Change Management in Enterprise IT Systems: Process Modeling and Capacity-optimal Scheduling

Praveen K. Muthusamy*, Koushik Kar*, Sambit Sahu†, Prashant Pradhan† and Saswati Sarkar‡
*Rensselaer Polytechnic Institute, Troy, NY 12180, USA, Email: {muthup,kark}@rpi.edu
†IBM TJ Watson Research, Hawthorne, NY 10532, USA, Email: {sambits,ppradhan}@us.ibm.com
‡University of Pennsylvania, Philadelphia, PA 19104, USA, Email: swati@seas.upenn.edu

***Abstract*—We provide a formal model for the Change Management process for Enterprise IT systems, and develop change scheduling algorithms that seek to attain the "change capacity" of the system. The change management process handles critical updates in the system that often use overlapping sets of servers, resulting in scheduling conflicts between the corresponding change classes. Furthermore, applications are typically associated with certain permissible downtime windows, which impose constraints on the timing of the change executions. Scheduling of changes for such systems represent a complex dynamic optimization question.**

In a limiting fluid regime, where changes are assumed non-atomic, we develop a scheduling policy that provably attains the change capacity of the system. We then propose and evaluate an atomic approximation of the optimal fluid scheduling policy, which is well suited for application to a real change management system. Simulation results demonstrate that the expected change execution delay and the capacity attained by the approximate policy is close to the best attainable values, when unavoidable capacity losses due to fragmentation effects are taken into account and is significantly better than a randomized scheduling policy.

I. Introduction

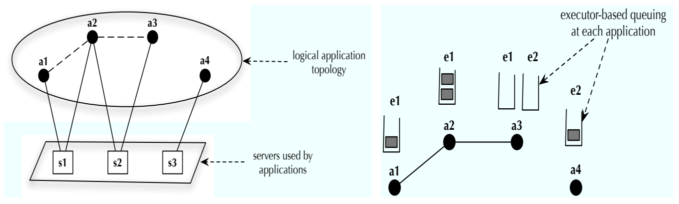
The management of today's Enterprise Information Technology (IT) eco-system is a very complex and challenging task due to (i) large number of subsystems involved, such as servers, network storage, firewalls, routers, (ii) interactions across multiple resources and domains, (iii) complex dependencies among applications stacks and IT resources, and (iv) Stringent downtime requirements as many of these applications are required to be available on a 24×7 basis. The two major management tasks in this context are: (i) Problem management – or handling of problem diagnosis and root cause analysis, and (ii) Change management – or timely implementation of updates to software stacks running on network devices, servers and storage.

The most critical component of the change management task is change scheduling. Our study is among the first ones to formally define the change scheduling process using mathematical foundations and analyze it to understand and propose improvements accounting for the practical constraints imposed in an Enterprise IT system. More precisely, we consider the problem of efficient management of a set of application *changes* under possible scheduling conflicts between them, and constraints on the change execution times. A change typically refers to a software update that must be implemented on a set of servers that depend on the specific application that the change is associated with. Scheduling conflicts among changes may be present, however, due to dependencies among applications and IT resources, availability of change executors (i.e., service specialists with appropriate skill sets) and additional constraints imposed due to service-level agreements. Moreover, each change must be implemented without interruption, and has an estimated time for completion, during which the application (and all other applications that use any server being updated) is unavailable to clients.

The key contribution of the paper lies in the novel quantitative formalization of the change scheduling process under very general and realistic assumptions and the characterization of the capacity-optimal change scheduling policy, albeit in the fluid regime. Our change scheduling objective is to attain the *change capacity* of the system, which represents (in loose terms) the ability to sustain the maximal set of change requests that can be scheduled by the system. (The notion of change capacity is defined precisely later in the paper.) Due to the atomic (i.e., indivisible) nature of the changes, and the asynchronous nature of the timing constraints on change execution times, the capacity-optimal change scheduling question in its generalized form is a very complex dynamic optimization problem. To obtain key insights to this difficult problem, and develop and motivate efficient approximate solutions, we analyze the scheduling question under an idealized *fluid* assumption, where changes are assumed to be *non-atomic* (or arbitrarily divisible). In this fluid system, we obtain a policy that is provably optimal in terms of change capacity. We then propose and study a approximate scheduling policy that tries to mimic the fluid system as closely as possible, while preserving the indivisibility of the changes. From a practical perspective, the excellent performance of the proposed atomic change scheduling algorithm (developed as an approximation to the capacity-optimal fluid scheduling algorithm) both in terms of attained capacity and expected delay, suggests that change management process based on this scheduling policy can be very effective in improving the performance of IT services that must be available to clients 24×7.

II. Change Scheduling Constraints

In this section, we briefly motivate the critical constraints on the change scheduling policy, which will be used later in the mathematical formulation. In the rest of the paper, we will associate changes with specific *applications*: our notion of an "application" is however very generic, and may stand for a user level process, a kernel level process, a database or protocol. Change implementation for any such application requires exclusive use of the "servers" that implement the application.


a1
a2
a3
a4
logical application topology
s1
s2
s3
servers used by applications
executor-based queuing at each application
e1
e2
e1 e2
e1
e2
a1
a2
a3
a4

(a) (b)

Fig. 1. Application conflict graph with change queues: (a) The sets of servers used by applications $a1, a2, a3, a4$ are $\{s1\}, \{s1, s2\}, \{s2\}, \{s3\}$ respectively; the dotted lines between applications indicate derived application conflicts; (b) The corresponding queuing architecture as needed by our algorithm, when executors $e1, e2$ can execute $\{a1, a2, a3\}$ and $\{a3, a4\}$, respectively.

Scheduling Conflicts: A scheduling conflict is said to exist between two changes if they can not be scheduled at the same time. Such conflicts typically arise due to the overlap in the resources on which they are to be implemented. Applications often use common server(s), thereby resulting in conflicts between the changes of those applications.

Timing Constraints: As discussed earlier, there can be constraints on when an application can be down ("application downtime"), to minimize the effect of the application downtime on the overall client base. Note that scheduling of an application change on a set of servers not only results in the unavailability of the application during the period the change is scheduled, but also that of all other applications that use any of the servers on which the change is implemented. As a result, an application change must be scheduled at a time that corresponds to a permissible downtime of that application, as well as all other "conflicting" applications.

III. Modeling and Formulation

A. System Model

Let N denote the set of applications that we consider, and let S be the set of all servers used by the applications. The set of servers used by application $i \in N$ is represented by $S_i \subseteq S$. Two applications are said to be in *conflict* with each other if they use any common server, i.e., $S_i \cap S_j \neq \phi$. Let $N_i \subset N$ denote the *conflict set* of application i, i.e., the set of applications (not including i itself) that conflict with application $i \in N$. Application change requests arrive at the change management/scheduling system according to a (typically unpredictable) random process. The goal of the change management system is to schedule these changes efficiently, in an online manner, without using any a priori knowledge of future change requests. Each change k is associated with a unique application, denoted by $N(k)$. A change k requires a finite time to be executed; although this time may not be known exactly in advance, it can be estimated, and these estimates can be used in making the scheduling decisions. Two changes k and k' are said to conflict with each other if the corresponding applications, $N(k)$ and $N(k')$, are in conflict with each other. Two conflicting changes cannot be scheduled at the same time. Since an application conflicts with itself (by default), two changes of the same application can not be scheduled together. The conflicts between applications can be conveniently modeled as a *conflict graph* $G = (V, L)$, where each vertex of the graph corresponds to an application ($|V| = |N|$), and an (undirected) edge exists between two vertices in G if they conflict with each other. Clearly, for a set of changes that can be scheduled at the same time, the corresponding applications constitute an *independent set* in G.

The definition of the scheduling conflicts between changes in our case, as described above, can be motivated as follows. Any change implementation/execution requires updating the servers that are used by the corresponding application. In other words, even though any one server may be used only for a part of the time over which a change is implemented, we assume that all servers in S_i are unavailable for any other purpose (including implementing other changes) during the entire period of time during which a change of application i is implemented. This implies that scheduling of conflicting changes can not be "pipelined" across different servers to improve efficiency. From this assumption, it follows that it is sufficient to consider the change scheduling question in the context of the *logical* application-topology instead of the *physical* server-topology. Thus, in the rest of this paper, we would consider and address the change scheduling question at this logical level (using the application-level conflict graph topology like the one shown in Figure 1(a)), without concerning ourselves directly with the servers on which these changes are physically implemented.

Each change is associated with timing constraints on when they can be executed, derived from allowed downtimes of the applications, as discussed next. We associate with an application $i \in N$ a set of allowed downtimes $\tilde{\Delta}_i = \{(s_i^1, t_i^1), (s_i^2, t_i^2), \ldots\}$ (where $s_i^m < t_i^m < s_i^{m+1}$, for all m). Typically, $\tilde{\Delta}_i$ will be a periodic pattern repeated on a daily or weekly basis (e.g., 2am to 4am every day), although periodicity of $\tilde{\Delta}_i$ is not required by our solution approach or analysis. Note that scheduling a change of application i also affects applications in its conflict set A_i, which are using servers in S_i (that remain unavailable during this change implementation). Therefore scheduling of a change of application i must also take into account the allowed downtimes of the applications in A_i. The *permissible time set* for (changes of) application i, i.e., times at which a change of application i can be in execution, is thus represented as $\Delta_i = \tilde{\Delta}_i \cap_{i' \in N_i} \tilde{\Delta}_{i'}$. With slight abuse of notation, we will also represent the permissible time set as a binary time-function $\Delta_i(\cdot)$, where $\Delta_i(t) = 1$ if t falls within an interval in the permissible time set, and 0 otherwise.

Our change management system is also associated set of change executors E, where $E_i \subseteq E$ represents the set of executors that are capable of implementing changes of application $i \in N$. In general, some executors may be capable of executing changes of multiple applications, while some changes can be implemented by multiple executors. In view of this, the change scheduling question also involves finding an *assignment* between applications and executors. The applications and executors being modeled as the two vertex sets of a bipartite graph, where an edge (i, j), $i \in N, j \in E$, represents the fact that changes of application i can be executed by executor j. Therefore, the set of changes scheduled at any time must correspond to a matching in the bipartite graph.

B. Capacity Region and Capacity-optimal Scheduling

The notion of capacity-optimal scheduling is based on the notion of a *capacity region*, which we define first. These notions are derived from [2], focused on throughput(capacity)-optimal scheduling in constrained queuing systems. For any application i, the change arrival process is random in nature, and has mean ρ_i, that represents the average "amount" of changes arriving per unit time. Since each change is associated with an execution time, the amount of changes arriving in any time interval equals the sum of the execution times of all changes arriving in that interval. Therefore, since (amount of) change arrivals are measured in units of time, the average rate of change arrivals for any application i, ρ_i, is a dimensionless positive real number. Let $\vec{\rho} = (\rho_i, i \in N)$ denote the vector of average change arrival rates. A change management (scheduling) system is said to be *stable* for a change arrival rate vector $\vec{\rho}$ under a scheduling policy ψ, if the backlog of all queues (or queue-lengths, measured in units of time) in the system at all times t remains upper bounded by a constant that is independent of t, when the change arrival rate vector is $\vec{\rho}$ and ψ is used as the scheduling policy. In such a case, scheduling policy ψ is said to *stabilize* the system for arrival rate vector $\vec{\rho}$. The *capacity region* of the system, denoted by Λ, is the set of all arrival rate vectors for which the system can be stabilized by *some* scheduling policy. Moreover, a rate vector outside the capacity region is not attainable, since all scheduling policies would lead to unbounded queues in the system for that arrival rate vector. Analytical characterization of the capacity region of the system can be found in the technical report [4]. A scheduling policy is said to be *capacity-optimal* if it stabilizes the change management system for all arrival rate vectors that are *strictly within* (i.e, *interior* of) the capacity region Λ.

IV. Change Scheduling Algorithms and Analysis

In this section, we first outline the optimal scheduling policy under the idealistic assumption that changes can be "broken up" arbitrarily. We provide a proof of capacity-optimality in this "fluid" regime, and use it to motivate a practical scheduling policy that respects change indivisibility, but is designed to closely approximate the fluid scheduling policy. There is a large body of work on the application of these optimality notions and Lyapunov techniques to wireless networks [3], [1]. In [1], the authors consider the scheduling question for multichannel wireless networks, but, the conflict constraints and mapping functions cannot be arbitrary unlike our model. In [3], the authors do not consider timing constraints.

A. *Optimal Scheduling in a Fluid System*

In the idealized fluid system, we assume that it is possible to schedule part of a change while keeping on hold the rest for later; in other words, changes are assumed to be non-atomic or arbitrarily divisible. Note that this fluid limit approximates the real scenario when the change execution times are much smaller than the duration of the permissible time windows of the applications, and allows us to neglect the capacity loss due to "fragmentation" effects. Loss due to fragmentation occurs, for example, when the permissible time window sizes are not a multiple of the change execution times. For example, if all the permissible time windows are 10 hours each, and each change takes 4 hours to execute, at least $(2/10) = 20\%$ of the change capacity of the system must be lost due to the atomic (indivisible) nature of the changes; this is referred to as "fragmentation loss". Therefore, the capacity-optimality results we derive using the fluid model holds in this limiting regime where fragmentation losses are absent. Thus the result we derive in this context also holds exactly in the special case of a slotted time system where all changes execution times are of unit duration (one slot), and the permissible time windows are slot-aligned and occupy an integral number of slots.

The change scheduling policy consists of two components that can be executed in parallel: (i) change queueing policy, and (ii) change selection policy. The queuing architecture involves executor-based queuing of changes at each application, i.e., an application i maintains E_i change queues, one for each executor that can schedule the changes of that application. Our queuing policy dictates that a change of application i is queued at one of the E_i queues immediately upon arrival. A queue for executor j at application i contains changes of i that will be executed by executor j. Let $Q_{ij}(t)$ denote the *backlog* of the queue for executor j at application i, at time t, representing the *time* it would take to schedule the changes backlogged in the queue. Under the fluid assumption, changes remaining in the queue can be fractional, and therefore, Q_{ij} is in general a real positive number. In our change queuing policy, incoming changes of application i are buffered at the queue j with the smallest value of the backlog, among all $|E_i|$ queues of that application, i.e $j = \arg\min_{j' \in E_i} Q_{ij'}(t)$.

We now describe the change selection process of the change scheduling policy. The change schedule is re-computed each time any of the timing constraints change, i.e., the schedule re-computation is performed when the permissible time window function $\Delta_i(t)$ changes (from 1 to 0, or 0 to 1), for any application i. Let $\tau_1, \tau_2, \ldots, \tau_m, \ldots$ denote the successive time instants at which any of the timing constraints of the system change. Then at any τ_m, the *Capacity-optimal Fluid Scheduling (CFS)* algorithm schedules *a set of non-conflicting queues such that they maximize the sum of the queue backlogs*. A schedule computed at τ_k will remain in effect from τ_m to τ_{m+1}, irrespective of change arrivals during this interval. At τ_{m+1}, a new schedule is recomputed taking in account the new timing constraints and the updated backlog values. Let $\mathcal{U}_m$ be a collection of all sets of queues that are *schedulable* during $(\tau_m, \tau_{m+1}]$. In other words, each element $U \in \mathcal{U}_m$ must satisfy the following two conditions: (i) all queues in U must be non-conflicting, i.e., for any two queues $(i,j), (i',j') \in U$ (where $j \in E_i$ and $j' \in E_{i'}$), $i \neq i'$, $j \neq j'$, and $i' \notin A_i$ (or equivalently $i \notin A_{i'}$); (ii) the permissible time window of the corresponding applications must cover $(\tau_m, \tau_{m+1}]$, i.e., for each queue (i,j) in U, $\Delta_i(t)$ must be 1 for all $t \in (\tau_m, \tau_{m+1}]$. Then the CFS policy corresponds to picking a schedule U_m^* at τ_m, where U_m^* is defined as follows:

$$U_m^* = \arg\max_{U \in \mathcal{U}_m} \sum_{(i,j) \in U} Q_{ij}(\tau_m). \tag{1}$$

Note that computing U^* corresponds to finding a *maximum weighted independent set* in the queue conflict graph, with the backlog Q_{ij} as the queue (node) weights. Our scheduling policy is very intuitive, as it attempts to give preference to queues with larger backlogs in selecting the changes to be executed. Figure 1(b) shows the queuing architecture for the system shown in Figure 1(a), and there are two executors $e1$ and $e2$, which can execute applications $\{a1, a2, a3\}$ and $\{a3, a4\}$, respectively. Since changes of application $a3$ can be executed by both $e1$ and $e2$, it maintains two queues.

Capacity-optimal Fluid Scheduling (CFS)
Change Queuing: For any application $i \in N$, buffer each incoming change of i at the queue (among all $|E_i|$ queues of application i) that has the smallest backlog.
Change Scheduling: At each $\tau_m, m \geq 0$, do the following:

1) Remove from consideration all queues of applications i which cannot be scheduled during $(\tau_m, \tau_{m+1}]$, i.e., $\Delta_i(t) = 0$ for $t \in (\tau_m, \tau_{m+1}]$.
2) Construct an extended conflict graph G' on the remaining queues, such that a conflict between any two queues (i, j) and (i', j') exists iff, either applications i and i' are the same or conflict with each other, or $j = j'$.
3) For each remaining queue (i, j), associate a weight equal to its backlog Q_{ij}.
4) Compute the maximum weighted independent set in G', with the queue backlog values as the node (queue) weights, as in (1). Let U^*_m denote this independent set.
5) Use U^*_m as the change schedule during $(\tau_m, \tau_{m+1}]$.

B. Capacity Analysis

The above scheduling policy can be shown to attain the maximum change capacity of the fluid system, as we formally state below. Although not necessary for stability, for simplicity of analysis we assume that the arriving changes only enter the queues at time instants $\tau_1, \tau_2, \ldots$. Our theoretical result also requires certain weak assumptions on the evolution of the permissible time windows and the change arrival processes. For any application i, the permissible and non-permissible time windows occur at a minimum granularity of δ ($\delta > 0$), bounded in length, and follow the strong law of large numbers (see [4]). We require that the amount of change arrivals of any application over any permissible or non-permissible time window is upper bounded, and the change arrival processes satisfy the strong law of large numbers (see [4]). Under these assumptions, we can show the following result.

Theorem 1: The scheduling policy CFS stabilizes the system for all change arrival rate vectors $\vec{\rho} \in \text{Int}(\Lambda)$.

[4] provides the complete proof of Theorem 1. Theorem 1 states that CFS stabilizes the system for all arrival rate vectors that are strictly within the capacity region, or in other words, CFS is capacity-optimal.

C. Scheduling with Indivisible Changes

In practice, execution of changes must be done atomically, i.e., once started, a change must be executed to finish without interruption. Next we present an approximation to the optimal fluid scheduling algorithm, CFS, that respects the indivisibility of the changes. The resulting algorithm, which we call *a-CFS*, attempts to schedule changes on an one-by-one basis so as to myopically mimic CFS as closely as possible. More specifically, the algorithm a-CFS works as follows. a-CFS maintains the same queueing architecture, and uses the same queuing policy on change arrivals as CFS. a-CFS runs CFS on the side, and keeps track of how much service time each of the queues (i, j) have received under both CFS and a-CFS. In its change selection policy, any time a scheduling opportunity is available, a-CFS gives preference to the queue(s) whose service (under a-CFS) is lagging the most from the service received by it under CFS (ties are broken arbitrarily). The scheduler also ensures that the change selected is such that its execution can be completed before the current permissible time window for the corresponding application closes. Note that a scheduling opportunity arises (i.e., a-CFS needs to select change(s) to be scheduled) only when one of the following events happen (i) An executor becomes available due to the completion of a change it was handling, (ii) The timing constraints change (i.e., the current time t corresponds to one of $\tau_1, \tau_2, \ldots$), (iii) a change arrival occurs in a queue associated with an executor who is currently idle. Also note that in any scheduling opportunity, a-CFS may end up scheduling zero, one or more changes. In any scheduling opportunity, we ensure that the resulting schedule is *maximal*, by iteratively running its change selection policy until no more changes can currently be scheduled without violating the conflict constraints on the queues or the timing constraints on the applications.

The a-CFS algorithm is detailed below. Here, $W_{ij}(t)$ and $\hat{W}_{ij}$ respectively denote the service provided to queue (i, j) by CFS and a-CFS, respectively, until time t. The change queuing policy in a-CFS remains the same as in CFS, so we only describe the change scheduling policy below.

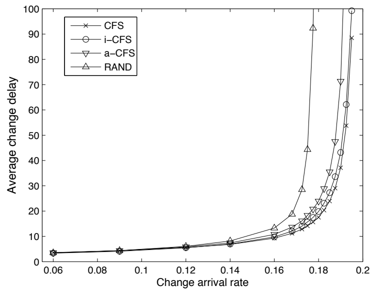
Approximate CFS (a-CFS)

- Simulate the CFS algorithm.
- At each scheduling opportunity, do the following:
  1) Remove from consideration all queues (i) which are currently in schedule or conflict with a queue currently in schedule, or (ii) whose permissible time set does not include the current time t, or (iii) whose head-of-line change cannot be executed before its current permissible time window closes.
  2) For each remaining queue (i, j), calculate the *lag value* L_{ij} as $L_{ij}(t) = W_{ij}(t) - \hat{W}_{ij}(t)$.
  3) Until schedule is maximal, iteratively add eligible queues to the schedule, in decreasing order of lag, while ensuring that the added queue does not conflict with any queue already in the schedule.

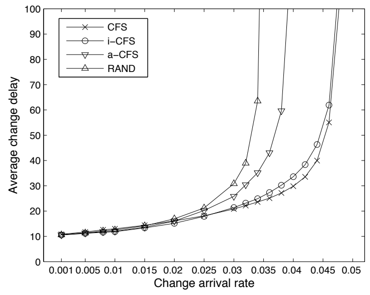
Computing the maximum weighted independent set, as required by CFS (and therefore by a-CFS as well) does not pose a serious limitation in Enterprise IT systems, since the number of change classes (applications) and executors are typically small. Executor availability constraints can also be easily incorporated into our formulation and solution.

V. Simulation Evaluation

In this section, we evaluate the performance of CFS and a-CFS algorithms, in terms of metrics like change capacity and expected change delay. The performance difference of a-CFS from that of CFS originates from two factors which make a-CFS non-optimal: (i) a-CFS performs scheduling one (or a few) changes at a time, and (ii) capacity loss due to fragmentation. To understand the effect of (i) and (ii) separately, we compare CFS and a-CFS with an algorithm where the non-optimality is only due to (i), but not (ii), since capacity loss due to fragmentation is avoided. This algorithm,



(a)



(b)

Fig. 2. a) Change delay comparison without timing constraints, b) Change delay comparison with timing constraints.

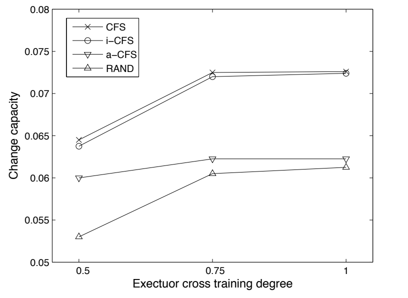


Fig. 3. Effect of executor cross-training degree on change capacity.

which we call i-CFS (or *incremental* CFS), schedules changes one-by-one as in a-CFS, but breaks up changes when the current permissible time window of the application closes. We also compare these algorithms with a baseline randomized algorithm, RAND. RAND works like a-CFS expect that when a scheduling opportunity becomes available, the change selection is done at random from the set of available changes. Comparison with RAND is intended to show the importance of picking changes based on lag values.

Simulations were carried out in the system with 4 applications shown in Figure 1. There are 2 possible maximal independent sets: $\{a1, a3, a4\}$ and $\{a2, a4\}$. Note that the mapping between the applications and executors is varied from that specified when we study the effect of executor cross-training on the performance. In our simulation setup, changes arrive in a continuous manner, according to a poisson process. The execution time of the changes arriving at $a1, a2, a3, a4$ are $1, 2, 3$, and 4 hours, respectively. For the simulations where we study the effect of timing constraints, the position of the permissible window during a day is chosen at random, for each application. The duration of the time window is set to either 8 or 10 hours. All delay results shown are obtained by averaging over at least $100,000$ hours to ensure steady state.

We first plot the average change delay as the rate of change arrivals is increased, in the baseline case when there are no timing constraints on the applications (i.e., the applications can be scheduled anytime), as shown in Figure 2(a). The change delay increases smoothly as the arrival rate increases, till the change capacity is reached. We observe that while all the algorithms perform similarly at low values of the change arrival rate, the average change delay curves for the different algorithms show divergence as the arrival rate increases. However, note that performance of i-CFS and a-CFS are very close to each other – this is expected since there are no capacity losses due to fragmentation in this case. These two algorithms also perform very close to the optimal fluid scheduling algorithm, CFS. Figure 2(b) shows similar results, but for the case with timing constraints. The permissible time windows chosen for the applications $a1, a2, a3, a4$ were $\{0-8\}, \{2-10\}, \{4-12\}, \{6-16\}$ respectively. In this case, note that i-CFS again performs very close to CFS. Thus performing change scheduling incrementally (one-by-one) based on lag values does not lead to significant degradation in performance compared to the optimal fluid policy, in absence of capacity losses due to fragmentation. The effectiveness of doing scheduling based on the lag values is demonstrated by the fact that a-CFS performs significantly better than RAND in terms of delay at high loads, and the change capacity attained. Note that the difference in the attained capacity that we see between a-CFS and CFS or i-CFS is due to capacity loss due to fragmentation effect. Figures 2(a) and 2(b) also show that the change capacity attained in the system (i.e., the value at which the delay "blows up") by i-CFS is very close to that of CFS. The capacity attained by a-CFS is lower than CFS and i-CFS due to fragmentation losses, but significantly better than that of RAND.

Figure 3 shows the change capacity attained, as the executor cross-training degree increases, while all other parameters remain fixed. The cross-training degree is computed as the average number of applications that an executor is capable of handling. The x-axis shows the cross-training degree expressed as a fraction of the maximum value (which is 4, since there are 4 applications in system). The comparative performance of the algorithms is similar to that observed earlier. We observe that the attained capacity improves as the cross-training degree increases, as we intuitively expect. However, the marginal benefits of cross-training show "diminishing returns", demonstrating that a moderate degree of cross-training provides an efficient way of attaining close-to-optimal performance.

The change capacity characterization that we provide, can be useful in answering system provisioning questions, such as the number of executors or the degree of executor cross-training required to attain a certain change capacity. Finally, note that in this work we do not explicitly attempt to minimize the expected change delay (which is a related but different, and possibly more complex, optimization question that remains open). However, capacity-optimal algorithms can be expected to result in low delays at high load, which is what we observe in our simulations when we compare the performance results of the a-CFS and RAND algorithms.

References

[1] K. Kar, X. Luo and S. Sarkar, "Throughput-optimal Scheduling in Multichannel Access Point Networks under Infrequent Channel Measurements", *Proc. IEEE Infocom 2007*, Anchorage, AK, May 2007.

[2] L. Tassiulas and A. Ephremides, "Stability properties of queueing systems and scheduling policies for maximum throughput in multihop radio networks", *IEEE Trans on Automatic Control*, 37(12),1992.

[3] L. Tassiulas, P. Bhattacharya, "Allocation of interdependent resources for maximum throughput," *Stochastic Models*, Vol. 16, No. 1, 2000.

[4] P. Kumar *et al.*, "Change Management in Network Services: Process Modeling and Capacity-optimal Scheduling", Technical Report, www.ecse.rpi.edu/∼koushik/TechRep-ChangeSched.pdf.