara, K. Inoue, B. Kaufmann, T. Okimoto, T. Schaub, T. Soh, N. Tamura, and P. Wanko, "teaspoon: solving the curriculum-based course timetabling problems with answer set programming," in *Annals of Operations Research*, 2019.

IX. MiniZinc Model for MME Change Schedule Discovery

We present the MiniZinc code for conducting our model-driven constrained optimization for MMEs. Note that, although we format this model to better reading, it is indeed generated automatically on-the-fly by Zapper.

```
%%%%%%%%%% Parameters %%%%%%%%%%%%%%%%
int: NM; % number of MMEs.
int: NP; % number of MME pools.

% For each MME, indicates the pool ID for that MME.
array[1..NM] of 1..NP: pool_id;

% Index of last possible time-slot for scheduling.
% 1st time-slot is counted as 1; 2nd as 2 and so.
int: maxT;

% True/1, if i-th MME does NOT have a conflict on j-th time-slot.
array[1..NM, 1..maxT] of 0..1: noConflict;

% Maximum number of MMEs that can be scheduled on j-th time-slot.
array[1..maxT] of 1..NM: mmeSlotCapacity;

% maximum number of MMEs that can be scheduled from i-th pool on
% j-th time-slot. We must allow 0 (zero), to block that time-slot.
array[1..NP, 1..maxT] of 0..NM: poolSlotCapacity;

% Maximum number of different pools that can be scheduled on
% j-th time-slot. We must allow 0 (zero), to block that time-slot.
array[1..maxT] of 0..NM: poolSlotDistinct;

%%%%%%%%%% Decision variables %%%%%%%%%%%%%%%%
% TRUE/1, if i-th MME gets scheduled on j-th time-slot.
array[1..NM, 1..maxT] of var 0..1: SCHEDULED;

%%%%%%%%%% Constraints %%%%%%%%%%%%%%%%
constraint % Schedule only on noConflict time-slots.
forall(i in 1..NM, j in 1..maxT) (
  SCHEDULED[i,j] <= noConflict[i,j]
);

constraint % Schedule each MME exactly one time-slot (or none).
forall(i in 1..NM) (
  sum(j in 1..maxT) (SCHEDULED[i,j]) <= 1
);

constraint % Satisfy MME time-slot capacity.
forall(j in 1..maxT) (
  sum(i in 1..NM) (SCHEDULED[i,j]) <= mmeSlotCapacity[j]
);

constraint % Satisfy MME pool capacity.
forall(j in 1..maxT, k in 1..NP) (
  sum(i in 1..NM where pool_id[i] == k) (SCHEDULED[i,j])
  <= poolSlotCapacity[k,j]
);

constraint % Satisfy MME distinct pool bound.
forall(j in 1..maxT) (
  sum(k in 1..NP) (
    bool2int(sum(i in 1..NM where
  pool_id[i] == k) (SCHEDULED[i,j]) >= 1)
  ) <= poolSlotDistinct[j]
);

%%%%%%%%%% Objective function %%%%%%%%%%%%%%%%
% The time-slot in which the MME is scheduled.
array[1..NM] of var 0..maxT: SLOTS_SCHEDULED =
  [sum(j in 1..maxT) (j * SCHEDULED[i,j]) | i in 1..NM];

% Number of scheduled MMEs.
var int: NUM_SCHEDULED = sum(i in 1..NM)
  (bool2int(SLOTS_SCHEDULED[i] > 0));

% Sum of all completion times, used as proxy for the
% schedule length. In optimization, it is
% mathematically equivalent to the average.
var int: TOTAL_COMPLETION_TIME =
  sum(i in 1..NM) (SLOTS_SCHEDULED[i]);

% Bi-objective function such that we first try to
% schedule as many MMEs as possible,
% then minimize the total completion time.
solve maximize
  (maxT * NM * NUM_SCHEDULED) - TOTAL_COMPLETION_TIME;
```