University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

CSE Technical reports | Computer Science and Engineering, Department of

2011

Response Time Analysis of Hierarchical Scheduling: the Synchronized Deferrable Servers Approach

Haitao Zhu
University of Nebraska-Lincoln, hzhu@cse.unl.edu

Steve Goddard
University of Nebraska – Lincoln, goddard@cse.unl.edu

Matthew B. Dwyer
University of Nebraska-Lincoln, matthewbdwyer@virginia.edu

Follow this and additional works at: http://digitalcommons.unl.edu/csetechreports

Zhu, Haitao; Goddard, Steve; and Dwyer, Matthew B., "Response Time Analysis of Hierarchical Scheduling: the Synchronized Deferrable Servers Approach" (2011). *CSE Technical reports*. 145.
http://digitalcommons.unl.edu/csetechreports/145

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Technical reports by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

Response Time Analysis of Hierarchical Scheduling: the Synchronized Deferrable Servers Approach *

TR-UNL-CSE-2011-0006

Haitao Zhu, Steve Goddard, Matthew B. Dwyer
University of Nebraska - Lincoln, Lincoln, NE 68588
{hzhu,goddard,dwyer}@cse.unl.edu

Abstract

Hierarchical scheduling allows the use of different schedulers and provides temporal isolation for different applications on a single hardware platform. We propose a hierarchical scheduling interface called synchronized deferrable servers. In addition to the common advantages of hierarchical scheduling, synchronized deferrable servers can combine partitioned and global multiprocessor scheduling in one system, and increase the schedulable system utilization. Response time analysis of tasks executed by these servers is presented and evaluated through simulation. We show that evenly allocating bandwidth across cores is "better" than other allocation schemes in terms of a task set's schedulability. In addition, under hierarchical scheduling, the threshold between lightweight and heavyweight tasks may be different from what they are under dedicated scheduling.

1 Introduction

Significant research has been done on hierarchical scheduling [14, 16, 17, 28, 30]. In hierarchical scheduling, an *interface* specifies how processor resources are provided over time, e.g., "a server provides capacity of 5 time units at a period of 20 time units." While designing hierarchical scheduling, the designers need to choose an interface that can meet the application requirements. In practice, it is desirable to choose interfaces that are easy to implement while permitting schedulability to be guaranteed.

With the increasing use of multi-core architectures in real-time systems, multiprocessor scheduling has received growing attention. Conventional multiprocessor scheduling can be divided into two categories: *partitioned scheduling* and *global scheduling*. Under partitioned scheduling, tasks are statically bound to a processor, and cannot migrate across different processors. In contrast, under global scheduling, tasks can migrate across different processors. It is well-known that both partitioned and global scheduling have their own advantages and disadvantages, and neither dominates the other. Hierarchical scheduling allows the combination of partitioned and global scheduling in one system, for example, hierarchical scheduling in an earliest-deadline-first (EDF) system has been reported in [30].

We propose a multi-core hierarchical scheduling interface for *fixed-priority preemptive* systems called Synchronized Deferrable Servers (SDS). We consider the deferrable server in this paper as it is a bandwidth-preserving technique with good performance and low implementation complexity [12]. Our work can be generalized to non-bandwidth-preserving servers such as periodic servers, which will be discussed in Section 5.3. In an SDS interface, each core[1] hosts one Deferrable Server (DS) [32] with the same period and possibly different capacity. In this paper we focus on the case where only one highest-priority DS exists on a core, while presenting generalization to arbitrary-priority mulitple DS on a core in Section 5.1. The tasks in a system are divided into two categories: the *migrating* tasks and the *non-migrating* tasks. A migrating task can migrate across different cores, and thus can be processed by different DS. A non-migrating task is not processed by SDS, and is statically bound to a core with migration disallowed.

While clock synchronization is in general difficult for multiprocessors *without a global clock*, various multi-core architectures with a global clock for all cores have been used in real projects.[2] A periodic timer can then be used to achieve tight synchronization of the SDS. Since the servers are bandwidth preserving, the primary source of synchronization error is delay in capacity replenishment. The periodic timer eliminates accumulated drift from a common period (except for clock drift). Under this scenario, the degree of synchronization achieved is the same as the synchronization of a single DS to its replenishment period or a set of tasks to their release periods.

Recent advances in multi-core architectures allow applications built from a diverse collection of tasks to be realized

*This material is based in part upon work supported by the National Science Foundation under Awards CCF-0541263 and CNS-0720654 and by the National Aeronautics and Space Administration under grant number NNX08AV20A.

[1] We shall interchangeably use the terms *processor* and *core* in the rest of this paper.

[2] In a communication with Brandenburg, the maintainer of $LITMUS^{RT}$ project [1], he wrote: "...The supported x86 and ARM platforms (as well as the UltraSPARC platform supported in prior versions) all have a global clock signal that is accessible to all processors (in the form of cycle counter registers). Linux bases its notion of time on this global clock source, so there is no drift among processors."

on a single hardware platform. For some applications, migrating tasks across different cores may incur significant overhead, e.g., from cache misses, and these tasks should be statically bound to fixed cores. For other tasks, e.g., those that process volatile data, allowing migration offers the usual advantages of global scheduling.

An example demonstrating this diversity of tasks arises in run-time monitoring [2]. In run-time monitoring, a set of *monitor tasks* are used to collect and process *events* generated by *target tasks*. To overcome the overhead incurred by monitoring activities, researchers have exploited multiple execution cores to *hide* monitoring costs. The work in [22] proposes to use a dedicated core for monitoring. While dedicating an entire core is clearly a simple solution, it is not the most resource-efficient strategy.

In predictable monitoring [33], one is interested in guaranteeing that errors detected through monitoring will be reported within a bounded latency—latency is directly related to the maximum response time of the monitoring tasks. Effective CPU utilization will enable more errors to be detected within their prescribed bounds. To achieve this, rather than dedicate a single core to monitor tasks one might instead spread the total monitoring bandwidth across each core. This resource efficient solution can be achieved using the relatively simple interface of SDS by first partitioning the target tasks among all the cores, and then use the unused but available bandwidth on each core for monitoring tasks.

Our Contributions: First, we propose a new multi-core hierarchical scheduling interface that allows the use of partitioned and global scheduling in one system. In conventional partitioned scheduling, some cores may have available but unused processor bandwidth after tasks are partitioned. Our interface can collect this bandwidth for migrating tasks and thus improve system utilization.

Our second contribution is the Response Time Analysis (RTA) of multi-core hierarchical scheduling for fixed-priority preemptive systems. Unlike existing RTA [13, 20] for identical multiprocessor *dedicated scheduling*[3], our RTA can be applied to multi-core systems where each core provides different bandwidth. We present a sufficient condition to bound from above a task's response time, which is also applicable to identical multiprocessor dedicated scheduling considered in [13, 20]. In addition we show that, under hierarchical scheduling, a task's response time is affected by lower-priority tasks as well as higher-priority tasks.

Our third contribution is a demonstration that, given a fixed amount of total bandwidth on all cores, evenly allocating bandwidth across cores is superior to approaches that dedicate full bandwidth on individual cores. Thus, for improved schedulability, dedicating entire cores for a whole migrating task sets, which seems a "natural choice" in engineering practice, should be avoided.

Finally, our experimental results reveal *heavyweight tasks*' effect on a task set's schedulability. Heavyweight tasks have utilization or density exceeding a certain threshold. We show that, under hierarchical scheduling, the threshold for judging a task as heavyweight depends not only on its utilization or density, but also on the average bandwidth per core, which holds even when some cores are dedicated to tasks.

This paper is organized as follows. We introduce related work in Section 2, and in Section 3, the background and system model. In Section 4, we present our RTA for SDS. In Section 5, we discuss how to generalize our work to the cases where each DS has an arbitrary priority or periodic servers are used instead of DS. In Section 6, we present our evaluation results, and based on these results, we discuss bandwidth allocation schemes and the effect of heavyweight tasks on tasks' schedulability. Section 7 concludes this paper and identifies future work.

2 Related Work

Significant research has been done on hierarchical scheduling [14, 16, 17, 28, 30]. On uniprocessors, RTA of preemptive tasks executed by fixed-priority bandwidth-preserving servers, e.g., DS, has been studied [14,16]. These approaches are based on the fact that the job with the maximum response time is released at a *critical instant*. In general, for multiprocessor scheduling, the critical instant of a task is unknown. Therefore, these approaches for uniprocessors cannot be applied to multiprocessor scheduling.

Baruah *et al.* studied Constant-Bandwidth Servers and Total Bandwidth Server on dynamic-priority multiprocessors [10,11]. Recently, a hierarchical scheduling framework combining partitioned and global scheduling on EDF systems is studied by Shin *et al.* in [30]. A more general framework is proposed by Lipari and Bini [28], which allows designers to trade off resource usage and flexibility in determining virtual platform parameters.

While there are other significant works on hierarchical scheduling, we are not able to list them all in this paper. However, to the best of our knowledge, none of them present RTA for multiprocessor hierarchical scheduling.

RTA for identical multiprocessor dedicated scheduling is studied in [13, 20]. The essential idea of these approaches is to bound from above higher-priority tasks' *interference* on the task of interest. However, these approaches cannot be applied to hierarchical scheduling.

A recent research topic is *semi-partitioned* scheduling [3, 4, 21, 24–26]. Both semi-partitioned scheduling and our hierarchical scheduling interface combine the use of partitioned and global scheduling in one system, and can improve system utilization. However, our work overcomes the limitations of semi-partitioned scheduling. First, the known semi-partitioned scheduling algorithms cannot be applied to dynamic task systems where tasks may join

[3] In this paper, dedicated scheduling means conventional scheduling where 100% bandwidth of each core is dedicated to process tasks, i.e., hierarchical scheduling is not used.

or leave the system during execution. If the task set is changed, the whole task set must be re-partitioned. Second, semi-partitioned scheduling algorithms assume that all tasks can migrate at the partitioning phase, even though only a subset of the tasks will migrate during execution. Third, semi-partitioned scheduling assumes each core dedicates full CPU bandwidth to tasks, and thus no temporal protection among different applications is provided.

In global dedicated scheduling for identical multiprocessors, the "Dhall Effect" refers to the fact that, when heavyweight tasks exist, the task set is less likely to be schedulable, even if the average task utilization is low. To circumvent this effect, researchers have invented algorithms that handle heavyweight tasks differently from lightweight tasks [7,8,19,31]. While the threshold for judging heavyweight tasks varies in previous work, determining that threshold does not involve the bandwidth of the processors. For example, in [7, 8, 31], a utilization of 0.5 is usually regarded as the threshold of being heavyweight or not. Our results show that determination of this threshold must consider both utilization and processor bandwidth.

3 Background and System Model

A DS [32] is a described by a 2-tuple (T_S, C_S), where T_S is the replenishment period, and C_S is the maximum capacity provided by the DS in a replenishment period. A DS works as follows. When a DS with available capacity obtains the CPU, it processes pending workload; if no workload is pending, it simply holds its capacity. A DS will be suspended if it exhausts the maximum capacity of C_S time units in a replenishment period, and then waits for the next replenishment. Take as time 0 a DS's first replenishment, it will replenish its capacity with C_S time units at time $i \cdot T_S$, $i \in \mathbb{N}_0$, and any unused capacity before a replenishment will be discarded.

We extend the DS concept to multi-core systems, and propose the concept of SDS. A set of m SDS, denoted by m-SDS, consists of m DS (T_S, C_S^i), $1 \leq i \leq m$, where T_S is the common replenishment period, and C_S^i is the maximum capacity of the i-th DS, denoted by s_i, in a replenishment period. Since each DS has the same replenishment period, an m-SDS can be described by an $(m+1)$-tuple $(T_S, C_S^1, C_S^2, ...C_S^m)$. Without loss of generality, we order the DS such that, $\forall i$, $C_S^i \geq C_S^{i+1}$. Each DS consumes and replenishes capacity like a conventional uniprocessor DS.

While it is possible to have multiple sets of SDS each set of which has a different replenishment period in one system, in this paper we focus the case where only one set of SDS exists, and each DS has the highest priority on its host core. In practice, a system designer has the freedom of choosing a DS's replenishment period. To use optimal fixed-priority scheduling algorithms, such as Rate Monotonic (RM), one can choose a sufficiently short replenishment period. In Section 5.1, we present generalization to arbitrary priority DS and multiple DS on one core.

3.1 Non-migrating Tasks and Migrating Tasks

All tasks in the system are preemptive, and are divided into two categories: *non-migrating tasks* and the *migrating tasks*. A non-migrating task is *not* processed by the SDS, and is statically bound to a core with migration disallowed.

A migrating task can migrate across different cores, and thus can be processed by different DS. Let n be the number of the migrating tasks, and the i-th migrating task is denoted by τ_i. Each migrating task is modeled as a sporadic task (T_i, C_i, D_i) where T_i is the minimum inter-arrival time, C_i is the Worst Case Execution Time (WCET), and D_i is the relative deadline. In practice, migration of a task has a certain overhead; and as in [15], the "cost of pre-emption, migration, and the runtime operation of the scheduler is assumed to be either negligible, or subsumed into the worst-case execution time of each task." In this paper, we consider *constrained-deadline* systems where $D_i \leq T_i$, while deferring the discussion of $D_i > T_i$ to future work. Each migrating task is assigned a priority such that, $\forall i, \tau_i$ has a higher priority than τ_{i+1}.

In our system model, τ_i cannot be processed by two or more DS *at the same time*.

In this paper, we focus on the RTA of *migrating* tasks, and when no confusion arises, we shall refer to a migrating task simply as a task. The schedulability analysis of *non-migrating* tasks is essentially a uniprocessor scheduling problem, which has been well-studied [5, 23, 27].

3.2 Scheduling Policy

The scheduling policy of SDS consists of two levels: *intra*-core scheduling, and *inter*-core scheduling.

The intra-core scheduling policy utilizes a fixed-priority uniprocessor scheduling algorithm to locally schedule non-migrating tasks and DS on each core.

The inter-core scheduling policy determines on which DS a job (of a migrating task) executes. The system maintains a global job queue $\mathcal{Q}$ and a dispatcher $\mathcal{P}$. The job at $\mathcal{Q}$'s head has a higher priority than any other jobs in $\mathcal{Q}$. The policy is described as follows:

1. After a new job is released, it is added to $\mathcal{Q}$.
2. After a job is dispatched, it is removed from $\mathcal{Q}$.
3. After a running job is suspended, it is put back in $\mathcal{Q}$.
4. $\mathcal{P}$ makes a dispatching decision when one of the following events occurs: 1) a job is added to $\mathcal{Q}$; 2) a job is finished; 3) a suspended DS obtains the CPU (e.g., a DS's capacity is replenished.)
5. $\mathcal{P}$ dispatches a job from $\mathcal{Q}$ to a DS as follows.
   - If there is an idle DS with available capacity, $\mathcal{P}$ dispatches the job at $\mathcal{Q}$'s head to that DS. If multiple such DS exist, $\mathcal{P}$ selects the DS with the smallest index.
   - If each DS with available capacity is processing a job, and the lowest-priority *running* job has a priority lower than the job at $\mathcal{Q}$'s head, it will be preempted.
   - If no DS has capacity, no dispatching will be made.

4 Response Time Analysis

Denote by J_k^i the i'th job of the k'th task τ_k, and J_k^i's release time and response time are respectively denoted by r_k^i and R_k^i. When there is no need to distinguish which job of τ_k it is, we omit the superscripts i and simply use the notations of J_k, r_k and R_k. The job of τ_k that has the maximum response time is denoted by J_k^{max}, and its response time is denoted by R_k^{max}.

A job's *scheduling window* is the interval between when it is released and when it is finished. By definition, the length of a job's scheduling window is its response time. A job's scheduling window can be divided into two parts: the *head* and the *body*. The head of a job J_k is the interval between r_k, the release time of the job, and the first replenishment after r_k. The body of J_k's scheduling window is the whole sub-interval following the head. Figure 1 (a) illustrates the head and body of the scheduling window of a job J_2 released at time 14 and finished at 64.

Following the classic time demand analysis used in [16, 27], our RTA is performed by solving a recurrence equation:

$$R_k^{max} = \Re(R_k^{max}). \tag{1}$$

The works in [16, 27] make use of a *critical instant* concept, which is the release time of J_k^{max}, the job with the maximum response time. Following this concept, we define the *critical head*, denoted by HD_k^C, the *critical body*, denoted by BD_k^C, to be the head and the body of J_k^{max}. With a little abuse of notation, we shall also use HD_k^C and BD_k^C to denote the *lengths* of their corresponding intervals, when no confusion arises. By definition,

$$R_k^{max} = HD_k^C + BD_k^C. \tag{2}$$

If we know the *exact* critical instant of a task, we can determine the *exact* HD_k^C and BD_k^C, and thus calculate the exact R_k^{max}. Unfortunately, in multiprocessor scheduling, the critical instant of a task is generally unknown, and thus, calculating the exact HD_k^C and BD_k^C is not possible. If, however, we can respectively calculate HD_k^C's upper bound, denoted by $\hat{HD}_k^C$, and BD_k^C's upper bound, denoted by $\hat{BD}_k^C$, we can then obtain an upper bound for R_k^{max}. In the following discussion, we present how to calculate $\hat{HD}_k^C$ and $\hat{BD}_k^C$.

4.1 $\hat{HD}_k^C$: Upper Bound of HD_k^C

Denote by t_k^{CI} the critical instant of τ_k, and by t_k^0 the last replenishment time before t_k^{CI}. By definition,

$$HD_k^C = t_k^0 + T_S - t_k^{CI}. \tag{3}$$

On a uniprocessor, if τ_k is processed by a DS, and there are other tasks consuming the same DS's capacity, t_k^{CI} is the earliest instant when the capacity of the current replenishment period is exhausted [29], which is C_S time units after a replenishment period begins. Therefore, on a uniprocessor, $t_k^{CI} = t_k^0 + C_S$.

However, t_k^{CI} on a set of SDS is not always the earliest instant when all DS exhaust their capacity, which is $t_k^0 + C_S^1$ (recall that C_S^1 is the largest capacity of all the DS). This is as illustrated by Example 4.1.

Example 4.1. *Consider two tasks:* $\tau_1 = (100, 18, 100)$ *and* $\tau_2 = (150, 34, 150)$ *processed by a 2-SDS* $= (20, 14, 10)$. *Take as time* 0 *the beginning of the replenishment period within which a job* J_2 *of* τ_2 *is released. The earliest instant when all DS's capacity is consumed is* 14*, and this scenario is illustrated in Figure 1 (a). If* $r_2 = 14$ *and all the DS's available capacity* before 14 *is consumed, then* J_2*'s response time is as most* 50*, as illustrated in Figure 1 (a). However, consider another scenario in Figure 1 (b), if* $r_2 = 10$*, and a job* J_1 *of* τ_1 *is also released at time* 10*, then* J_2*'s response time is* 54*, as illustrated in Figure 1 (b).*

While the exact value of t_k^{CI} is generally unknown, the following Lemma 4.1 states that $t_k^{CI} \geq t_k^0 + C_S^m$, where C_S^m is the smallest capacity of all the DS.

Lemma 4.1. $t_k^{CI} \geq t_k^0 + C_S^m$

Proof: See Appendix A. □

Corollary 4.1. $HD_k^C \leq \hat{HD}_k^C$ *where*

$$\hat{HD}_k^C = T_S - C_S^m \tag{4}$$

Proof: By (3), $t_k^{CI} = t_k^0 + T_S - HD_k^C$, and by Lemma 4.1, $t_k^0 + T_S - HD_k^C \geq t_k^0 + C_S^m \implies HD_k^C \leq T_S - C_S^m$. □

A longer head does not necessarily lead to a longer response time, as a longer head also means possibly more available capacity in the head. Thus more workload can be processed within the head, and this in turn may decrease the length of the body. Therefore, Corollary 4.1 does *not* state that we can simply use $\hat{HD}_k^C = T_S - C_S^m$ as the *exact* critical head. However, if we use $\hat{HD}_k^C$ as the upper bound of the critical head, *and* "discard" all the capacity within the critical head, that is, all the workload within J_k^{max}'s scheduling window is processed in the body BD_k^C, then we can bound HD_k^C and BD_k^C from above at the same time. How to bound BD_k^C from above is discussed next.

4.2 $\hat{BD}_k^C$: Upper Bound of BD_k^C

By (2), the sufficient condition to bound from above a job J_k's response time R_k is also the sufficient condition to bound from above the body BD_k^C. Next we present a sufficient condition to bound R_k from above.

4.2.1 Sufficient Condition for Bounding R_k

For further discussion, we define the concepts of *work-conserving* and *capacity-conserving*.

Work-conserving: A scheduling algorithm is work-conserving if it will never idle a processor whenever there is pending workload on that processor.

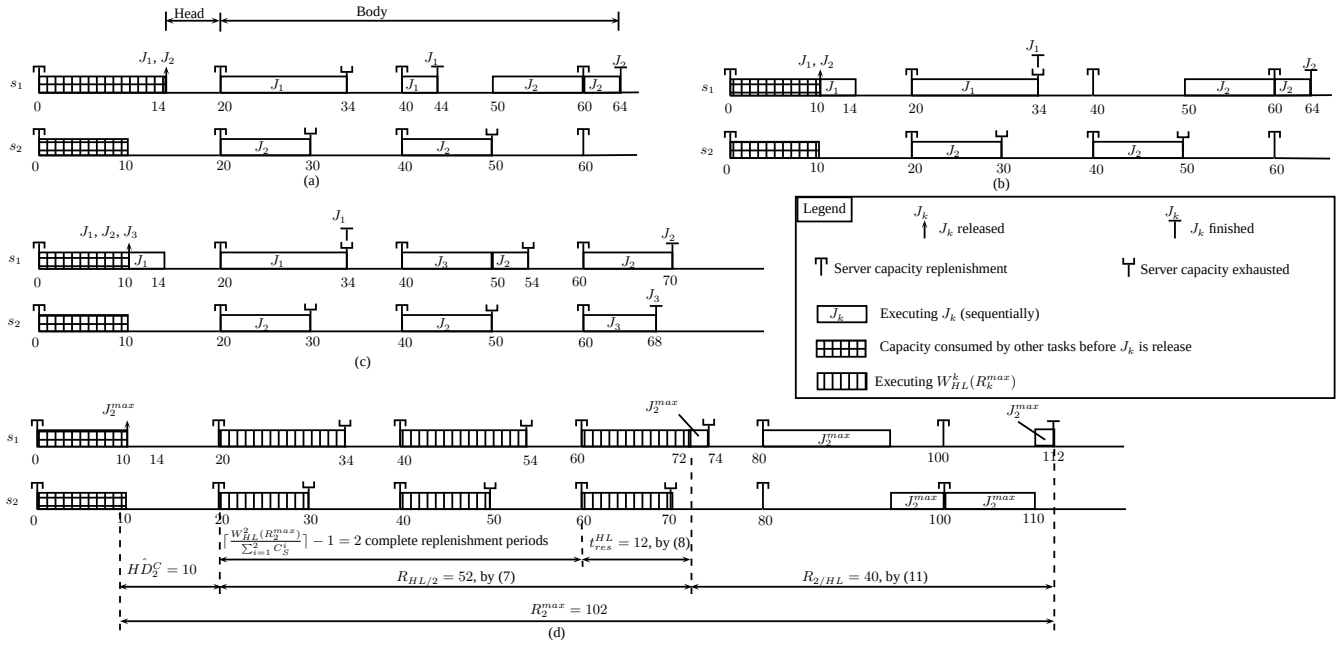

Figure 1. RTA of Tasks on a 2-SDS

Capacity-conserving: Denote by $Cpty([t_s, t_e])$ the maximum capacity available for processing workload within an interval $[t_s, t_e]$ (when all processors are running under full load), a scheduling algorithm is capacity-conserving if: $t''_e > t'_e \implies Cpty([t_s, t''_e]) \geq Cpty([t_s, t'_e])$. Intuitively, the capacity-conserving property says that, beginning at an instant, capacity provided in a longer interval is no less than in a shorter interval.

Most common scheduling algorithms, e.g., RM and Deadline Monotonic (DM), as well as DS, are both work- and capacity-conserving. The concepts of work- and capacity-conserving will be used in the following Lemma 4.2 that bounds R_k (and thus BD^C_k) from above.

Lemma 4.2. *Let $W_{excl.k}$ be the total workload of jobs other than J_k within J_k's scheduling window. If J_k and a fixed amount of $W_{excl.k}$ is processed within J_k's scheduling window by a work- and capacity-conserving algorithm, R_k has its upper bound if J_k does not start execution before* all *the workload $W_{excl.k}$ is* completely *finished.*

Proof: See Appendix B. □

The intuition of Lemma 4.2 is illustrated by an example in Figure 2. Consider a workload of $W_{excl.k} = 6$ units and a task $\tau_k = (10, 5, 10)$ processed by a 2-SDS $(5, 4, 3)$. Suppose $W_{excl.k}$ and J_k are both ready at time 0 when a replenishment occurs. In Figure 2 (a), if J_k and $W_{excl.k}$ are processed at the same time on different DS, then J_k is finished at time 7. However, consider another scenario in Figure 2 (b) where J_k does not start execution until $W_{excl.k}$ is completely finished (at time 3), J_k's finish time is delayed until 9, even though $W_{excl.k}$ is finished earlier under this scenario.

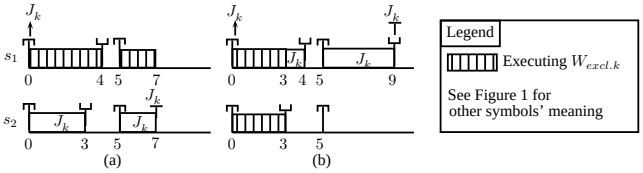

Figure 2. Example for Intuition of Lemma 4.2

It is worth noting that Lemma 4.2 is also applicable to identical multiprocessor dedicated scheduling considered in [13, 20], and provides for these platforms a sufficient condition to bound τ_k's maximum response time from above. For identical multiprocessor dedicated scheduling, if J_k does not start execution until $W_{excl.k}$ is completely finished, then all the processors must be busy executing $W_{excl.k}$, which takes $\frac{W_{excl.k}}{m}$ time units (here $W_{excl.k}$ is interpreted as the higher-priority tasks' workload processed under the worst-case scenario within an interval[4], while m is interpreted as the number of processors). R_k is then bounded from above by $\frac{W_{excl.k}}{m} + C_k$, as reported in [13,20][5].

Note that the sufficient condition stated in Lemma 4.2 may *not* necessarily occur for a particular task set and SDS. Lemma 4.2 states that, under the scenario where the sufficient condition holds, R_k will not be less than what it is under any other scenario where the same amount of $W_{excl.k}$ and J_k are processed.

4.2.2 Bounding BD^C_k from Above

After the sufficent condition to bound R_k from above is given, we show how to bound BD^C_k from above.

[4] In [13], this interval is J^{max}_k's scheduling window, while in [20], it is an extended busy window.

[5] $\frac{W_{excl.k}}{m}$ bounds from above J^{max}_k's *interference* in [13,20]. Those works differ mainly in how to bound $W_{excl.k}$ from above.

Denote respectively by $W_{HP}^k(R_k^{max})$ and $W_{LP}^k(R_k^{max})$ the upper bounds of the higher- and lower- priority[6] tasks' workload processed within J_k^{max}'s scheduling window under the worst-case scenario. Let

$$W_{HL}^k(R_k^{max}) = W_{HP}^k(R_k^{max}) + W_{LP}^k(R_k^{max}) \quad (5)$$

How to calculate $W_{HP}^k(R_k^{max})$, $W_{LP}^k(R_k^{max})$ and thus $W_{HL}^k(R_k^{max})$ will be presented in Section 4.2.3 and 4.2.4. For simplicity, readers can assume for now that they are known, and this will not affect understanding the following Corollary 4.2.

Corollary 4.2. *Under the scenario where J_k^{max} does not start execution before* all *the workload $W_{HL}^k(R_k^{max})$ is* completely *finished, let $R_{HL/k}$ be the time to process the workload $W_{HL}^k(R_k^{max})$, and $R_{k/HL}$ be the time to process J_k* after $W_{HL}^k(R_k^{max})$ is processed, *$BD_k^C \leq \hat{BD}_k^C$ where*

$$\hat{BD}_k^C = R_{HL/k} + R_{k/HL} \quad (6)$$

Proof: As aforementioned, the sufficient condition for J_k^{max}'s response time to achieve its upper bound is also the sufficient condition for BD_k^C to achieve its upper bound. Based Lemma 4.2, we have this corollary. □

$R_{HL/k}$ and $R_{k/HL}$ can be respectively calculated by Lemma 4.3 and Lemma 4.4.

Lemma 4.3.

$$R_{HL/k} = (\lceil \frac{W_{HL}^k(R_k^{max})}{\sum_{i=1}^m C_S^i} \rceil - 1) \cdot T_S + t_{res}^{HL} \quad (7)$$

where

$$t_{res}^{HL} = \begin{cases} \frac{W_{res}^{HL}}{m}, & \text{if } W_{res}^{HL} \leq \delta(m) \\ C_S^{i+1} + \frac{W_{res}^{HL} - \delta(i+1)}{i}, & \text{if } \delta(i+1) < W_{res}^{HL} \leq \delta(i), \\ & \forall i : 1 \leq i \leq m-1 \end{cases} \quad (8)$$

$$W_{res}^{HL} = W_{HL}^k(R_k^{max}) - (\lceil \frac{W_{HL}^k(R_k^{max})}{\sum_{i=1}^m C_S^i} \rceil - 1) \cdot \sum_{i=1}^m C_S^i \quad (9)$$

$$\forall i, 1 \leq i \leq m : \delta(i) = \sum_{j=i}^m C_S^j + C_S^i \cdot (i-1) \quad (10)$$

Proof Sketch Under work-conserving scheduling, if J_k^{max} does not start execution before $W_{HL}^k(R_k^{max})$ is *completely* finished, it must be the case that, before J_k^{max} starts execution, *all* DS with available capacity are *busy* executing $W_{HL}^k(R_k^{max})$. In other words, before $W_{HL}^k(R_k^{max})$ is finished, each DS's capacity is used to process $W_{HL}^k(R_k^{max})$. This is the key observation for the calculation of $R_{HL/k}$. For a complete proof, please see Appendix C.

[6]Lower-priority tasks need to be considered as they also consume capacity, as will explained in Section 4.2.4.

Lemma 4.4.

$$R_{k/HL} = \begin{cases} C_k, & \text{if } C_S^{rmn,k} \geq C_k \\ T_S - t_{res}^{HL} + CRP_k \cdot T_S + C_k - & \\ CRP_k \cdot \min(\sum_{i=1}^m C_S^i, T_S), & \text{otherwise} \end{cases} \quad (11)$$

where

$$CRP_k = \lceil \frac{C_k - C_S^{rmn,k}}{\min(\sum_{i=1}^m C_S^i, T_S)} \rceil - 1 \quad (12)$$

$$C_S^{rmn,k} = \min(\sum_{i=1}^m C_S^i - W_{res}^{HL}, T_S - t_{res}^{HL}) \quad (13)$$

and t_{res}^{HL} and W_{res}^{HL} are respectively given by (8) and (9).

Proof: See Appendix D. □

Calculation of $R_{HL/k}$ and $R_{k/HL}$ relies on $W_{HL}^k(R_k^{max})$ which (based on (5)) is determined by $W_{HP}^k(R_k^{max})$ and $W_{LP}^k(R_k^{max})$ to be discussed in the next two sub-sections.

4.2.3 To Calculate $W_{HP}^k(R_k^{max})$

Let $RW_{HP}^k(L)$ denote the upper bound of the *requested* workload of the tasks with a priority higher than τ_k in an interval of length L.

Under the worst-case scenario, for the higher-priority tasks, *all* the requested workload in J_k^{max}'s scheduling window will be processed no later than when J_k^{max} is finished, that is,

$$W_{HP}^k(R_k^{max}) = RW_{HP}^k(R_k^{max}). \quad (14)$$

It is important to note that the *requested* workload and the *processed* workload under the worst-case scenario may be different for lower-priority tasks, as will be shortly discussed in Section 4.2.4. This is the reason for distinguishing the concepts of requested and processed workloads.

Let $RW_i^k(L)$ denote the upper bound of a task τ_i's workload requested in an interval of L, then

$$RW_{HP}^k(L) = \sum_{i<k} RW_i^k(L). \quad (15)$$

The calculation of $RW_i^k(L)$ has been studied in [6,9,13, 20]. Among these works, [9,20] have tighter results than the other works. In [9, 20], the authors extend the beginning of J_k^{max}'s scheduling window to an earlier instant so as to obtain a tighter upper bound of the *carry-in* (and thus the workload) of the *higher*-priority tasks. However, it is unknown whether an instant before J_k^{max}'s release time can be found such that both the higher- and lower-priority tasks' carry-in can be bounded from above *tightly at the same time*. As we shall see shortly, both the higher- and lower-priority tasks must be considered in the RTA of SDS. Therefore, the techniques in [9, 20] cannot be applied here.

Without more efficient and accurate techniques at hand, we resort to the technique proposed in [13]:

$$\forall i \neq k : RW_i^k(L) = \\ N_i(L) \cdot C_i + \min(C_i, L + D_i - C_i - N_i(L) \cdot T_i) \quad (16)$$

where $N_i(L) = \lfloor \frac{L+D_i-C_i}{T_i} \rfloor$.

While (16) can be used to bound from above the requested workloads of both the higher- and lower-priority tasks at the same time, for the lower-priority tasks, we can obtain a tighter upper bound of $W^k_{LP}(R^{max}_k)$ by utilizing the fact that not all requested workload of the lower-priority tasks needs to be done within J^{max}_k's scheduling window even under the worst-case scenario. This is discussed next.

4.2.4 To Calculate of $W^k_{LP}(R^{max}_k)$

Under dedicated scheduling, a job J_k's response time is not affected by lower-priority tasks. Under hierarchical scheduling, however, its response time may be affected by lower-priority tasks, since while it is executing on one DS, there may be lower-priority tasks running on other DS, which will decrease the total capacity available to J_k, and thus increase J_k's response time, as illustrated in Example 4.2.

Example 4.2. *Consider again the two tasks and SDS given in Example 4.1, and now there is a third task* $\tau_3 = (100, 18, 100)$. *In Figure 1 (c), if a job* J_3 *is also released at time* 10 *together with* J_1 *and* J_2, $R_2 = 60$ *is longer than* $R_2 = 54$ *in Figure 1 (b).*

Example 4.2 indicates a significant difference between the RTA for hierarchical scheduling and the RTA for dedicated scheduling [13, 14, 16, 20] where only higher-priority tasks need to be considered.

Let $RW^k_{LP}(L)$ denote the upper bound of the *requested* workload of the tasks with priority lower than τ_k in an interval of length L:

$$RW^k_{LP}(L) = \sum_{i>k} RW^k_i(L) \tag{17}$$

where $RW^k_i(L)$ is given by (16). In general, within a job J^{max}_k's scheduling window, $RW^k_{LP}(R^{max}_k)$ is different from $W^k_{LP}(R^{max}_k)$, the *processed* workload of the lower-priority tasks. Lower-priority tasks can run only when J^{max}_k is running, and it is possible that only a portion of the requested workload in J^{max}_k's scheduling window is processed, depending on how much *cumulative capacity* is available for the lower-priority tasks in J^{max}_k's scheduling window. The cumulative capacity for a task within an interval is the amount of this task's workload that can be processed within this interval (when no DS is idle in this interval). Let $CCL_k(L)$ denote the cumulative capacity for processing lower-priority tasks within an interval of length L, then

$$W^k_{LP}(L) = \min(RW^k_{LP}(L), CCL_k(L)) \tag{18}$$

Within J^{max}_k's scheduling window, $CCL_k(R^{max}_k)$ can be bounded from above based on the following observations:

1. Lower-priority tasks run only when J^{max}_k is running;
2. While J^{max}_k is running, at most $m-1$ DS are executing lower-priority tasks;
3. J^{max}_k runs exactly for C_k time units.

Based on these observations, $CCL_k(R^{max}_k)$ is bounded by

$$CCL_k(R^{max}_k) = (m-1) \cdot C_k \tag{19}$$

4.3 Putting the Pieces Together

We now give a theorem that bounds R^{max}_k from above.

Theorem 4.3.

$$R^{max}_k \le H\hat{D}^C_k + B\hat{D}^C_k$$

where $H\hat{D}^C_k$ *and* $B\hat{D}^C_k$ *are given by (4) and (6).*

Proof: It follows from (2), Corollaries 4.1 and 4.2. □

R^{max}_k is then bounded by the smallest solution to

$$x = H\hat{D}^C_k + B\hat{D}^C_k \tag{20}$$

(20) can be solved by iteration starting with $x = H\hat{D}^C_k + C_k$, and it terminates if a solution is found, or $x > D_k$.

4.3.1 An Example Illustrating the Calculation

In this section we demonstrate how to use our RTA with an example. In this example, the task sets and SDS parameters are the same as those in Example 4.2.

As aforementioned, the approach to calculating the maximum response time R^{max}_k is by solving a recurrent equation (20). In this example, $k = 2$.

By Corollary 4.1, $H\hat{D}^C_2 = T_S - C^2_S = 10$. The above equation is solved by iteration starting at $R^{max}_2 = H\hat{D}^C_2 + C_2 = 44$. For each value of R^{max}_2, we compute the Right-Hand Side (RHS) of (20), and if (20) is satisfied, then the iteration terminates. If not satisfied, R^{max}_2 is increased by 1, and a new iteration begins. This process repeats until a value of R^{max}_2 satisfying (20) is found, or $R^{max}_2 > D_2$. In this example, $R^{max}_2 = 102$ is the smallest value that satisfies (20). We now demonstrate that, at the last step, i.e., $R^{max}_2 = 102$, how BD^C_2 is calculated based on Equations (14) through (20). This is demonstrated step by step as follows.

1. By (14), (15) and (16), $W^2_{HP}(102) = 36$.
2. By (17), $RW^2_{LP}(102) = 36$; by (19), $CCL_k(102) = 34$; by (18), $W^2_{LP}(102) = \min(36, 34) = 34$.
3. By (5), $W^2_{HL}(102) = 70$.
4. Compute $R_{HL/2}$ by Lemma 4.3. By (9), $W^{HL}_{res} = 22$. By (8) and (10), $t^{HL}_{res} = 12$. As illustrated in Figure 1 (d), the time is equal to two complete replenishment periods: $[20, 40)$ and $[40, 60)$, plus the time t^{HL}_{res} to process the residual work in $[60, 80)$. While calculating t^{HL}_{res}, we first calculate $\delta(m = 2) = 20$ by (10). Since $W^{HL}_{res} = 22 > \delta(m = 2) = 20$, we need to find an i' such that $1 \le i \le m-1$, and $\delta(i'+1) < W^{HL}_{res} \le \delta(i')$,

which in general can be done by first calculating $\delta(i)$ for all $1 \leq i \leq m$ and then by binary search. In this example, $i' = 1$. By (8), $t_{res}^{HL} = 12$.

To sum up, $R_{HL/2} = 52$.

5. Compute $R_{2/HL}$ by Lemma 4.4. By (13), $C_S^{rmn,2} = 2$. By (12), $CRP_2 = 1$. By (11), $R_{2/HL} = 40$. This step is also illustrated in Figure 1 (d).

6. By (6), $\hat{BD}_2^C = R_{HL/2} + R_{2/HL} = 92$.

7. $RHS = \hat{HD}_2^C + \hat{BD}_2^C = 102$ is equal to Left-Hand Side (LHS), the iteration terminates. $R_2^{max} = 102$ is the maximum response time.

5 Generalization

5.1 RTA for DS with an Arbitrary Priority

The RTA in Section 4 assumes that each DS has the highest priority on its host core, and thus only one DS is allowed on each core. In this section, we remove the highest-priority assumption and extend our RTA to arbitrary priority SDS. Provided that a DS has an arbitrary priority on its host core, mulitple DS on the same core are allowed.

When a DS s_i does not have the highest priority, it may suffer *interference*, the time during which a DS cannot execute due to the execution of other higher-priority *non-migrating* tasks or servers. Let IS_i, $1 \leq i \leq m$, be the maximum interference which s_i could suffer within a replenishment period. Note that all the non-migrating tasks and servers that can interfere with s_i are bound to the same host core of s_i. The calculation of IS_i is a uniprocessor scheduling problem, and has been solved by Davis and Burns in [16].

We now analyze how the RTA can be adapted for the case when $IS_i > 0$. We consider the head and the body respectively.

5.1.1 Bounding the Critical Head from Above

Let $IS_{max} = \max_{1 \leq i \leq m} \{IS_i\}$, a simple approach to bound from above the critical head is to add IS_{max} to $\hat{HD}_k^C$, which is the critical head calculated by (4) in Section 4.1. Note that the critical head cannot be longer than the replenishment period, therefore, the upper bound of the critical head for the case where DS has an arbitrary priority is given by $\min\{IS_{max} + \hat{HD}_k^C, T_S\}$.

5.1.2 Bounding the Critical Body from Above

When a DS s_i does not have the highest priority, it is still guaranteed capacity of C_S^i within each replenishment period by uniprocessor scheduling policy. Therefore, the number of *complete* replenishment periods within the body is the same as the number of the complete replenishment periods when each DS has the highest priority. The interference by other higher-priority non-migrating tasks and serves increase the upper bound of τ_k's body by adding at most IS_{max} to the time needed to finish the residual work in the last replenishment period of τ_k's scheduling window, that is, IS_{max} is added to the RHS of (11).

In summary, if each DS of a set of SDS has an arbitrary priority on its host core, the maximum response time of a migrating task is bounded from above by its maximum response time when each DS has the highest priority, plus $2 \cdot IS_{max}$.

5.2 Multiple DS on a Core

Since each DS of a SDS can be assigned with an arbitrary priority on its core, mulitple DS belonging to different SDS can run on one core, and each DS has different priority. For each set of SDS, the system maintains a job queue and a dispatcher associated with that SDS. Each dispatcher is responsible for dispatching jobs processed by the SDS associated with that dispatcher. On each core, the DS of different sets of SDS are scheduled by a fixed-priority uniprocessor scheduling algorithm.

5.3 Synchronized Periodic Server

In this section we present how to generalize the RTA in Section 4 to hierarchical scheduling where Periodic Server (PS) is used. To simplify discussion, we still assume that each PS has the highest priority on its host core.

A PS is also described by a 2-tuple (T_S, C_S) where T_S and C_S are the replenishment period and capacity respectively. A PS consumes and replenishes capacity like a DS does, but the difference is that, for a periodic server, if no workload is pending, all its capacity will be discarded, and will not be replenished until the next replenishment period begins; that is, a periodic sever cannot preserve its capacity in a period.

We can also design a hierarchical scheduling interface by replacing the aforementioned DS with PS that have a synchronized period, and for simplicity, we call such an interface the Synchronized Periodic Servers (SPS). The intra-core and inter-core scheduling policies described in Section 3.2 can also be applied to SPS.

The aforementioned RTA for SDS can be generalized to SPS by making some changes, which will be discussed shortly. As in the RTA for SDS, we also divide a job's scheduling window into the two parts: the head and the body, and bound them from above respectively.

5.3.1 Bounding the Head from Above

Let t_0 be the instant when a replenishment occurs. Since a PS is not bandwidth-preserving, if no workload is pending at t_0, all the capacity in the currently replenishment period will be discarded. If some workload arrives at time $t_0 + \epsilon$ where ϵ is an arbitrarily small positive number, it needs to wait for $T_S - \epsilon$ time units before the next replenishment occurs. Therefore, under SPS, the head of a job's

scheduling window is upper bounded from above by T_S, different from the head's upper bound under SDS which is calculated by (4).

5.3.2 Bounding the Body from Above

PS is also work- and capacity-conserving, and the sufficient condition to bound a job's response time and thus the body from above in Lemma 4.2 is also applicable to bounding from above the body of a job under SPS. The upper bound of the (critical) body, $B\hat{D}_k^C$, also consists of two parts: $R_{HL/k}$ and $R_{k/HL}$. While Lemma 4.3 still holds for calculation of $R_{HL/k}$, Lemma 4.4 needs modification before being used to calculate of $R_{k/HL}$, which is described as follows.

Under the scenario where all other tasks' workload is completely finished before J_k^{max} starts execution, after all the other tasks' workload $W_{HL}^k(R_k^{max})$ is handled, there is only one PS executing J_k^{max}, and other PS will discard their capacity because no workload is pending for them. According to the inter-core scheduling policy in Section 3.2, while dispatching J_k^{max}, if multiple servers with available capacity are idle, the server with the smallest index (i.e., the one with the largest capacity) is selected to execute J_k^{max}. As a result, in the last replenishment period wherein the workload $W_{HL}^k(R_k^{max})$ is finished, the remaining capacity available to process J_k^{max} is $C_S^1 - t_{res}^{HL}$, and in each following complete replenishment period, the capacity available to process J_k^{max} is C_S^1. To sum up,

$$R_{k/HL} = \begin{cases} C_k, & \text{if } C_S^{rmn,k} \geq C_k \\ T_S - t_{res}^{HL} + CRP_k \cdot T_S + C_k - & \\ CRP_k \cdot C_S^1, & \text{otherwise} \end{cases} \tag{21}$$

where

$$CRP_k = \lceil \frac{C_k - C_S^{rmn,k}}{C_S^1} \rceil - 1 \tag{22}$$

$$C_S^{rmn,k} = C_S^1 - t_{res}^{HL} \tag{23}$$

and t_{res}^{HL} and W_{res}^{HL} are respectively given by (8) and (9).

After $R_{HL/k}$ and $R_{k/HL}$ is calculated, the upper bound of BD_k^C can be calculated by (6).

5.3.3 Putting the Pieces Together

So far we have seen how to calculate the upper bounds of the head and body, R_k^{max} can then be obtained by solving (20).

6 Evaluation and Discussion

In this section we evaluate our SDS under different settings. While designing SDS, given a fixed amount of total bandwidth and the freedom of choosing how to allocate the bandwidth to each DS, a question of interest is: How does a bandwidth allocation scheme affect the task set's schedulability? Detailed discussion of this question is presented in Section 6.2.

Another question of interest is: How does a task's utilization/density affect a task set's schedulability? In identical multiprocessor dedicated scheduling, *heavyweight* tasks, i.e., tasks with high utilization, will decrease the chance that a task set is schedulable, even if the total utilization of the task set is low. This is recognized as the "Dhall Effect" [18]. For task sets scheduled by SDS, we are interested not only in whether a similar effect exists, but also in what "heavyweight" means in this context. Detailed discussion of this question is presented Section 6.3.

6.1 Experiment Settings

We examined 4-SDS, 8-SDS and 16-SDS with average bandwidth per DS equal to 0.15, 0.3 and 0.5. For a specific set of SDS, the replenishment period is varied among 1000, 2000, ..., and 10000. Given a set of SDS with a certain average bandwidth, three types of bandwidth allocation schemes are considered. In the first allocation scheme, denoted by *EQUAL*, each *DS* has the same bandwidth. In the second scheme, denoted by *FIRST-FIT*, the total bandwidth is allocated to each DS in a *First-Fit* style: The first DS has as much bandwidth as possible (up to 1), and the second DS has as much of the rest of the bandwidth as possible, and so on. In the third scheme, denoted by *RANDOM*, the total bandwidth is randomly distributed across all DS.

Task sets of two different sizes, $n = 10$ and 20, are randomly generated. Since we are considering constrained-deadline task sets in this paper, *density* instead of utilization is used as an evaluation parameter. A task's density is the ratio of this task's WCET to the smaller value of its deadline and period. Note that it is also possible to use utilization as an evaluation parameter.

We say that task sets with the same size n and the same average density per task belong to the same *task set class*. In our experiment, each task set class has 1000 task sets.

To generate tasks' density, we use the *UUniFast-Discard* algorithm [15]. After a task's density is determined, its deadline is randomly generated between 10000 and 100000 with uniform distribution. Each task's period is randomly chosen between its deadline and 1.5 times its deadline.

The discussion in the next two sub-sections is based on a representative subset of the results. For more results, please refer to Appendix E.

6.2 Bandwidth Allocation

The aforementioned three allocation schemes for an 8-SDS with an average bandwidth of 0.3 are examined. Figure 3 shows the percentages of the task sets respecting their deadlines, termed the *acceptance ratio*, when applying our RTA to 1000 task sets, each of which contains 20 tasks with an average density per task of 0.065. Figure 3 indicates that, given a fixed amount of total bandwidth, *EQUAL* is the most likely to schedule a task set. This is because *EQUAL* has the highest degree of parallelism (among different tasks), and the interference suffered by a task is shorter

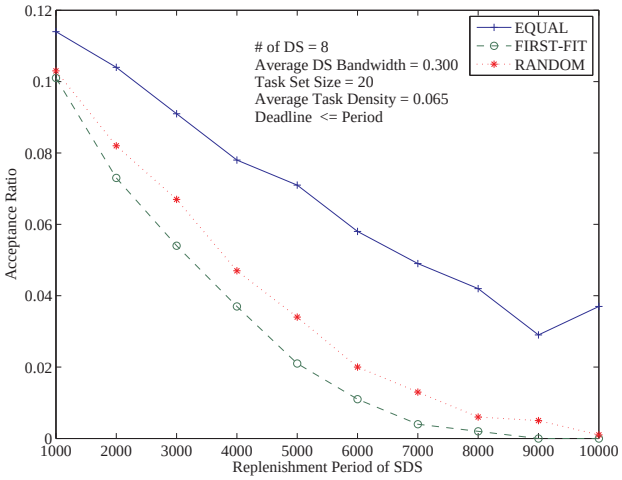

Figure 3. Acceptance ratio v.s. replenishment period under different allocation schemes (for heavyweight tasks)

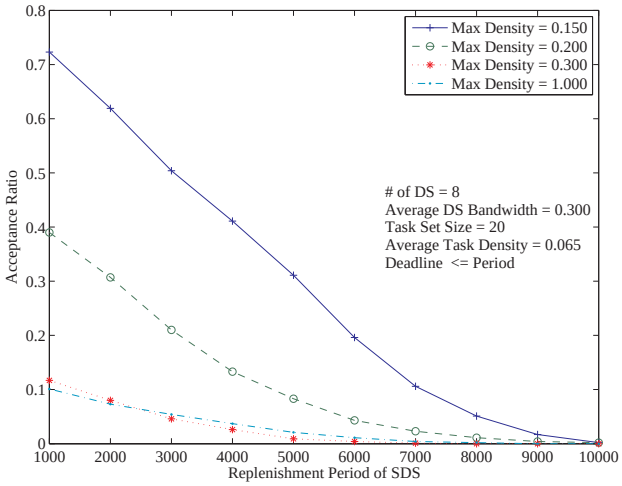

Figure 4. Acceptance ratio v.s. replenishment period under FIRST-FIT allocation scheme

under this allocation scheme than the other two schemes. In contrast, *FIRST-FIT* has the lowest degree of parallelism, and thus the interference on a task is longer.

The above result and conclusion seem to be contrary to the result in a recent work [28], wherein the authors argue that *FIRST-FIT* would be preferable in terms of tasks' schedulability. However, there is no contradiction here. The result shown in Figure 3 is a statistical result, therefore, the above conclusion may not hold for a specific task. For example, consider the highest-priority task τ_1. Under *FIRST-FIT*, the bandwidth that can be consumed by lower-priority tasks during τ_1's execution is lower than the other two schemes. In this regard *FIRST-FIT* favors the higher-priority task's schedulability. Further study is needed to understand the relationship between the bandwidth allocation scheme and a task at a particular priority.

6.3 Lightweight versus Heavyweight Tasks

The acceptance ratios in Figure 3 are extremely low, and it turns out that this is related to the *UUniFast-Discard* algorithm used to generate the task density.

While the task sets generated by the *UUniFast-Discard* algorithm are regarded as *unbiased*, we noticed that this algorithm tends to generate task sets with at least one *heavyweight* task. Given a set of SDS with an average bandwidth U_S^{avg}, we define a heavyweight task running on the SDS to be a task with a density greater than $U_S^{avg}/2$. Our results show that if a heavyweight task is present, then this task is unlikely to meet its deadline; but if no heavyweight task exists in a task set, then the task set is more likely to be schedulable. This is illustrated by Figure 4. In Figure 4, an 8-SDS with an average bandwidth of 0.3 is studied, and the bandwidth allocation scheme is *FIRST-FIT*. Four task set classes with the same size and the same average density are studied. These four task set classes differ from each other in the maximum possible densities of the tasks in each class, which are 0.15, 0.2, 0.3 and 1 respectively.

As illustrated in Figure 4, when the maximum density is no less than the average bandwidth, 0.3 in this case, the acceptance ratio is low. The acceptance ratio increases as the maximum density decreases. And when the maximum density is 1/2 of the average bandwidth, 0.15 in this case, the acceptance ratio is much higher than the case where the maximum density is equal to the average bandwidth for smaller replenishment periods.

In another two experiments (whose results are not presented due to space limitations), the acceptance ratios of the four task set classes studied in Figure 4 are also calculated under EQUAL and RANDOM allocation schemes, and the trend of the plotted curves are similar to Figure 4. All of these results suggest that the threshold for a task to be heavyweight turns out to be irrelevant to the allocation scheme. In Figure 4, under *FIRST-FIT*, there are two dedicated cores (bandwidth of 1), but the threshold of a heavyweight task is 0.15 instead of 0.5. For global scheduling on identical multiprocessors, it is well-known that, if a task set contains tasks with large utilizations or densities, it may be unschedulable even if the average task utilization or density is low [18]. An identical multiprocessor can be regarded as a special instance of SDS where each DS has bandwidth of 1. In this regard, our result extends the previous result. A quantitative analysis of how a task's utilization or density affects the system schedulability will be considered in future work.

7 Conclusion and Future Work

We propose a fixed-priority preemptive hierarchical multiprocessor scheduling interface called SDS, and present the RTA for migrating tasks. We identify the effect of lower-priority tasks in hierarchical scheduling RTA. Guidelines for designing SDS are discussed, and the schedulability affect of heavyweight in hierarchical scheduling is studied.

The RTA presented in this paper suffers pessimism in three respects: First, during the critical head, no capacity

is taken into account. Although this inaccuracy can be alleviated by selecting a shorter replenishment period, future work will be attempted to bound the head more tightly.

Second, the higher-priority tasks' workload is bounded with the approach in [13], which has been shown to not be tight [20]. However, the tighter result in [20] cannot be used here as mentioned in Section 4.2.3. Future work will investigate tighter bound on the higher-priority tasks' workload.

Third, to simplify the computational complexity, we assume that during τ_k's execution, all other DS are executing lower-priority tasks. Future work will improve the tightness by utilizing the fact that some DS may not execute due to exhaustion of capacity while τ_k is executing.

A Proof to Lemma 4.1

We prove the lemma by showing that if a job J_k is released at time $r_k^{'}, t_k^0 \leq r_k^{'} < t_k^0 + C_S^m$, its response time $R_k^{'}$ is less than its response time $R_k^{''}$ when it is released at time $r_k^{''} = t_k^0 + C_S^m$.

$r_k^{''}$ is the earliest instant when the DS with the smallest capacity can exhaust its capacity in a replenishment period, so within $[r_k^{'}, r_k^{''})$, all DS have available capacity. Now consider an interval of length L. The available cumulative capacity within $[r_k^{'}, r_k^{'} + L)$ is greater than the available cumulative capacity within $[r_k^{''}, r_k^{''} + L)$ for *any* L.[7] As a result, to process the same amount of workload, the time it takes when $r_k^{'} < r_k^{''}$ is no greater than what it is when $r_k^{'} = r_k^{''}$. Therefore, J_k^{max} cannot be released before $r_k^{''} = t_k^0 + C_S^m$, that is, $t_k^{CI} \geq t_k^0 + C_S^m$.

B Proof to Lemma 4.2

Consider two scenarios: **I**. J_k does not execute before $W_{excl.k}$ is *completely* finished, and **II**. J_k executes for a certain amount of time before $W_{excl.k}$ is finished. For both scenarios, J_k's scheduling window has 3 types of intervals:

1) The intervals wherein at least one processor is executing but no processor is executing J_k. Let $I_{I,busy}$ and $I_{II,busy}$ respectively be the *total* lengths of such intervals in J_k's scheduling window under Scenario I and II;

2) The intervals wherein no processor is executing $W_{excl.k}$ or J_k.[8] Let $I_{I,wait}$ and $I_{II,wait}$ be the *total* lengths of such intervals in J_k's scheduling window under Scenario I and II;

3) The intervals wherein one processor is executing J_k. The *total* length of such intervals in J_k's scheduling window is J_k's WCET, C_k, for both scenarios.

Let $R_{I,k}$ and $R_{II,k}$ respectively be J_k's response times under the above two scenarios. We have

$$R_{I,k} = I_{I,busy} + I_{I,wait} + C_k \tag{24}$$

[7] Note that this property may not hold if $r_k^{'} \geq t_k^0 + C_S^m$.

[8] due to, e.g., that no capacity is available to process $W_{excl.k}$ or J_k. Such an interval does not exist in dedicated identical multiprocessor scheduling, but can exist in hierarchical scheduling.

$$R_{II,k} = I_{II,busy} + I_{II,wait} + C_k \tag{25}$$

We prove $R_{II,k} \leq R_{I,k}$ by contradiction. Suppose $R_{II,k} > R_{I,k}$. Under Scenario II, when J_k is executing, there may or may not be other processors executing $W_{excl.k}$, but under work-conserving scheduling, for either case, $I_{I,busy} \geq I_{II,busy}$, as the same amount of $W_{excl.k}$ is processed under Scenario I and II. Based on (24) and (25):

$$I_{I,busy} \geq I_{II,busy}, R_{II,k} > R_{I,k} \implies I_{II,wait} > I_{I,wait} \tag{26}$$

Under dedicated scheduling, $I_{I,wait} = I_{II,wait} = 0$, so (26) leads to a contradiction.

Under hierarchical scheduling, (26) is possible only when $Cpty([r_k, r_k + R_k^I]) > Cpty([r_k, r_k + R_k^{II}])$ where r_k is J_k's release time. However, since $R_{II,k} > R_{I,k}$, this violates the capacity-conserving property. This finishes the proof.

C Proof to Lemma 4.3

Under work-conserving scheduling, if J_k^{max} does not start execution before $W_{HL}^k(R_k^{max})$ is *completely* finished, then before J_k^{max} starts execution, *all* DS with available capacity are *busy* executing $W_{HL}^k(R_k^{max})$. In other words, before $W_{HL}^k(R_k^{max})$ is finished, each DS's capacity is used to process $W_{HL}^k(R_k^{max})$.

Starting from the beginning of the interval BD_k^C, in each complete replenishment period, a total capacity of $\sum_{i=1}^m C_S^i$ units is used to processed $W_{HL}^k(R_k^{max})$ and it requires at most $(\lceil \frac{W_{HL}^k(R_k^{max})}{\sum_{i=1}^m C_S^i} \rceil - 1)$ *complete* replenishment periods. $R_{HL/k}$ consists of these complete replenishment periods and the time, denoted by t_{res}^{HL}, to process the residual workload, denoted by W_{res}^{HL}, which cannot be finished within the contiguous replenishment periods, as will be discussed next.

We now explain the intuitive meaning of W_{res}^{HL} in (9), $\delta(i)$ in (10), t_{res}^{HL} in (8), and how they are calculated.

W_{res}^{HL} in (9). In the last replenishment period during which the workload $W_{HL}^k(R_k^{max})$ is finished, *the residual workload* W_{res}^{HL} to be processed in this period is given by (9): In (9), $(\lceil \frac{W_{HL}^k(R_k^{max})}{\sum_{i=1}^m C_S^i} \rceil - 1) \cdot \sum_{i=1}^m C_S^i$ gives the amount of workload processed before the last replenishment period begins (i.e., the amount of workload processed within the contiguous complete replenishment periods), subtracting which from $W_{HL}^k(R_k^{max})$ gives W_{res}^{HL} in (9). Note that $0 < W_{res}^{HL} \leq \sum_{i=1}^m C_S^i$. □

$\delta(i)$ in (10). The function $\delta(i)$ in (10) calculates the total cumulative capacity between the beginning of a replenishment period, denoted by t_0, and the earliest instant when the i-th DS exhausts its capacity in the same replenishment period, which is $t_0 + C_S^i$. As illustrated in Figure 5 (a), when all DS are running under full load starting from t_0, within the interval $[t_0, t_0 + C_S^i)$, the DS s_i, s_{i+1}, ... and s_m has cumulative capacity of $\sum_{j=i}^{m} C_S^j$ units. Now consider the

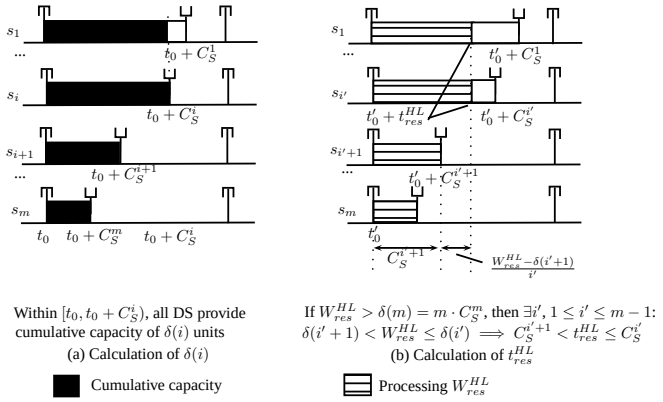

Figure 5. Calculation of $\delta(i)$ and t_{res}^{HL}

DS s_1, s_2, ... s_{i-1}. Each of them has a capacity no less than C_S^i units, and thus within $[t_0, t_0 + C_S^i)$, the DS s_1, s_2, ... s_{i-1} have a total cumulative capacity of $C_S^i \cdot (i-1)$ units. By summing up all the DS's cumulative capacity, the function $\delta(i)$, $1 \le i \le m$, is indeed given by (10). □

t_{res}^{HL} in (8). We now calculate the time, denoted by t_{res}^{HL}, to process the residual workload W_{res}^{HL}. Since some DS may exhaust their capacity before the residual workload is finished, t_{res}^{HL} cannot be calculated simply by $W_{res}^{HL} / \sum_{i=1}^{m} C_S^i$.

t_{res}^{HL} is calculated by considering the following facts:

1. Denote by t_0' the beginning instant of the last replenishment period during which all the workload $W_{HL}^k(R_k^{max})$ is finished,, then under the worst-case scenario, within the interval $[t_0', t_0' + t_{res}^{HL})$, *all the DS are busy executing the workload of W_{res}^{HL} units, and J_k^{max} is not executing until all the workload W_{res}^{HL} is finished*. In other words, all the DS are executing W_{res}^{HL} and then stop execution *simultaneously* at $t_0' + t_{res}^{HL}$.

2. W_{res}^{HL} is equal to the cumulative capacity of all the DS within $[t_0', t_0' + t_{res}^{HL}]$; in other words, t_{res}^{HL} is the time it takes for all the DS to provide cumulative capacity of W_{res}^{HL} units from time t_0'.

3. Within the interval $[t_0', t_0' + C_S^m)$, all DS have available capacity, so if $W_{res}^{HL} \le \delta(m) = m \cdot C_S^m$, then $t_{res}^{HL} = W_{res}^{HL}/m$, as given on the first line of (8).

4. Now consider the case when $W_{res}^{HL} > \delta(m) = m \cdot C_S^m$. Note that, given a value of i, $1 \le i \le m$, $\delta(i)$ has a unique value, i.e., $\delta(i)$ is surjective. If $W_{res}^{HL} > \delta(m) = m \cdot C_S^m$, there must exist one and only one i', $1 \le i' \le m-1$, such that $\delta(i'+1) < W_{res}^{HL} \le \delta(i')$, which in turn implies $C_S^{i'+1} < t_{res}^{HL} \le C_S^{i'}$. t_{res}^{HL} consists of two parts. As illustrated in Figure 5 (b), within the interval $[t_0', t_0' + C_S^{i'+1})$, $\delta(i'+1)$ units of workload are processed, and within $[t_0' + C_S^{i'+1}, t_0' + C_S^{i'})$, $W_{res}^{HL} - \delta(i'+1)$ units of workload are processed by i' DS, which takes $(W_{res}^{HL} - \delta(i'+1))/i'$ time units. Summing up $C_S^{i'+1}$ and $(W_{res}^{HL} - \delta(i'+1))/i'$, we have t_{res}^{HL} for the case when $\delta(i+1) < W_{res}^{HL} \le \delta(i)$, $1 \le i \le m-1$.

5. By definition, $0 < t_{res}^{HL} \le C_S^1$. The above argument covers all the possible values of t_{res}^{HL}.

To sum up, given a value of W_{res}^{HL}, the value of (8) can be uniquely determined, and t_{res}^{HL} is given by (8). □

$R_{HL/k}$ is given by summing up the length of the contiguous complete replenishment periods and t_{res}^{HL}, as in (7).

D Proof to Lemma 4.4

In the last replenishment period during which the workload $W_{HL}^k(R_k^{max})$ is finished, the amount of the residual workload, W_{res}^{HL}, in this period is given by (9). After the workload W_{res}^{HL} is finished, the remaining capacity in the same replenishment period is $\sum_{i=1}^{m} C_S^i - W_{res}^{HL}$. Since J_k^{max} cannot execute on more than one processor *at the same time*, the remaining capacity that can be used by J_k^{max} is given by $C_S^{rmn,k}$ in (13), where $T_S - t_{res}^{HL}$ is the length of the interval between when J_k^{max} starts execution and the first replenishment after J_k^{max} starts execution.

If $C_S^{rmn,k} \ge C_k$, J_k^{max} will be finished before the next replenishment period, then the time to process J_k^{max} *after the workload of $W_{HL}^k(R_k^{max})$ units is finished* is C_k.

If $C_S^{rmn,k} < C_k$, J_k^{max} will not finish before the next replenishment. In each complete replenishment period after J_k^{max} starts execution, the capacity that can be used by J_k^{max} is $\min(\sum_{i=1}^{m} C_S^i, T_S)$. The time to process J_k^{max} consists of three components: I) the length of the interval between when J_k^{max} starts execution and the first replenishment instant after J_k^{max} starts execution, which is $T_S - t_{res}^{HL}$, II) the number of the *complete* replenishment periods, CRP_k in (12), and III) the time to process J_k^{max}'s residual workload in the last replenishment period wherein J_k^{max} is finished, which is $C_k - CRP_k \cdot \min(\sum_{i=1}^{m} C_S^i, T_S)$. The time to process J_k^{max} *after the workload of $W_{HL}^k(R_k^{max})$ units is finished* is then the sum of all these three components.

E Acceptance Ratio Under Different Settings

This section lists the acceptance ratio under different settings. The SDS settings are summarized in Table 1. Note that an SDS is described by a combination of a value from each row of Table 1, and totally there are $3 \times 3 \times 3 = 27$ different combinations and thus 27 SDS.

Table 1. SDS Settings

Number of DS	4, 8, 16
Average DS Bandwidth	0.15, 0.30, 0.50
Allocation Scheme	*EQUAL*, *FIRST-FIT*, *RANDOM*

The task sets' settings are summarized in Table 2. Note that in this table, the values of average task density are artificially selected with the purpose to represent real world

tasks as much as possible, while still keeping them as comprehensive as possible. In Section 6.1, we defined that task

Table 2. Task Settings

Task Set Size	10, 20, 40
Average Task Density	0.010, 0.050, 0.060, 0.065, 0.070, 0.080, 0.100, 0.150
Maximum Task Density	0.150, 0.300, 1.000

sets with the same size n and the same average task density belong to the same *task set class*, now we re-define a task set class to be the task sets with the same size n, the same average task density *and* the same maximum task density. Therefore, a combination of a value from each row in Table 2 describes a task set class. In this experiment, there are $3 \times 8 \times 3 = 72$ task set classes each of which contains 1000 task sets. However, since *UUniFast-Discard* algorithm does not guarantee that a valid task set can always be generated within a given number of trials, to prevent generation of task sets from running too long, not all the 72 task set classes are studied in this experiment.

We evaluate the acceptance ratio of the above tasks set class when the tasks are processed by the aforementioned SDS. For a given number of DS and a given average DS bandwidth, the acceptance ratio for the three different bandwidth allocation schemes is shown in one figure. All the figures are listed below.

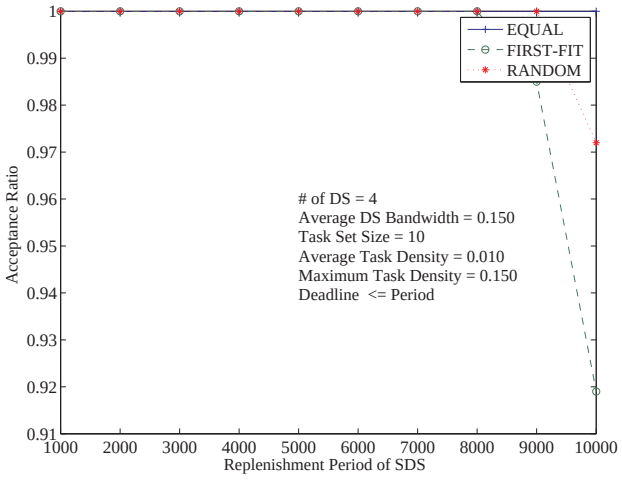

Figure 6. Acceptance ratio v.s. replenishment period

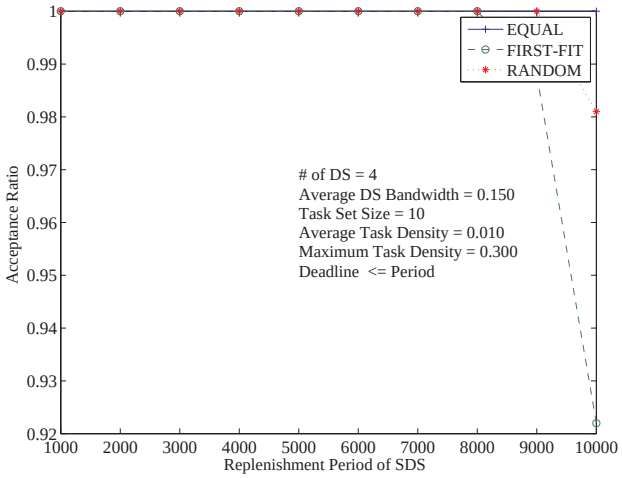

Figure 7. Acceptance ratio v.s. replenishment period

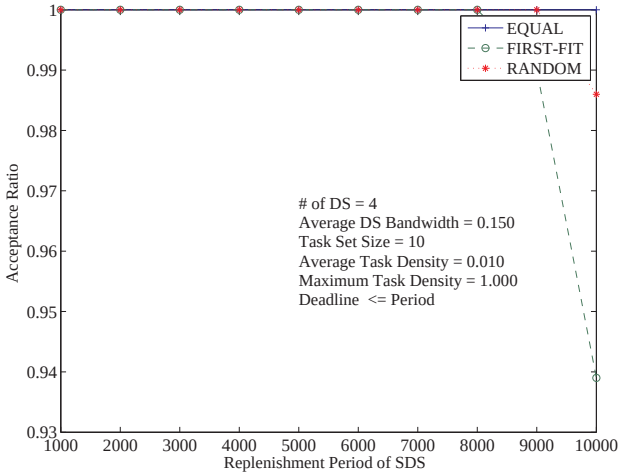

Figure 8. Acceptance ratio v.s. replenishment period

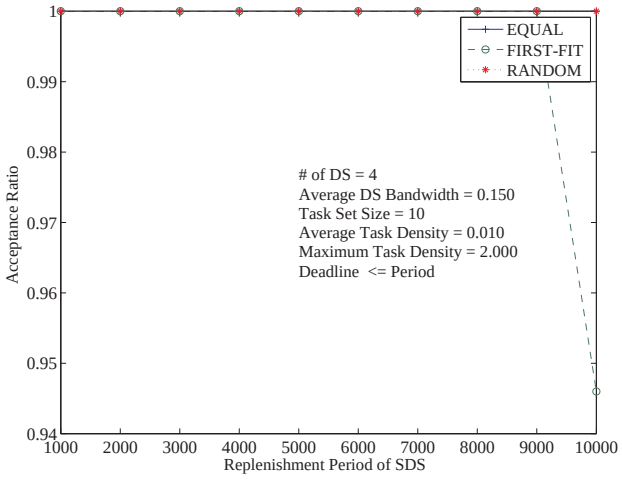

Figure 9. Acceptance ratio v.s. replenishment period

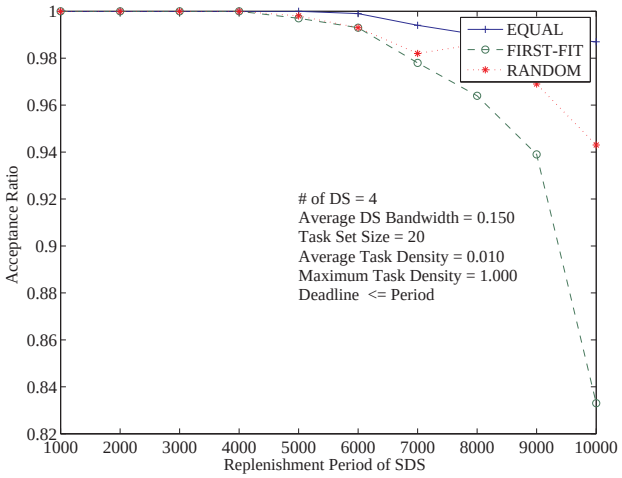

Figure 10. Acceptance ratio v.s. replenishment period

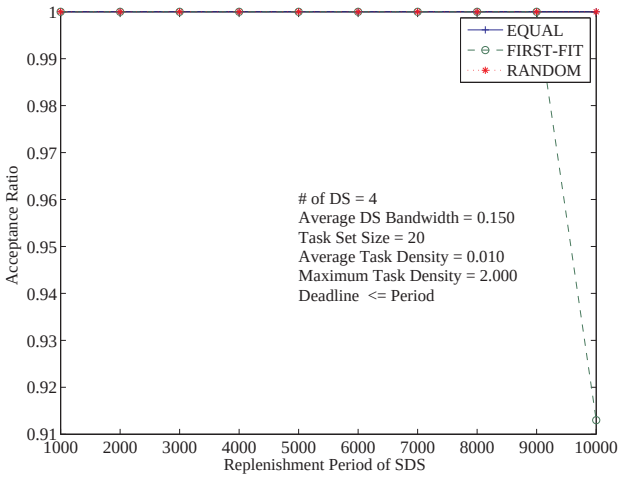

Figure 11. Acceptance ratio v.s. replenishment period

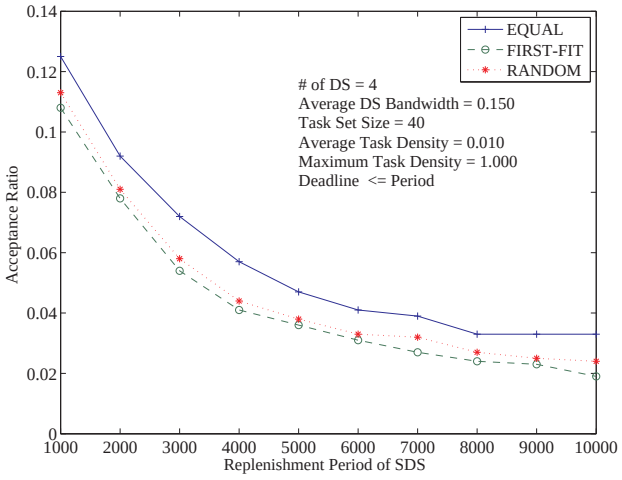

Figure 12. Acceptance ratio v.s. replenishment period

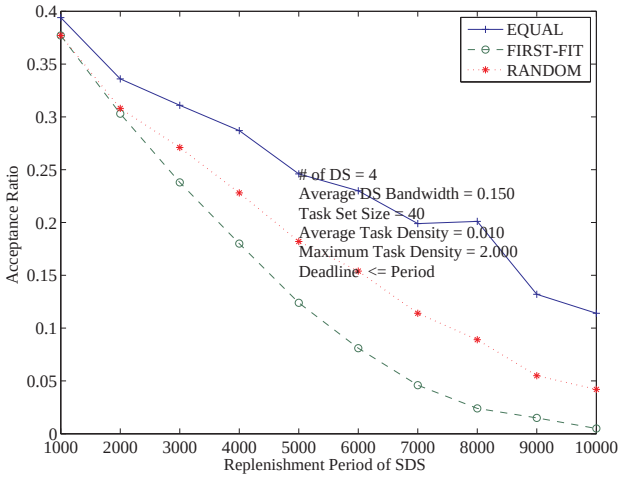

Figure 13. Acceptance ratio v.s. replenishment period

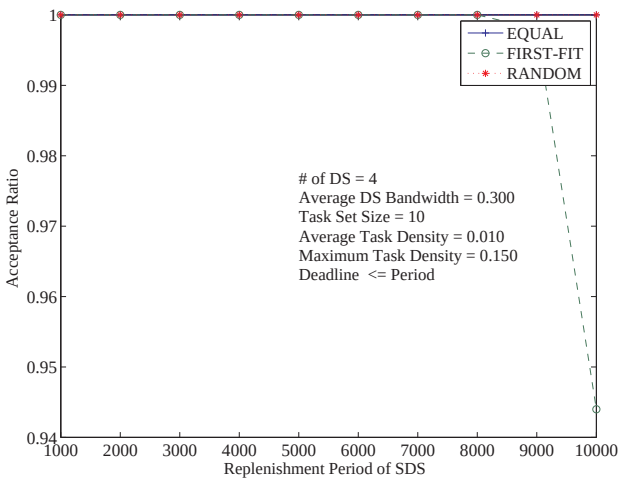

Figure 14. Acceptance ratio v.s. replenishment period

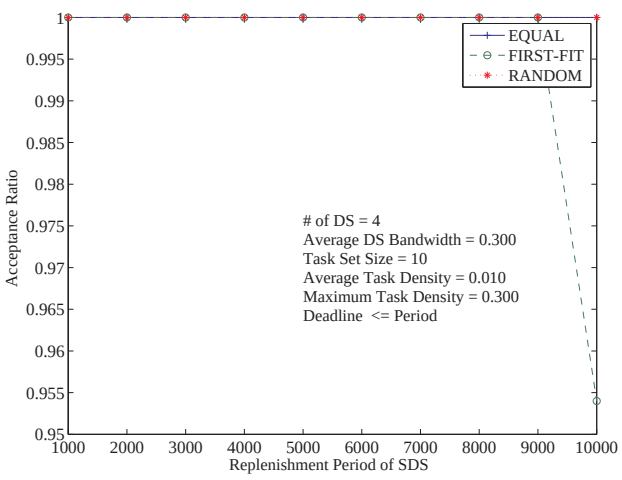

Figure 15. Acceptance ratio v.s. replenishment period

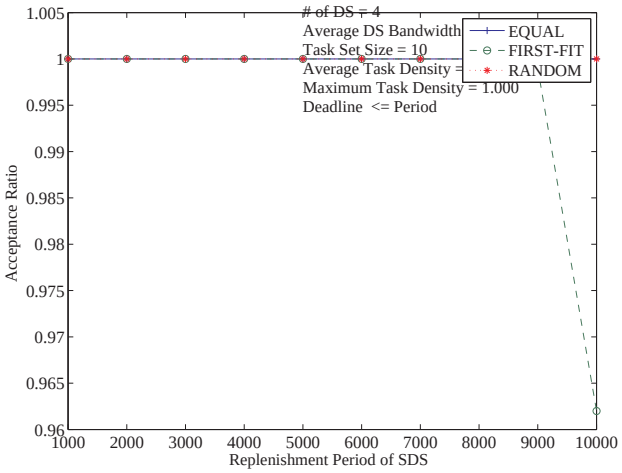

Figure 16. Acceptance ratio v.s. replenishment period

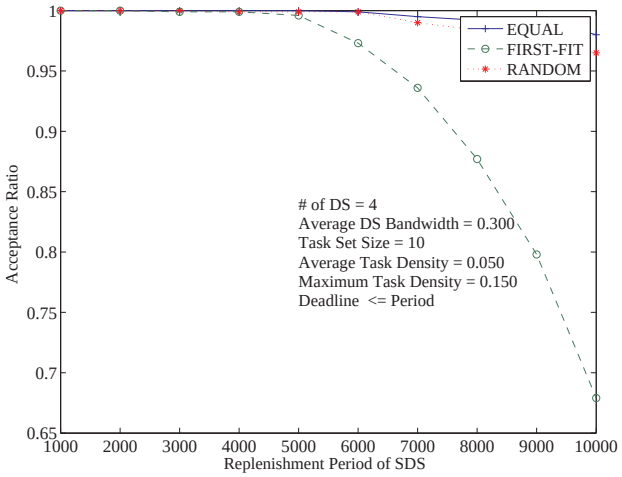

Figure 17. Acceptance ratio v.s. replenishment period

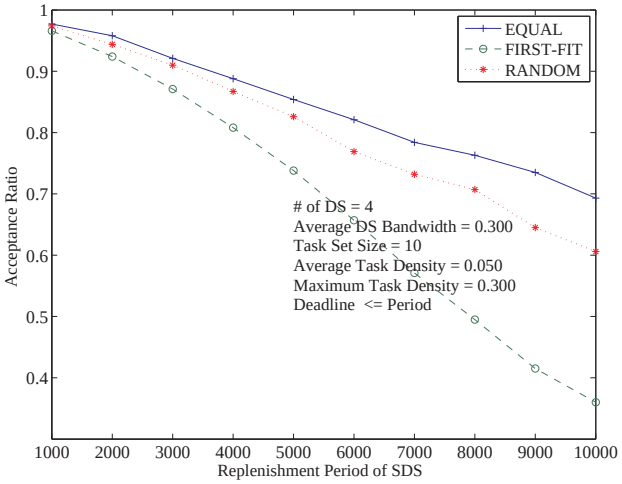

Figure 18. Acceptance ratio v.s. replenishment period

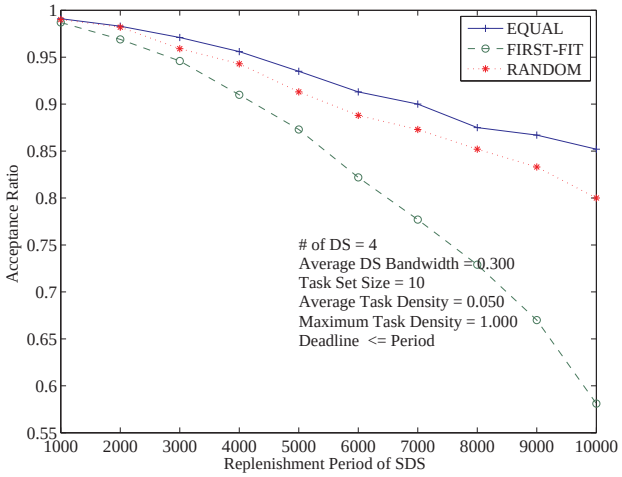

Figure 19. Acceptance ratio v.s. replenishment period

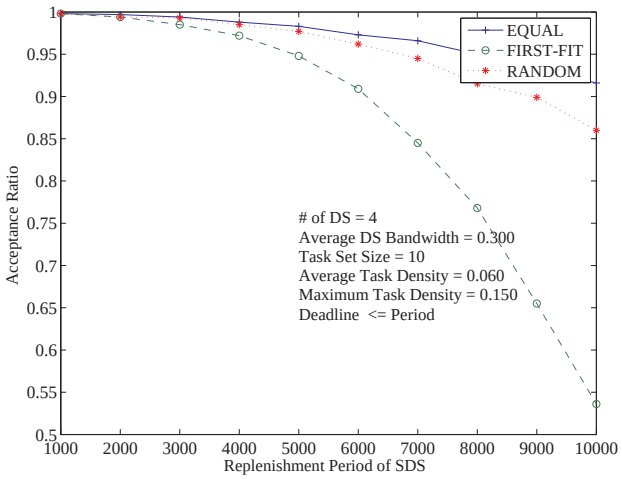

Figure 20. Acceptance ratio v.s. replenishment period

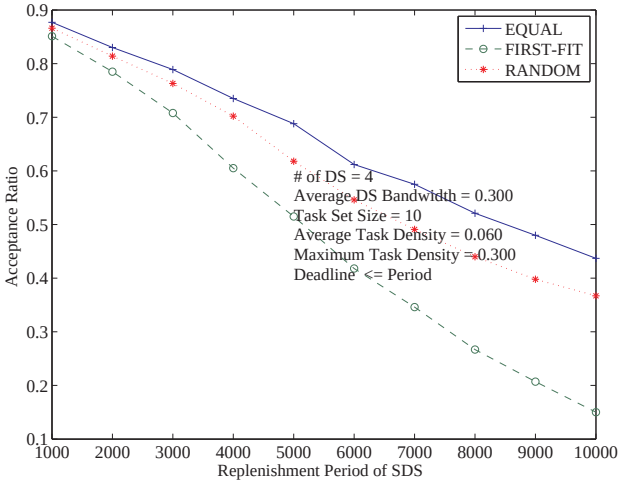

Figure 21. Acceptance ratio v.s. replenishment period

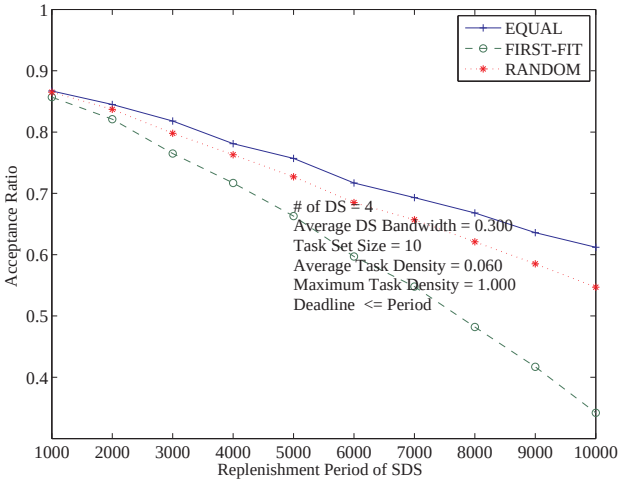

Figure 22. Acceptance ratio v.s. replenishment period

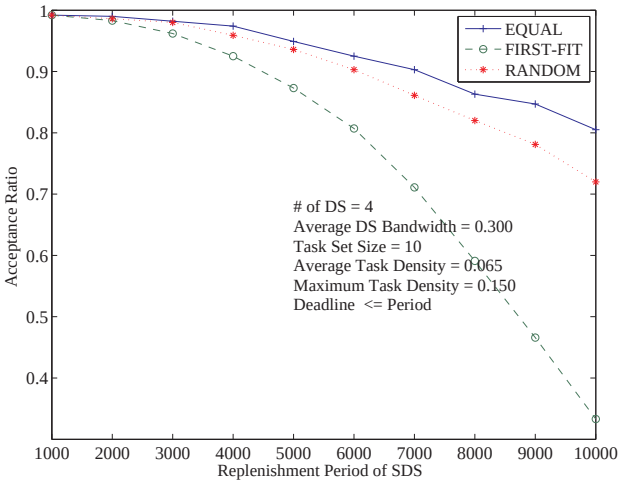

Figure 23. Acceptance ratio v.s. replenishment period

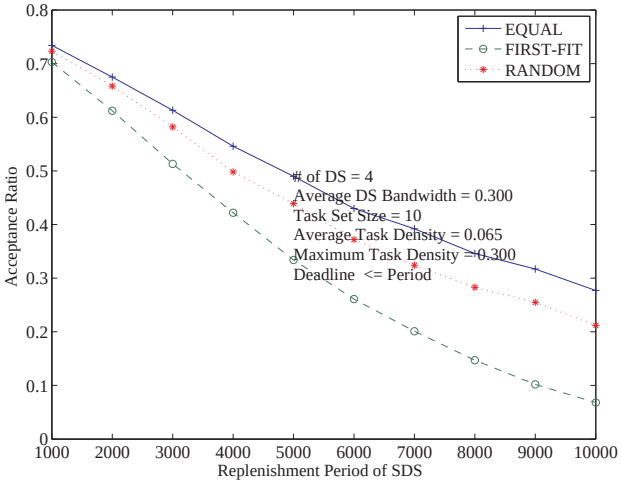

Figure 24. Acceptance ratio v.s. replenishment period

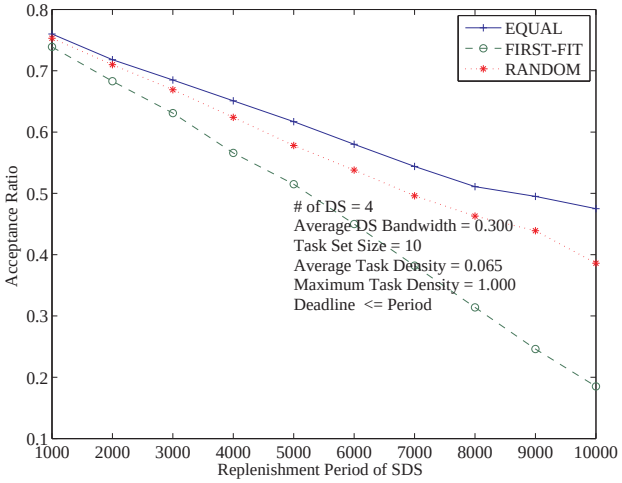

Figure 25. Acceptance ratio v.s. replenishment period

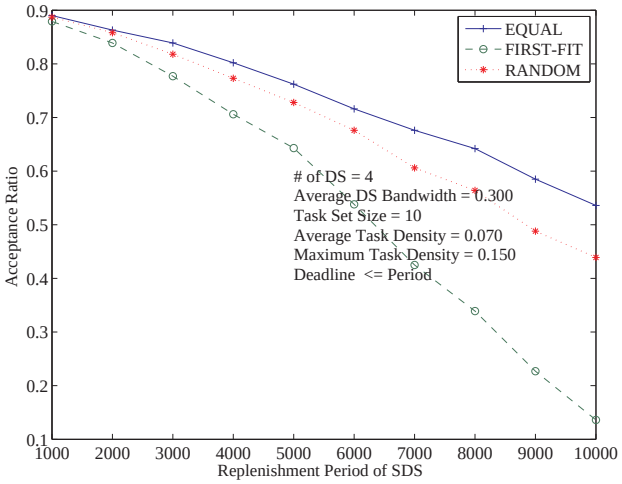

Figure 26. Acceptance ratio v.s. replenishment period

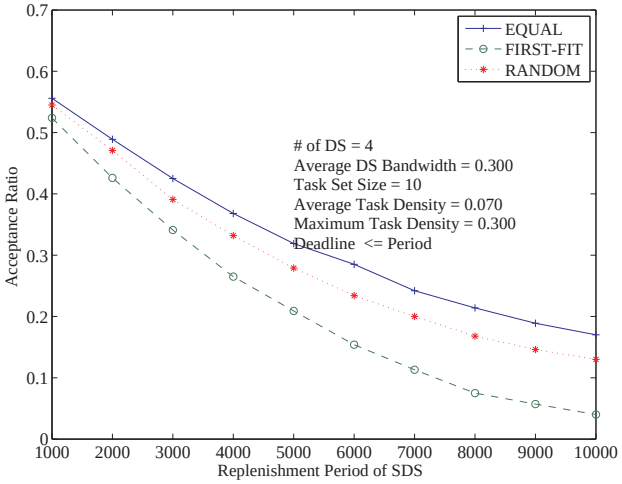

Figure 27. Acceptance ratio v.s. replenishment period

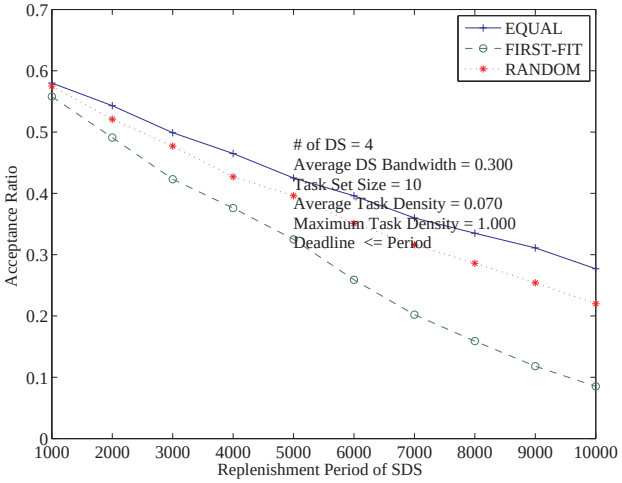

Figure 28. Acceptance ratio v.s. replenishment period

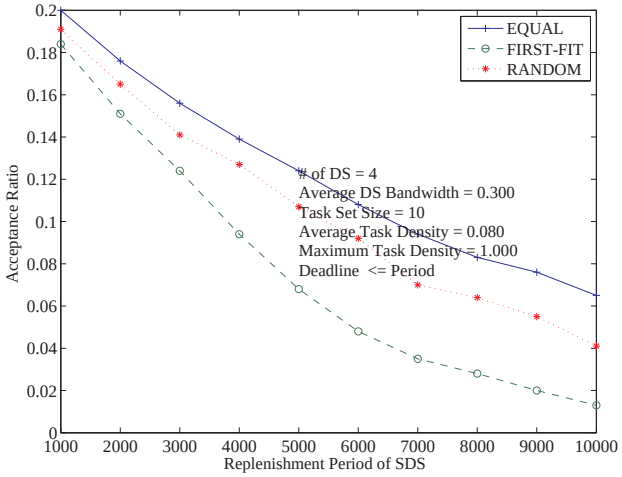

Figure 31. Acceptance ratio v.s. replenishment period

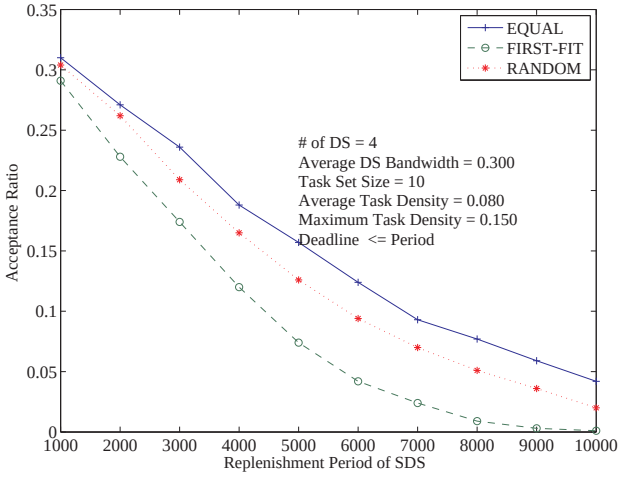

Figure 29. Acceptance ratio v.s. replenishment period

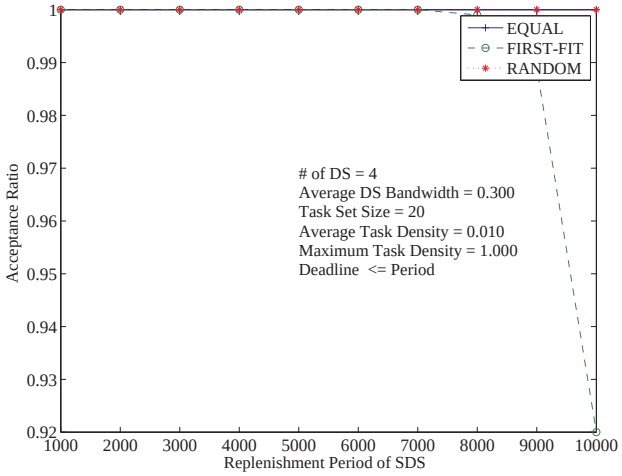

Figure 32. Acceptance ratio v.s. replenishment period

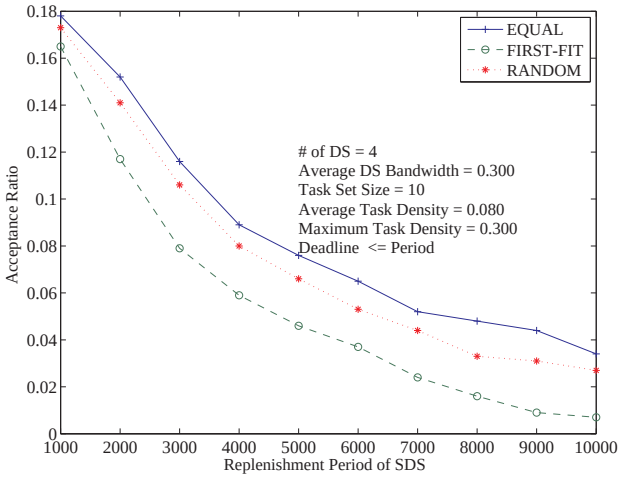

Figure 30. Acceptance ratio v.s. replenishment period

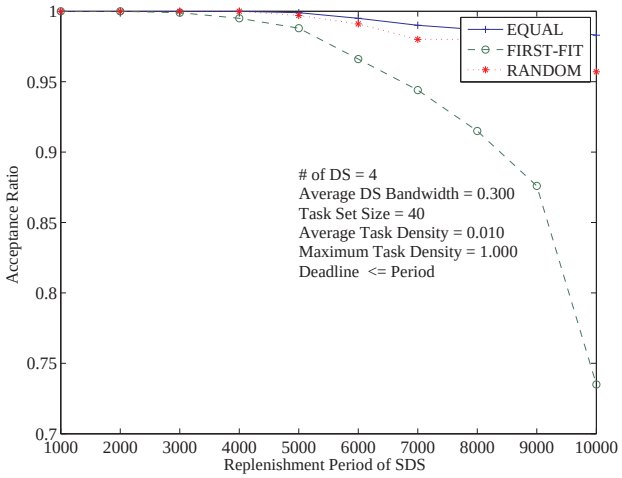

Figure 33. Acceptance ratio v.s. replenishment period

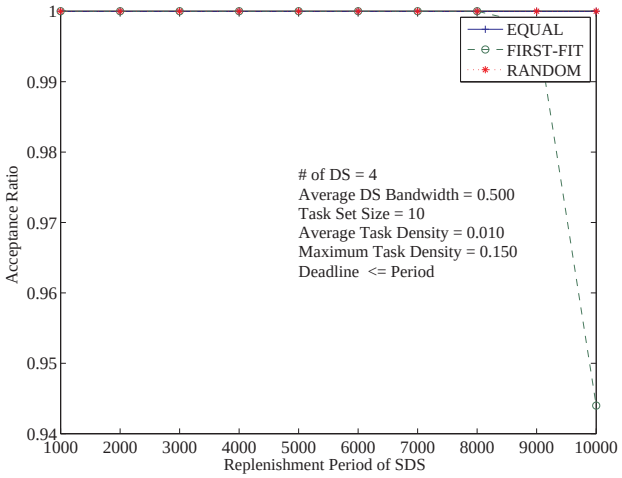

Figure 34. Acceptance ratio v.s. replenishment period

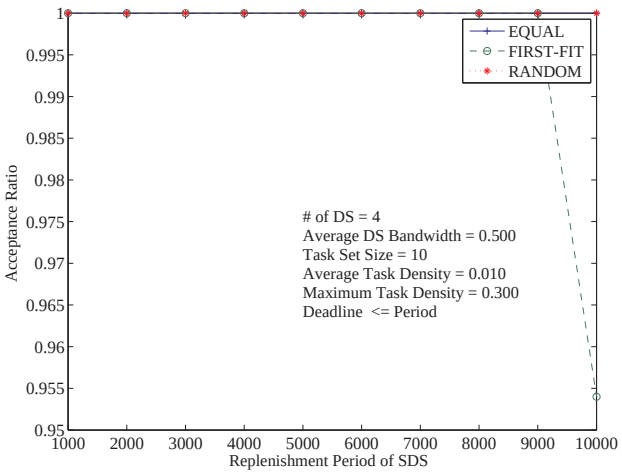

Figure 35. Acceptance ratio v.s. replenishment period

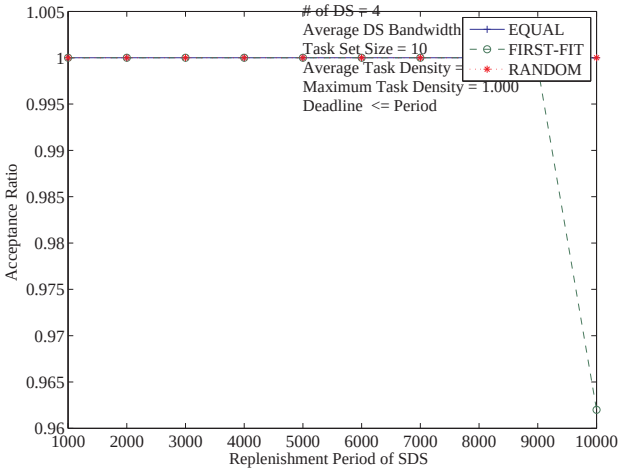

Figure 36. Acceptance ratio v.s. replenishment period

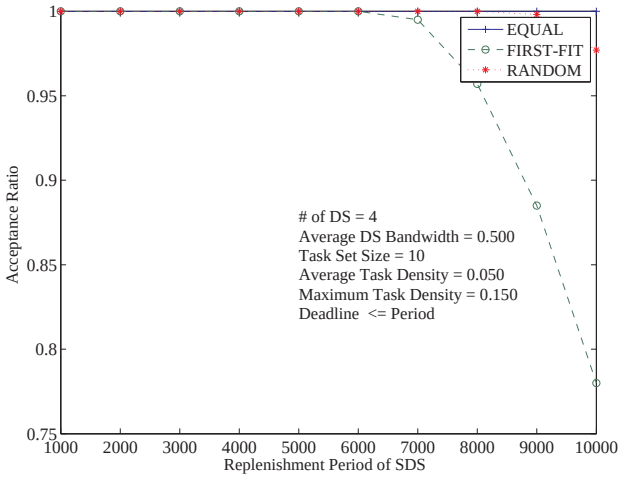

Figure 37. Acceptance ratio v.s. replenishment period

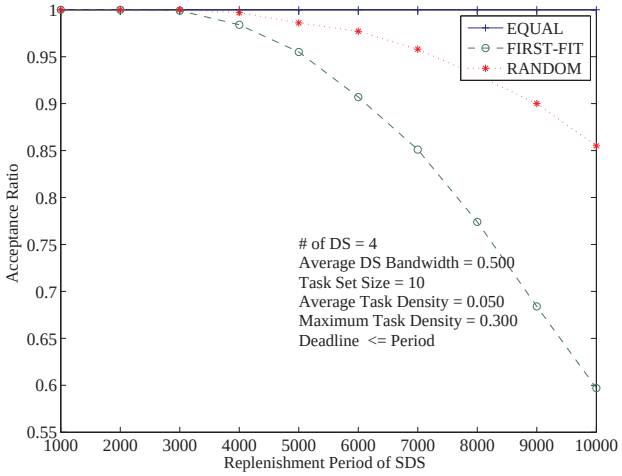

Figure 38. Acceptance ratio v.s. replenishment period

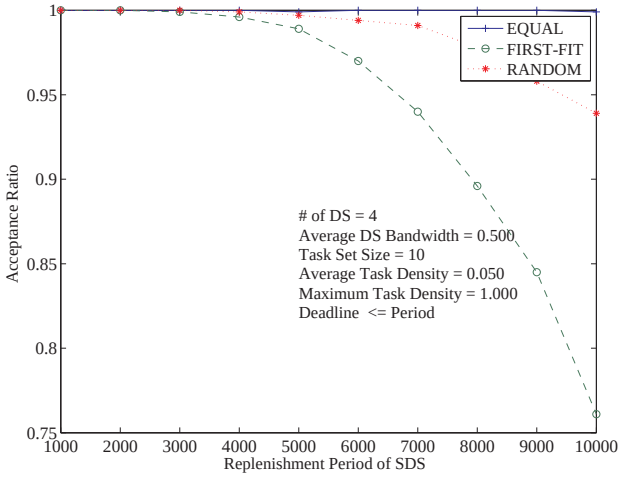

Figure 39. Acceptance ratio v.s. replenishment period

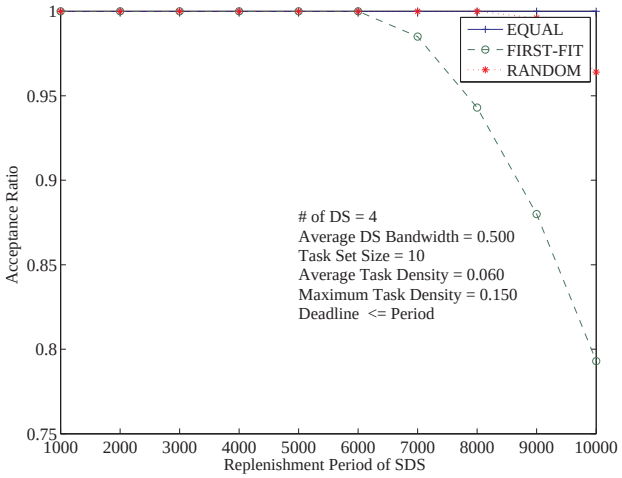

Figure 40. Acceptance ratio v.s. replenishment period

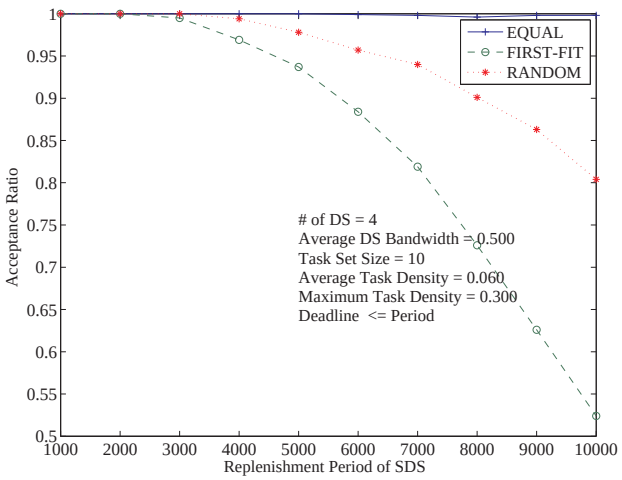

Figure 41. Acceptance ratio v.s. replenishment period

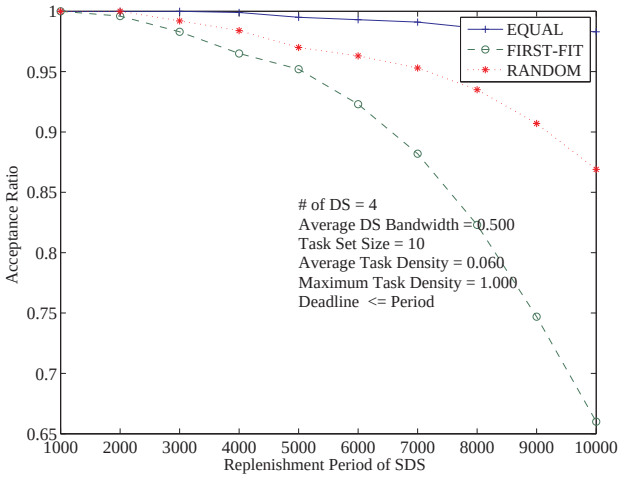

Figure 42. Acceptance ratio v.s. replenishment period

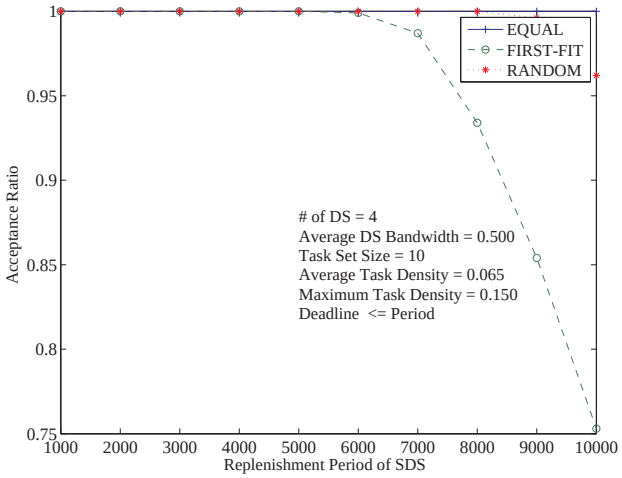

Figure 43. Acceptance ratio v.s. replenishment period

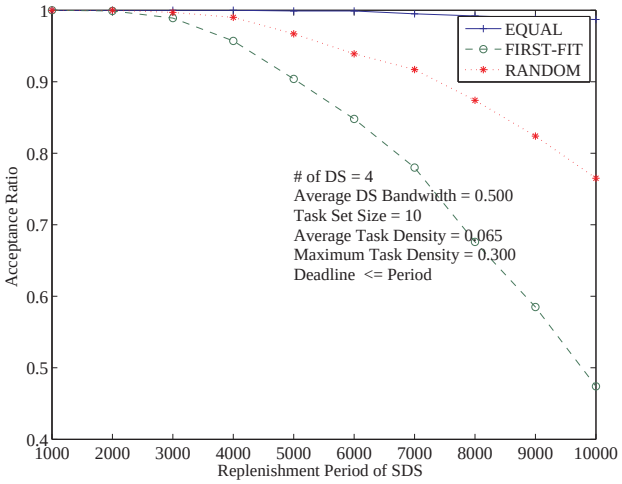

Figure 44. Acceptance ratio v.s. replenishment period

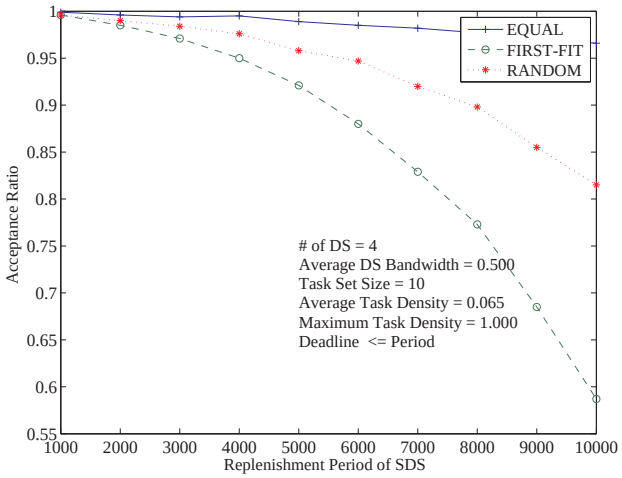

Figure 45. Acceptance ratio v.s. replenishment period

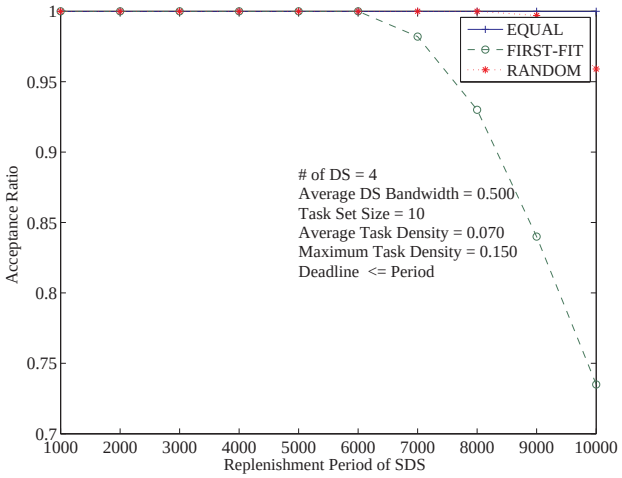

Figure 46. Acceptance ratio v.s. replenishment period

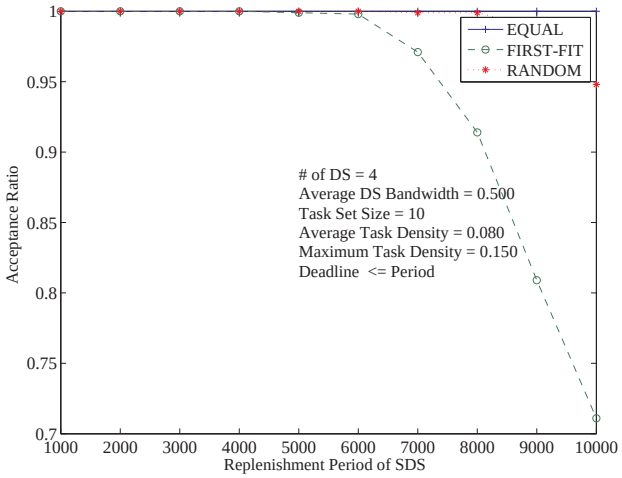

Figure 49. Acceptance ratio v.s. replenishment period

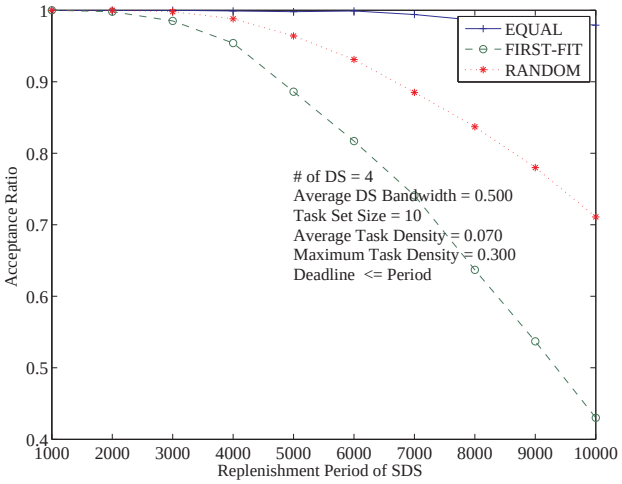

Figure 47. Acceptance ratio v.s. replenishment period

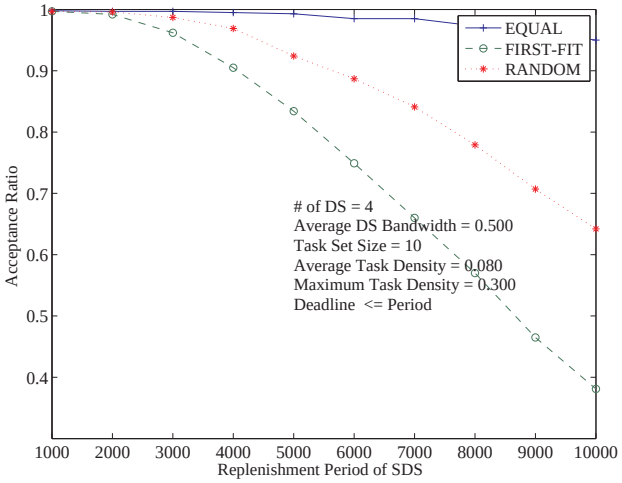

Figure 50. Acceptance ratio v.s. replenishment period

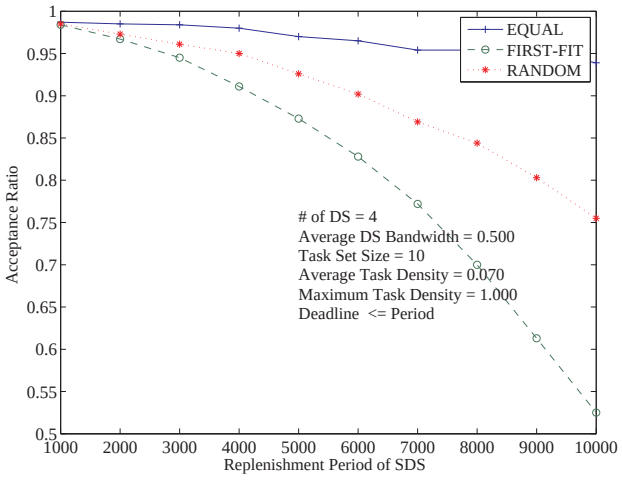

Figure 48. Acceptance ratio v.s. replenishment period

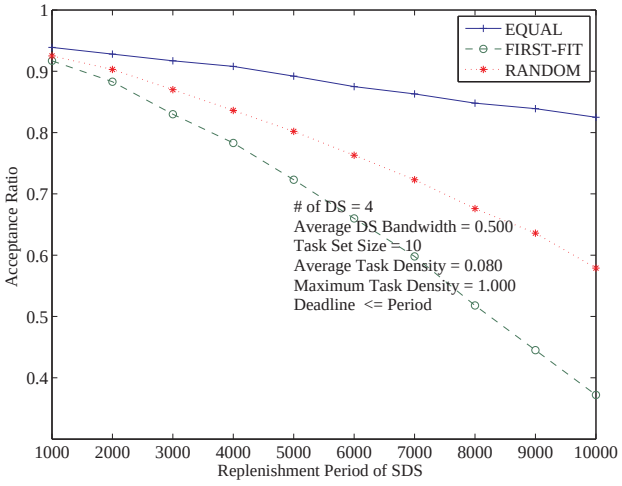

Figure 51. Acceptance ratio v.s. replenishment period

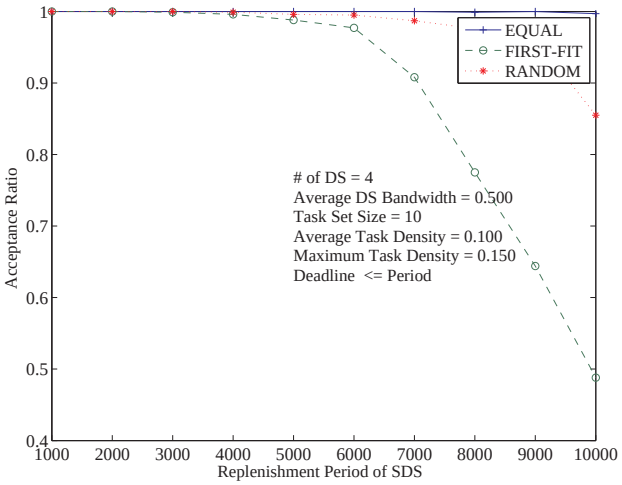

Figure 52. Acceptance ratio v.s. replenishment period

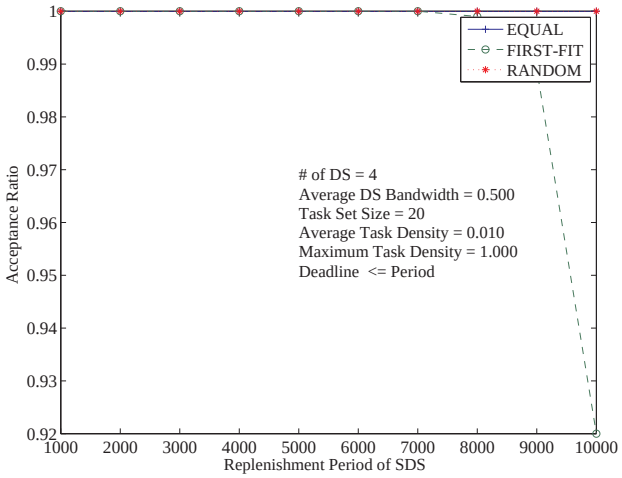

Figure 55. Acceptance ratio v.s. replenishment period

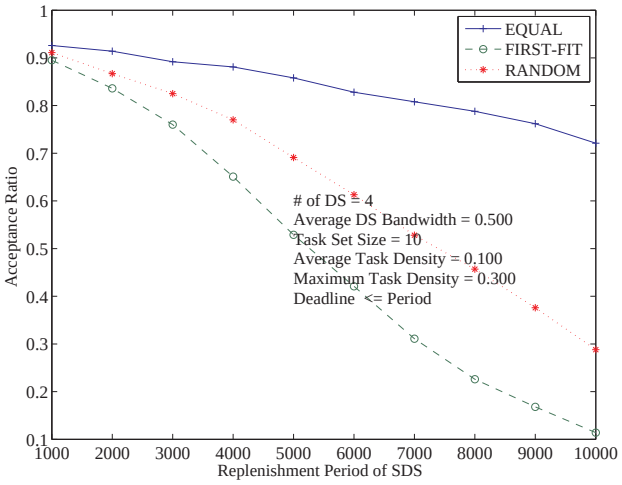

Figure 53. Acceptance ratio v.s. replenishment period

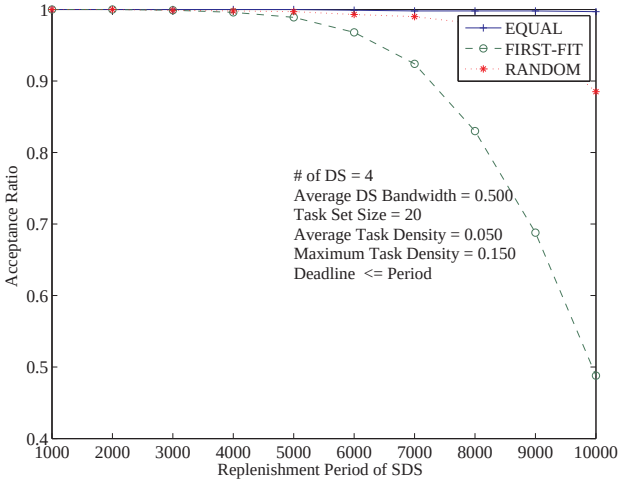

Figure 56. Acceptance ratio v.s. replenishment period

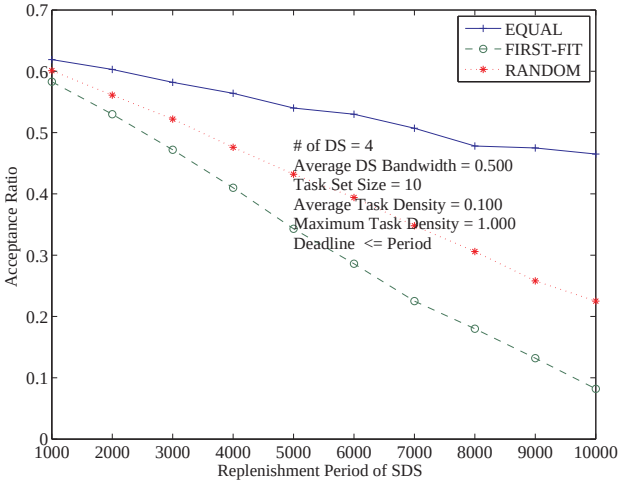

Figure 54. Acceptance ratio v.s. replenishment period

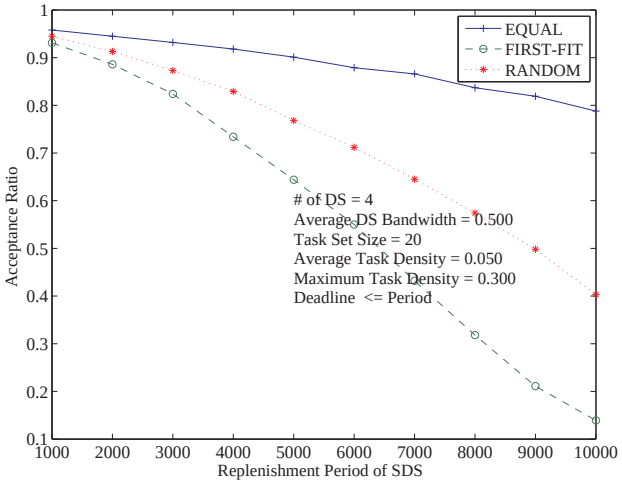

Figure 57. Acceptance ratio v.s. replenishment period

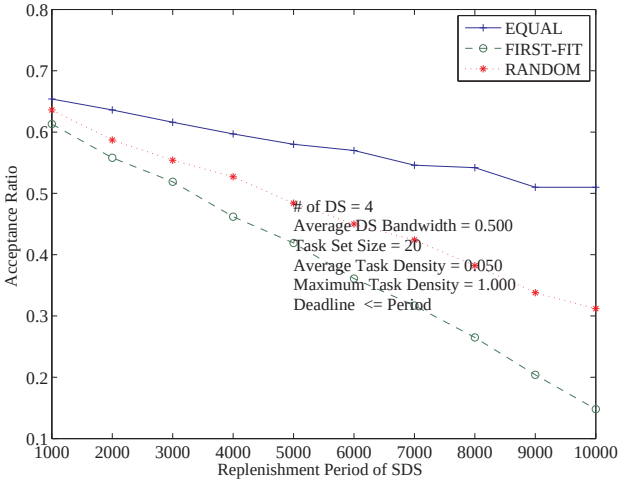

Figure 58. Acceptance ratio v.s. replenishment period

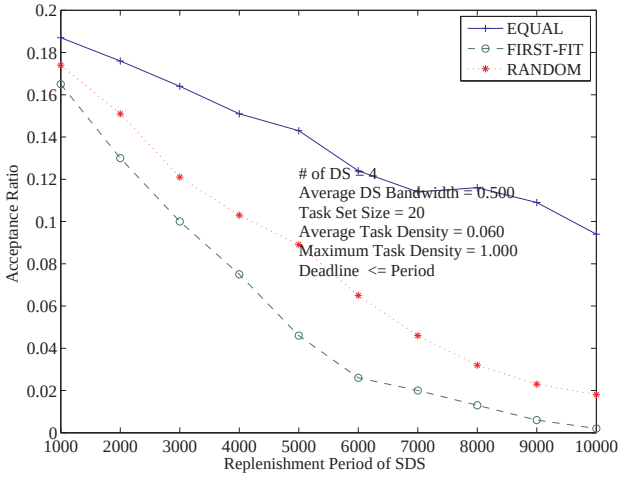

Figure 61. Acceptance ratio v.s. replenishment period

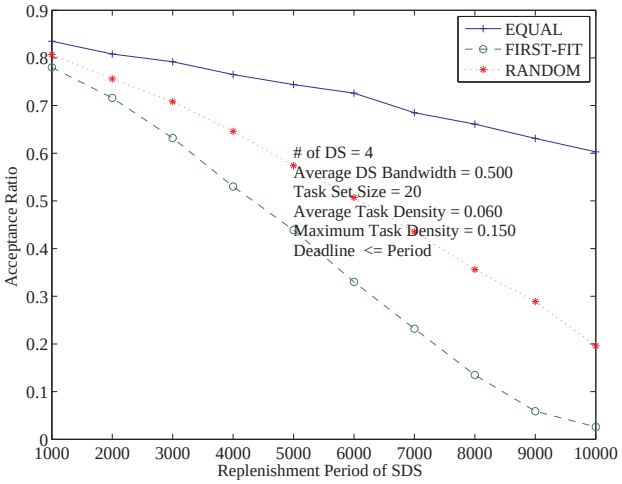

Figure 59. Acceptance ratio v.s. replenishment period

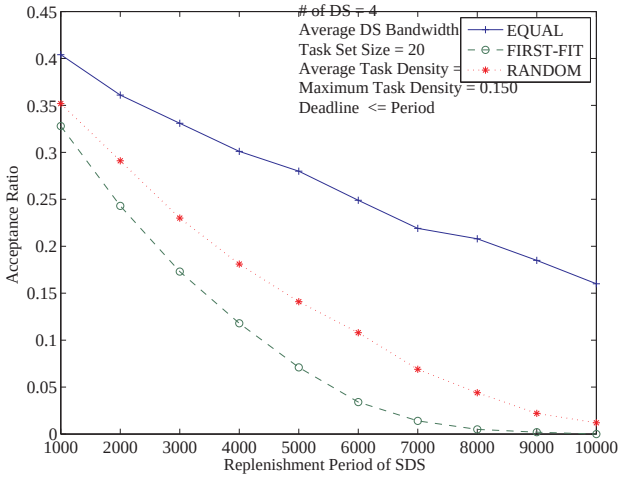

Figure 62. Acceptance ratio v.s. replenishment period

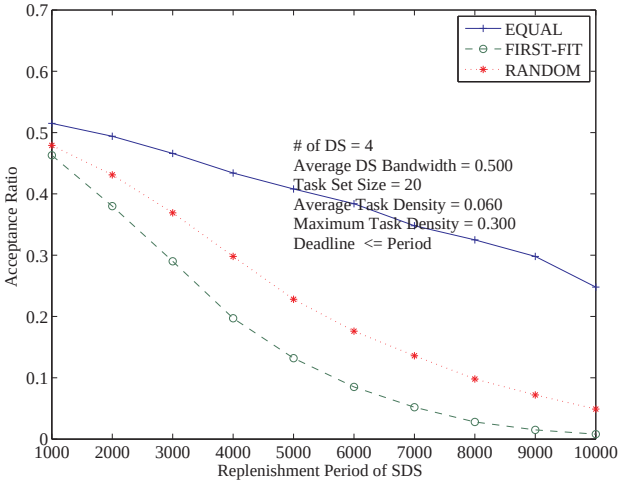

Figure 60. Acceptance ratio v.s. replenishment period

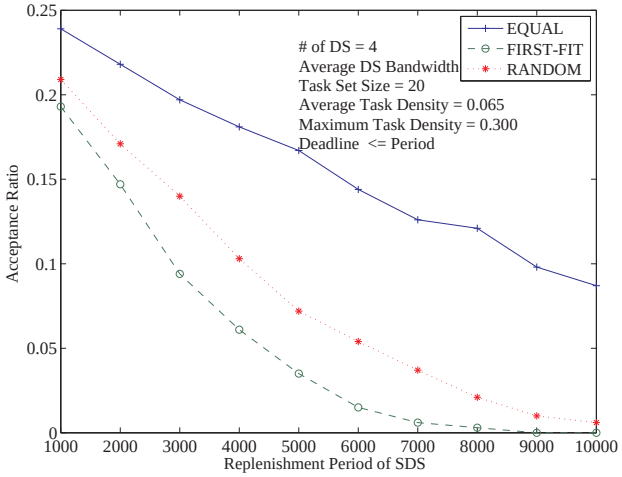

Figure 63. Acceptance ratio v.s. replenishment period

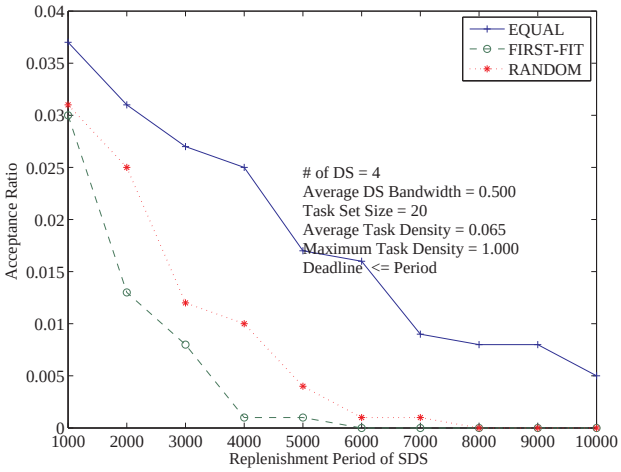

Figure 64. Acceptance ratio v.s. replenishment period

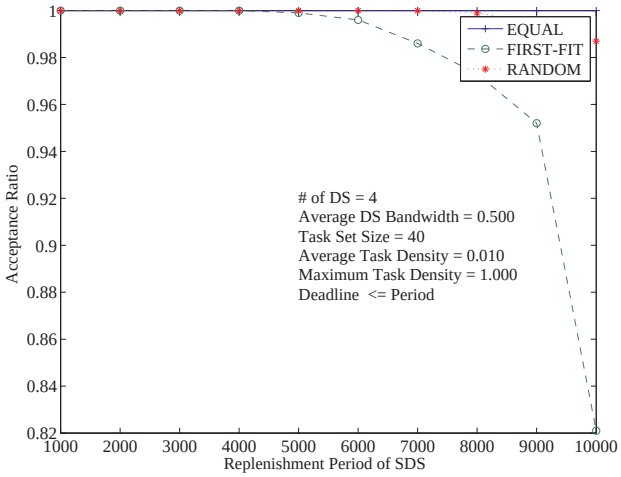

Figure 65. Acceptance ratio v.s. replenishment period

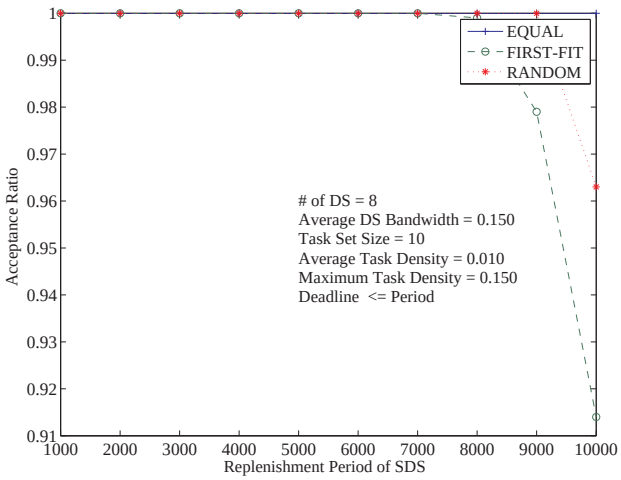

Figure 66. Acceptance ratio v.s. replenishment period

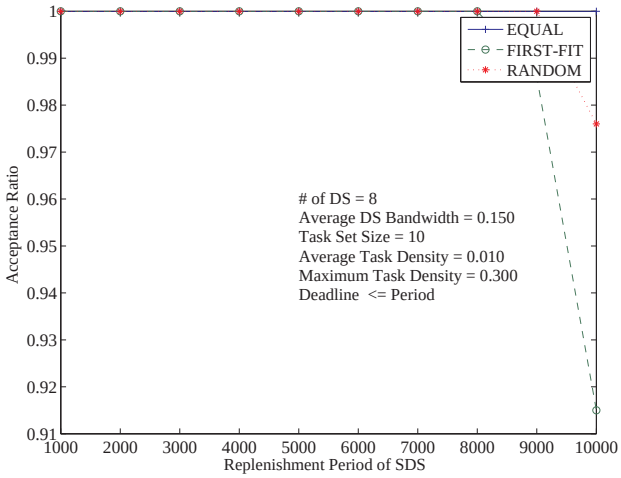

Figure 67. Acceptance ratio v.s. replenishment period

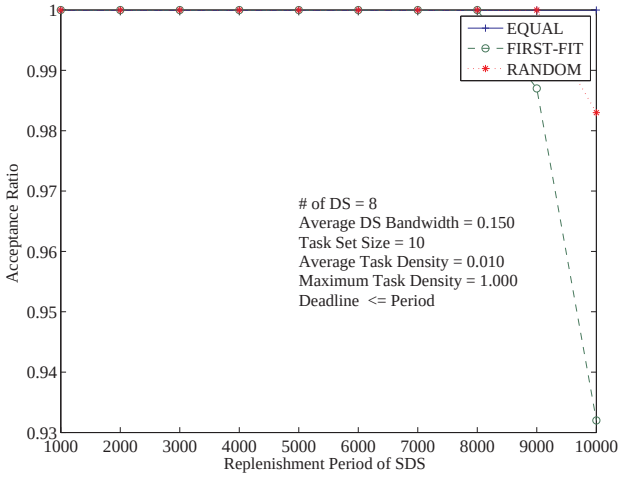

Figure 68. Acceptance ratio v.s. replenishment period

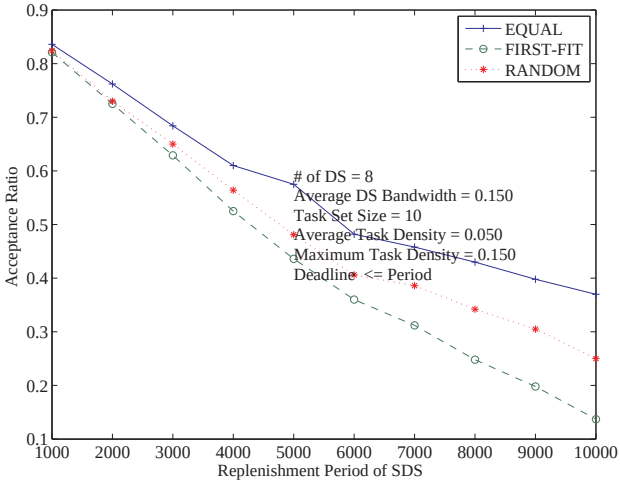

Figure 69. Acceptance ratio v.s. replenishment period

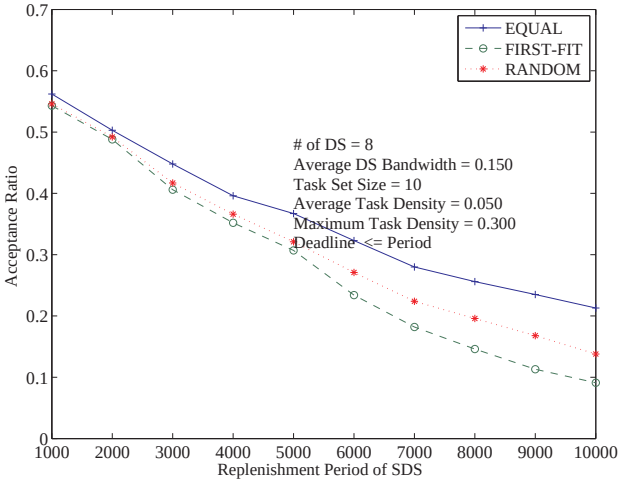

Figure 70. Acceptance ratio v.s. replenishment period

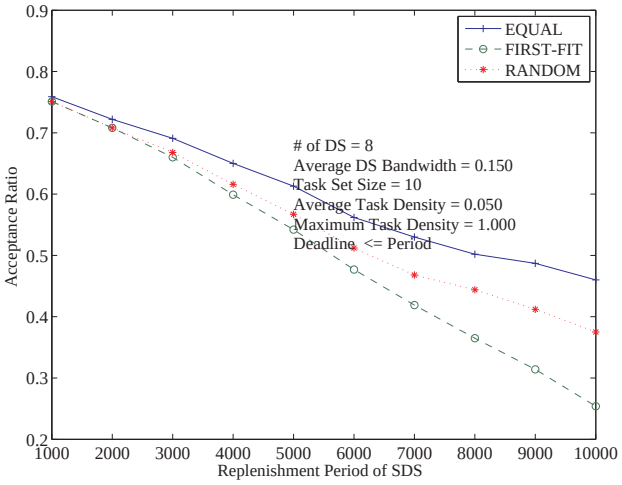

Figure 71. Acceptance ratio v.s. replenishment period

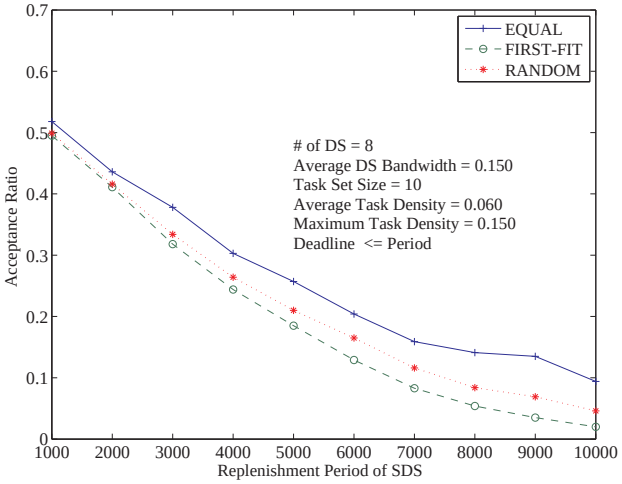

Figure 72. Acceptance ratio v.s. replenishment period

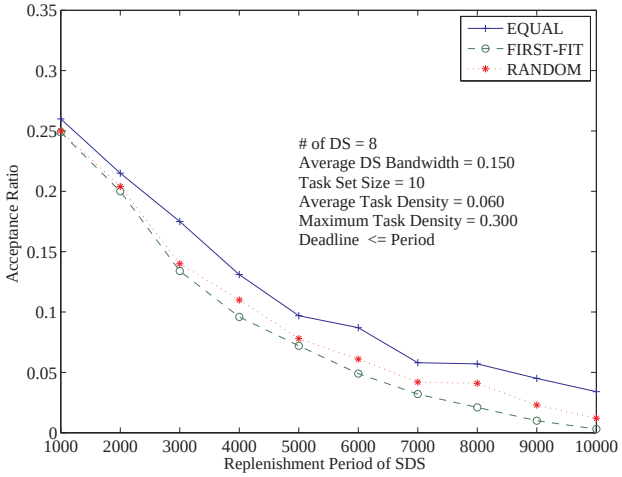

Figure 73. Acceptance ratio v.s. replenishment period

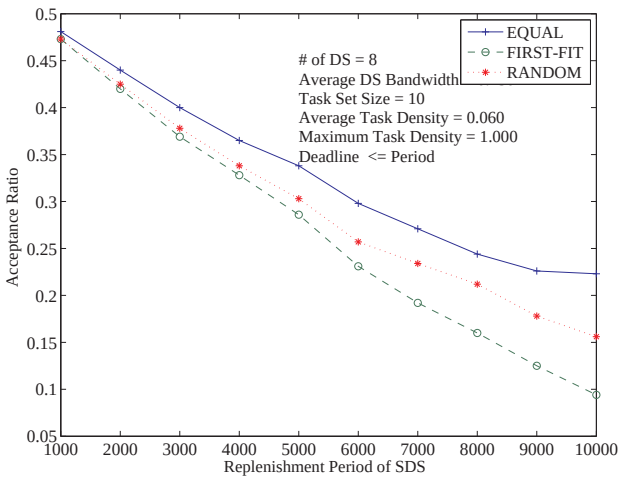

Figure 74. Acceptance ratio v.s. replenishment period

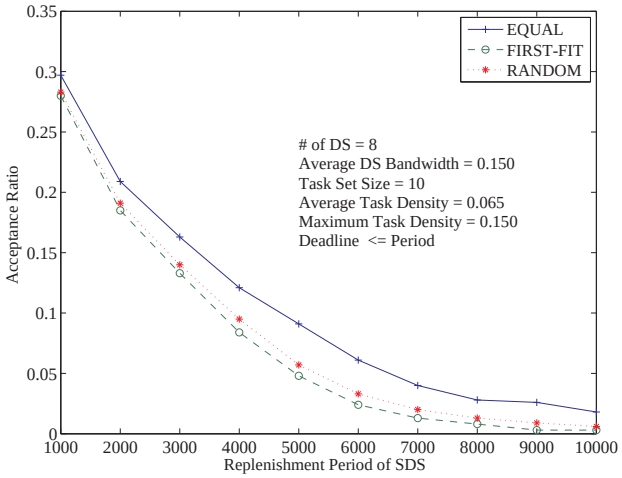

Figure 75. Acceptance ratio v.s. replenishment period

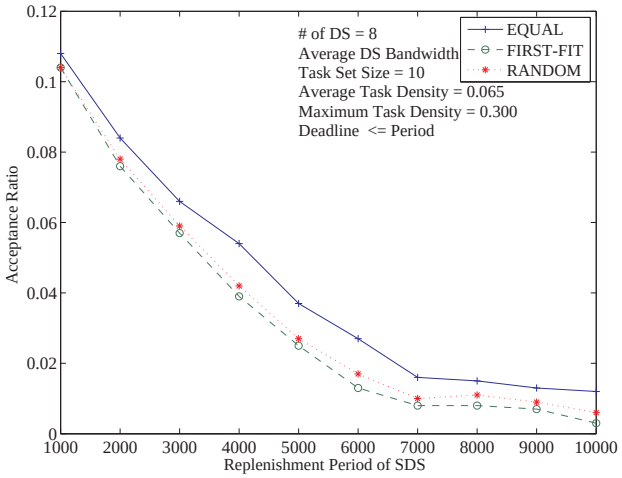

Figure 76. Acceptance ratio v.s. replenishment period

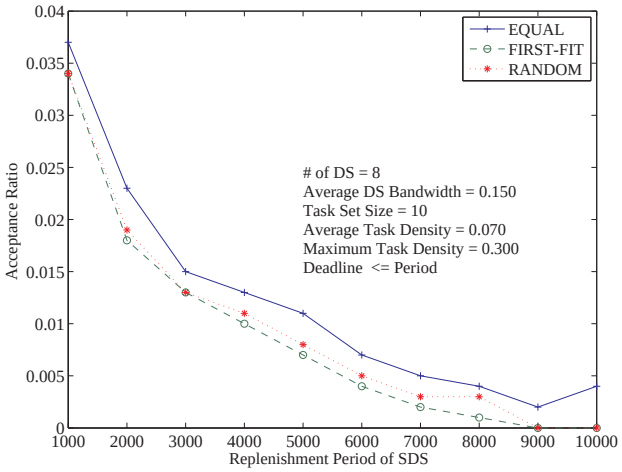

Figure 79. Acceptance ratio v.s. replenishment period

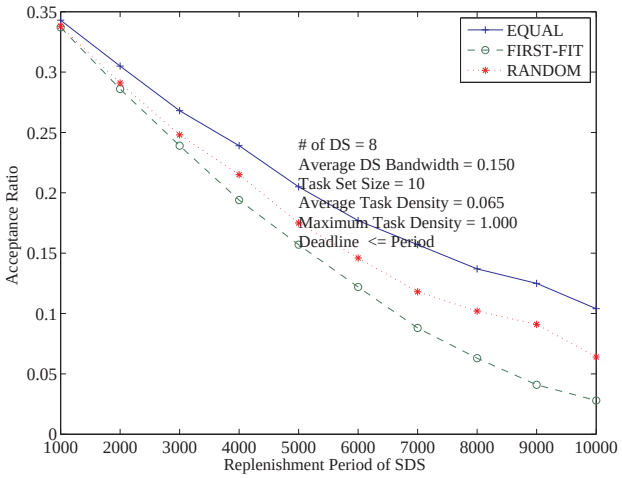

Figure 77. Acceptance ratio v.s. replenishment period

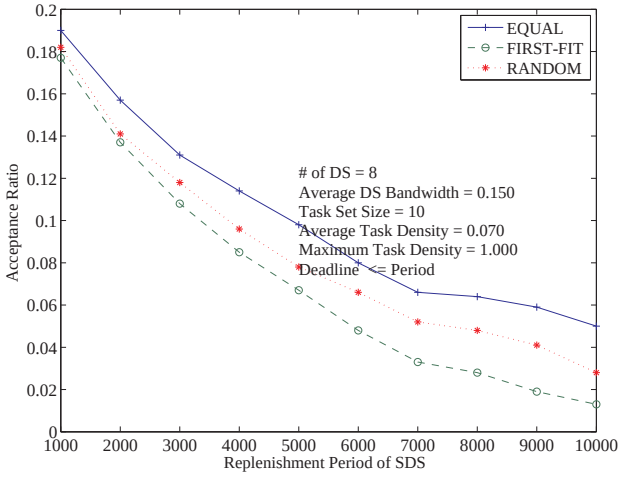

Figure 80. Acceptance ratio v.s. replenishment period

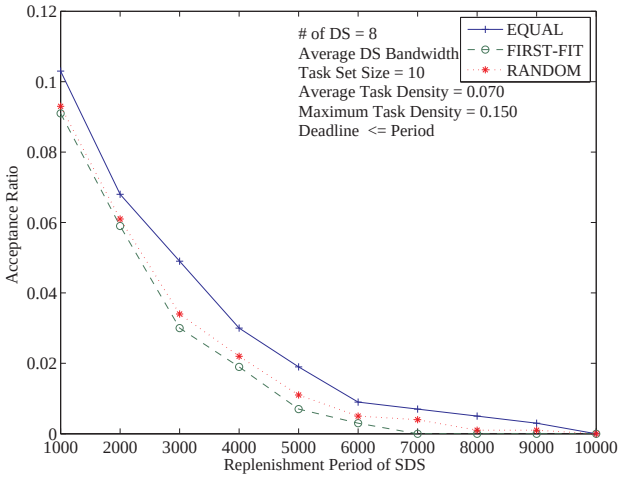

Figure 78. Acceptance ratio v.s. replenishment period

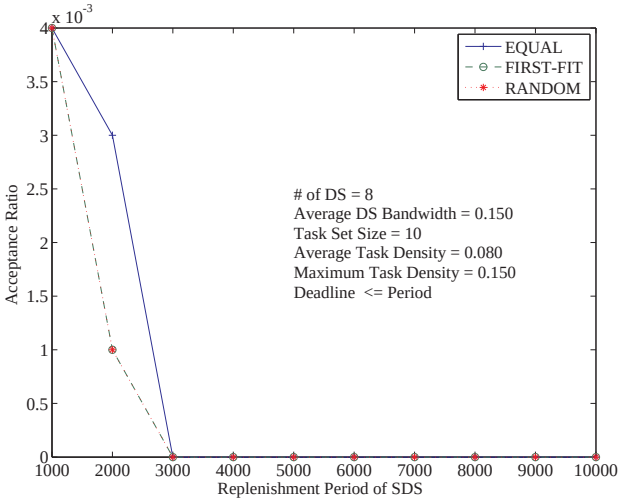

Figure 81. Acceptance ratio v.s. replenishment period

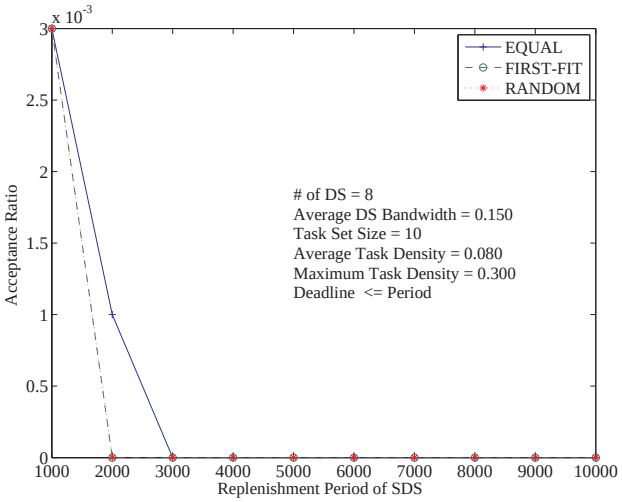

Figure 82. Acceptance ratio v.s. replenishment period

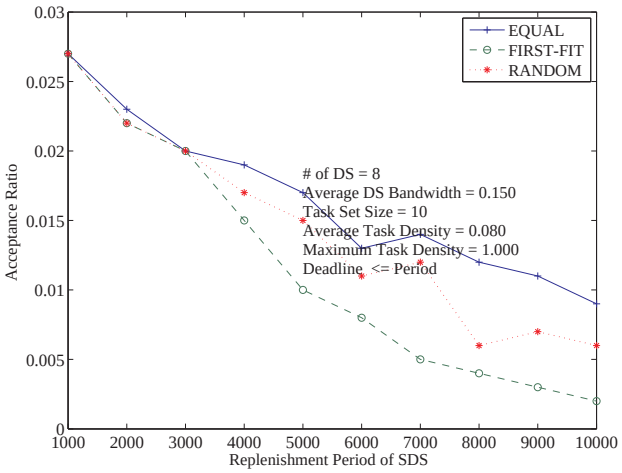

Figure 83. Acceptance ratio v.s. replenishment period

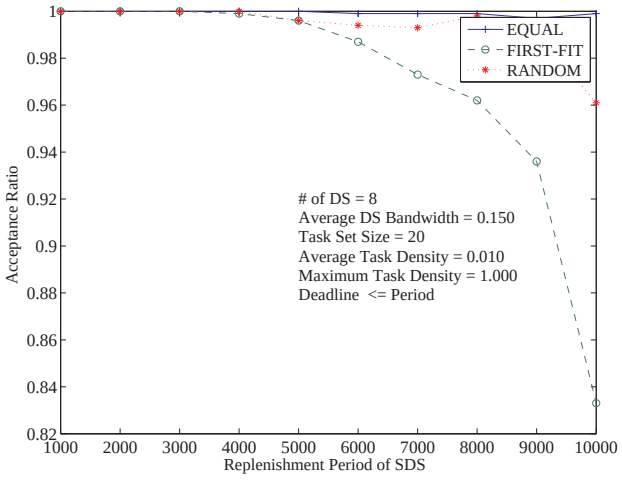

Figure 84. Acceptance ratio v.s. replenishment period

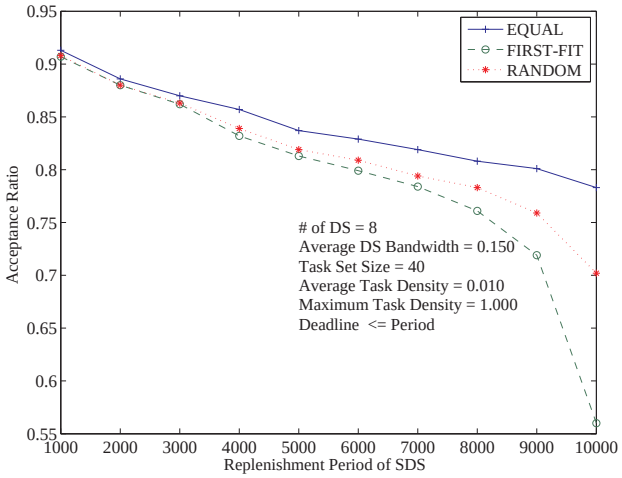

Figure 85. Acceptance ratio v.s. replenishment period

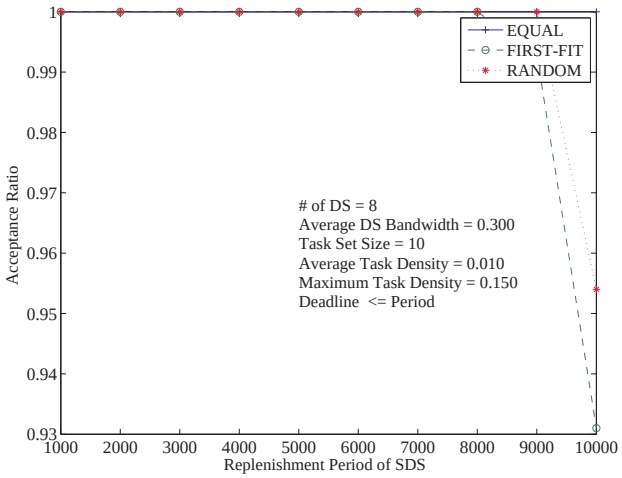

Figure 86. Acceptance ratio v.s. replenishment period

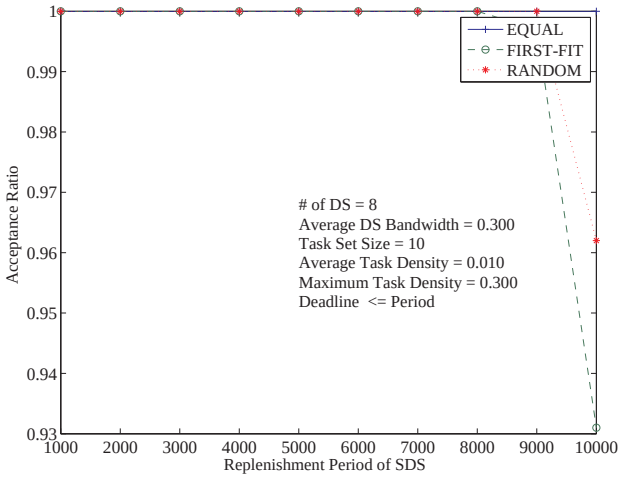

Figure 87. Acceptance ratio v.s. replenishment period

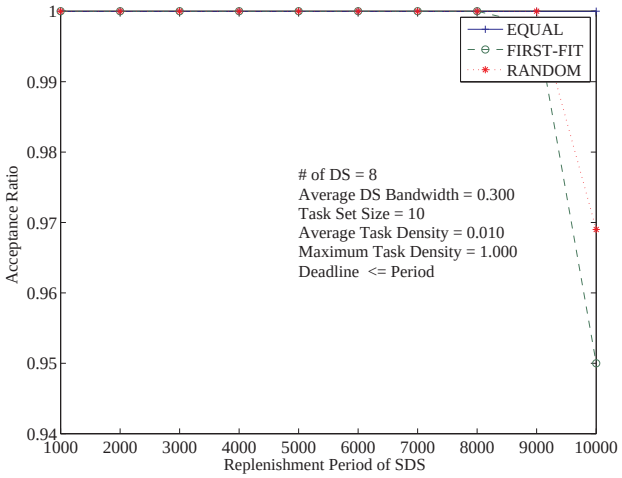

Figure 88. Acceptance ratio v.s. replenishment period

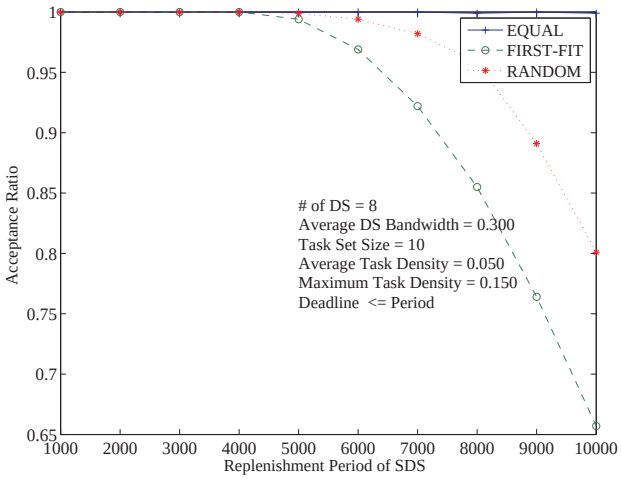

Figure 89. Acceptance ratio v.s. replenishment period

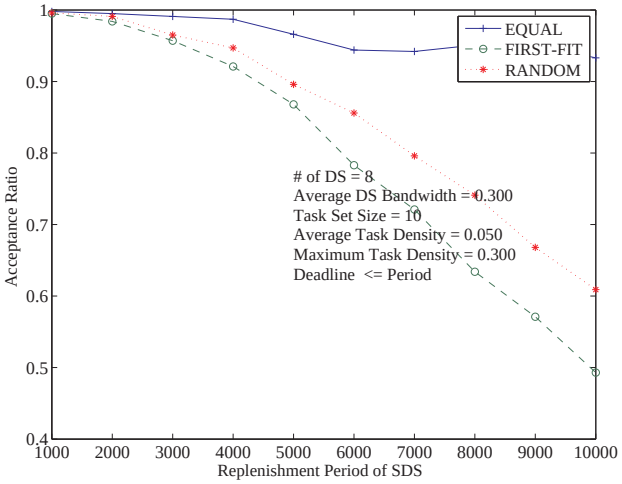

Figure 90. Acceptance ratio v.s. replenishment period

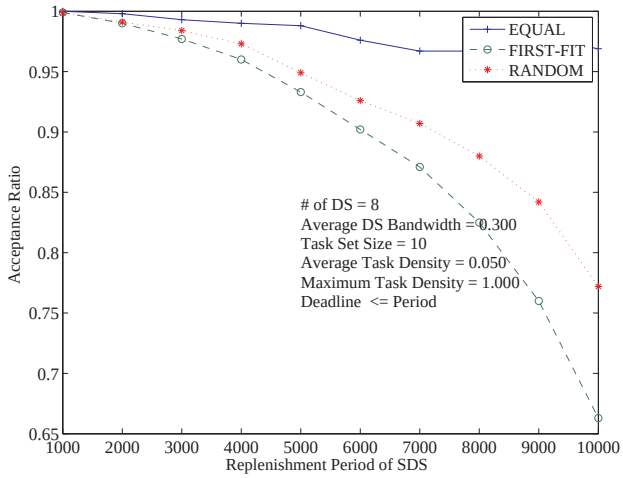

Figure 91. Acceptance ratio v.s. replenishment period

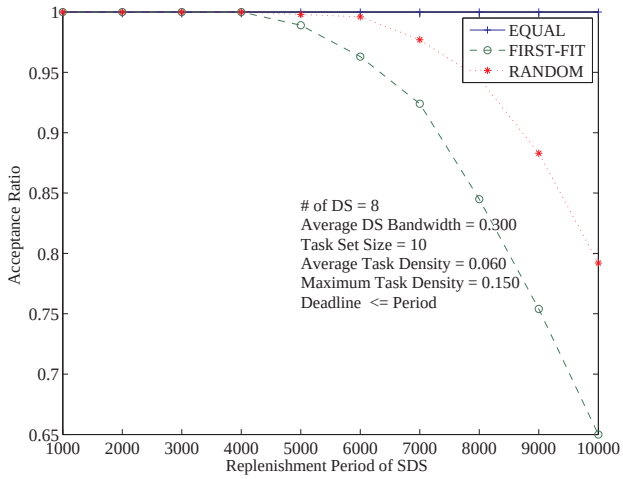

Figure 92. Acceptance ratio v.s. replenishment period

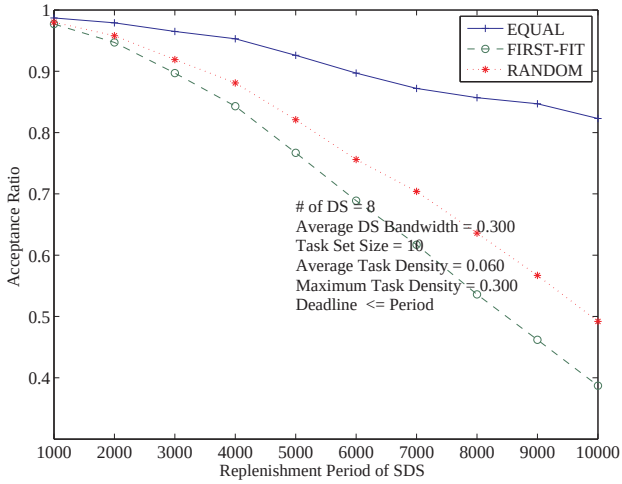

Figure 93. Acceptance ratio v.s. replenishment period

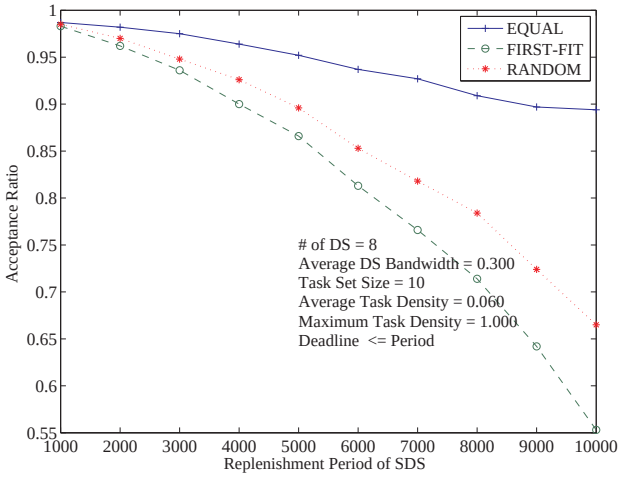

Figure 94. Acceptance ratio v.s. replenishment period

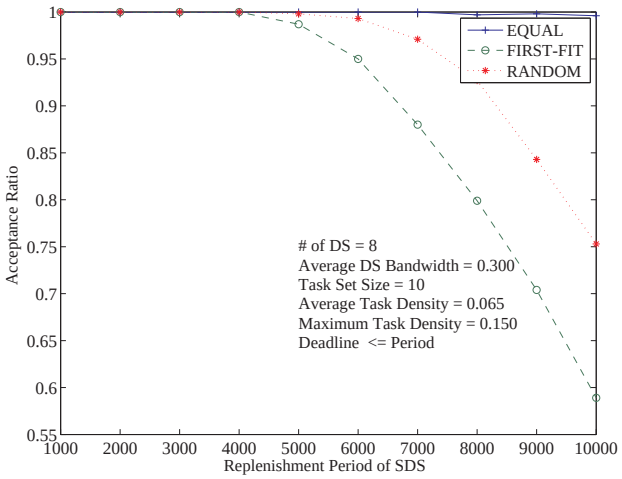

Figure 95. Acceptance ratio v.s. replenishment period

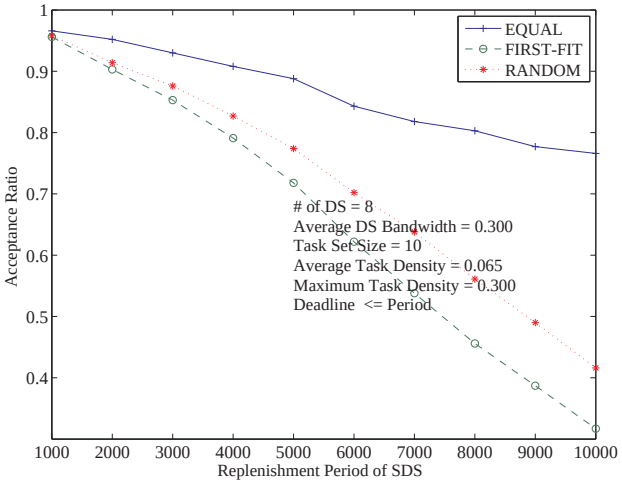

Figure 96. Acceptance ratio v.s. replenishment period

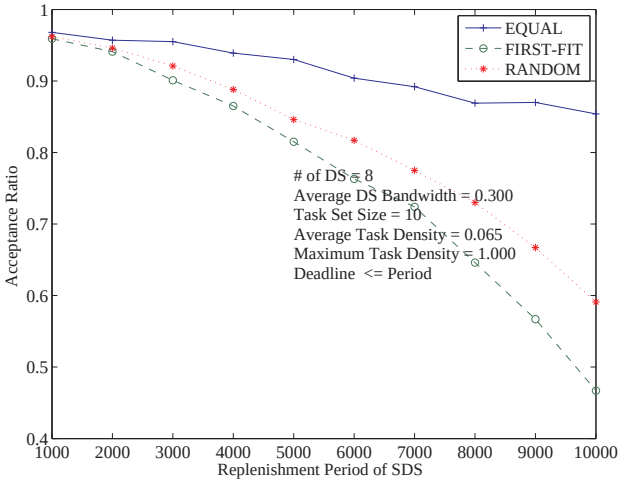

Figure 97. Acceptance ratio v.s. replenishment period

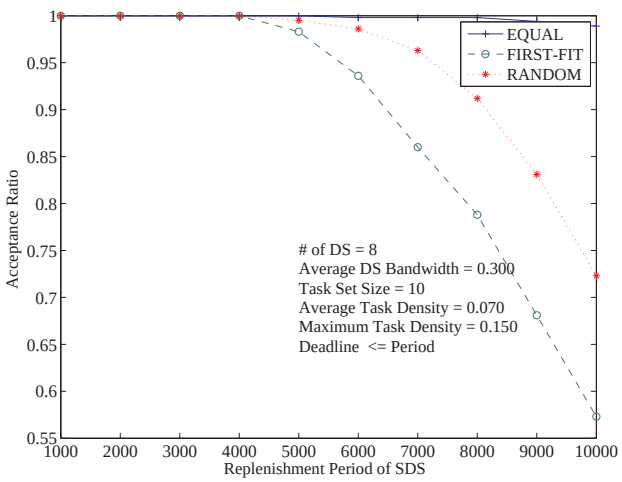

Figure 98. Acceptance ratio v.s. replenishment period

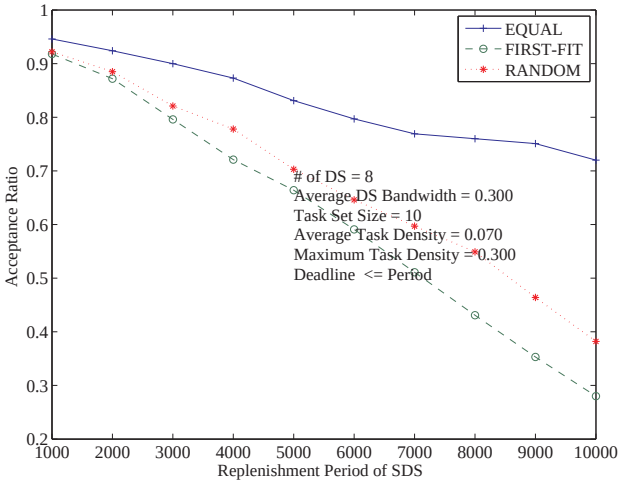

Figure 99. Acceptance ratio v.s. replenishment period

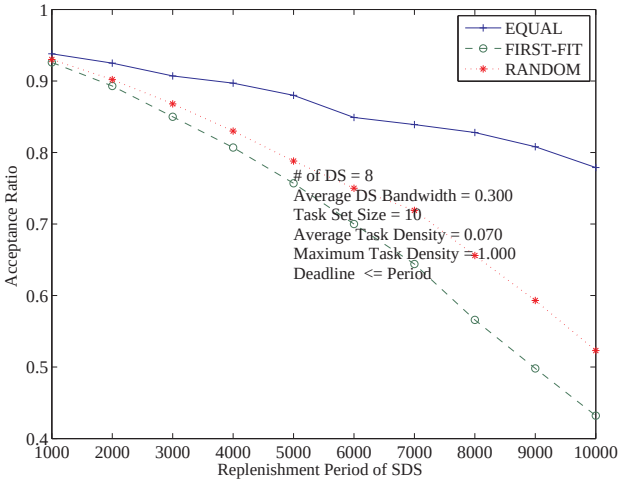

Figure 100. Acceptance ratio v.s. replenishment period

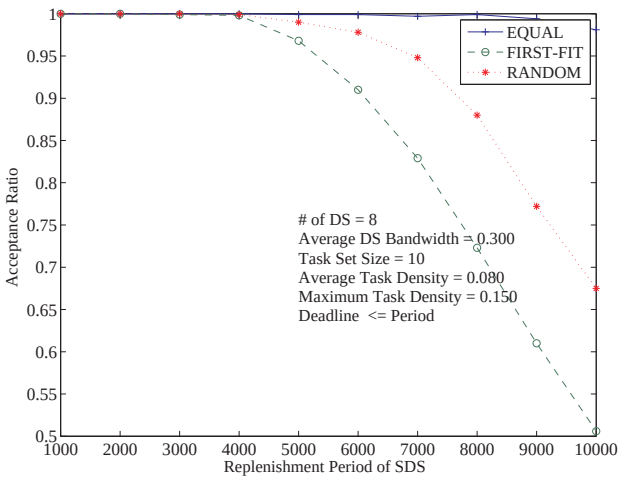

Figure 101. Acceptance ratio v.s. replenishment period

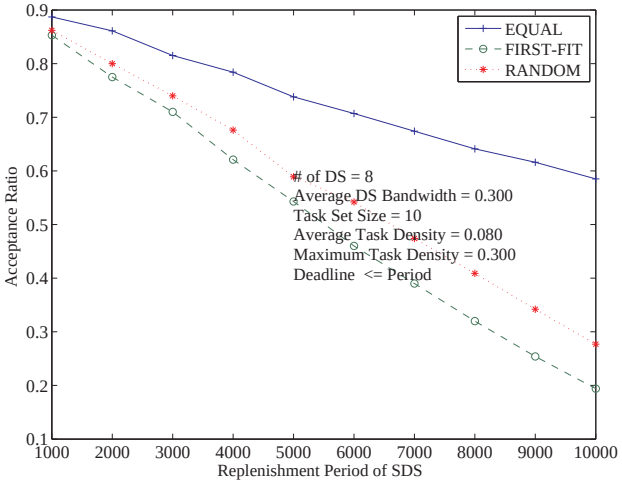

Figure 102. Acceptance ratio v.s. replenishment period

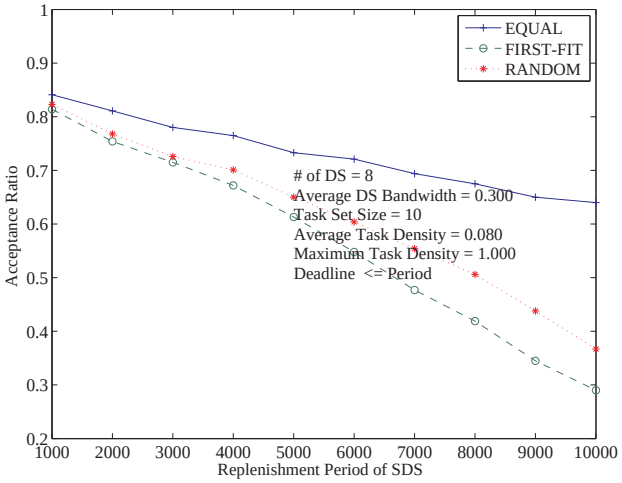

Figure 103. Acceptance ratio v.s. replenishment period

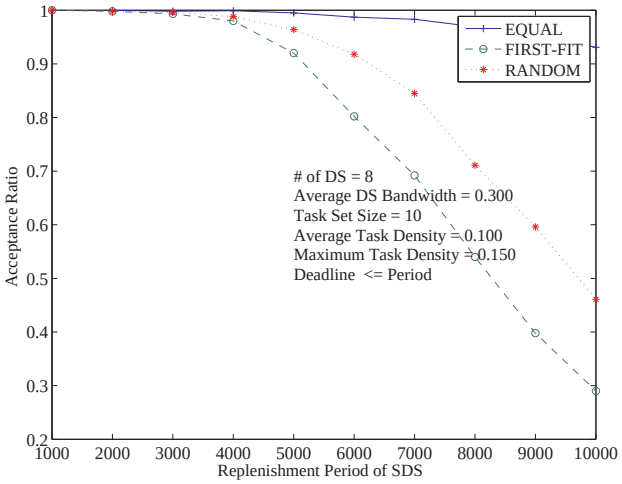

Figure 104. Acceptance ratio v.s. replenishment period

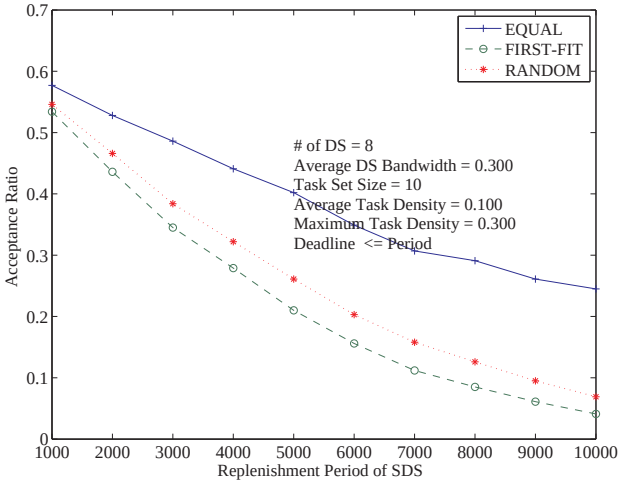

Figure 105. Acceptance ratio v.s. replenishment period

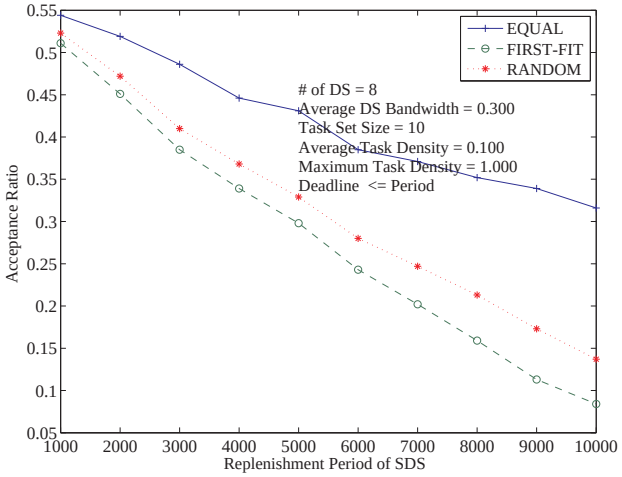

Figure 106. Acceptance ratio v.s. replenishment period

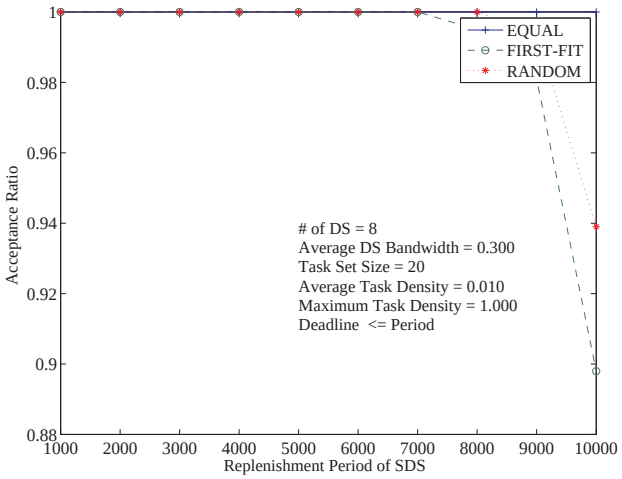

Figure 107. Acceptance ratio v.s. replenishment period

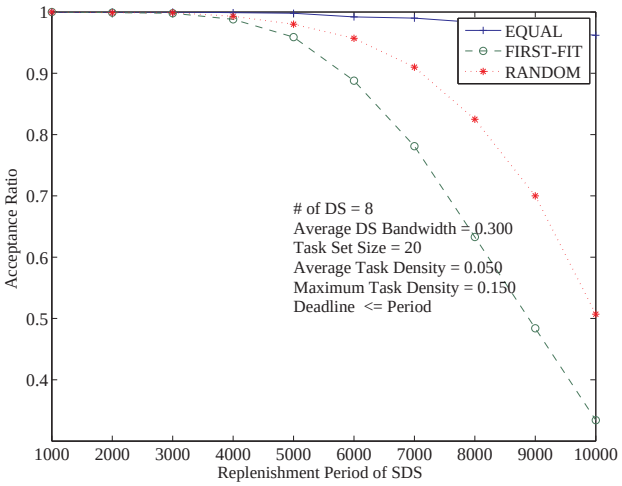

Figure 108. Acceptance ratio v.s. replenishment period

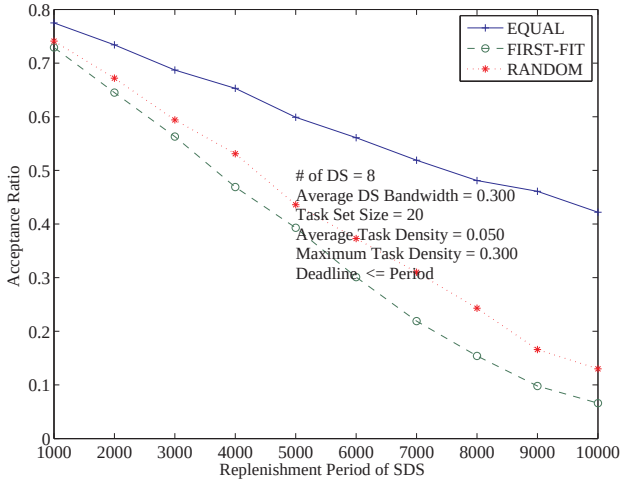

Figure 109. Acceptance ratio v.s. replenishment period

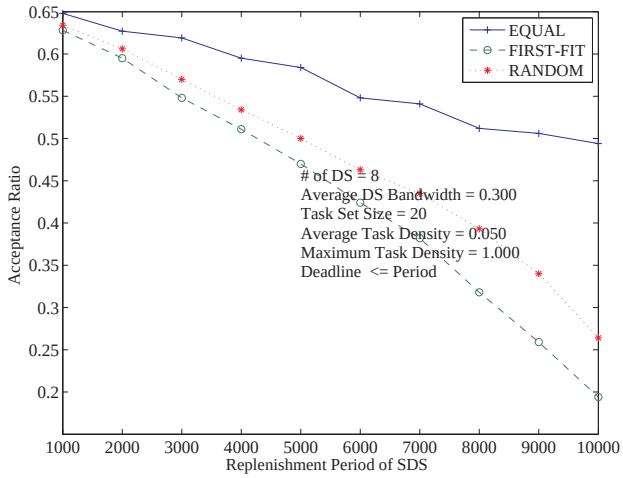

Figure 110. Acceptance ratio v.s. replenishment period

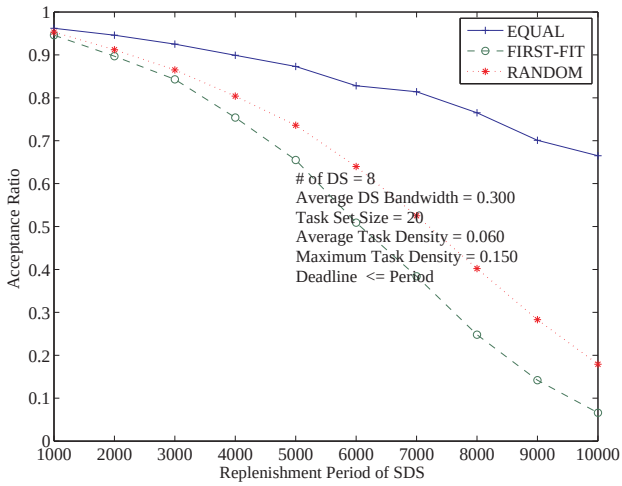

Figure 111. Acceptance ratio v.s. replenishment period

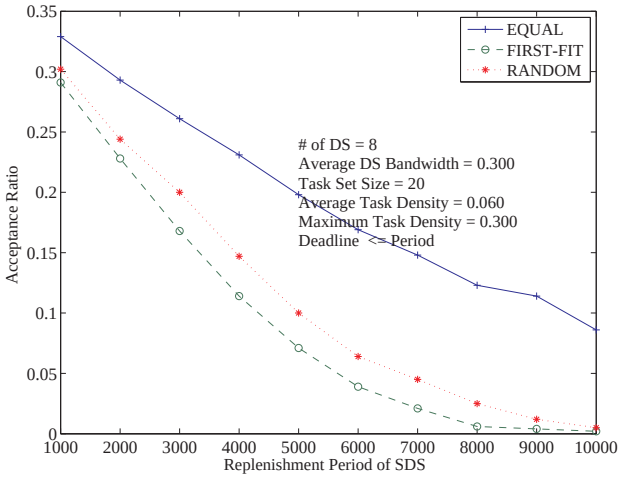

Figure 112. Acceptance ratio v.s. replenishment period

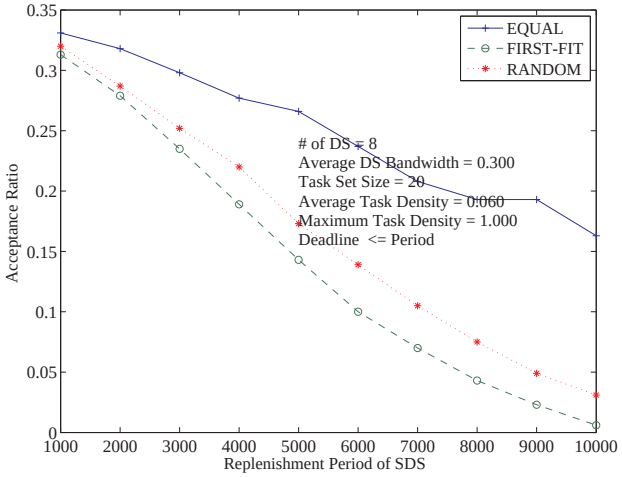

Figure 113. Acceptance ratio v.s. replenishment period

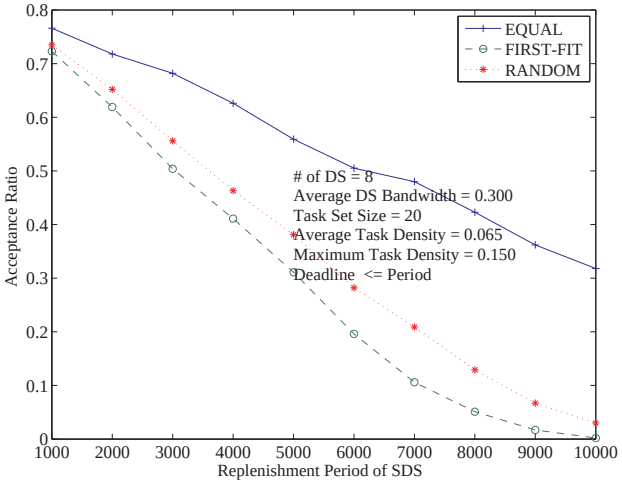

Figure 114. Acceptance ratio v.s. replenishment period

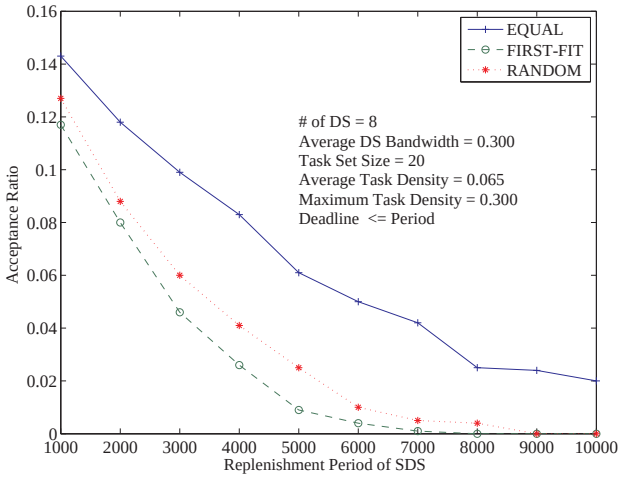

Figure 115. Acceptance ratio v.s. replenishment period

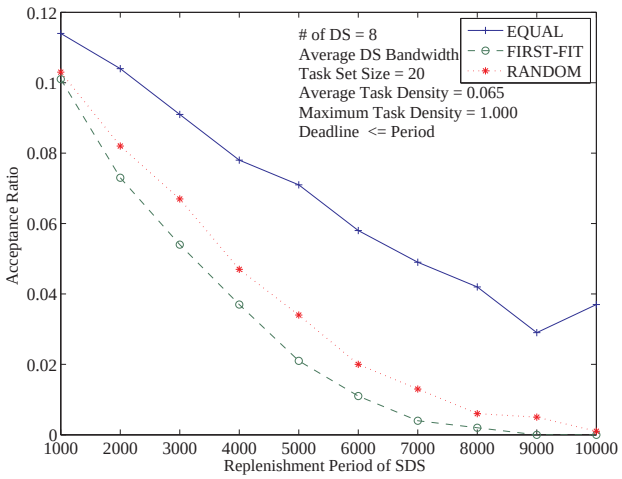

Figure 116. Acceptance ratio v.s. replenishment period

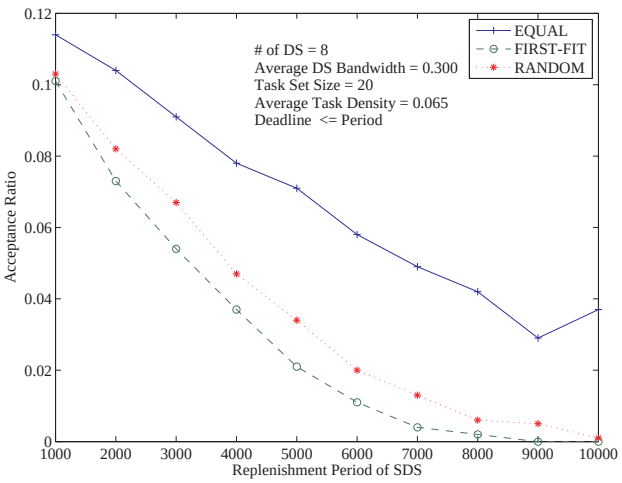

Figure 117. Acceptance ratio v.s. replenishment period

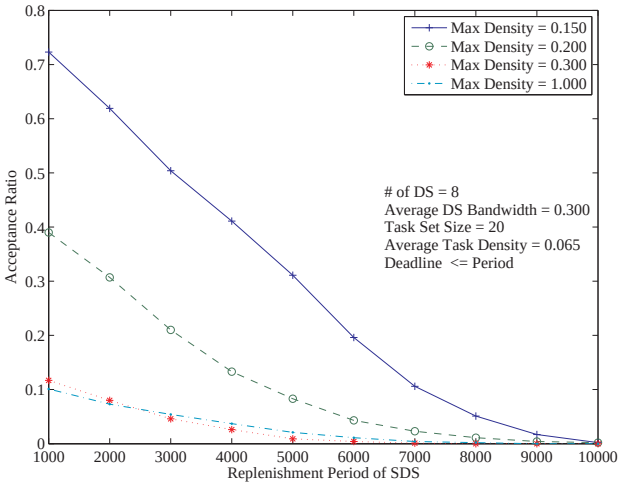

Figure 118. Acceptance ratio v.s. replenishment period

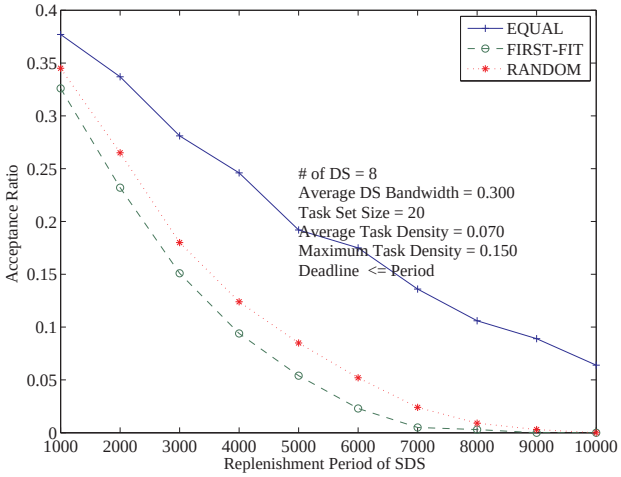

Figure 119. Acceptance ratio v.s. replenishment period

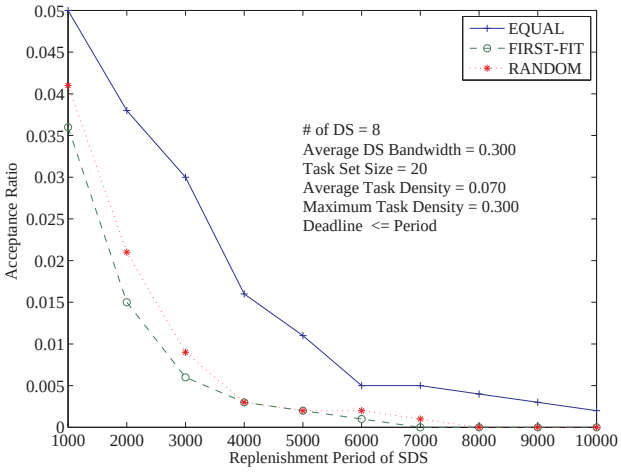

Figure 120. Acceptance ratio v.s. replenishment period

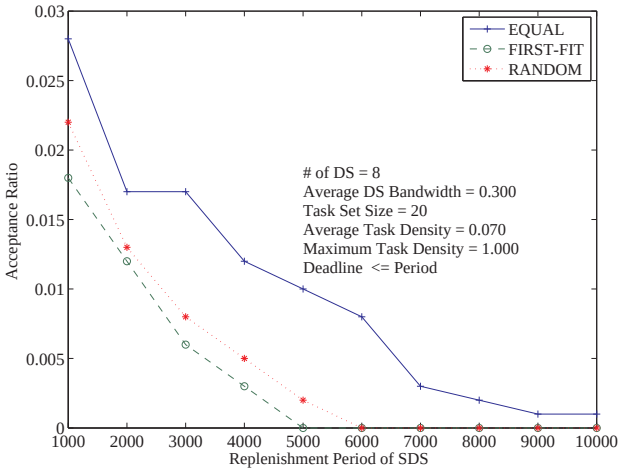

Figure 121. Acceptance ratio v.s. replenishment period

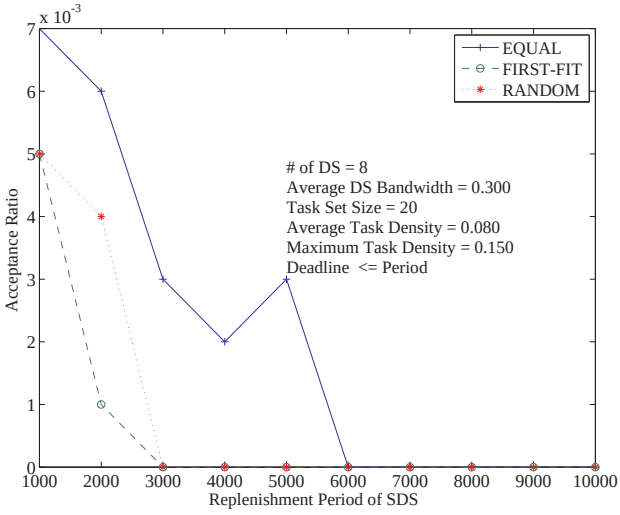

Figure 122. Acceptance ratio v.s. replenishment period

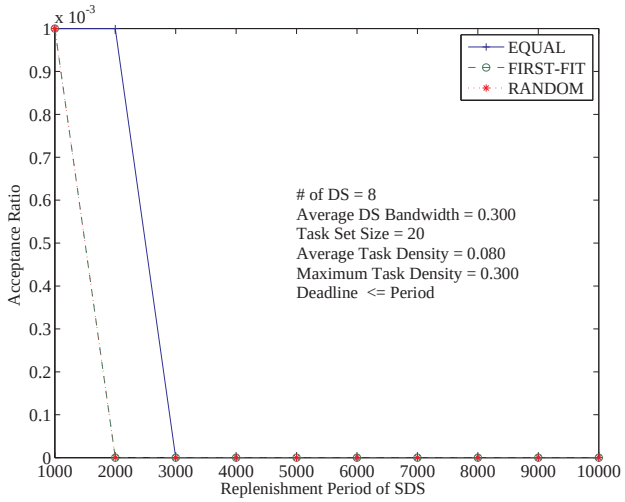

Figure 123. Acceptance ratio v.s. replenishment period

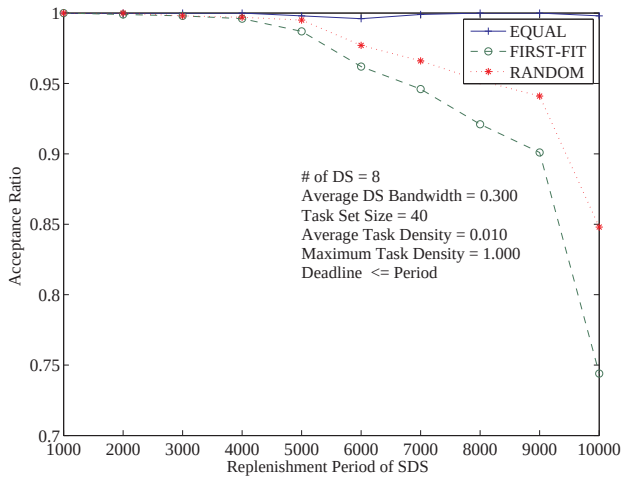

Figure 124. Acceptance ratio v.s. replenishment period

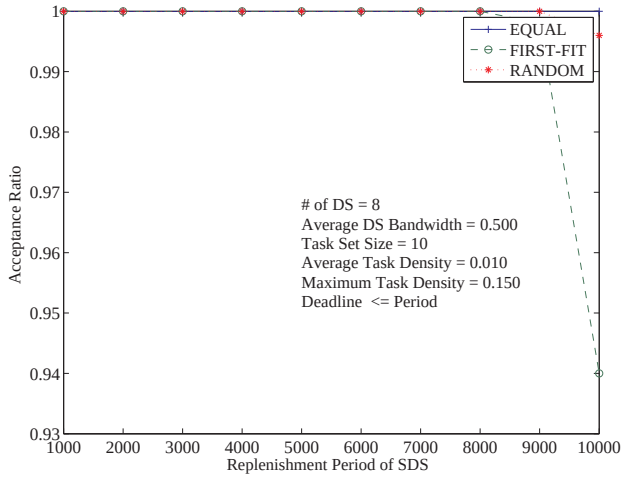

Figure 125. Acceptance ratio v.s. replenishment period

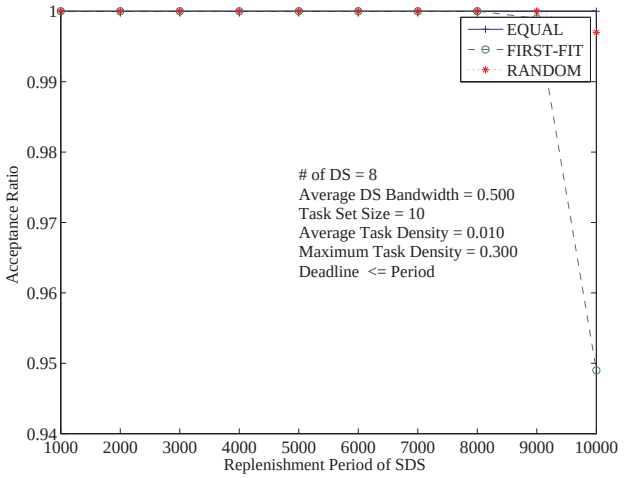

Figure 126. Acceptance ratio v.s. replenishment period

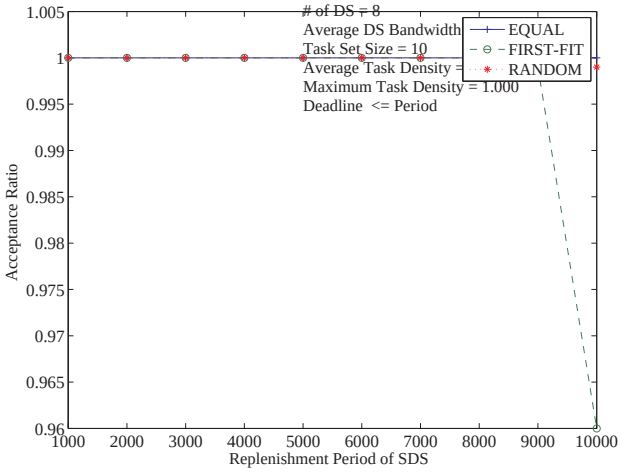

Figure 127. Acceptance ratio v.s. replenishment period

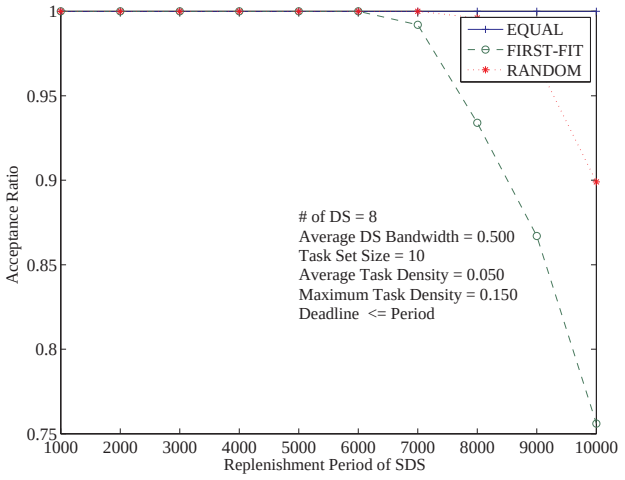

Figure 128. Acceptance ratio v.s. replenishment period

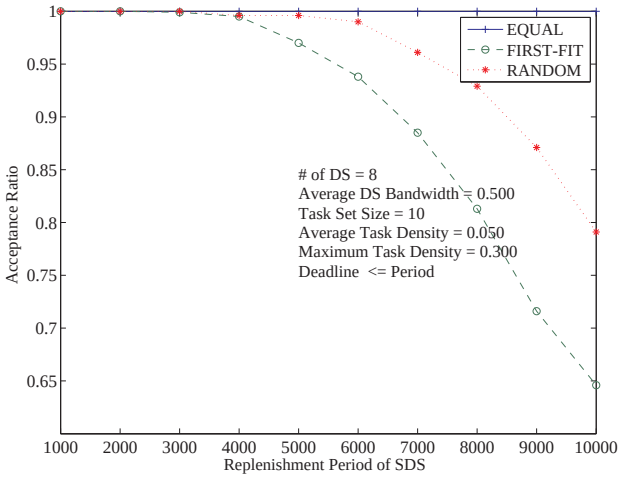

Figure 129. Acceptance ratio v.s. replenishment period

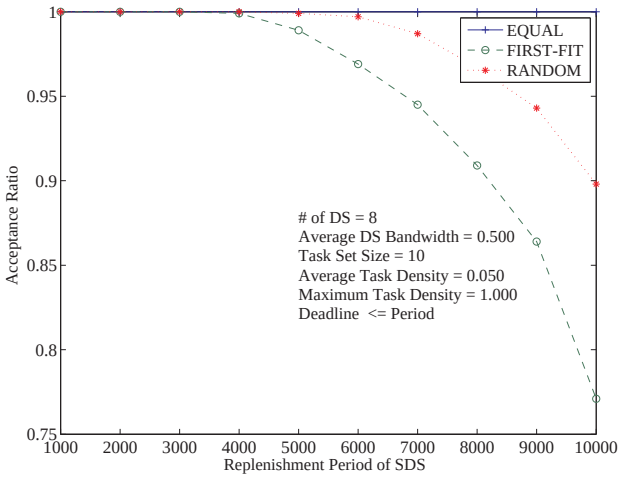

Figure 130. Acceptance ratio v.s. replenishment period

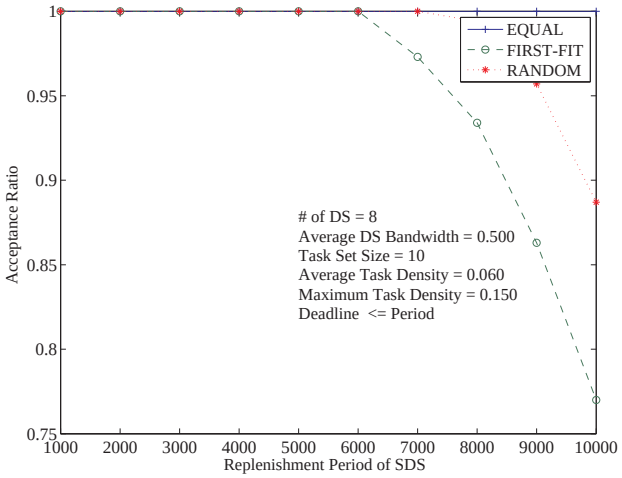

Figure 131. Acceptance ratio v.s. replenishment period

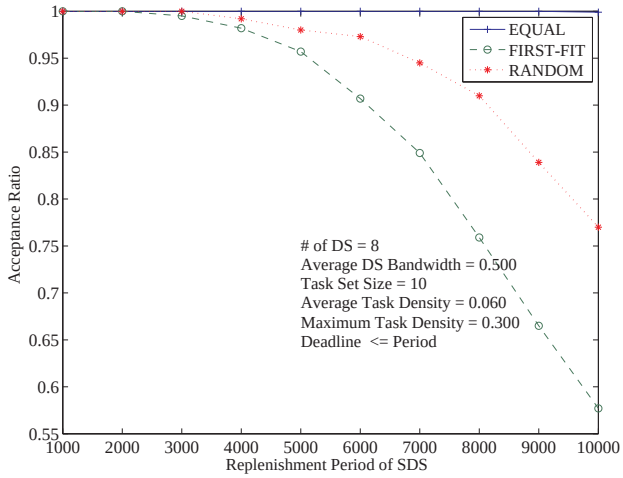

Figure 132. Acceptance ratio v.s. replenishment period

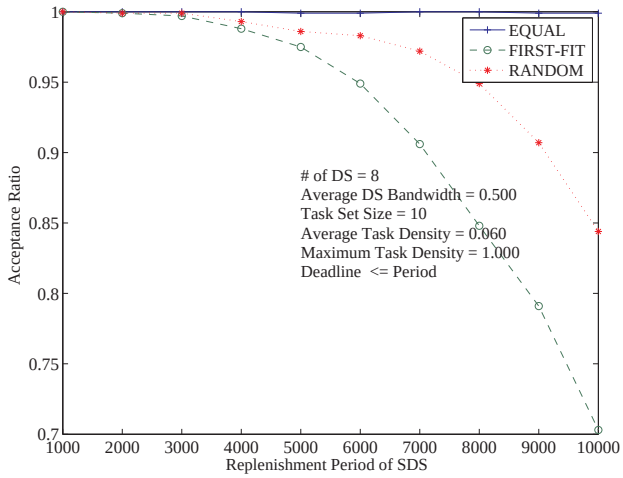

Figure 133. Acceptance ratio v.s. replenishment period

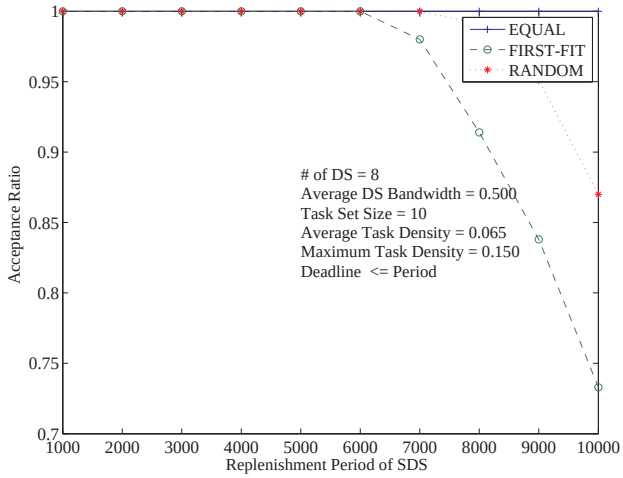

Figure 134. Acceptance ratio v.s. replenishment period

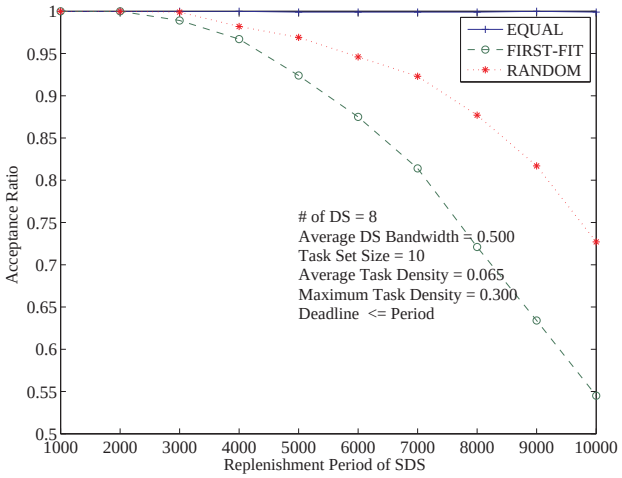

Figure 135. Acceptance ratio v.s. replenishment period

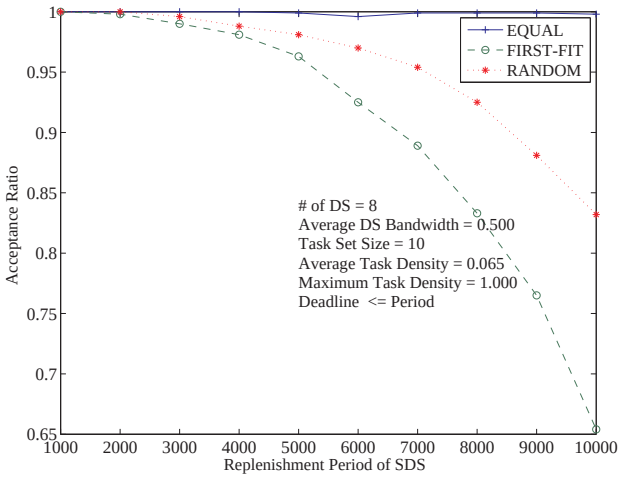

Figure 136. Acceptance ratio v.s. replenishment period

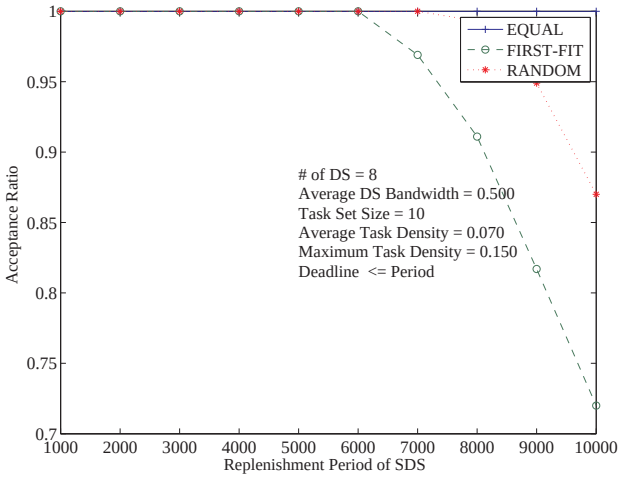

Figure 137. Acceptance ratio v.s. replenishment period

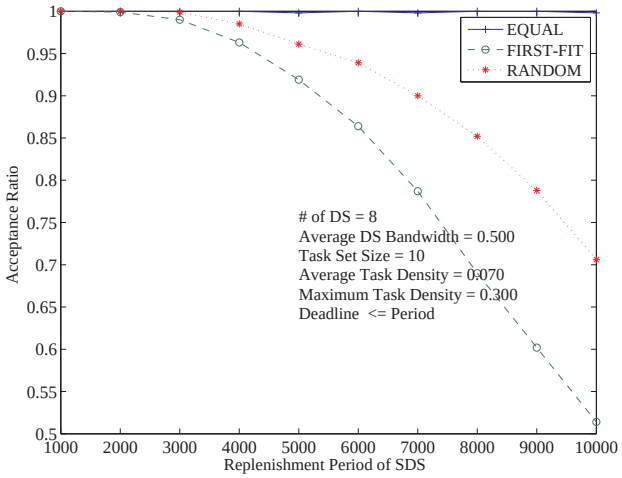

Figure 138. Acceptance ratio v.s. replenishment period

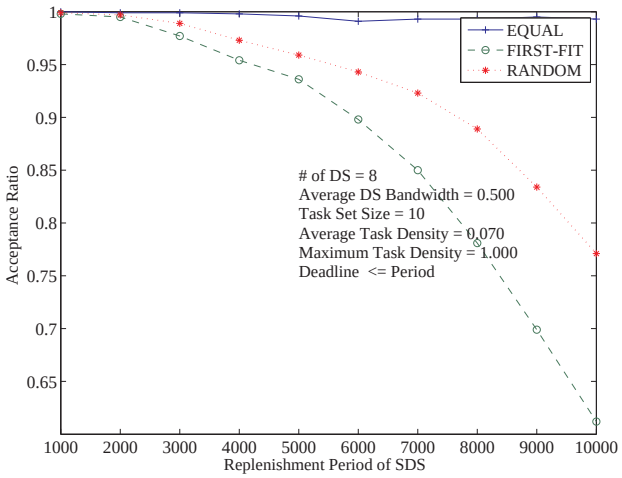

Figure 139. Acceptance ratio v.s. replenishment period

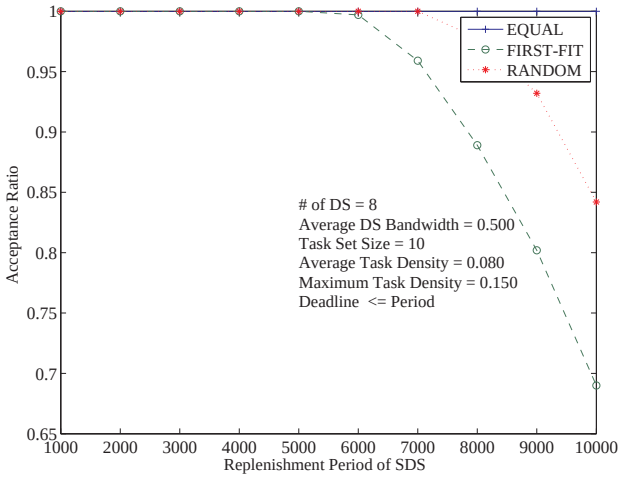

Figure 140. Acceptance ratio v.s. replenishment period

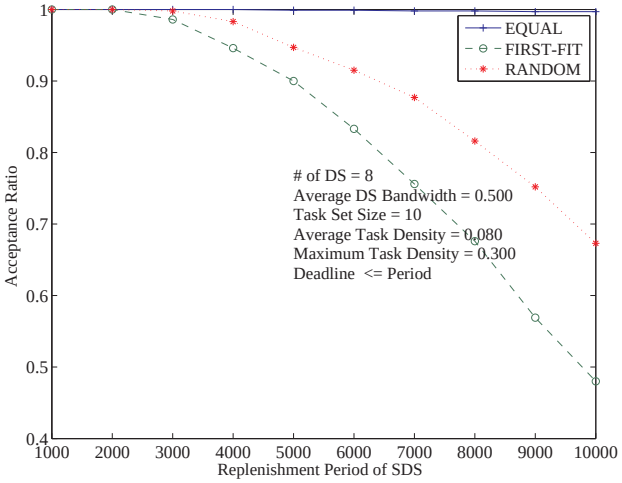

Figure 141. Acceptance ratio v.s. replenishment period

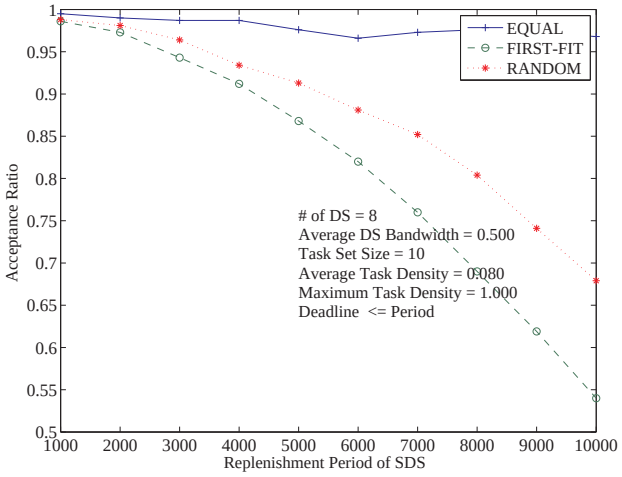

Figure 142. Acceptance ratio v.s. replenishment period

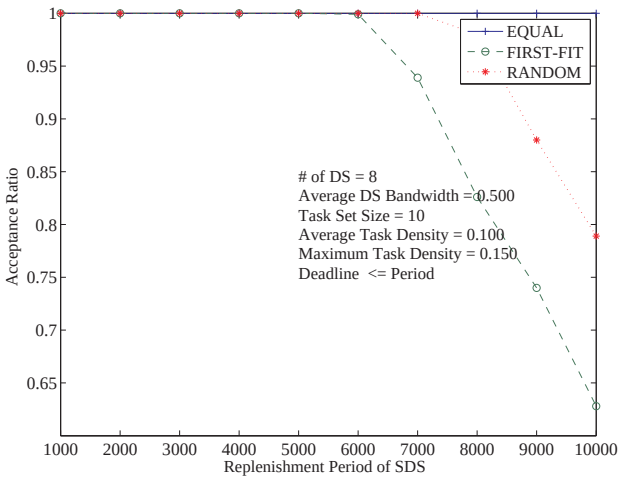

Figure 143. Acceptance ratio v.s. replenishment period

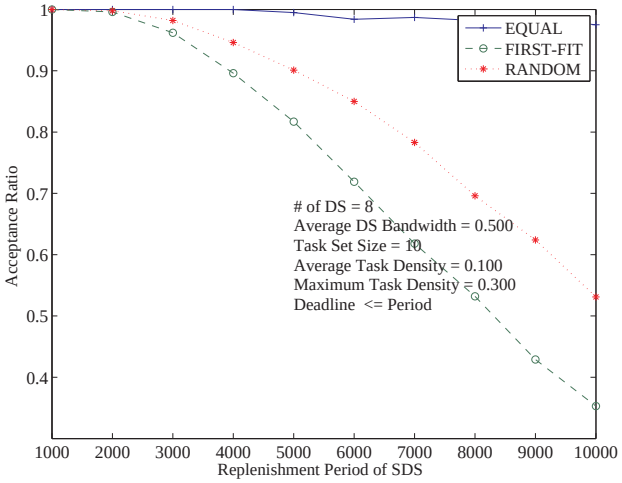

Figure 144. Acceptance ratio v.s. replenishment period

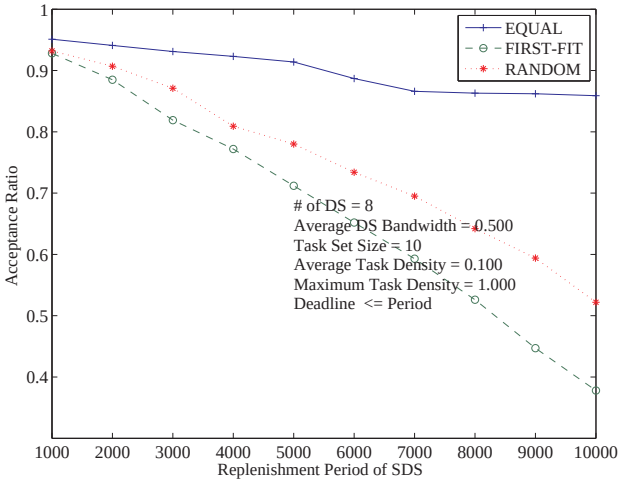

Figure 145. Acceptance ratio v.s. replenishment period

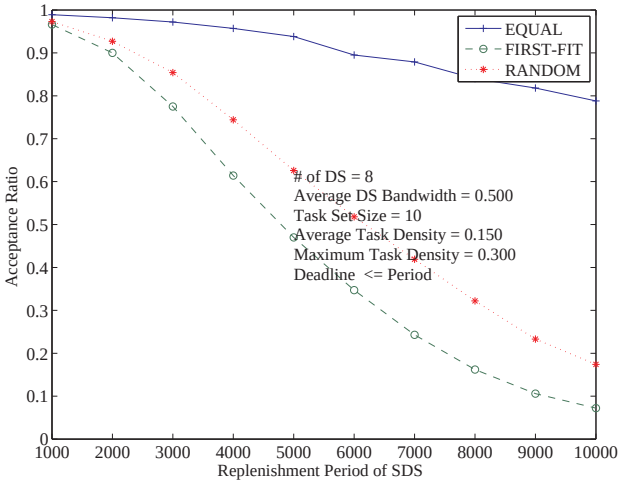

Figure 146. Acceptance ratio v.s. replenishment period

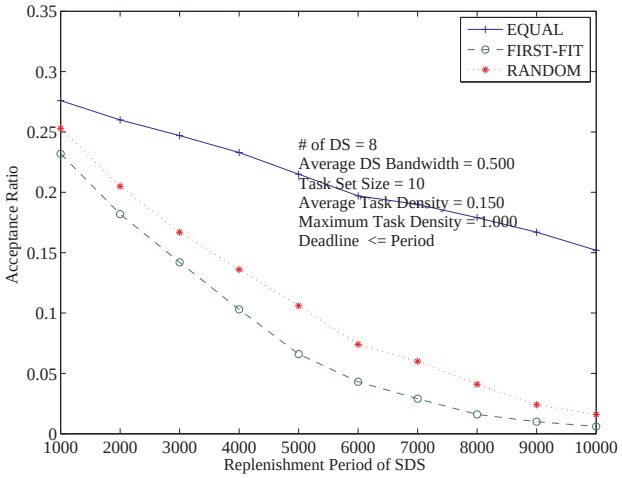

Figure 147. Acceptance ratio v.s. replenishment period

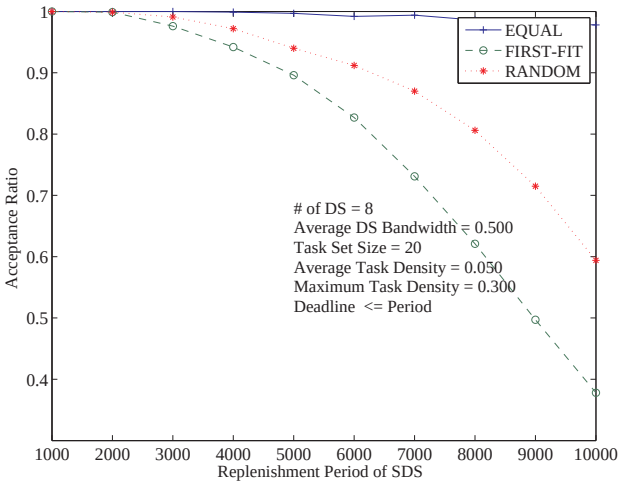

Figure 150. Acceptance ratio v.s. replenishment period

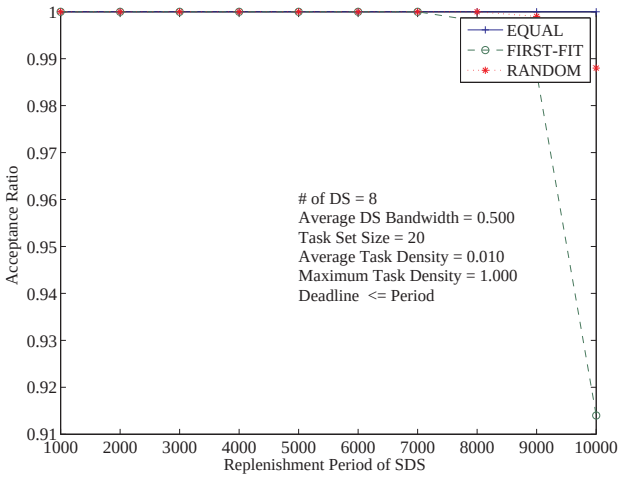

Figure 148. Acceptance ratio v.s. replenishment period

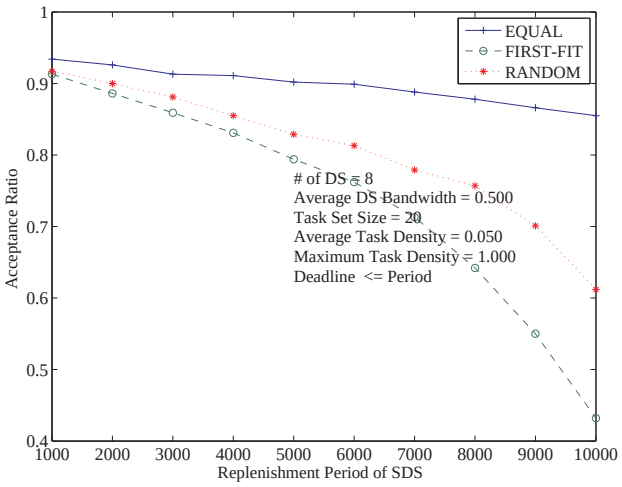

Figure 151. Acceptance ratio v.s. replenishment period

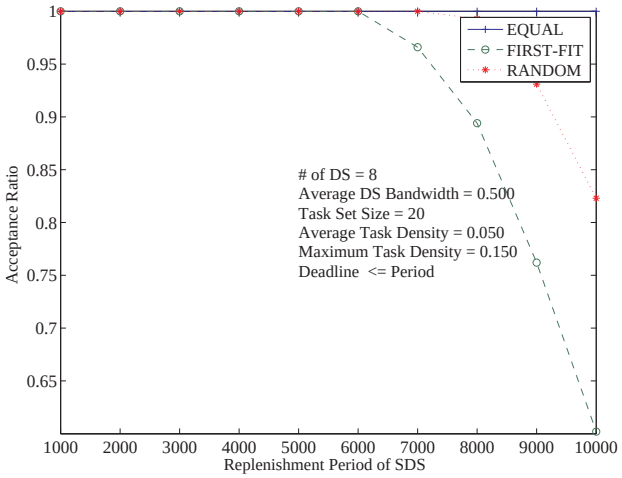

Figure 149. Acceptance ratio v.s. replenishment period

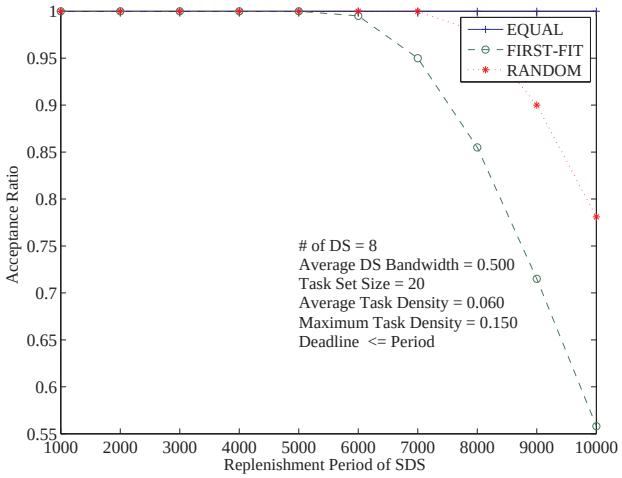

Figure 152. Acceptance ratio v.s. replenishment period

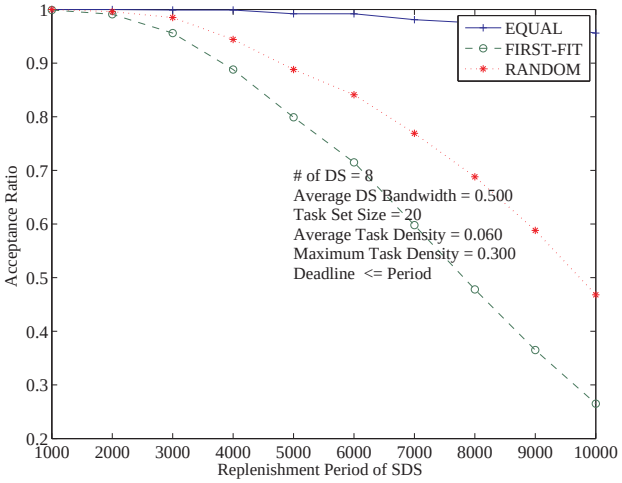

Figure 153. Acceptance ratio v.s. replenishment period

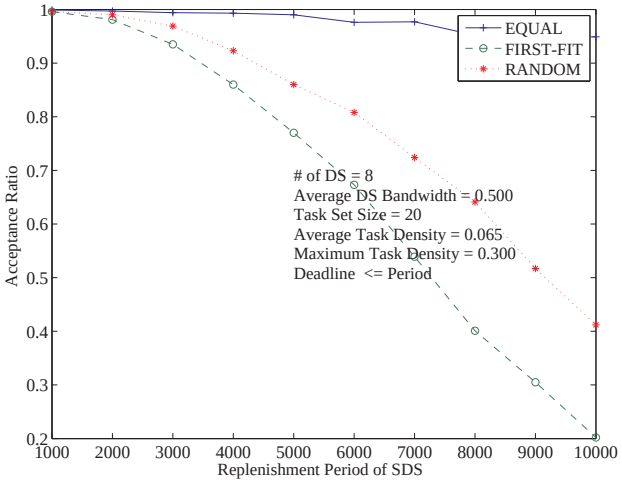

Figure 156. Acceptance ratio v.s. replenishment period

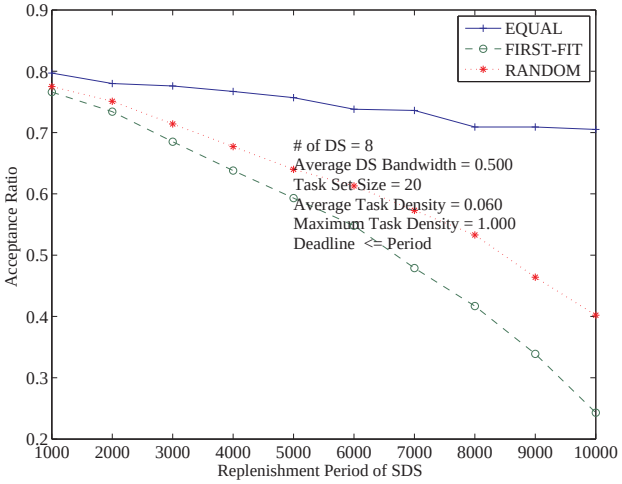

Figure 154. Acceptance ratio v.s. replenishment period

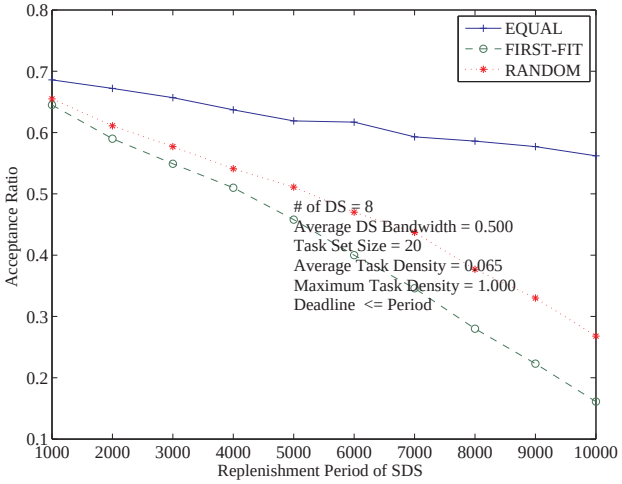

Figure 157. Acceptance ratio v.s. replenishment period

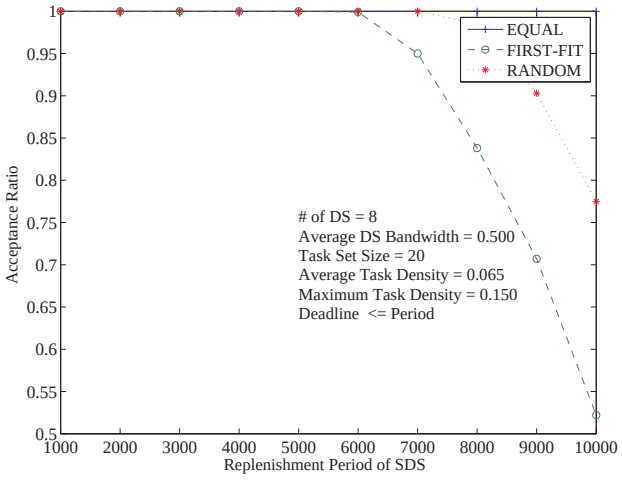

Figure 155. Acceptance ratio v.s. replenishment period

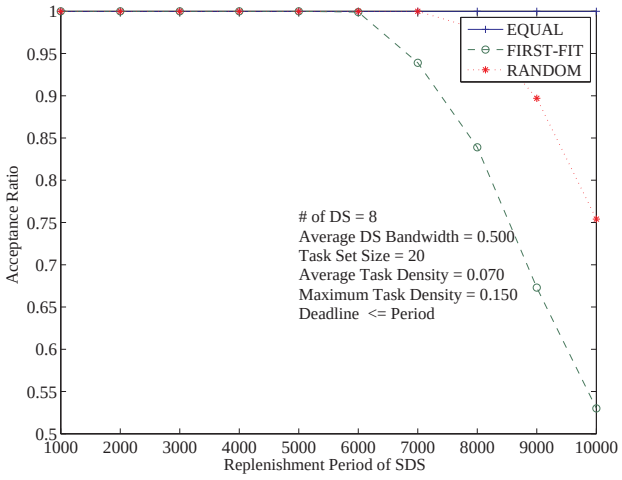

Figure 158. Acceptance ratio v.s. replenishment period

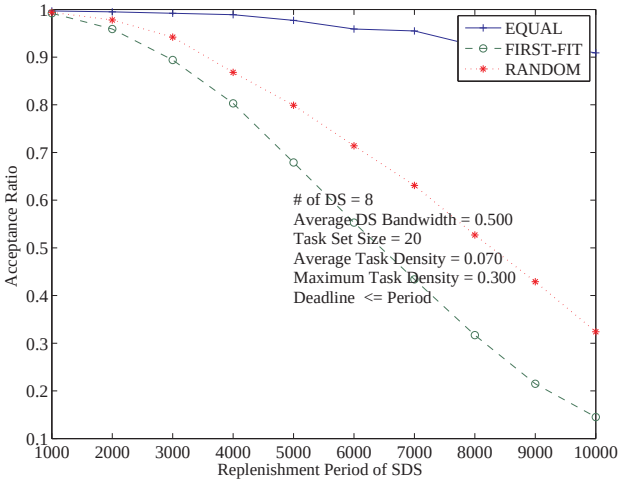

Figure 159. Acceptance ratio v.s. replenishment period

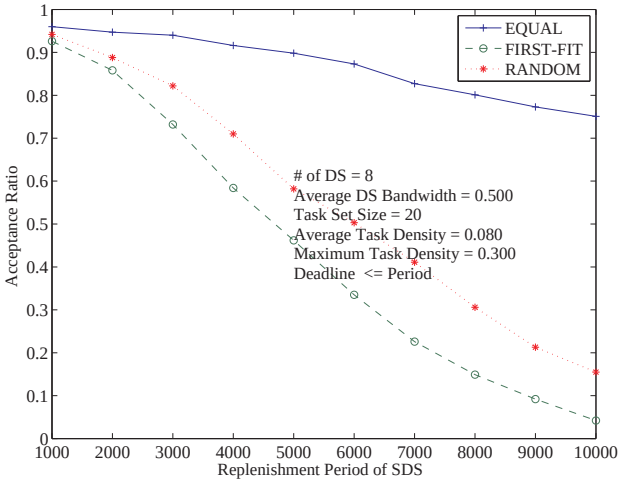

Figure 162. Acceptance ratio v.s. replenishment period

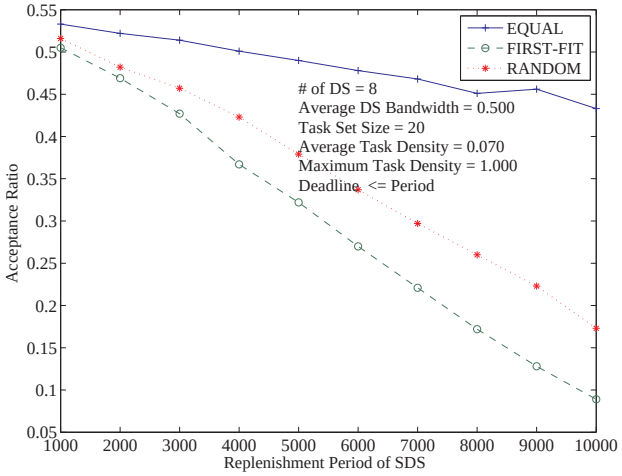

Figure 160. Acceptance ratio v.s. replenishment period

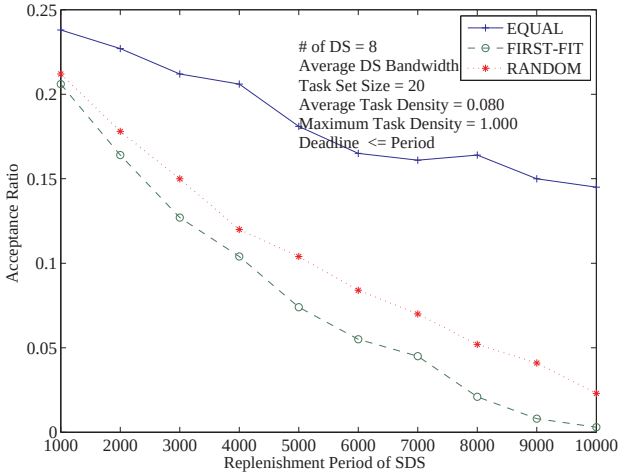

Figure 163. Acceptance ratio v.s. replenishment period

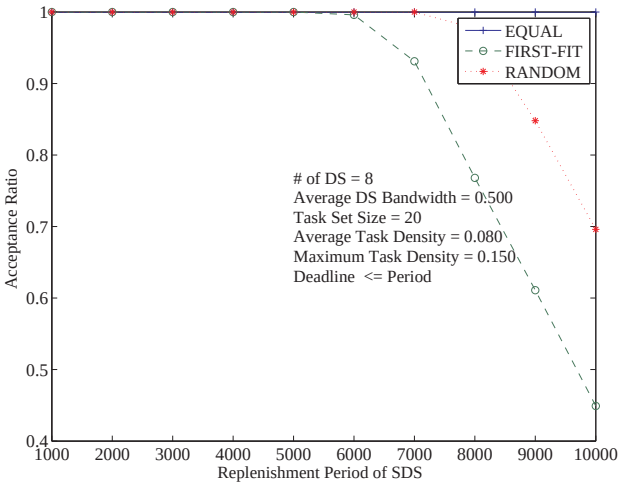

Figure 161. Acceptance ratio v.s. replenishment period

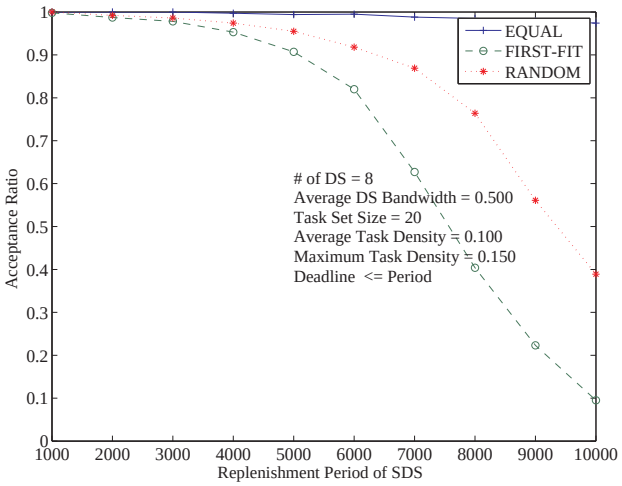

Figure 164. Acceptance ratio v.s. replenishment period

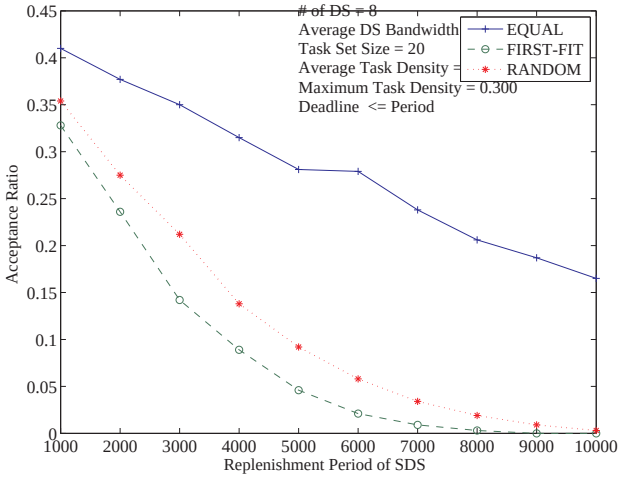

Figure 165. Acceptance ratio v.s. replenishment period

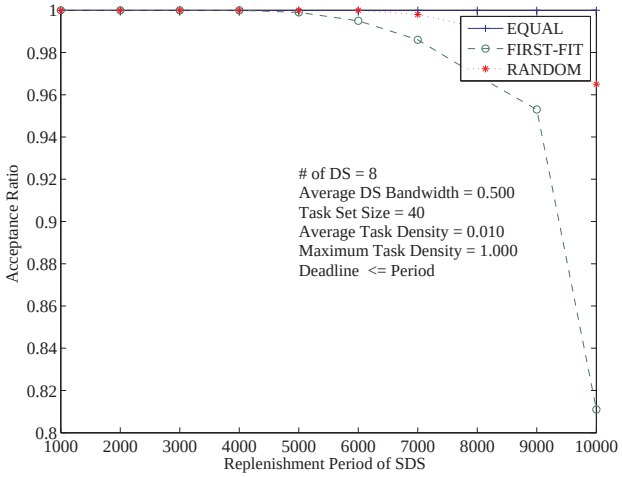

Figure 167. Acceptance ratio v.s. replenishment period

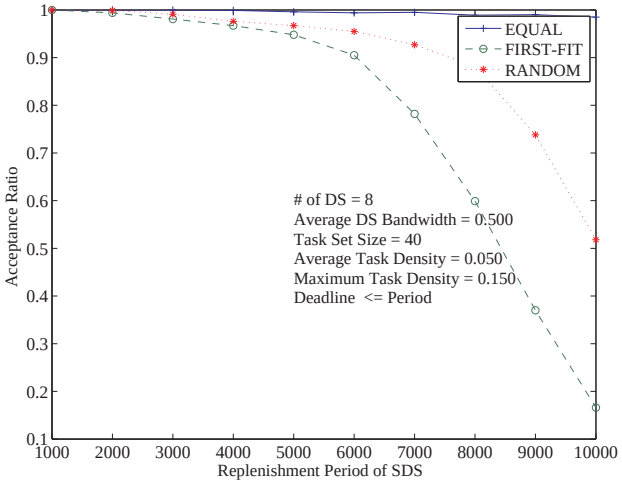

Figure 168. Acceptance ratio v.s. replenishment period

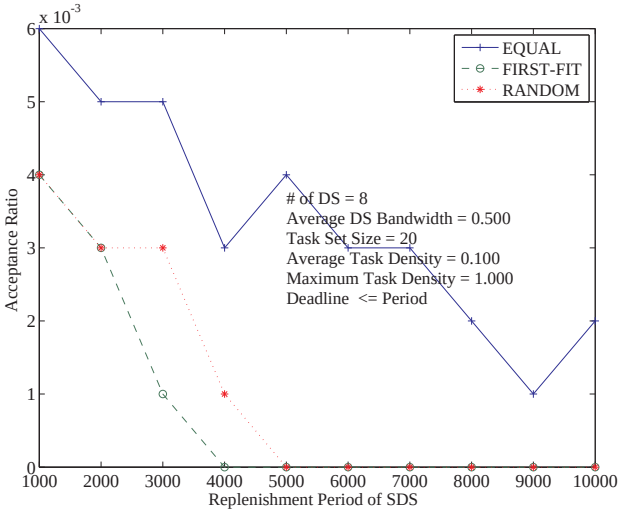

Figure 166. Acceptance ratio v.s. replenishment period

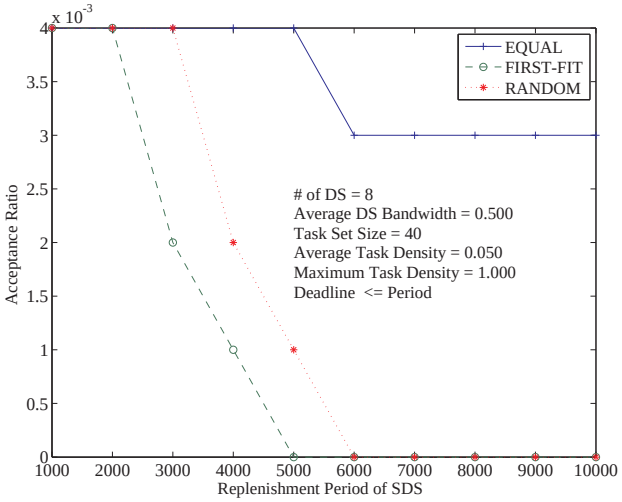

Figure 169. Acceptance ratio v.s. replenishment period

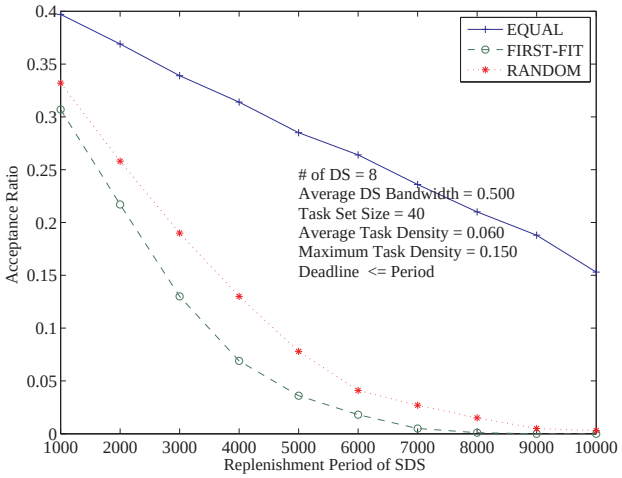

Figure 170. Acceptance ratio v.s. replenishment period

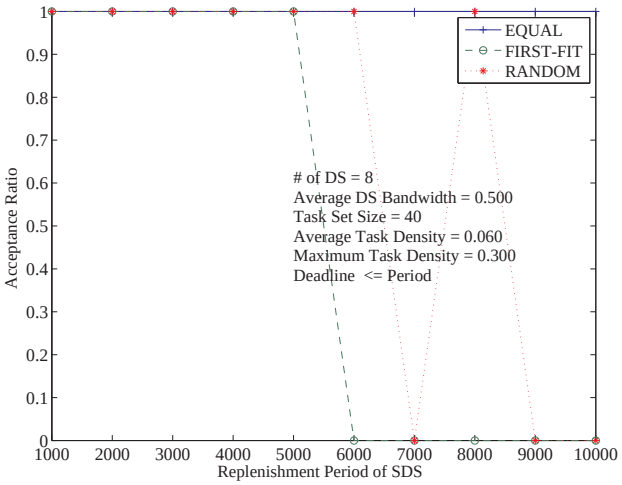

Figure 171. Acceptance ratio v.s. replenishment period

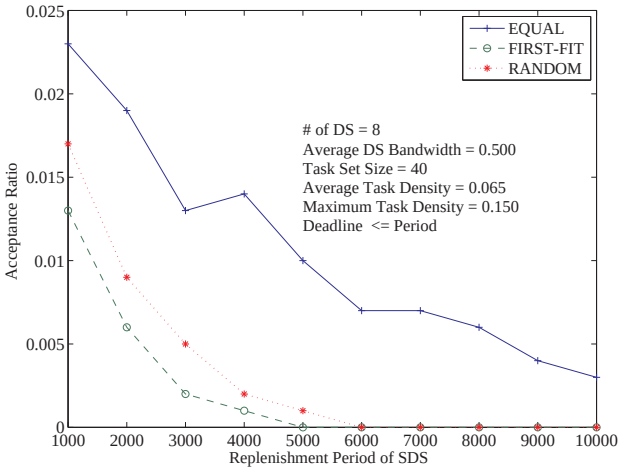

Figure 172. Acceptance ratio v.s. replenishment period

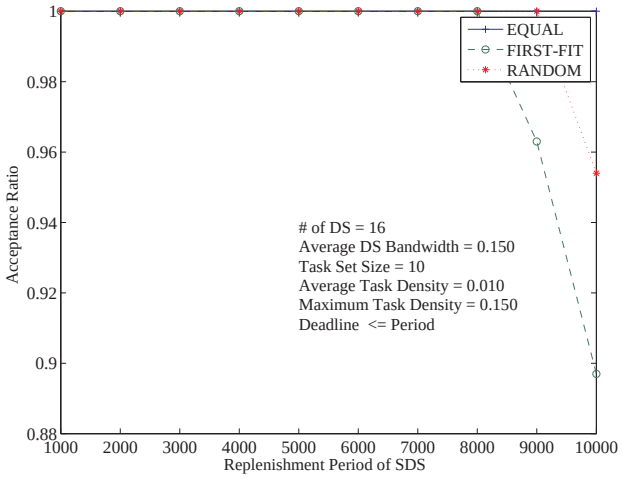

Figure 173. Acceptance ratio v.s. replenishment period

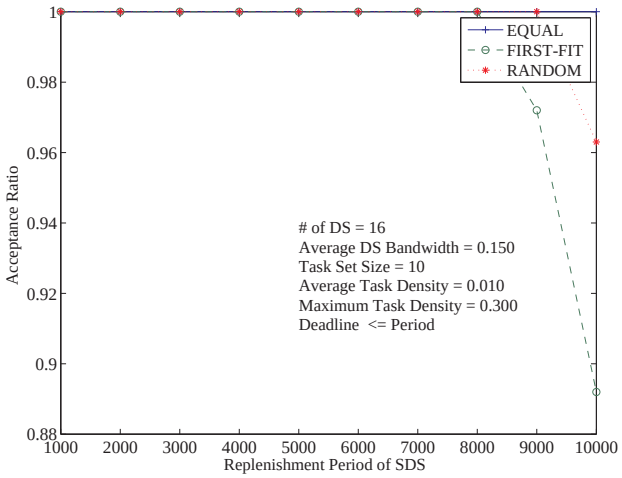

Figure 174. Acceptance ratio v.s. replenishment period

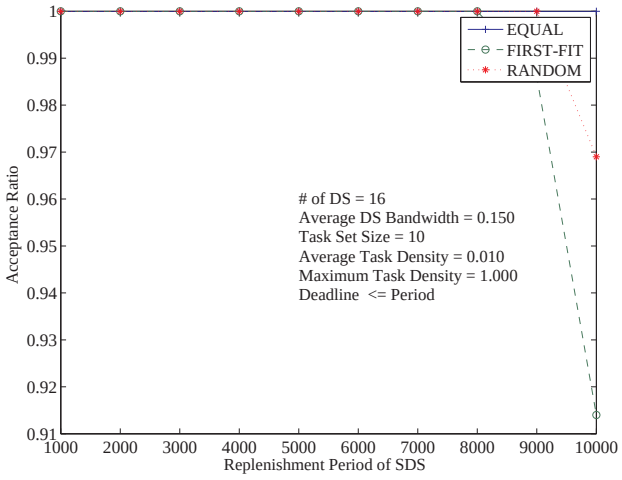

Figure 175. Acceptance ratio v.s. replenishment period

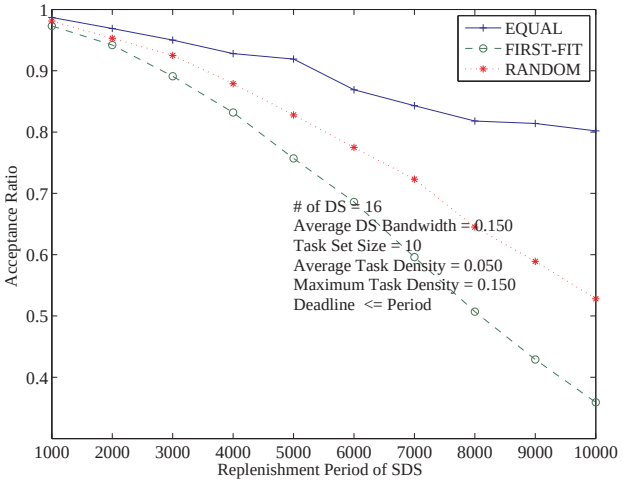

Figure 176. Acceptance ratio v.s. replenishment period

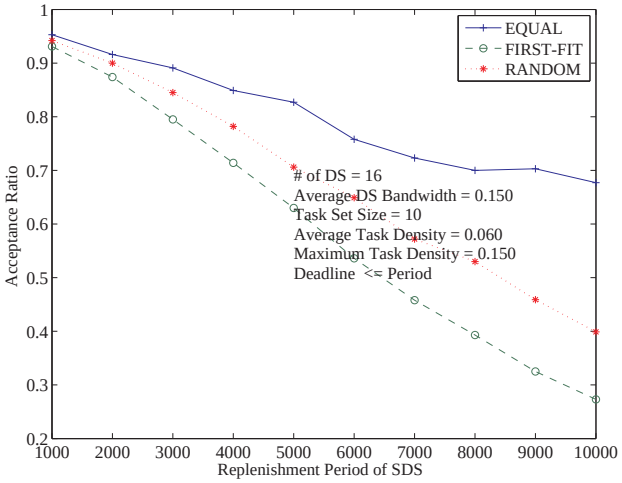

Figure 179. Acceptance ratio v.s. replenishment period

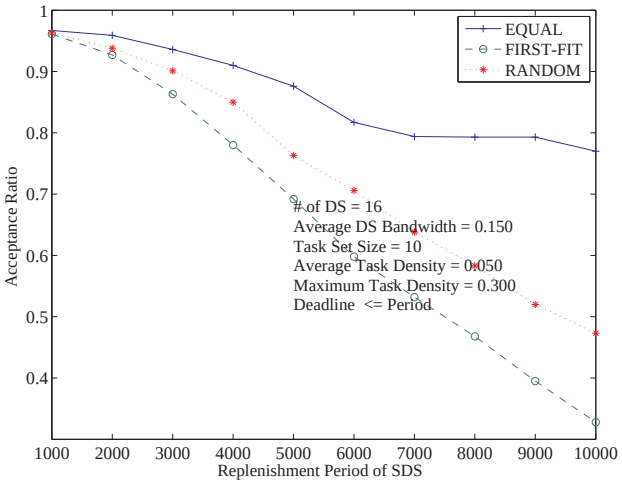

Figure 177. Acceptance ratio v.s. replenishment period

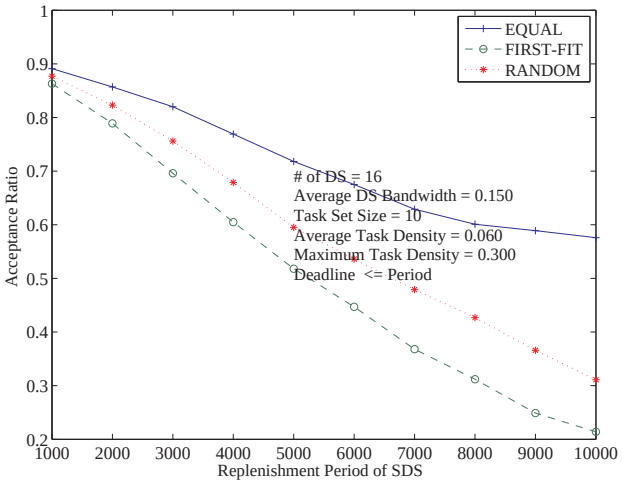

Figure 180. Acceptance ratio v.s. replenishment period

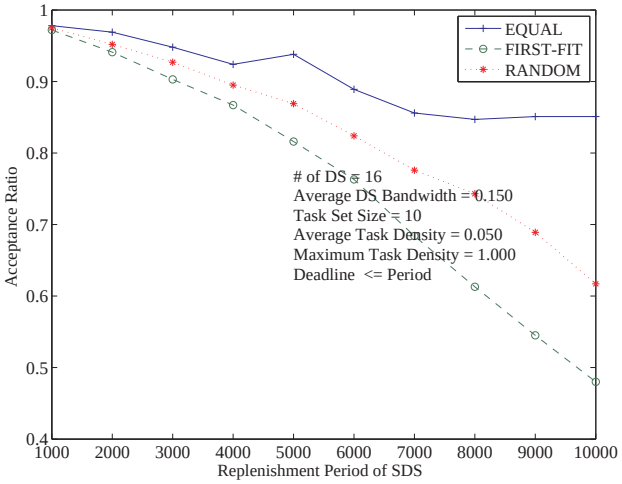

Figure 178. Acceptance ratio v.s. replenishment period

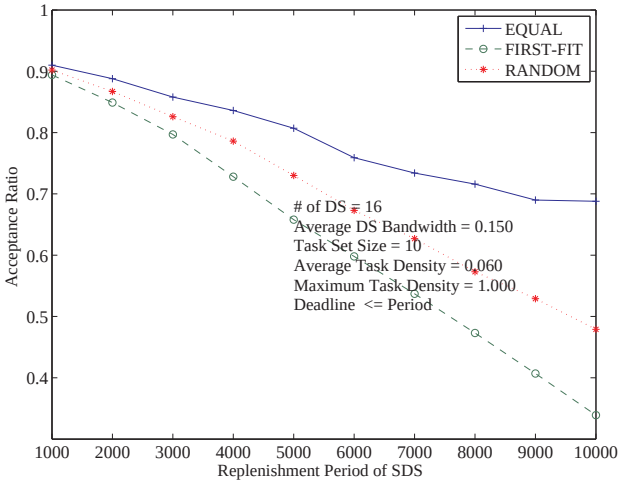

Figure 181. Acceptance ratio v.s. replenishment period

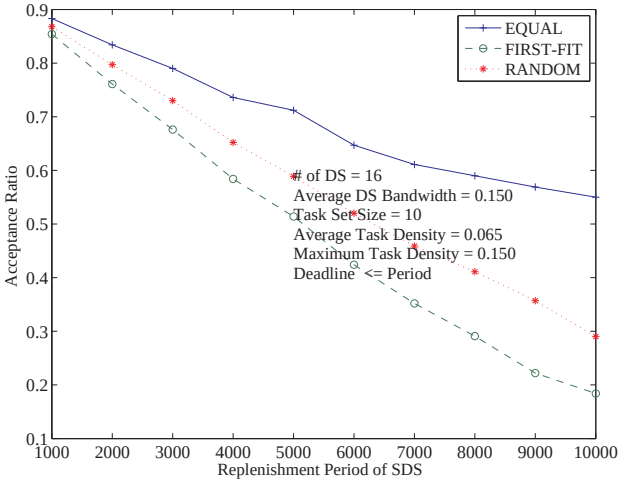

Figure 182. Acceptance ratio v.s. replenishment period

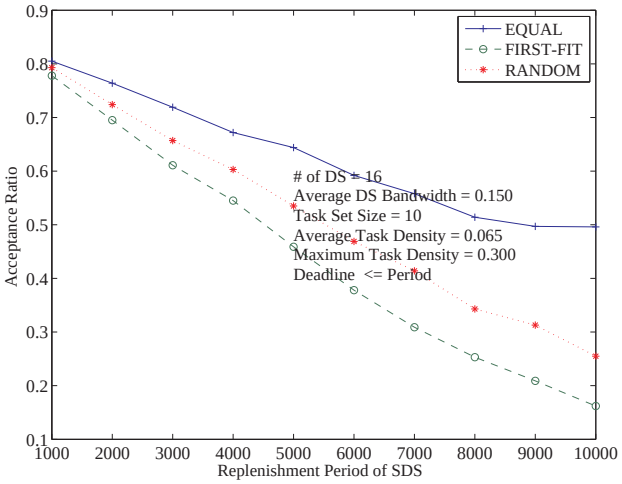

Figure 183. Acceptance ratio v.s. replenishment period

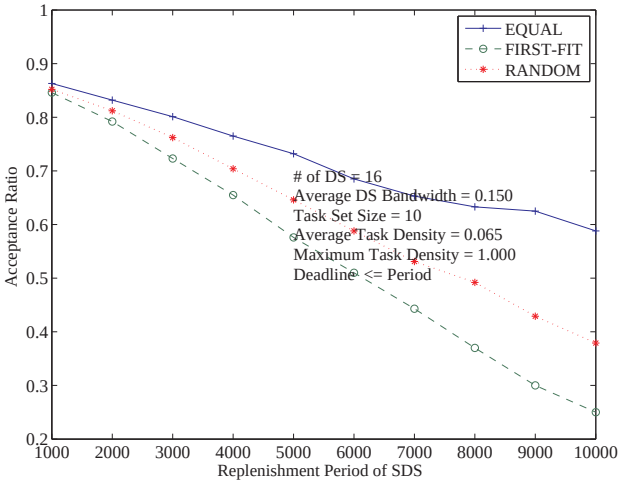

Figure 184. Acceptance ratio v.s. replenishment period

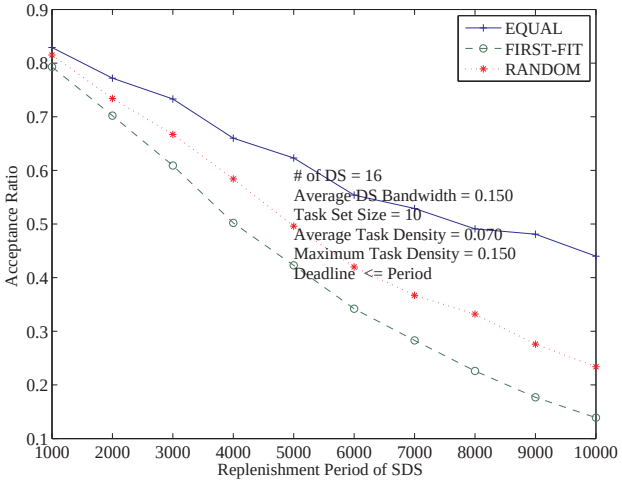

Figure 185. Acceptance ratio v.s. replenishment period

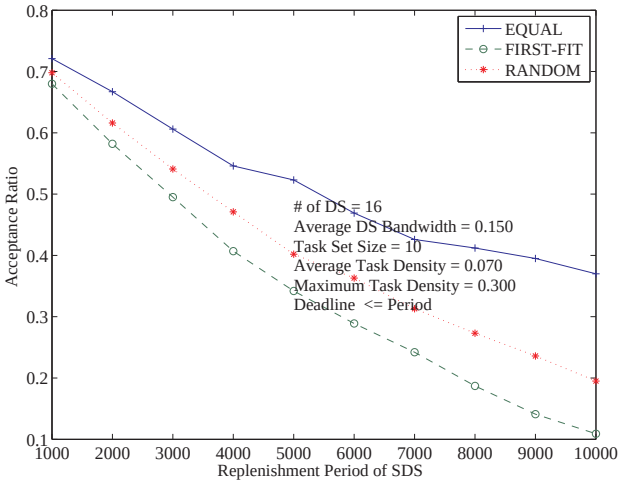

Figure 186. Acceptance ratio v.s. replenishment period

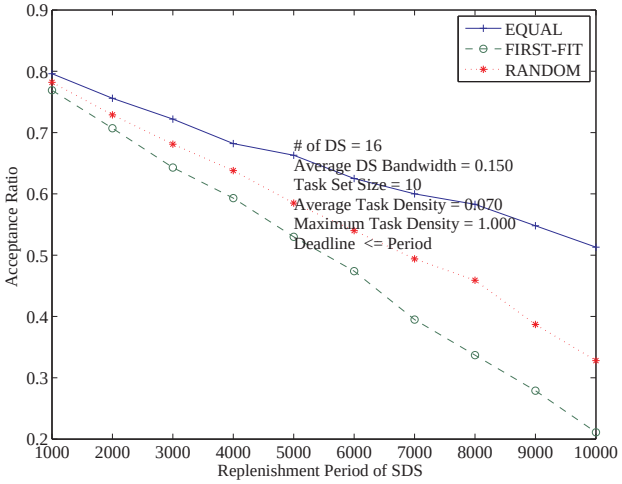

Figure 187. Acceptance ratio v.s. replenishment period

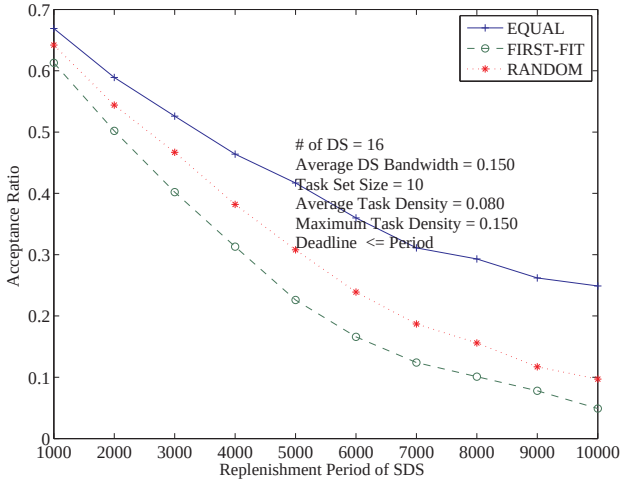

Figure 188. Acceptance ratio v.s. replenishment period

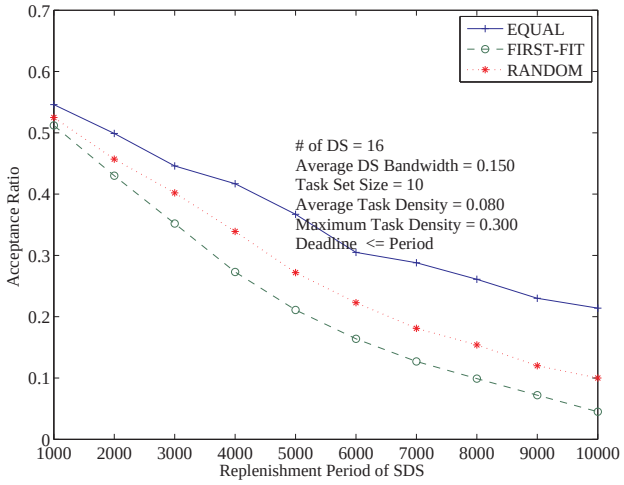

Figure 189. Acceptance ratio v.s. replenishment period

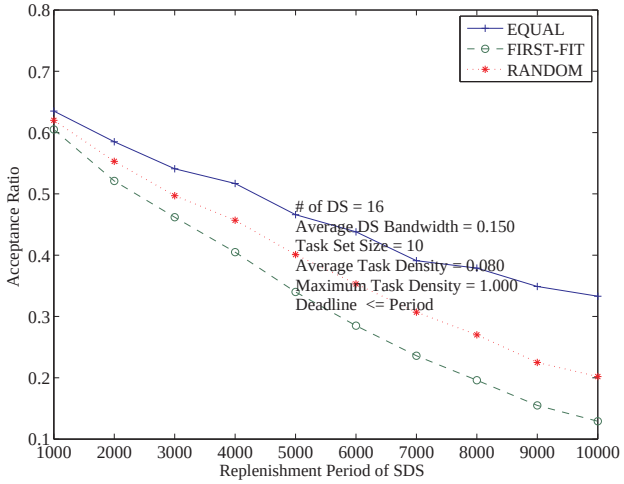

Figure 190. Acceptance ratio v.s. replenishment period

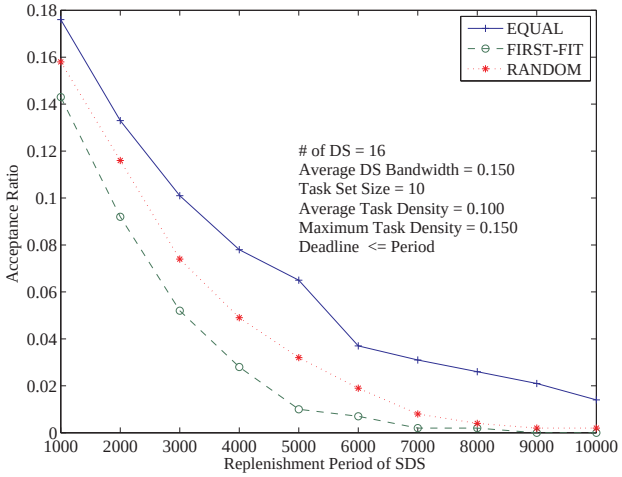

Figure 191. Acceptance ratio v.s. replenishment period

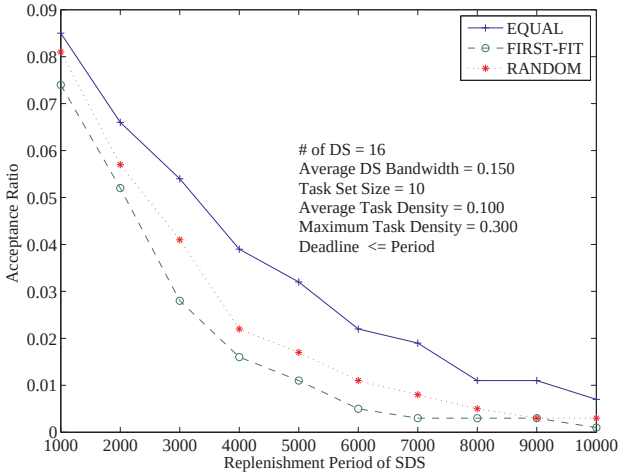

Figure 192. Acceptance ratio v.s. replenishment period

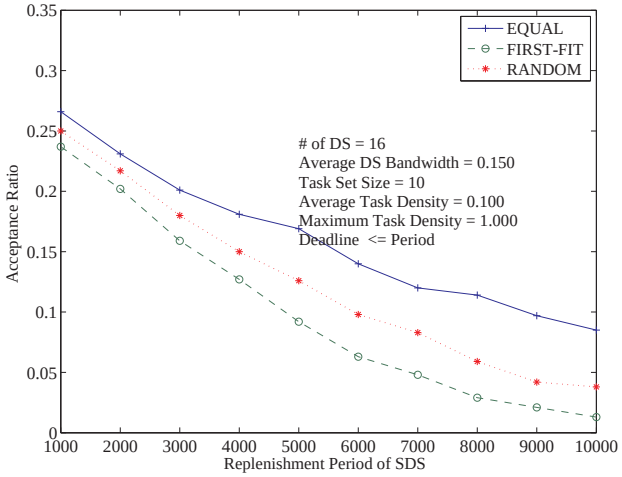

Figure 193. Acceptance ratio v.s. replenishment period

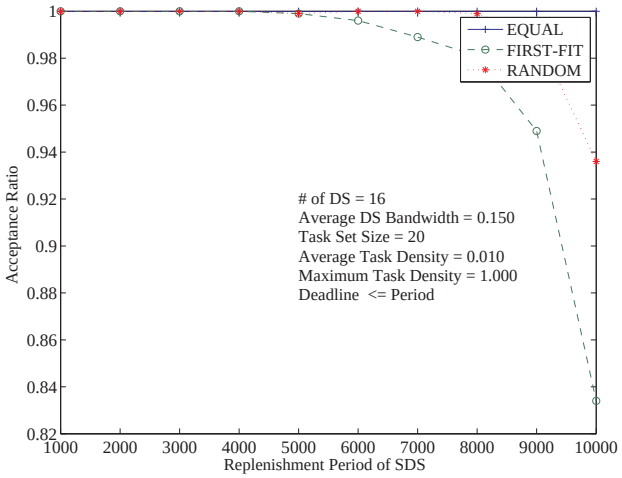

Figure 194. Acceptance ratio v.s. replenishment period

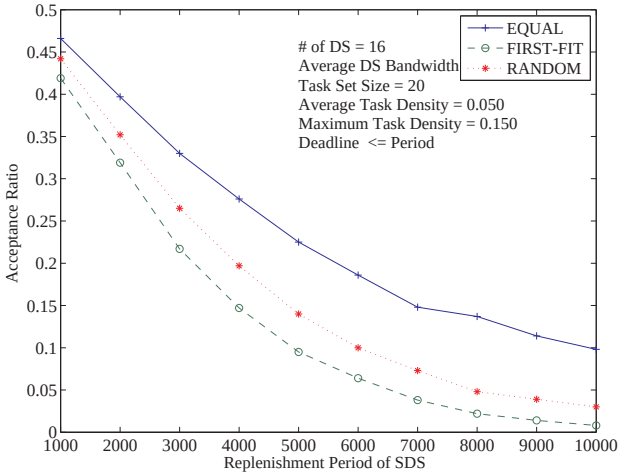

Figure 195. Acceptance ratio v.s. replenishment period

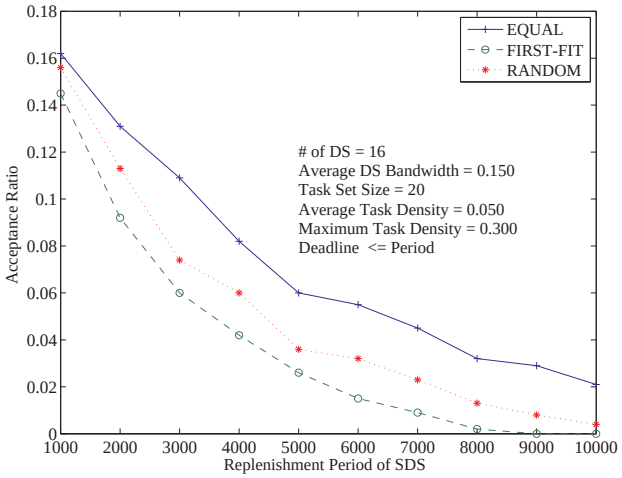

Figure 196. Acceptance ratio v.s. replenishment period

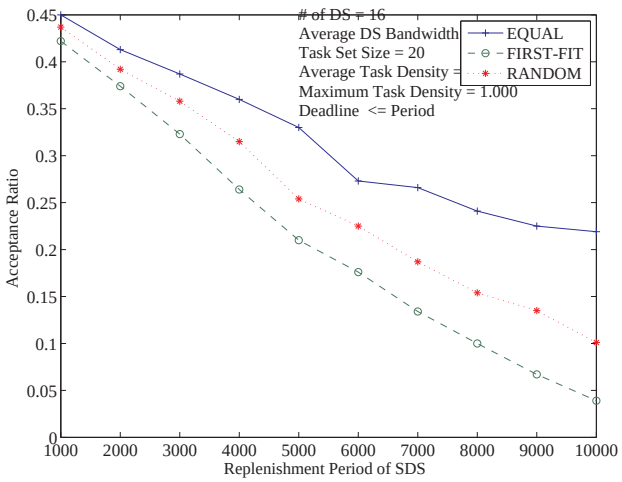

Figure 197. Acceptance ratio v.s. replenishment period

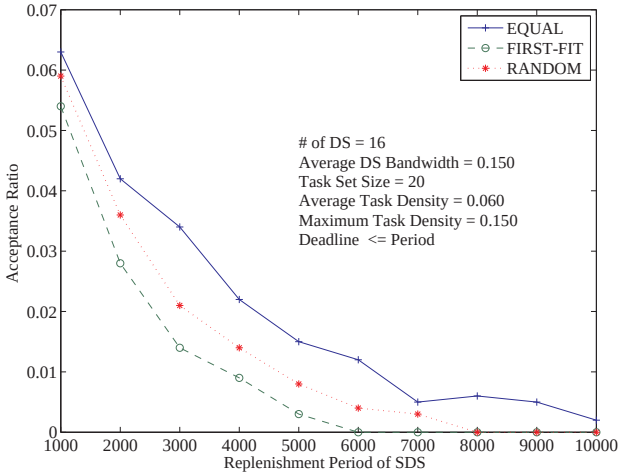

Figure 198. Acceptance ratio v.s. replenishment period

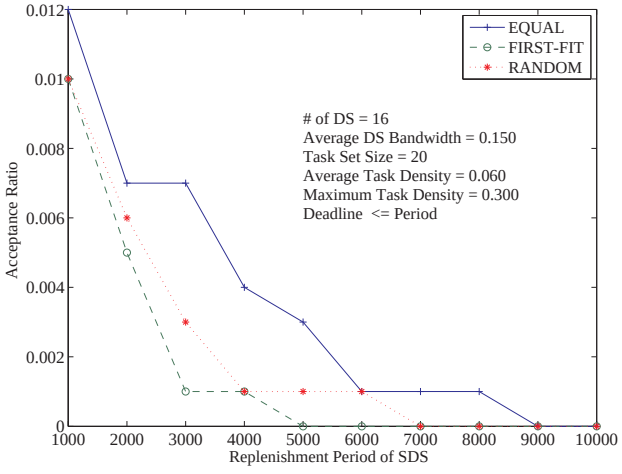

Figure 199. Acceptance ratio v.s. replenishment period

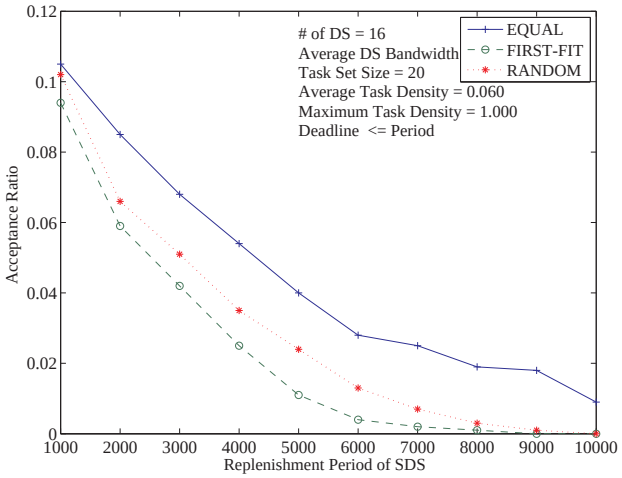

Figure 200. Acceptance ratio v.s. replenishment period

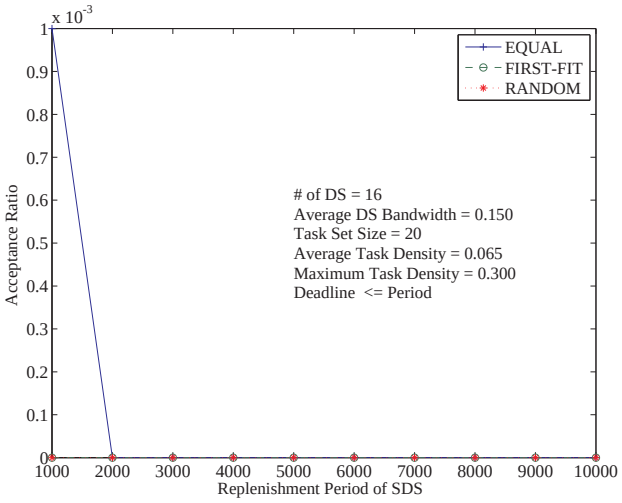

Figure 202. Acceptance ratio v.s. replenishment period

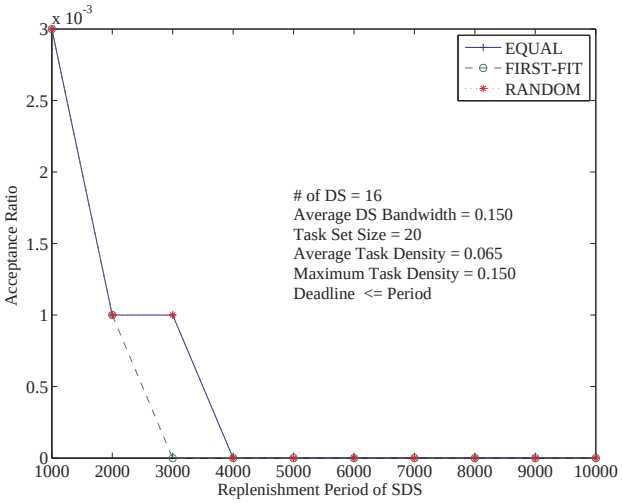

Figure 201. Acceptance ratio v.s. replenishment period

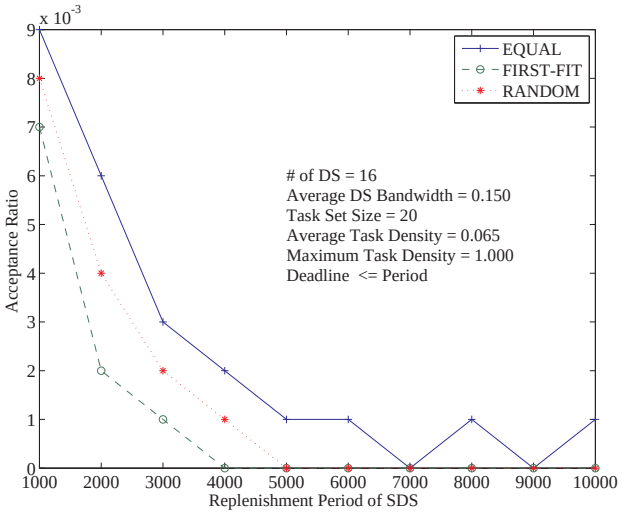

Figure 203. Acceptance ratio v.s. replenishment period

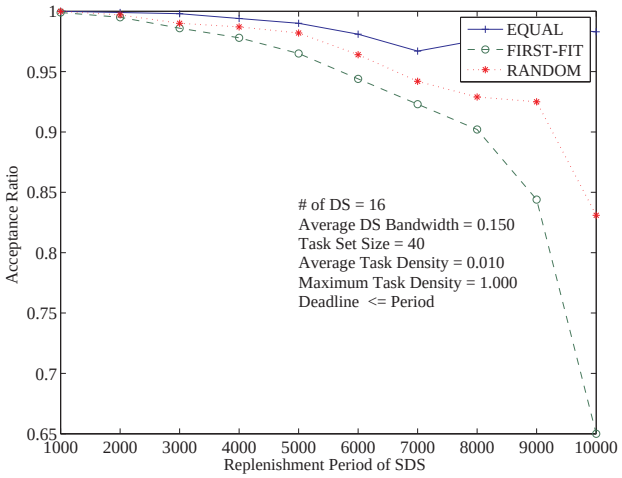

Figure 204. Acceptance ratio v.s. replenishment period

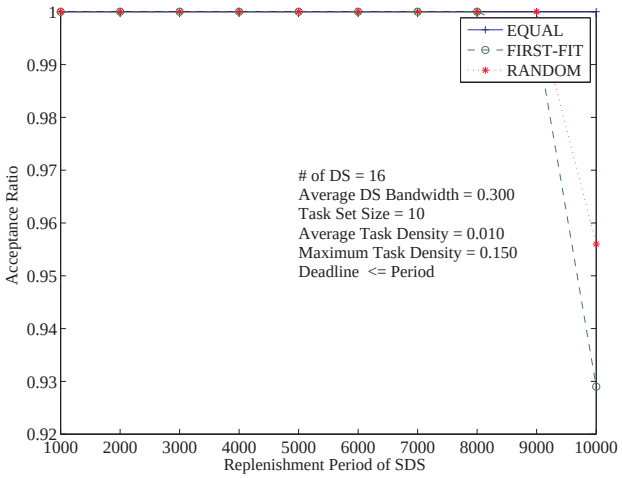

Figure 205. Acceptance ratio v.s. replenishment period

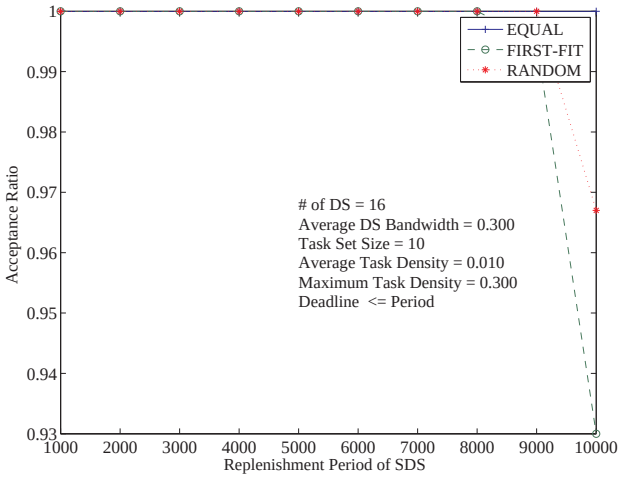

Figure 206. Acceptance ratio v.s. replenishment period

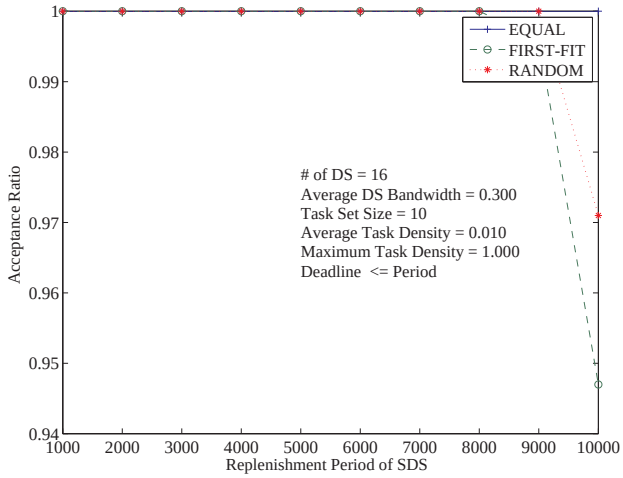

Figure 207. Acceptance ratio v.s. replenishment period

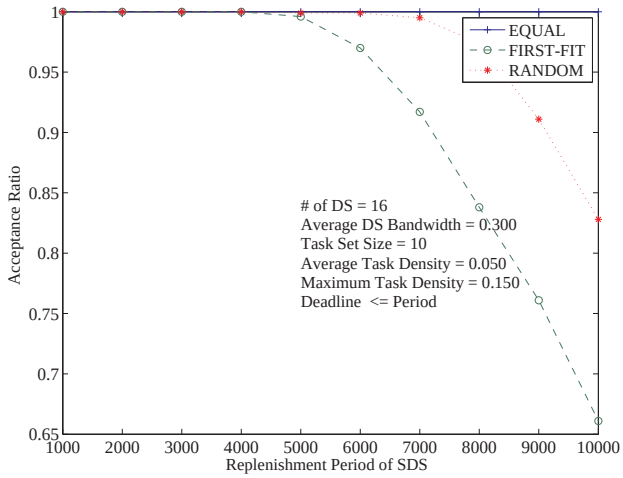

Figure 208. Acceptance ratio v.s. replenishment period

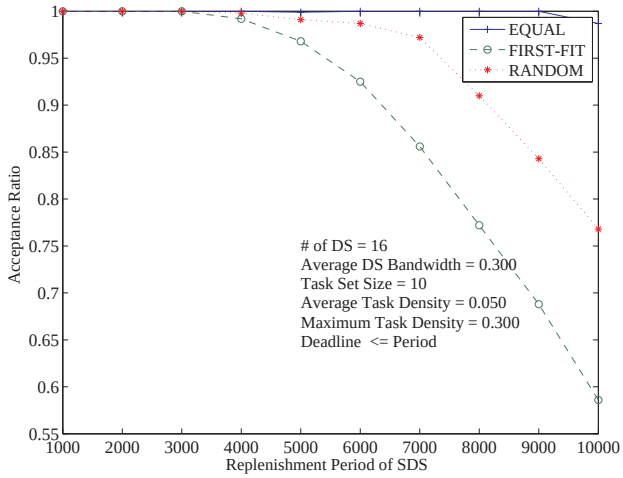

Figure 209. Acceptance ratio v.s. replenishment period

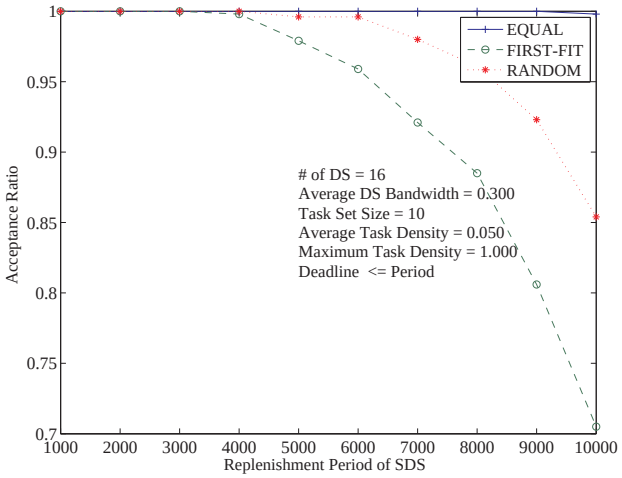

Figure 210. Acceptance ratio v.s. replenishment period

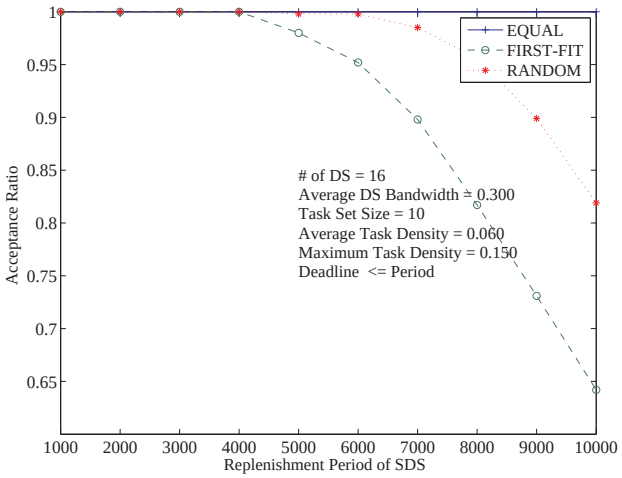

Figure 211. Acceptance ratio v.s. replenishment period

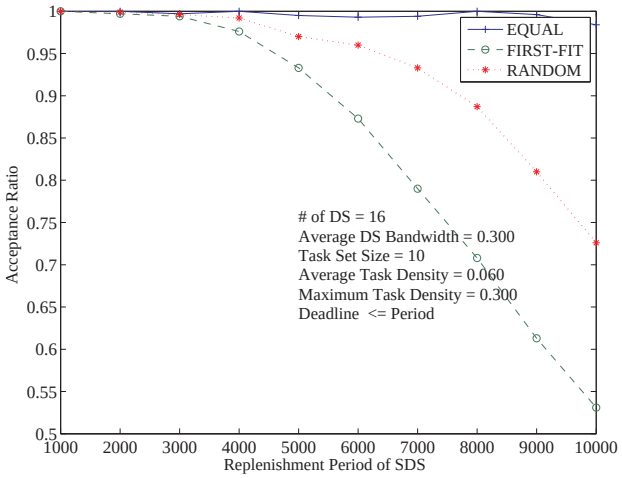

Figure 212. Acceptance ratio v.s. replenishment period

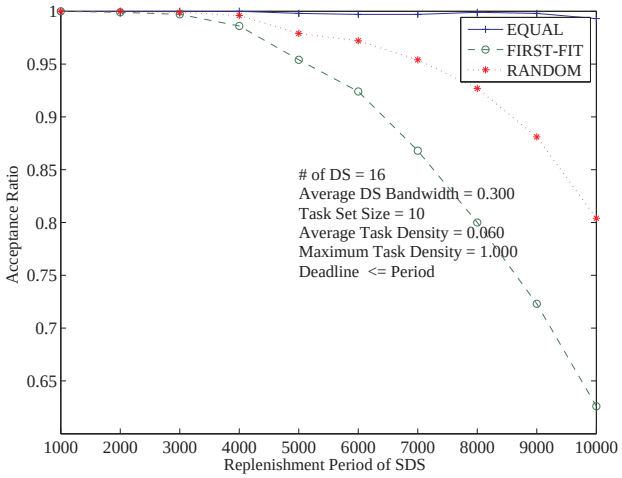

Figure 213. Acceptance ratio v.s. replenishment period

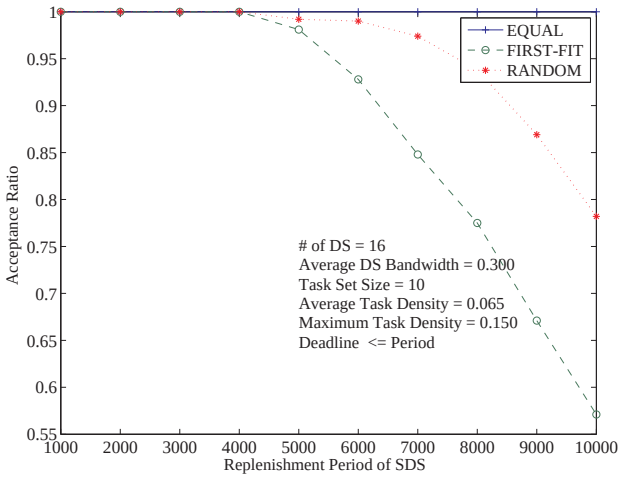

Figure 214. Acceptance ratio v.s. replenishment period

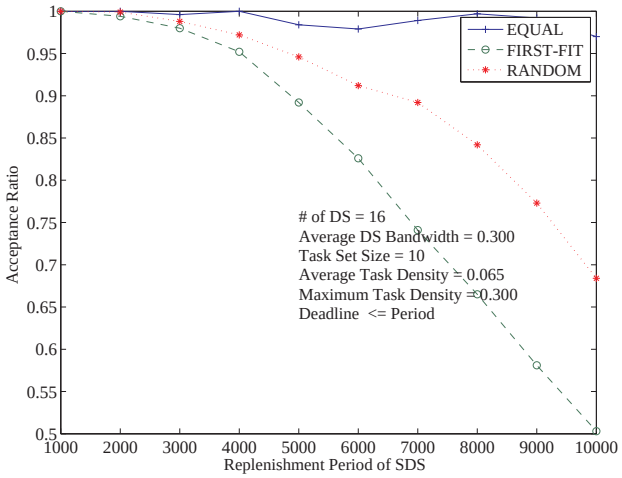

Figure 215. Acceptance ratio v.s. replenishment period

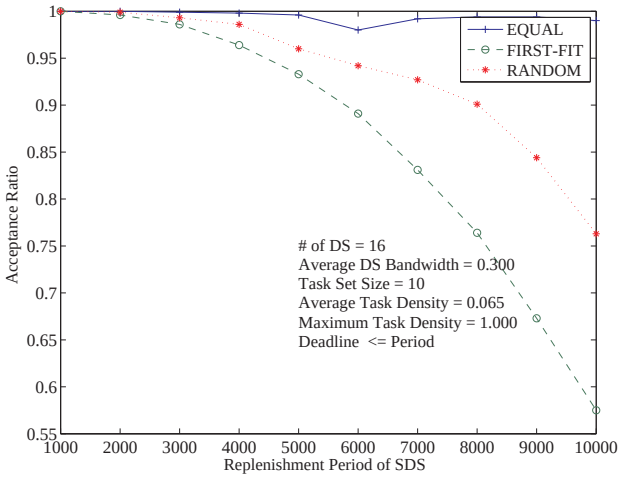

Figure 216. Acceptance ratio v.s. replenishment period

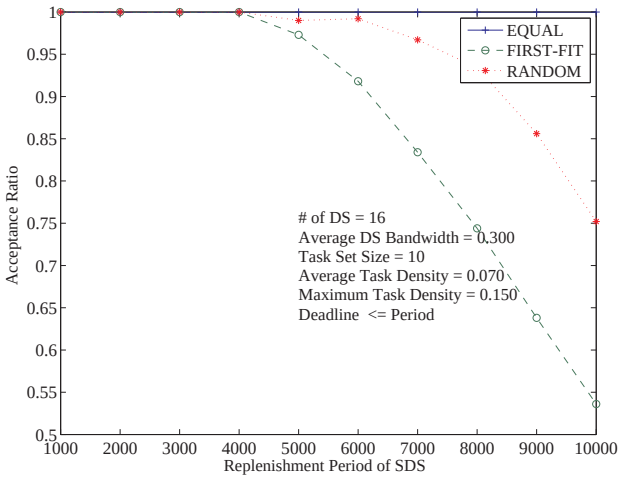

Figure 217. Acceptance ratio v.s. replenishment period

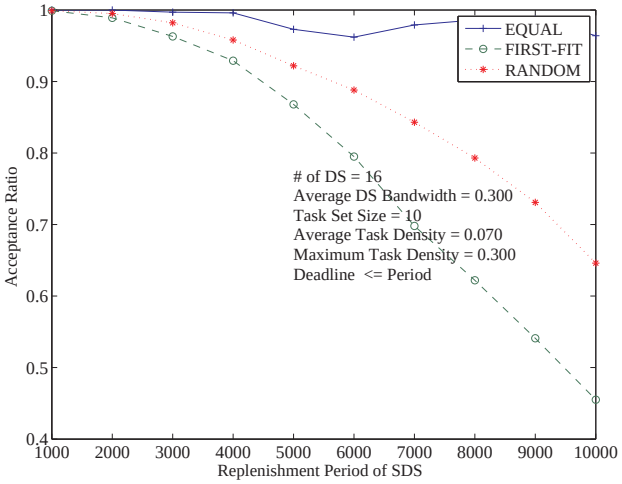

Figure 218. Acceptance ratio v.s. replenishment period

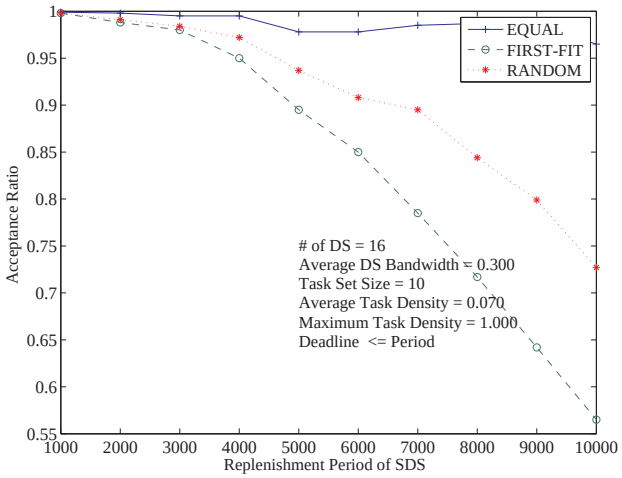

Figure 219. Acceptance ratio v.s. replenishment period

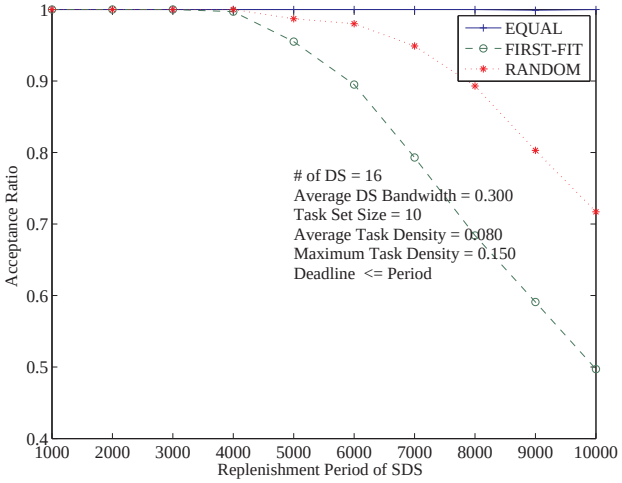

Figure 220. Acceptance ratio v.s. replenishment period

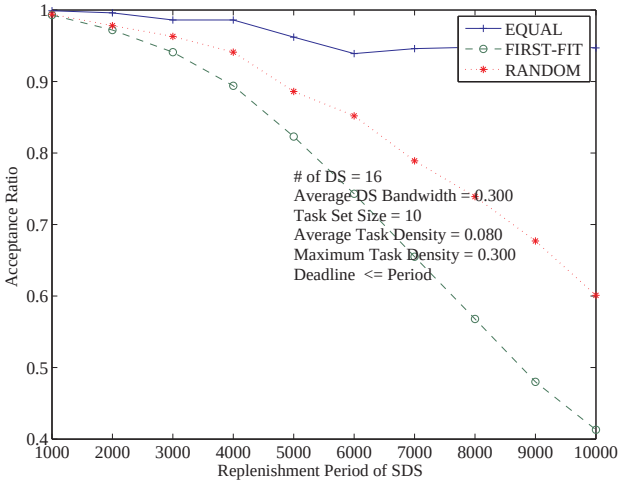

Figure 221. Acceptance ratio v.s. replenishment period

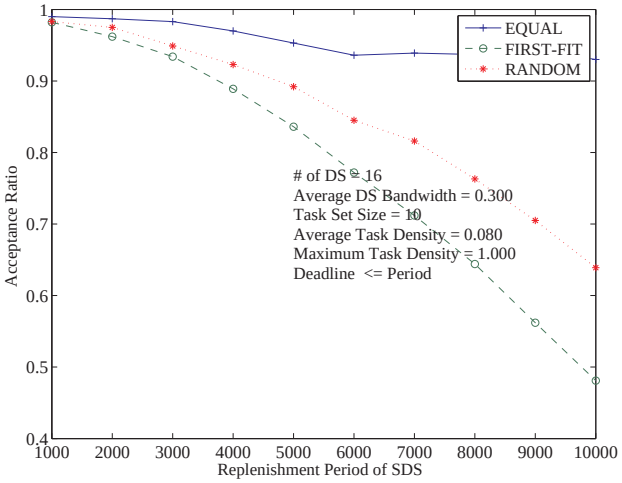

Figure 222. Acceptance ratio v.s. replenishment period

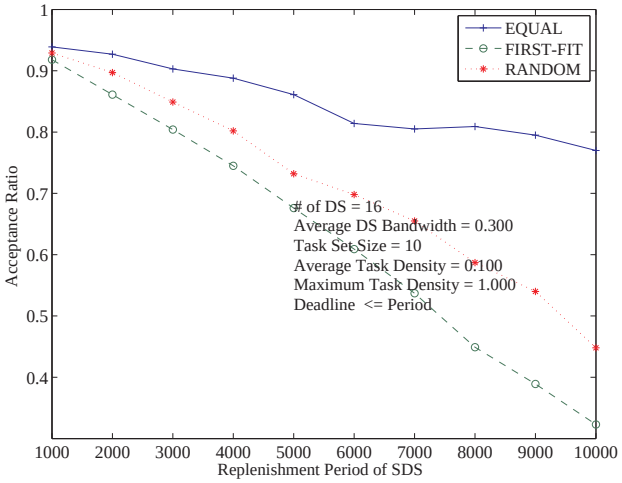

Figure 225. Acceptance ratio v.s. replenishment period

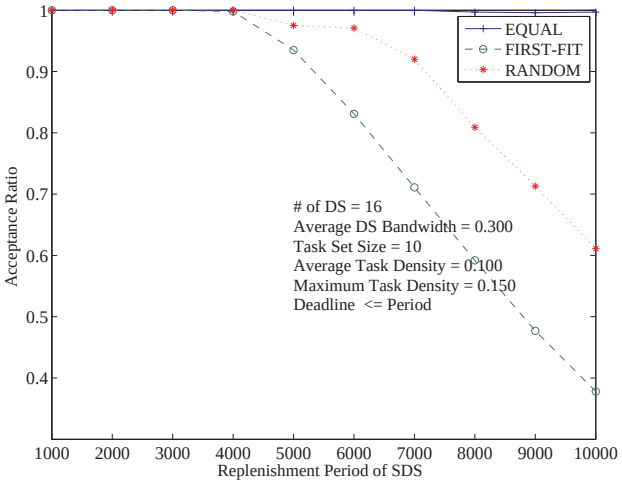

Figure 223. Acceptance ratio v.s. replenishment period

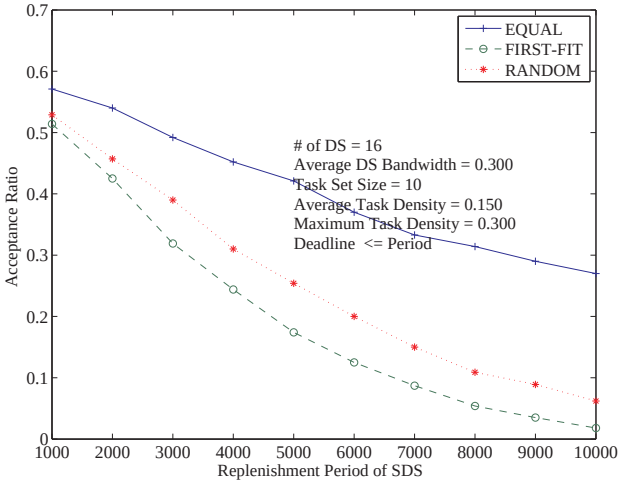

Figure 226. Acceptance ratio v.s. replenishment period

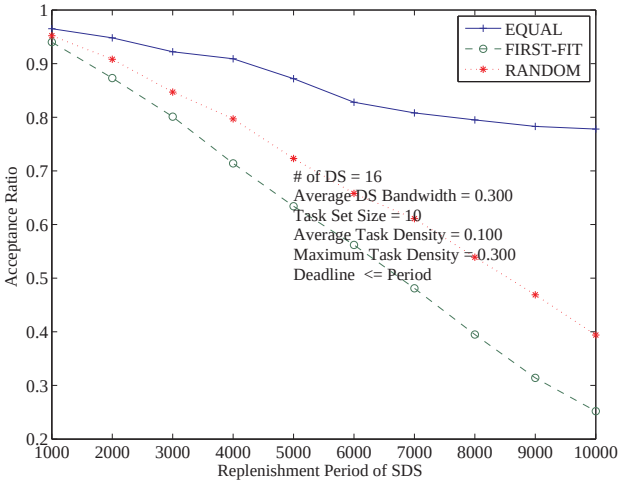

Figure 224. Acceptance ratio v.s. replenishment period

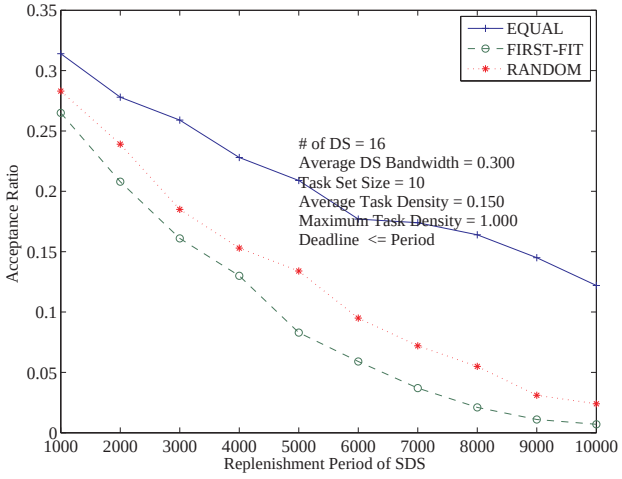

Figure 227. Acceptance ratio v.s. replenishment period

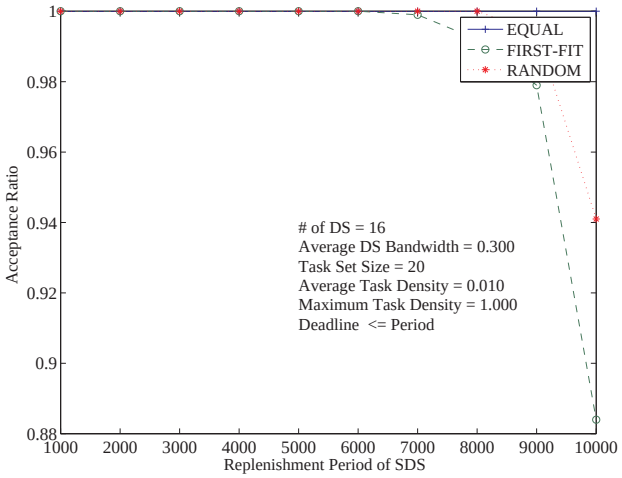

Figure 228. Acceptance ratio v.s. replenishment period

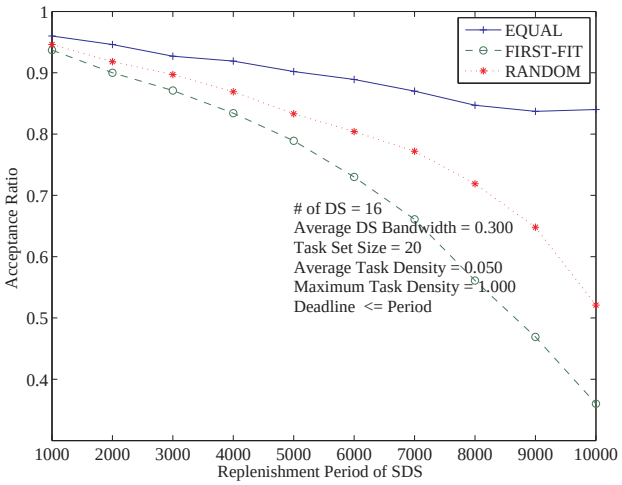

Figure 231. Acceptance ratio v.s. replenishment period

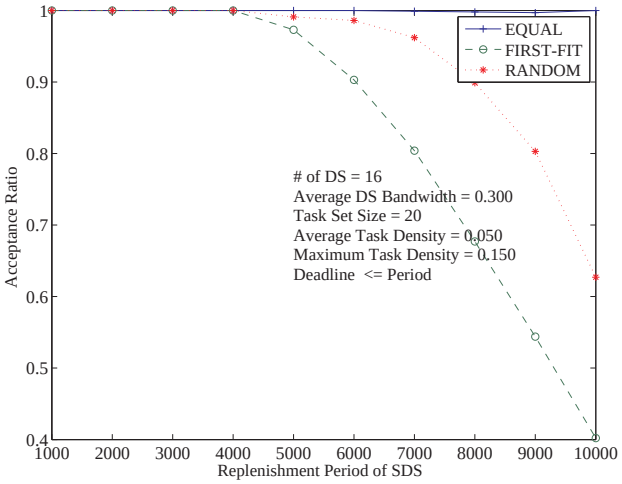

Figure 229. Acceptance ratio v.s. replenishment period

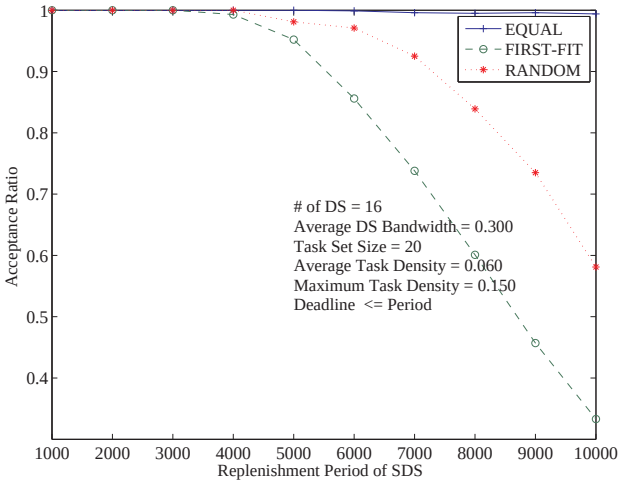

Figure 232. Acceptance ratio v.s. replenishment period

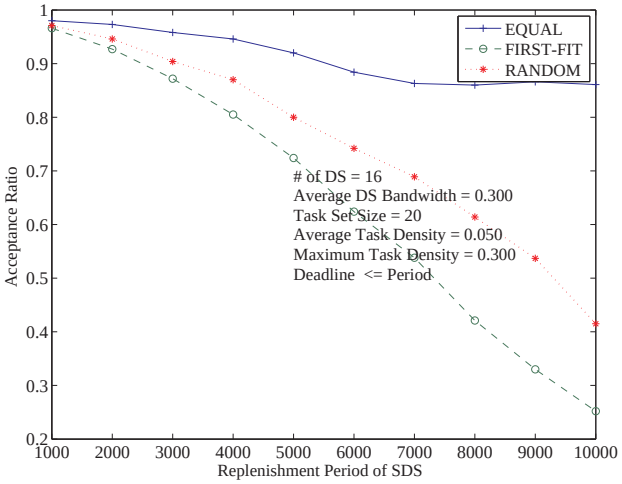

Figure 230. Acceptance ratio v.s. replenishment period

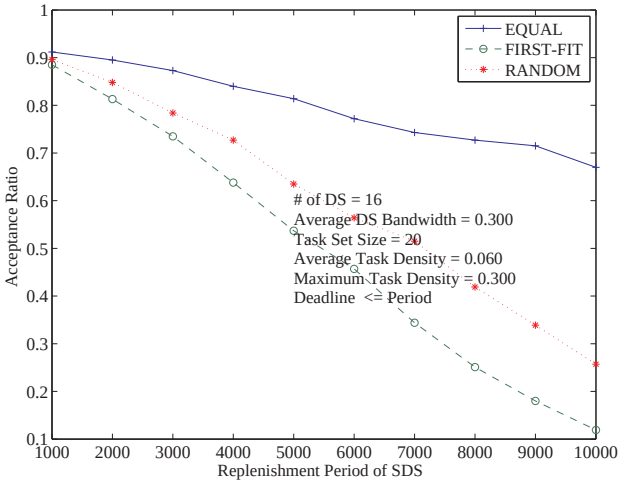

Figure 233. Acceptance ratio v.s. replenishment period

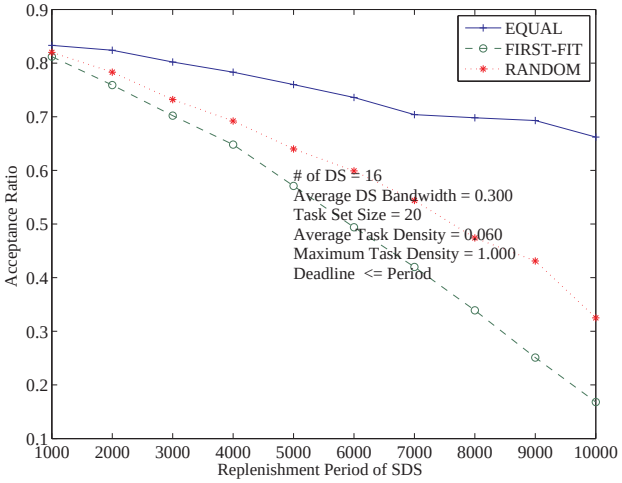

Figure 234. Acceptance ratio v.s. replenishment period

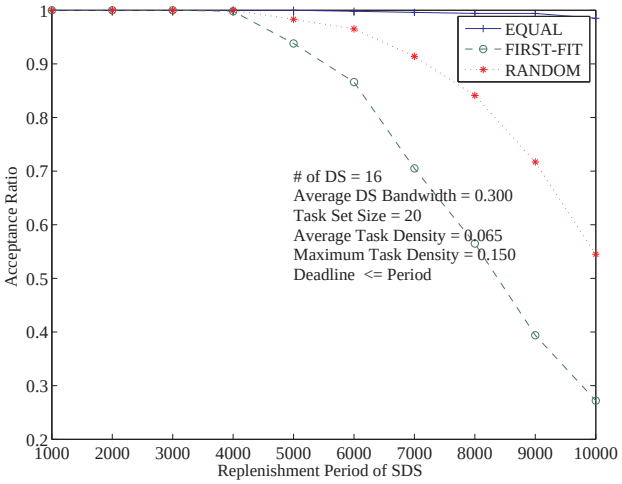

Figure 235. Acceptance ratio v.s. replenishment period

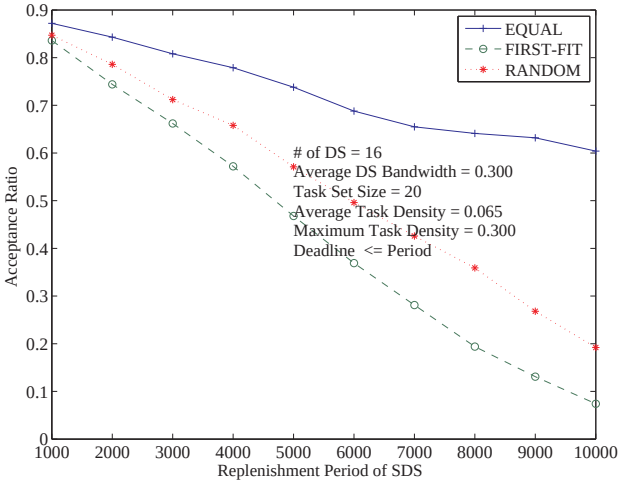

Figure 236. Acceptance ratio v.s. replenishment period

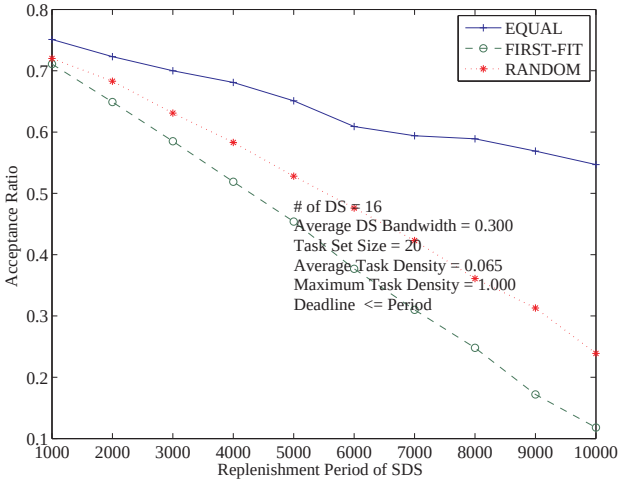

Figure 237. Acceptance ratio v.s. replenishment period

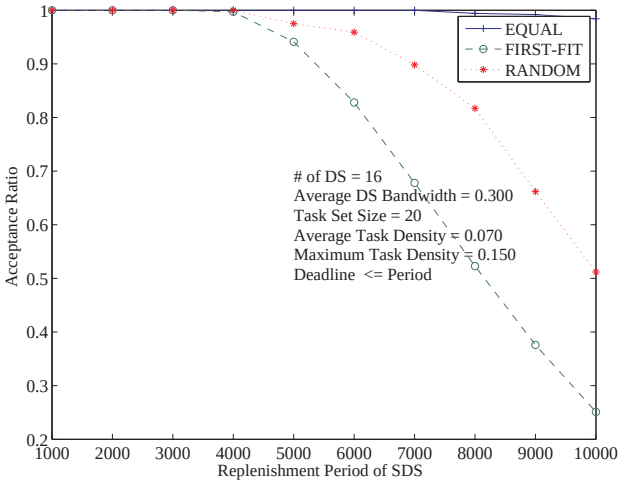

Figure 238. Acceptance ratio v.s. replenishment period

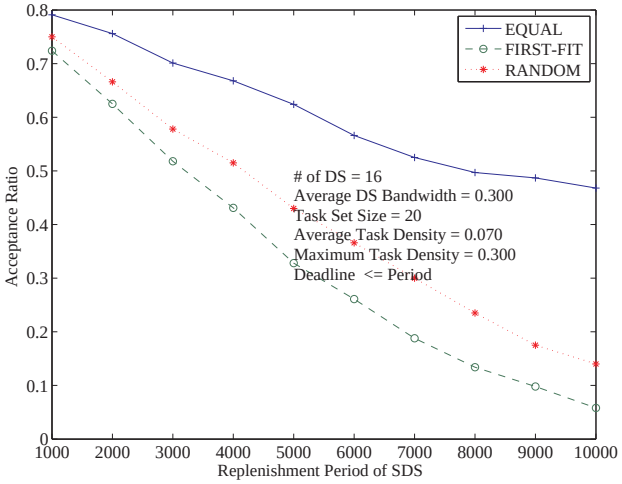

Figure 239. Acceptance ratio v.s. replenishment period

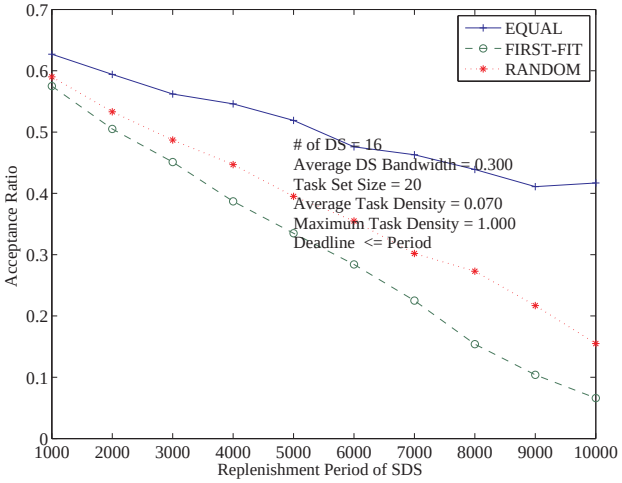

Figure 240. Acceptance ratio v.s. replenishment period

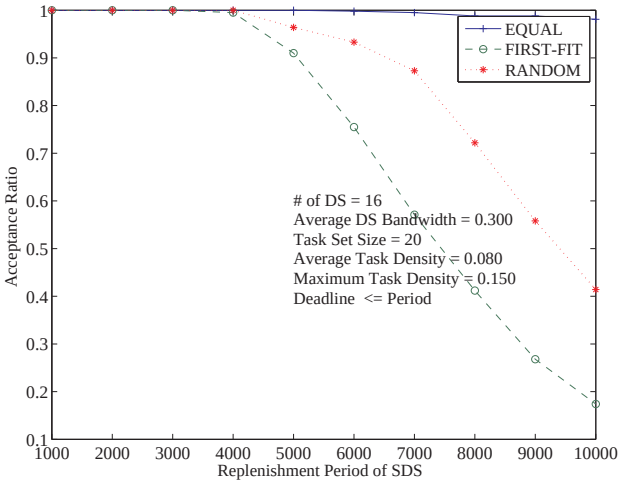

Figure 241. Acceptance ratio v.s. replenishment period

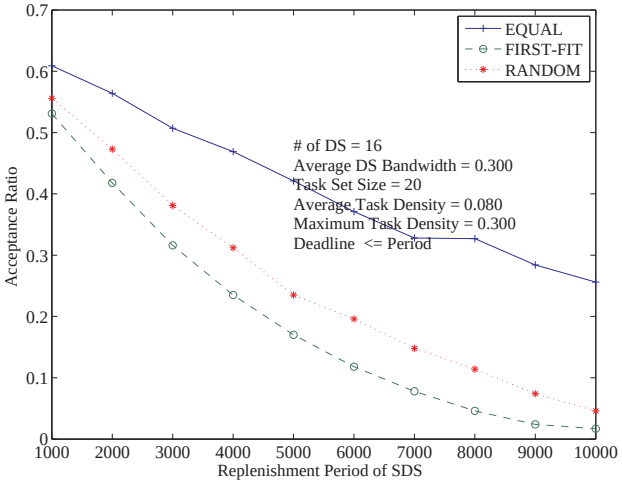

Figure 242. Acceptance ratio v.s. replenishment period

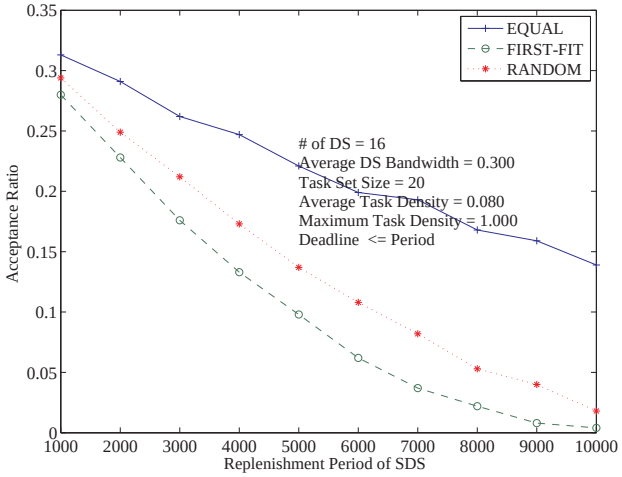

Figure 243. Acceptance ratio v.s. replenishment period

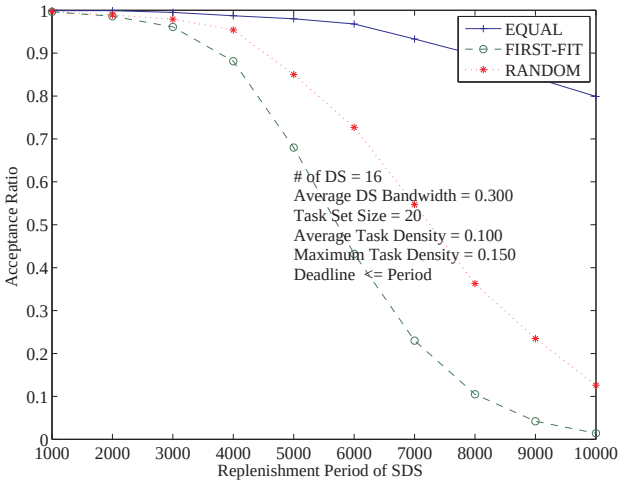

Figure 244. Acceptance ratio v.s. replenishment period

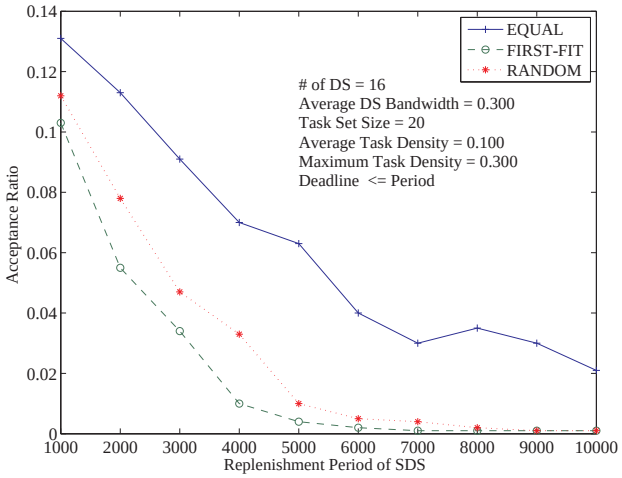

Figure 245. Acceptance ratio v.s. replenishment period

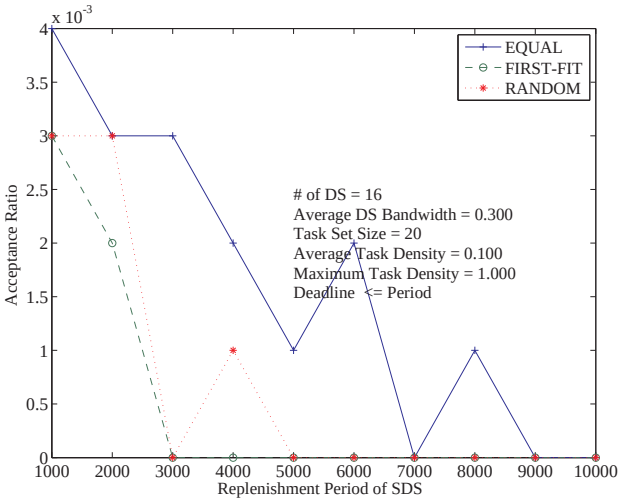

Figure 246. Acceptance ratio v.s. replenishment period

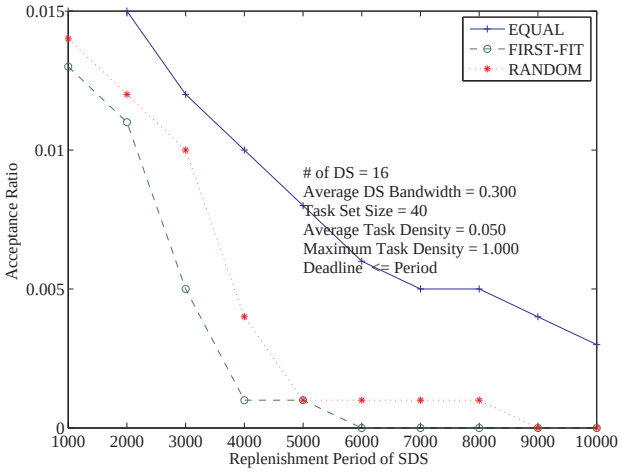

Figure 249. Acceptance ratio v.s. replenishment period

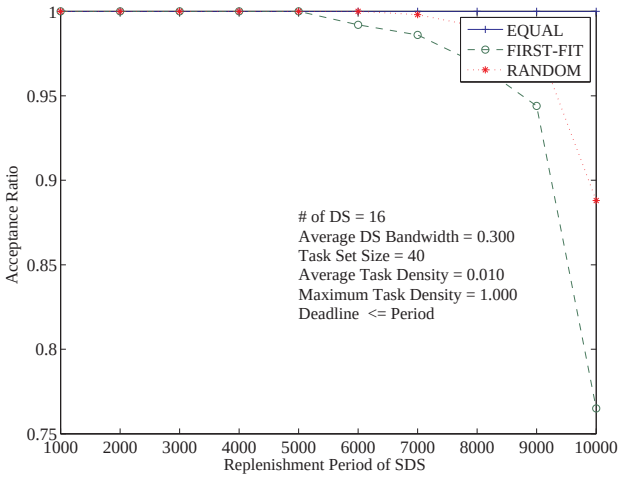

Figure 247. Acceptance ratio v.s. replenishment period

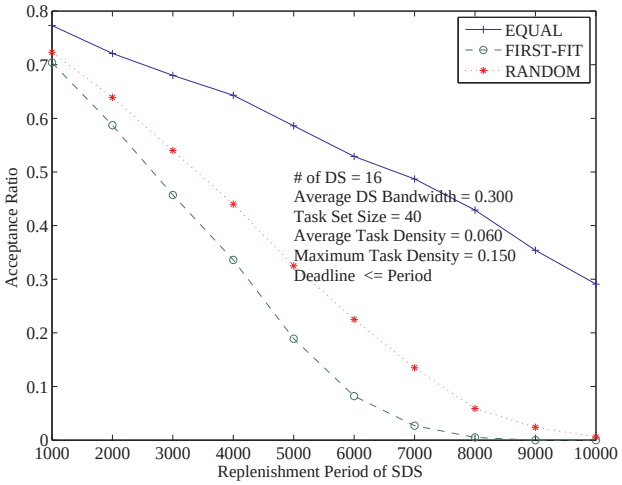

Figure 250. Acceptance ratio v.s. replenishment period

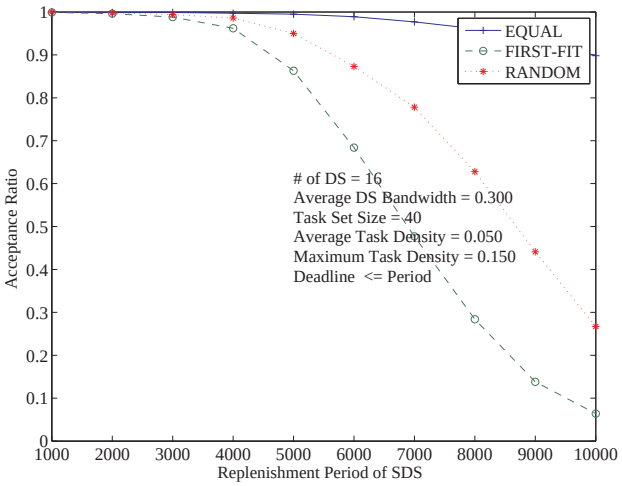

Figure 248. Acceptance ratio v.s. replenishment period

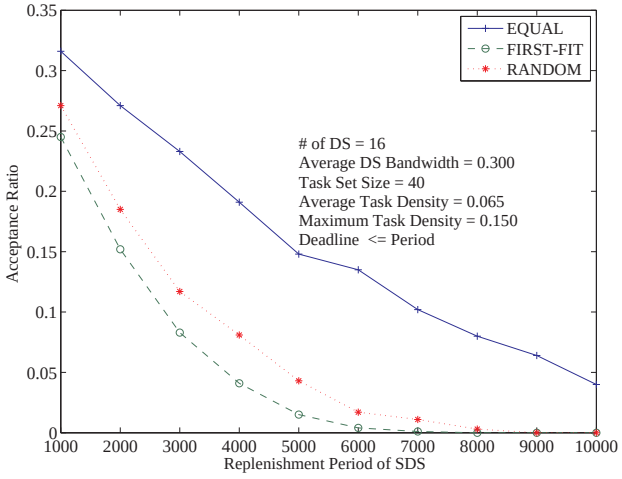

Figure 251. Acceptance ratio v.s. replenishment period

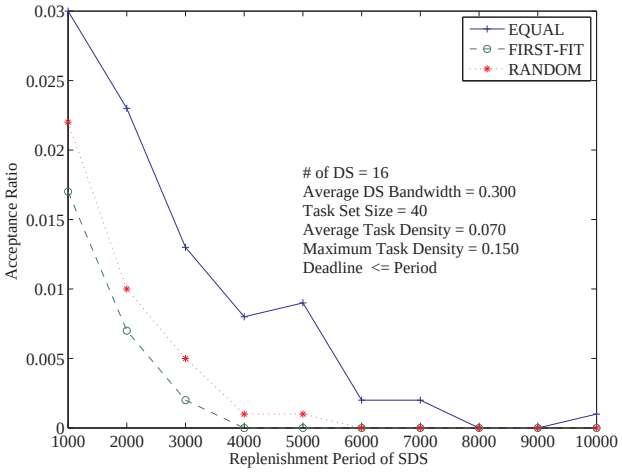

Figure 252. Acceptance ratio v.s. replenishment period

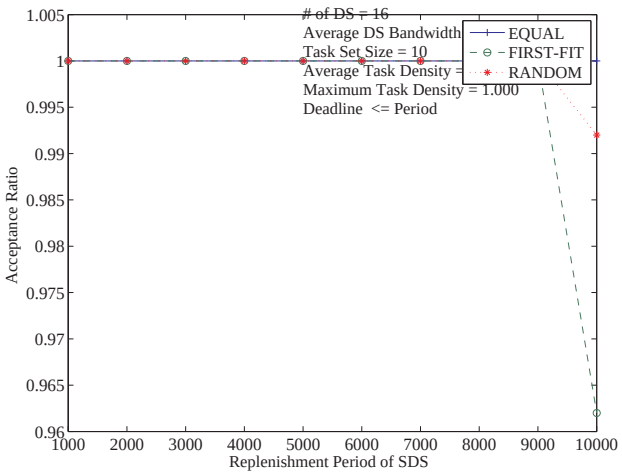

Figure 255. Acceptance ratio v.s. replenishment period

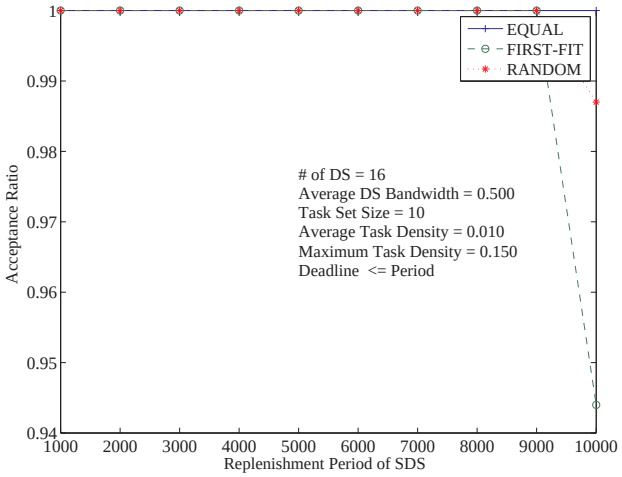

Figure 253. Acceptance ratio v.s. replenishment period

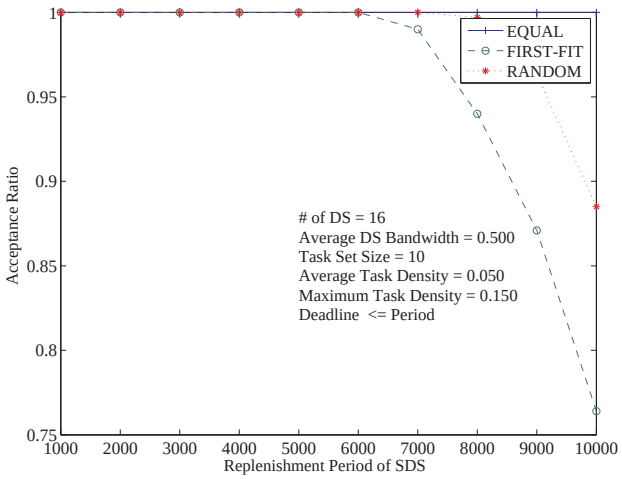

Figure 256. Acceptance ratio v.s. replenishment period

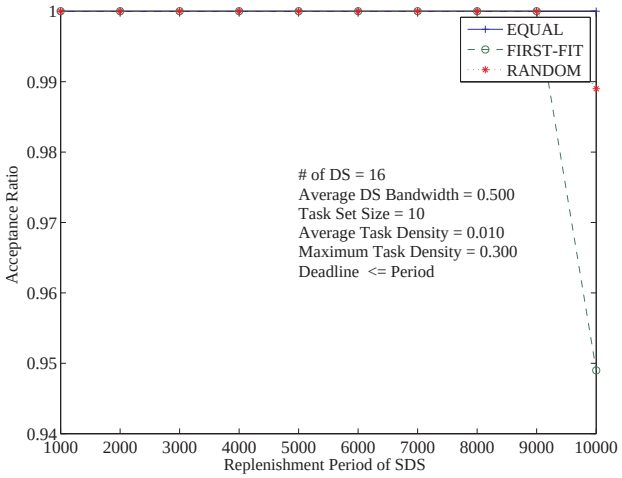

Figure 254. Acceptance ratio v.s. replenishment period

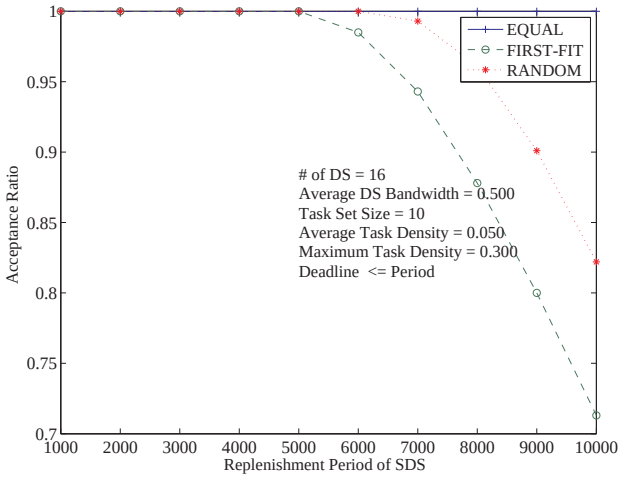

Figure 257. Acceptance ratio v.s. replenishment period

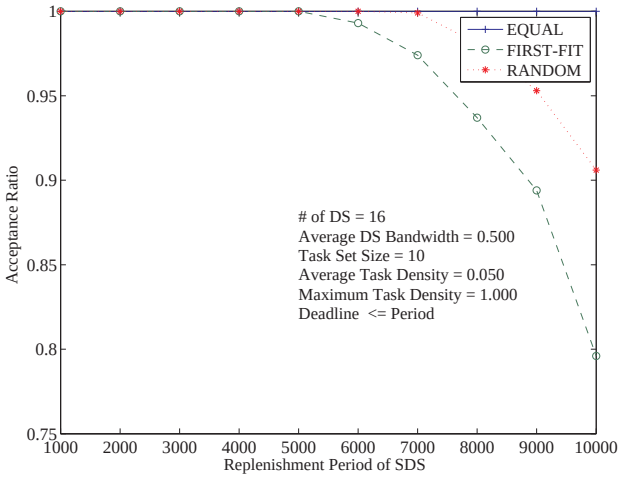

Figure 258. Acceptance ratio v.s. replenishment period

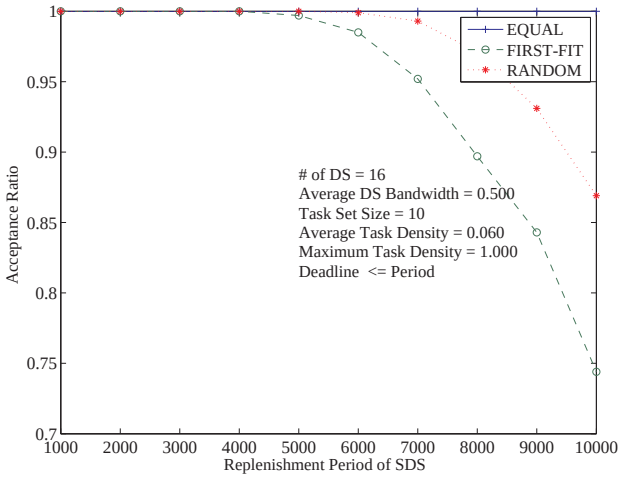

Figure 261. Acceptance ratio v.s. replenishment period

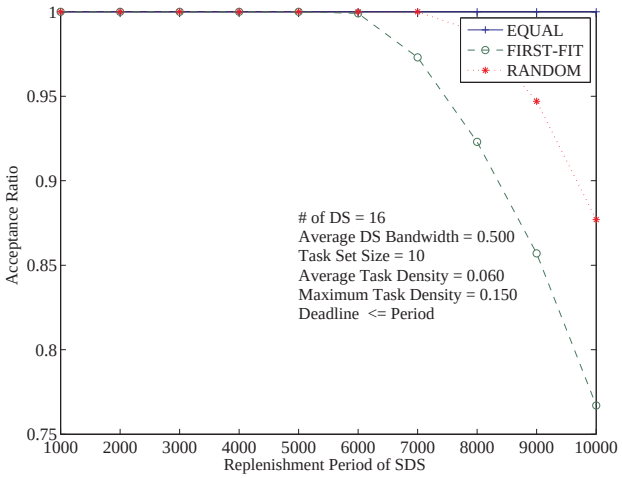

Figure 259. Acceptance ratio v.s. replenishment period

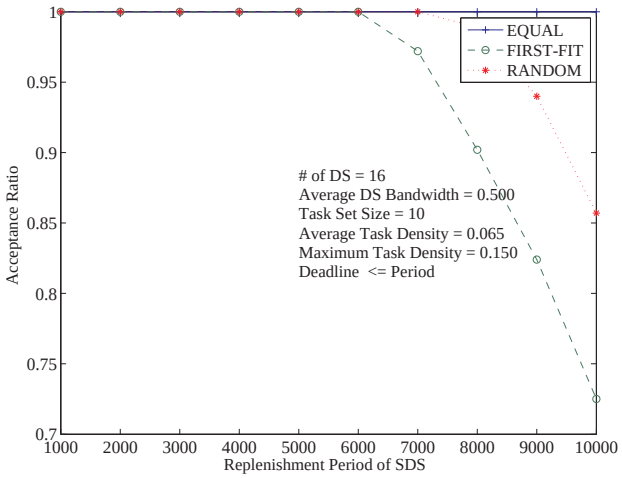

Figure 262. Acceptance ratio v.s. replenishment period

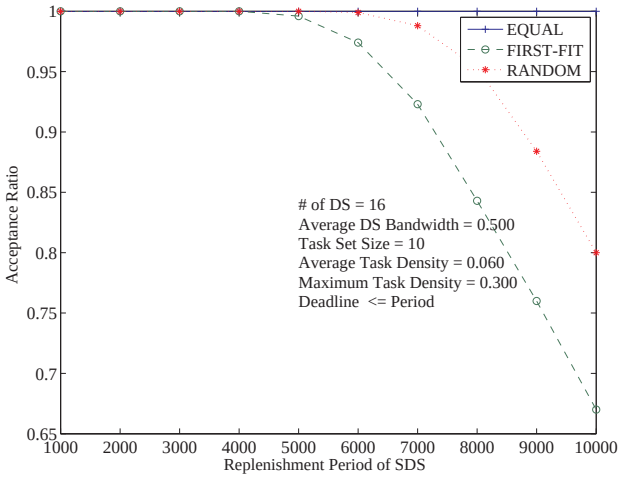

Figure 260. Acceptance ratio v.s. replenishment period

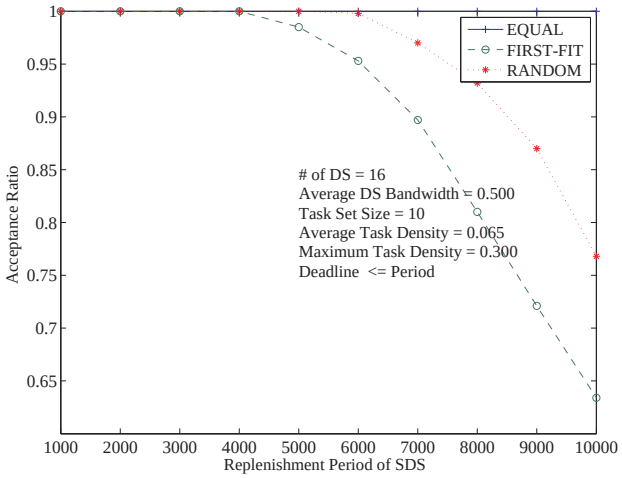

Figure 263. Acceptance ratio v.s. replenishment period

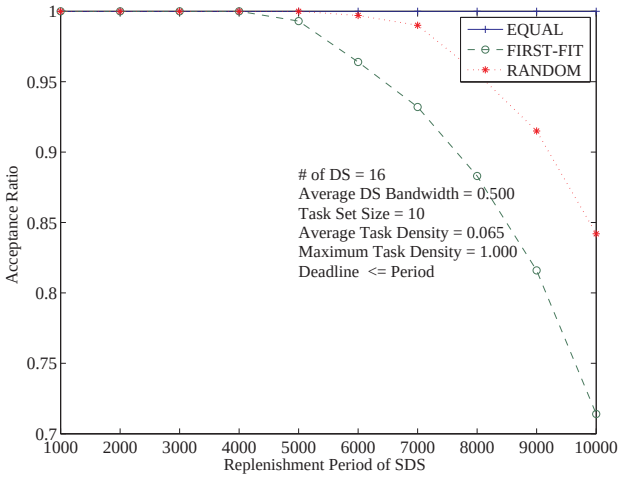

Figure 264. Acceptance ratio v.s. replenishment period

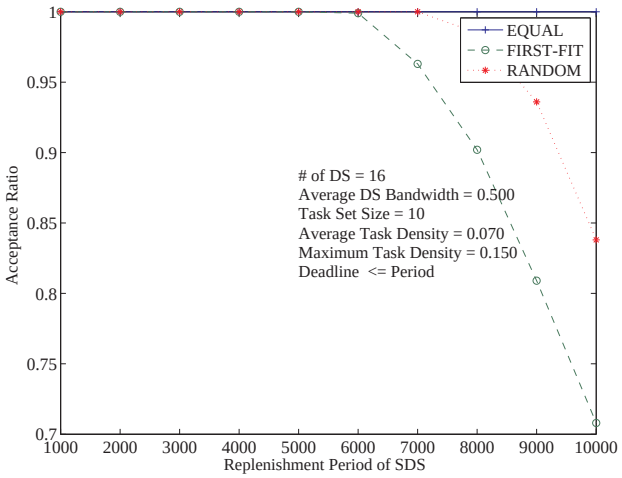

Figure 265. Acceptance ratio v.s. replenishment period

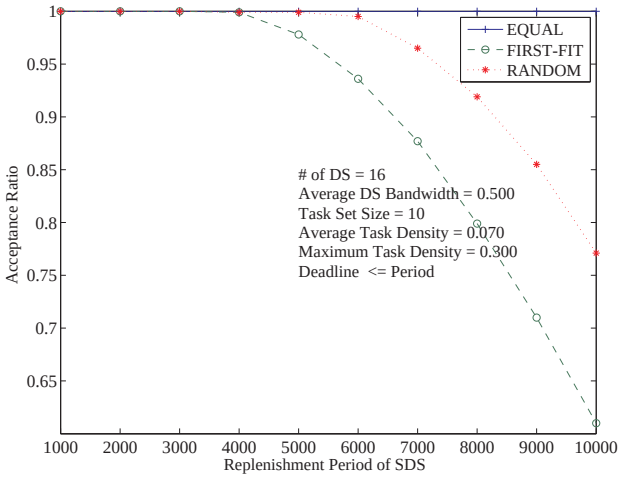

Figure 266. Acceptance ratio v.s. replenishment period

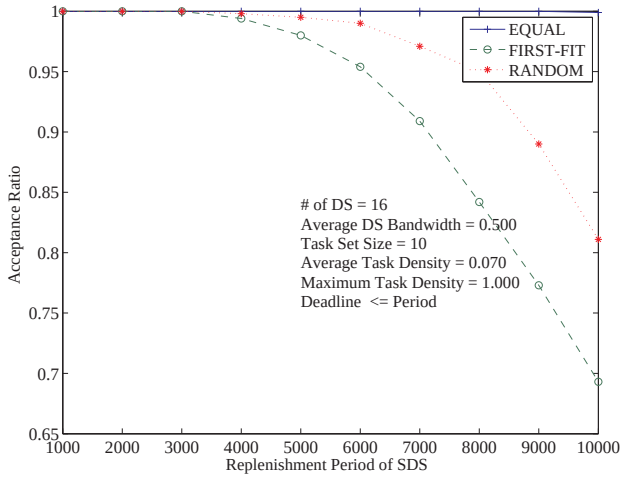

Figure 267. Acceptance ratio v.s. replenishment period

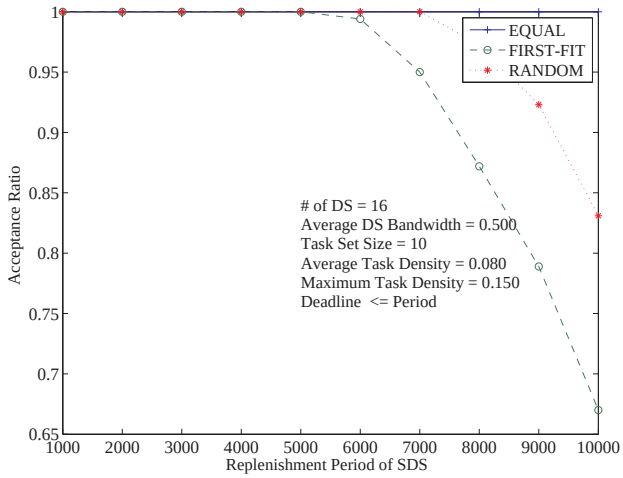

Figure 268. Acceptance ratio v.s. replenishment period

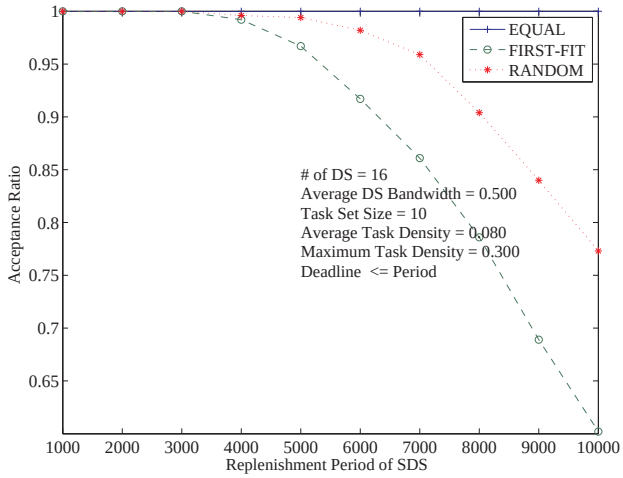

Figure 269. Acceptance ratio v.s. replenishment period

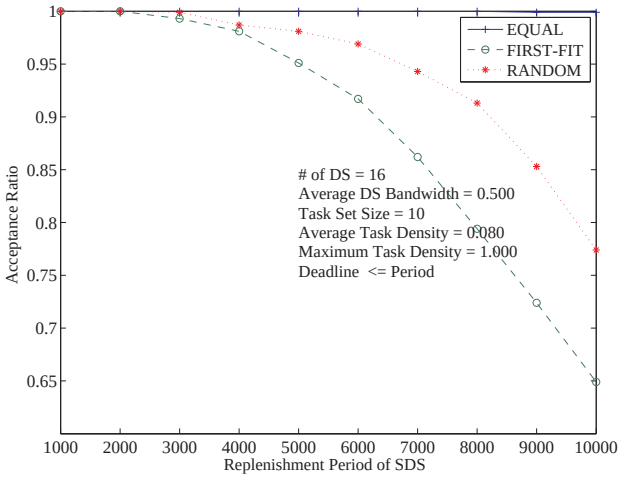

Figure 270. Acceptance ratio v.s. replenishment period

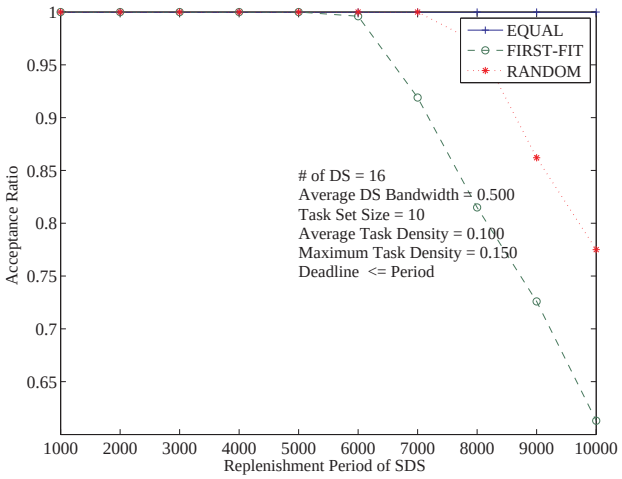

Figure 271. Acceptance ratio v.s. replenishment period

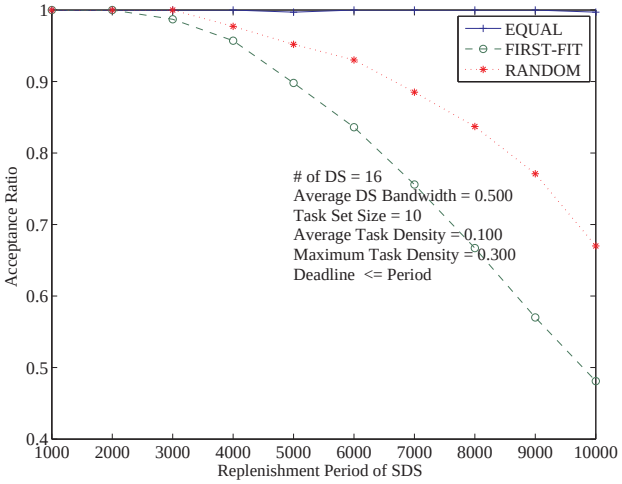

Figure 272. Acceptance ratio v.s. replenishment period

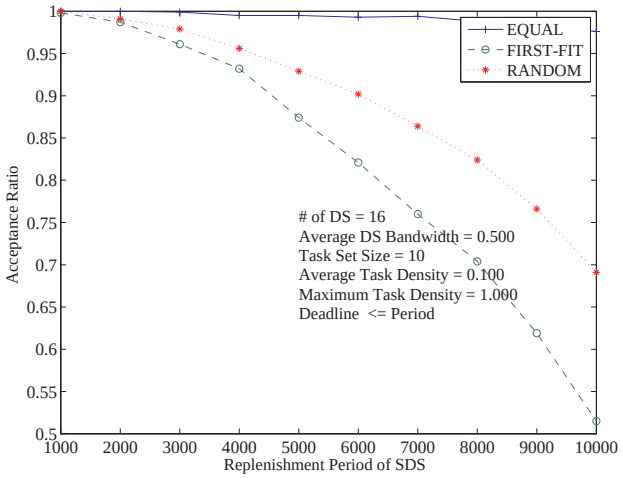

Figure 273. Acceptance ratio v.s. replenishment period

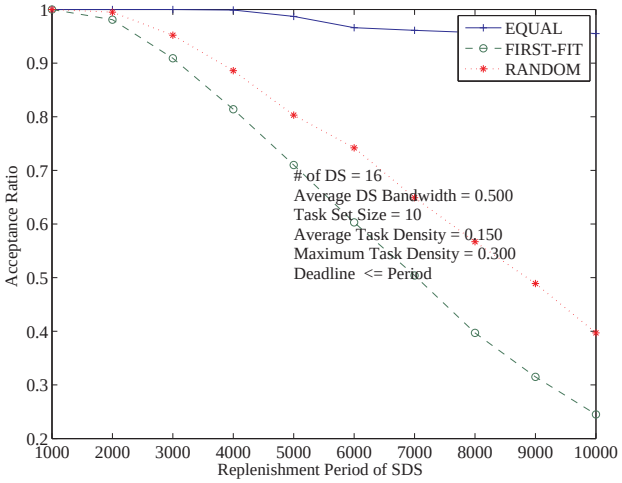

Figure 274. Acceptance ratio v.s. replenishment period

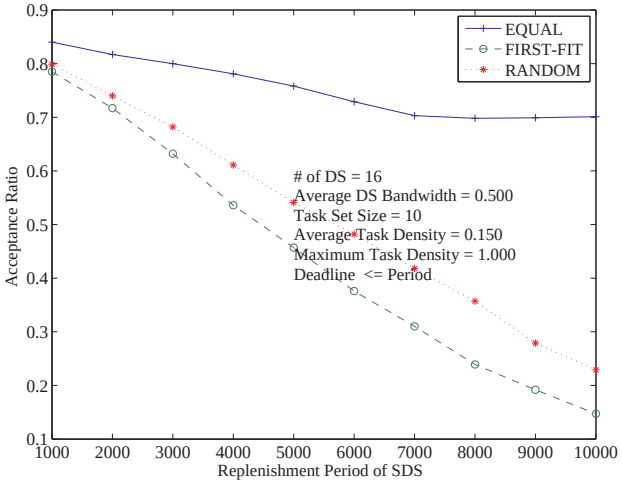

Figure 275. Acceptance ratio v.s. replenishment period

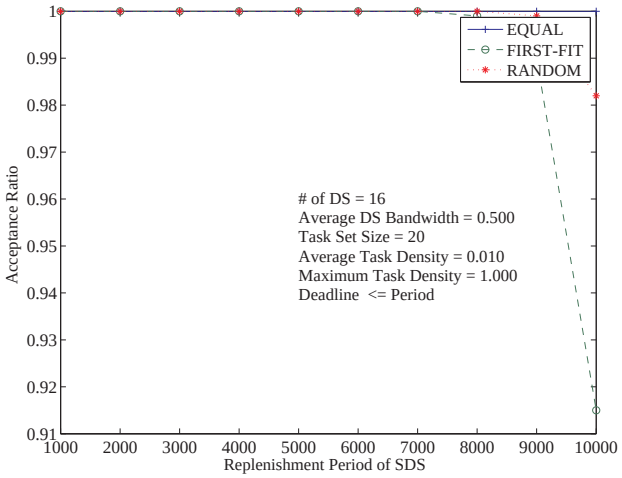

Figure 276. Acceptance ratio v.s. replenishment period

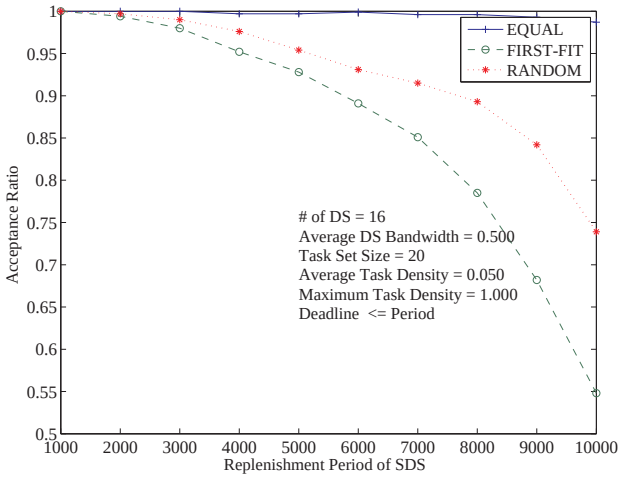

Figure 279. Acceptance ratio v.s. replenishment period

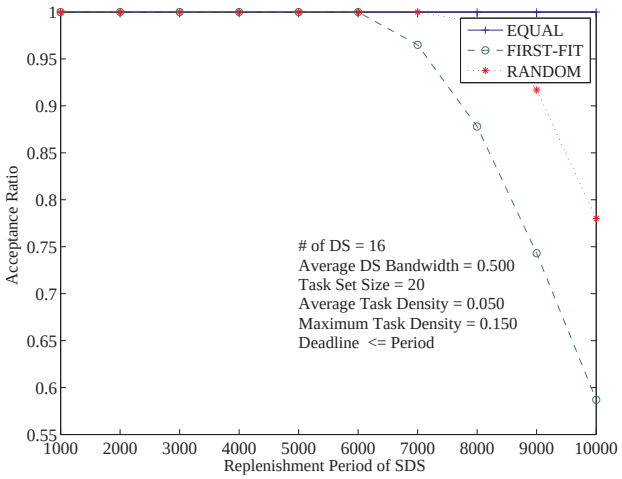

Figure 277. Acceptance ratio v.s. replenishment period

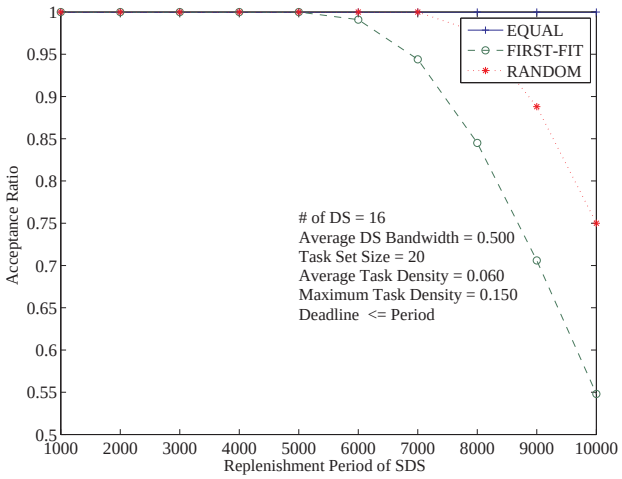

Figure 280. Acceptance ratio v.s. replenishment period

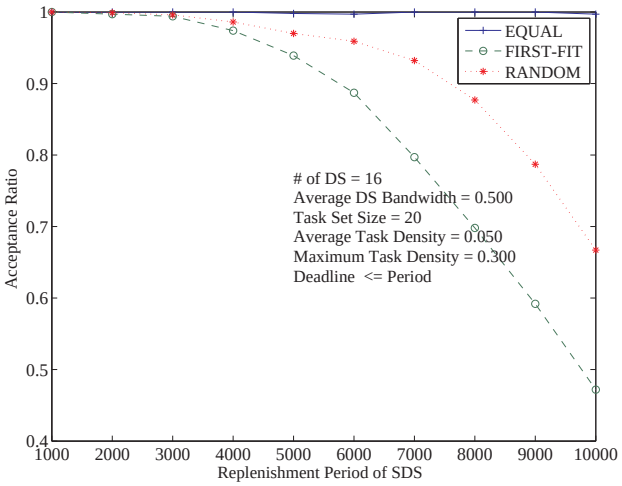

Figure 278. Acceptance ratio v.s. replenishment period

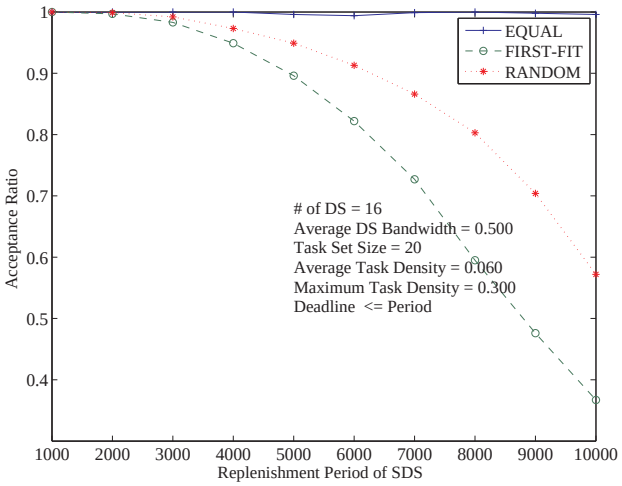

Figure 281. Acceptance ratio v.s. replenishment period

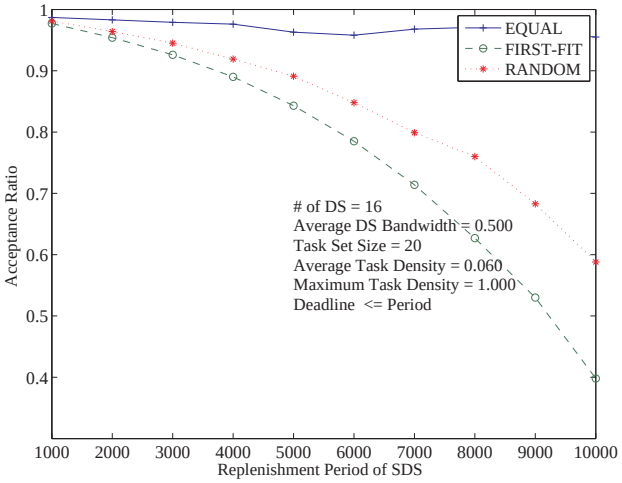

Figure 282. Acceptance ratio v.s. replenishment period

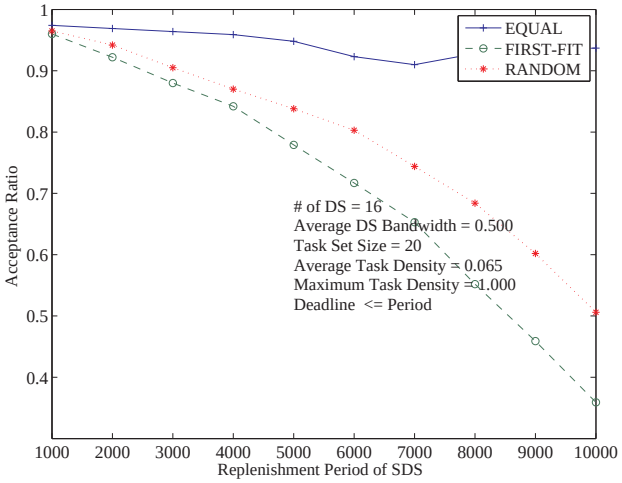

Figure 285. Acceptance ratio v.s. replenishment period

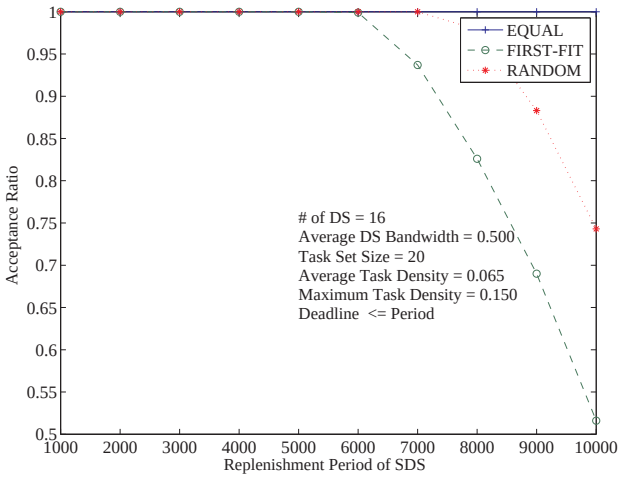

Figure 283. Acceptance ratio v.s. replenishment period

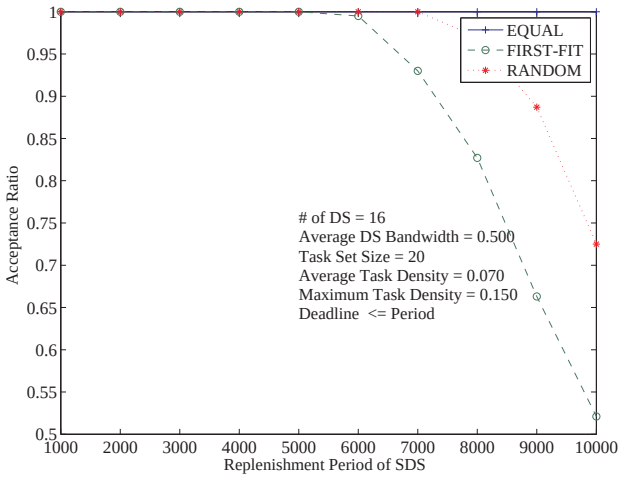

Figure 286. Acceptance ratio v.s. replenishment period

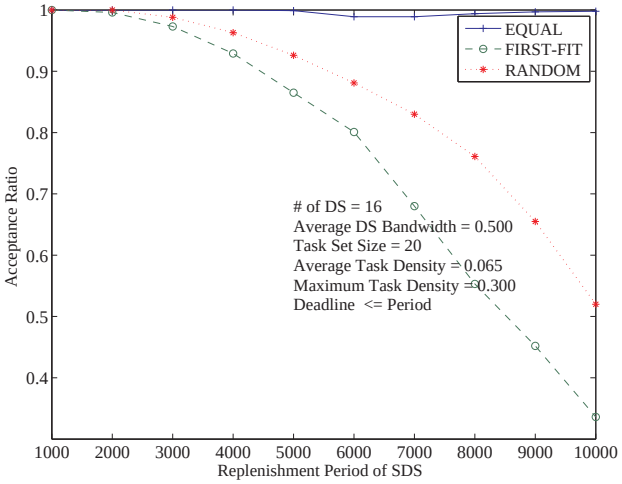

Figure 284. Acceptance ratio v.s. replenishment period

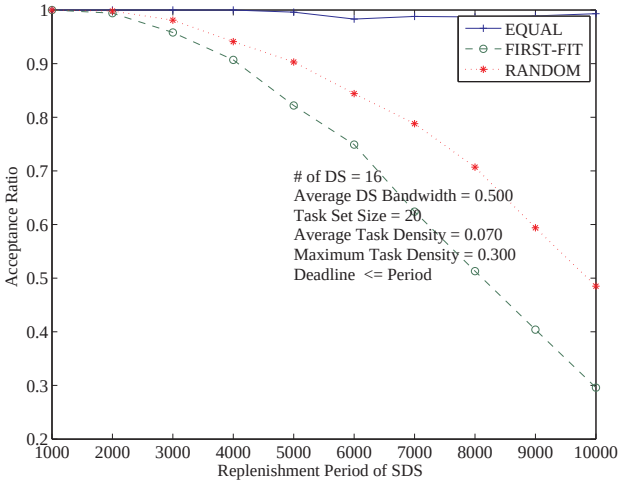

Figure 287. Acceptance ratio v.s. replenishment period

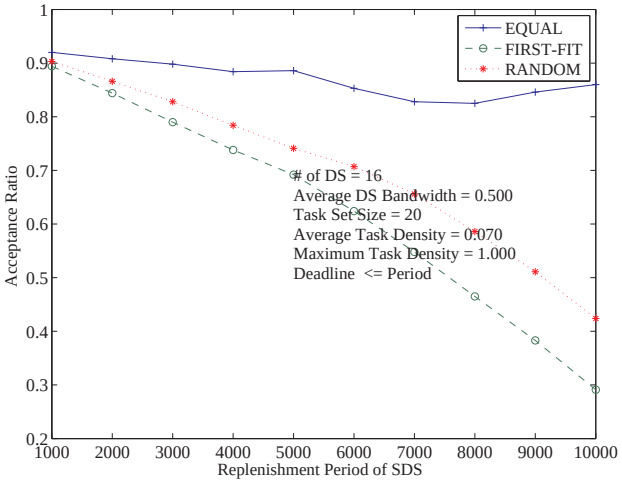

Figure 288. Acceptance ratio v.s. replenishment period

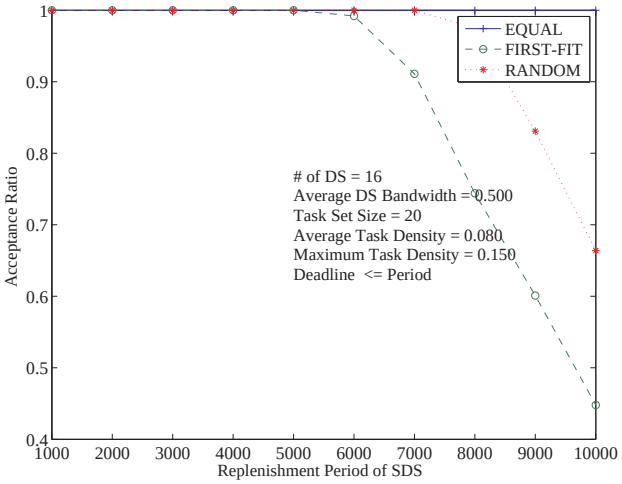

Figure 289. Acceptance ratio v.s. replenishment period

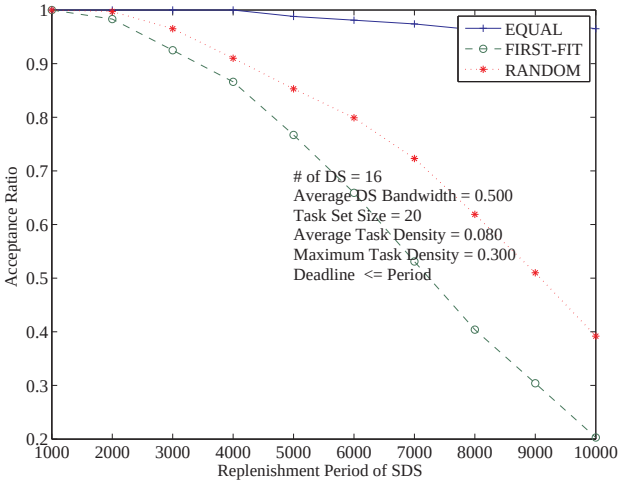

Figure 290. Acceptance ratio v.s. replenishment period

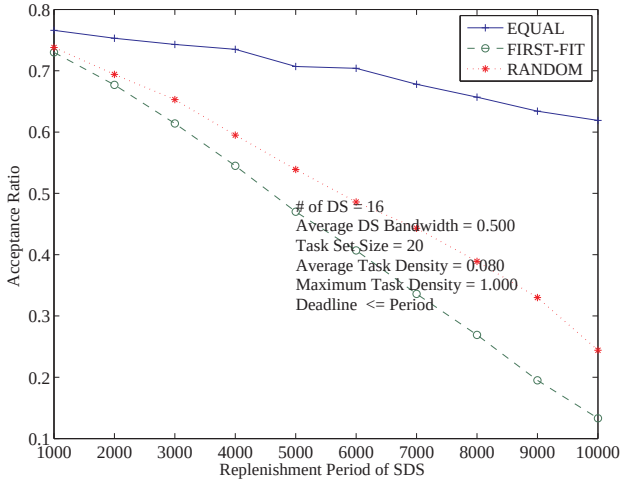

Figure 291. Acceptance ratio v.s. replenishment period

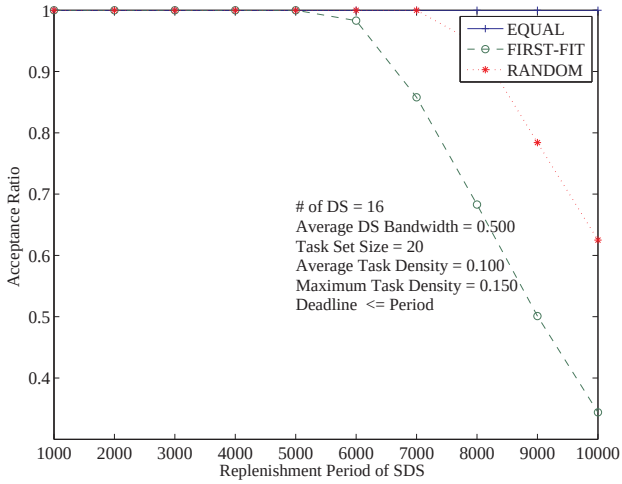

Figure 292. Acceptance ratio v.s. replenishment period

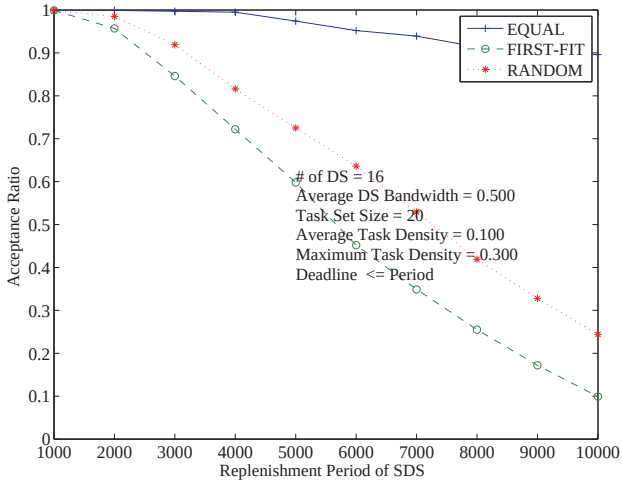

Figure 293. Acceptance ratio v.s. replenishment period

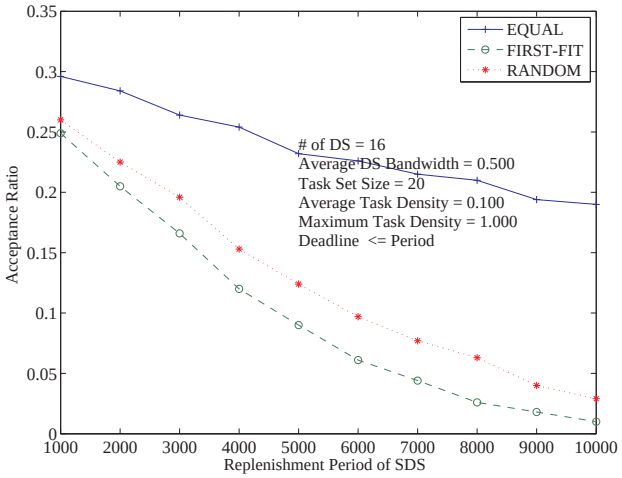

Figure 294. Acceptance ratio v.s. replenishment period

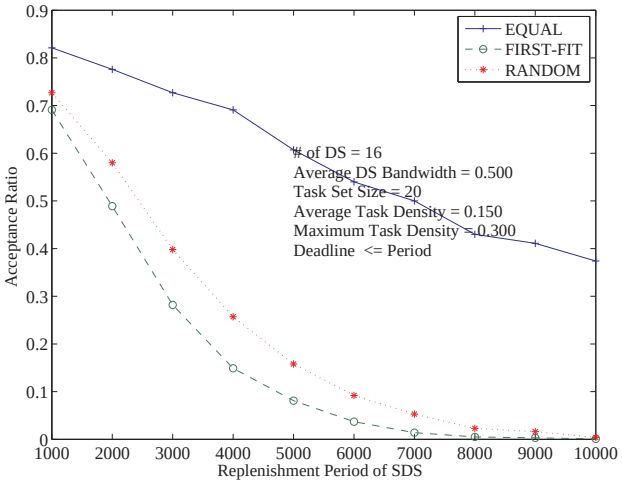

Figure 295. Acceptance ratio v.s. replenishment period

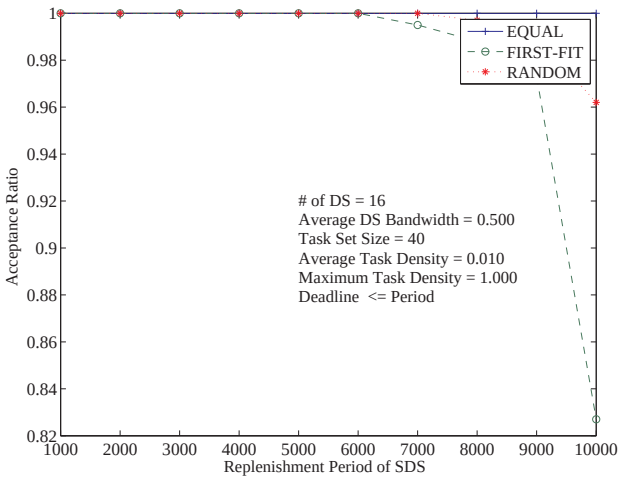

Figure 296. Acceptance ratio v.s. replenishment period

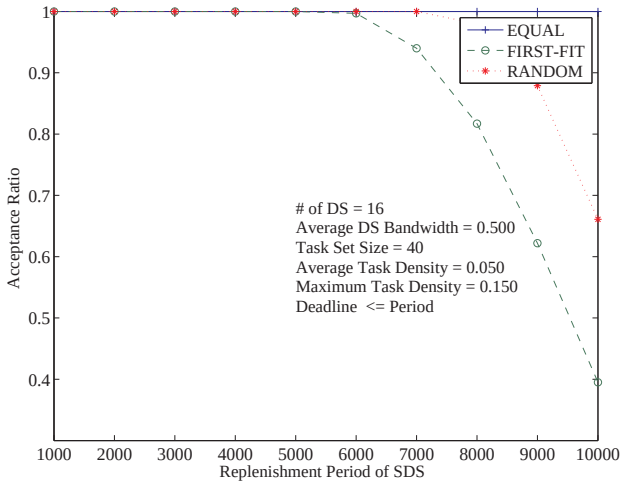

Figure 297. Acceptance ratio v.s. replenishment period

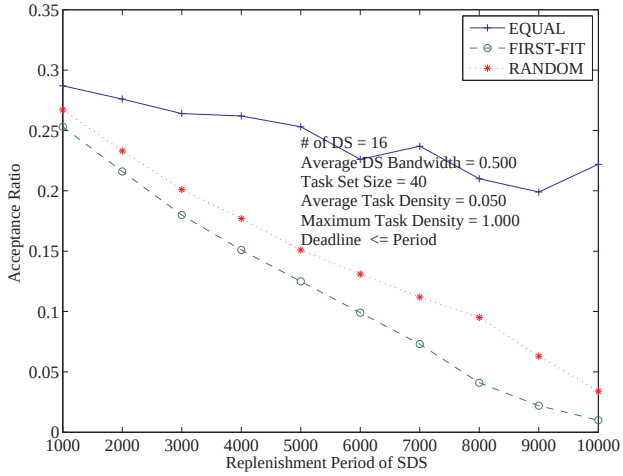

Figure 298. Acceptance ratio v.s. replenishment period

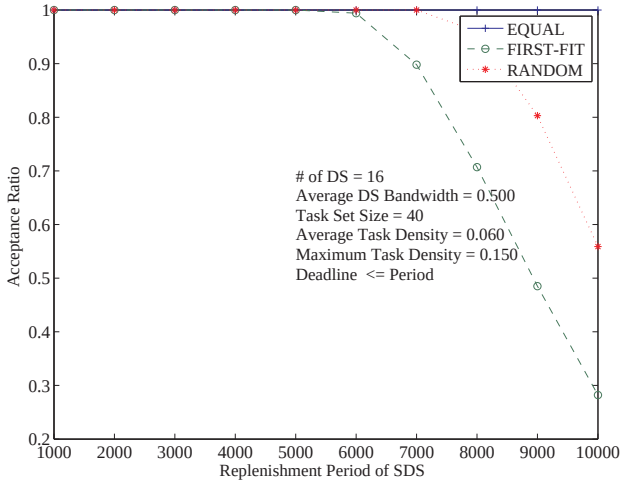

Figure 299. Acceptance ratio v.s. replenishment period

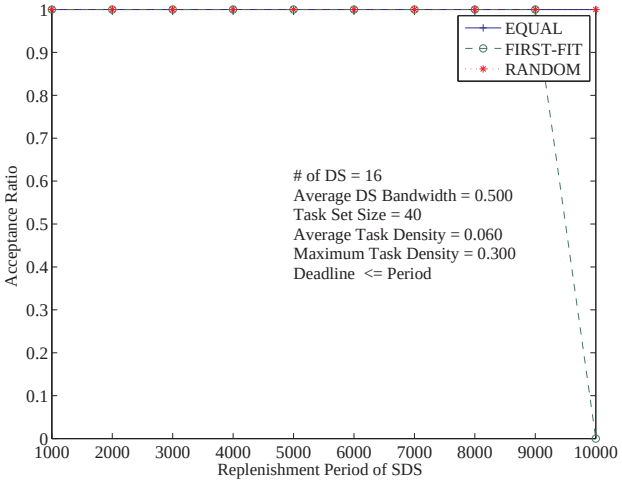

Figure 300. Acceptance ratio v.s. replenishment period

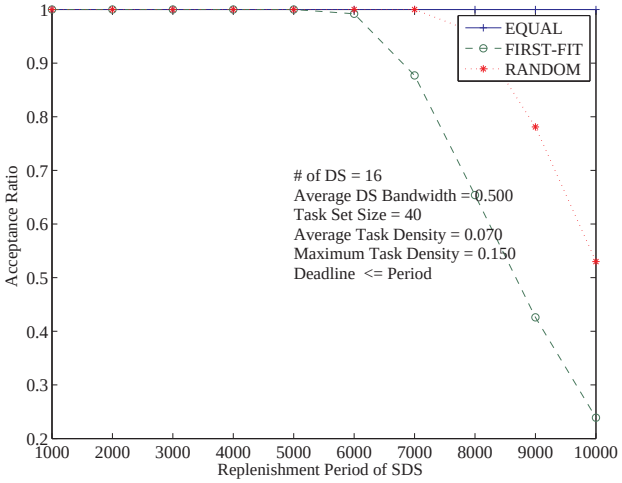

Figure 303. Acceptance ratio v.s. replenishment period

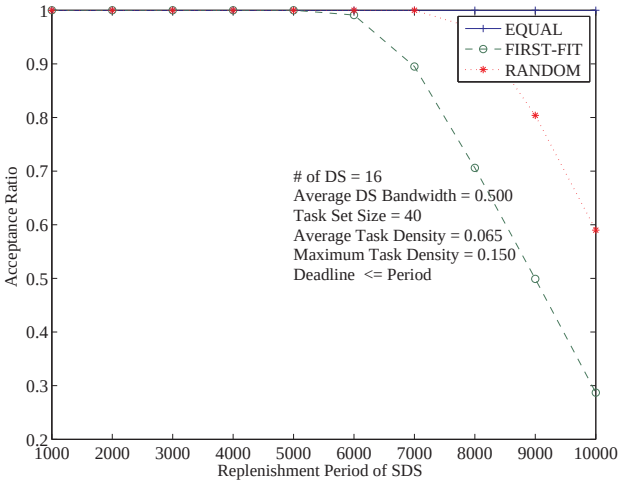

Figure 301. Acceptance ratio v.s. replenishment period

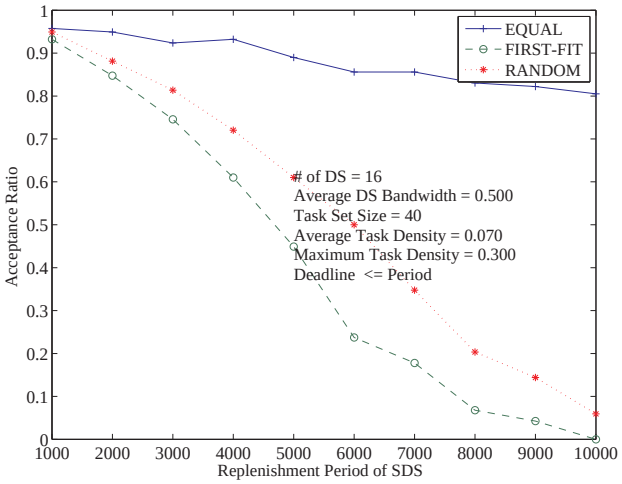

Figure 304. Acceptance ratio v.s. replenishment period

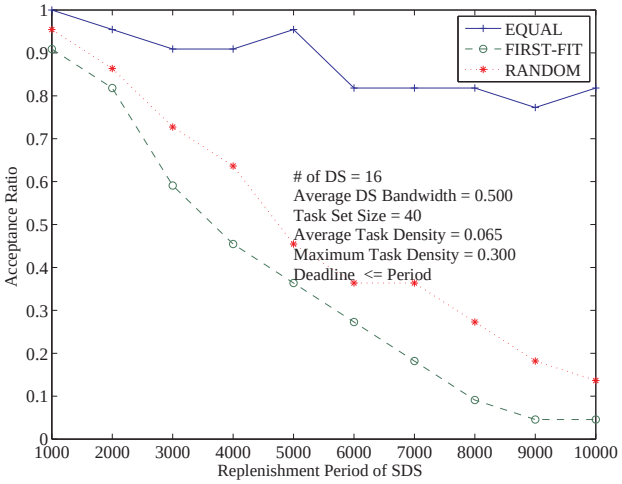

Figure 302. Acceptance ratio v.s. replenishment period

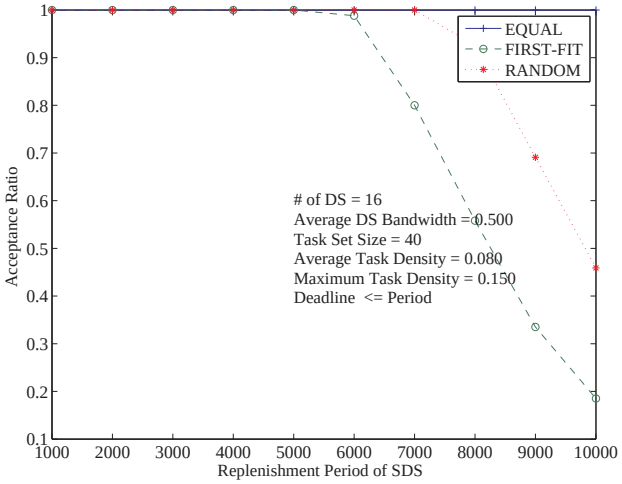

Figure 305. Acceptance ratio v.s. replenishment period

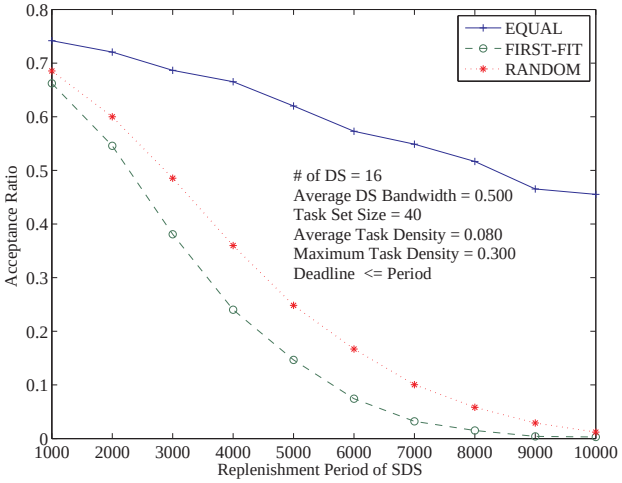

Figure 306. Acceptance ratio v.s. replenishment period

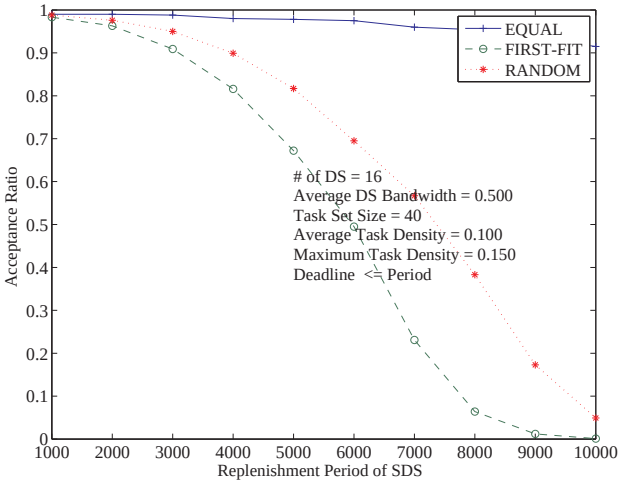

Figure 307. Acceptance ratio v.s. replenishment period

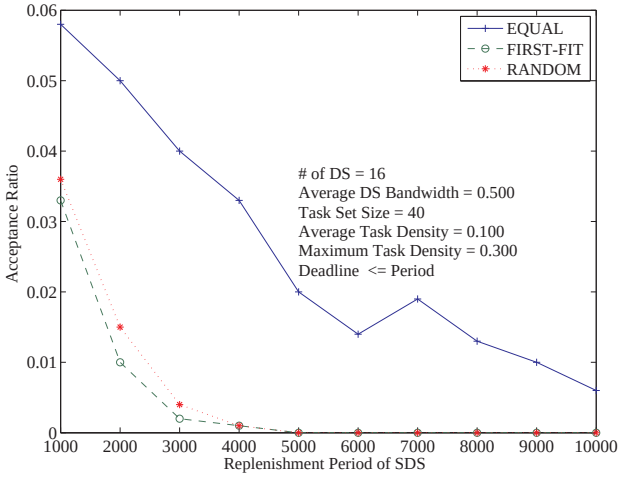

Figure 308. Acceptance ratio v.s. replenishment period

References

[1] `http://www.cs.unc.edu/~anderson/litmus-rt/`.

[2] `http://www.runtime-verification.org`.

[3] B. Andersson and K. Bletsas. Sporadic multiprocessor scheduling with few preemptions. In *ECRTS*, pages 243–252, 2008.

[4] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *RTCSA*, pages 322–334, 2006.

[5] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[6] T. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *RTSS*, pages 120–129, 2003.

[7] T. Baker. An analysis of EDF schedulability on a multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, pages 760–768, 2005.

[8] T. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Realtime and Embedded Systems*. Citeseer, 2007.

[9] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *RTSS*, pages 119–128, 2007.

[10] S. Baruah, J. Goossens, and G. Lipari. Implementing constant-bandwidth servers upon multiprocessor platforms. In *RTAS*, pages 154–163, 2002.

[11] S. Baruah and G. Lipari. A multiprocessor implementation of the total bandwidth server. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[12] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *RTSS*, page 68, Washington, DC, USA, 1999. IEEE Computer Society.

[13] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *RTSS*, pages 149–160, 2007.

[14] P. Cuijpers and R. Bril. Towards Budgeting in Real-Time Calculus: Deferrable Servers. In *Lecture Notes in Computer Science*, volume 4763, page 98. Springer, 2007.

[15] R. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *RTSS*, pages 398–409. IEEE, 2009.

[16] R. I. Davis and A. Burns. Hierarchical fixed priority preemptive scheduling. In *RTSS*, pages 389–398, 2005.

[17] Z. Deng, J. W. S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In J. W. S. Liu, editor, *EWRTS*, pages 191–199. Sun, J., 1997.

[18] S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, pages 127–140, 1978.

[19] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2):187–205, 2003.

[20] N. Guan, M. Stigge, W. Yi, and G. Yu. New response time bounds for fixed priority multiprocessor scheduling. In *RTSS*, pages 387 –397, 2009.

[21] N. Guan, M. Stigge, W. Yi, and G. Yu. Fixed-priority multiprocessor scheduling with liu & laylands utilization bound. In *RTAS*, 2010.

[22] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 155–174, 2009.

[23] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390, 1986.

[24] S. Kato and N. Yamasaki. Portioned Static-Priority Scheduling on Multiprocessors. In *IPDPS*, 2008.

[25] S. Kato and N. Yamasaki. Semi-Partitioned Fixed-Priority Scheduling on Multiprocessors. In *RTAS*, pages 23–32. IEEE Computer Society Washington, DC, USA, 2009.

[26] K. Lakshmanan, R. R. Rajkumar, and J. P. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *ECRTS*, pages 239 – 248, 2009.

[27] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *RTSS*, pages 166–171, Dec 1989.

[28] G. Lipari and E. Bini. A framework for hierarchical scheduling on multiprocessors: from application requirements to run-time allocation. In *RTSS*, pages 249–258, 2010.

[29] S. Saewong, R. Rajkumar, J. Lehoczky, and M. Klein. Analysis of hierarchical fixed-priority scheduling. In *ECRTS*, pages 152–160, Washington, DC, USA, 2002. IEEE Computer Society Press.

[30] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *ECRTS*, pages 181–190. IEEE, 2008.

[31] A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors* 1. *Information Processing Letters*, 84(2):93–98, 2002.

[32] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, 1995.

[33] H. Zhu, M. B. Dwyer, and S. Goddard. Predictable runtime monitoring. In *ECRTS*, pages 173–183, 2009.