Dynamic Voltage Scaling for the Schedulability of Jitter-Constrained Real-Time Embedded Systems*

Bren Mochocki and Xiaobo Sharon Hu

Department of CSE
University of Notre Dame
Notre Dame, IN 46556
{bmochock, shu}@cse.nd.edu

Razvan Racu and Rolf Ernst
Technical University of Braunschweig
Institute of Computer and
Communication Network Engineering
D-38106 Braunschweig, Germany
{razvan, ernst}@ida.ing.tu-bs.de

***Abstract*— Jitter is a critical problem for the design of both distributed embedded systems and real-time control systems. This work considers meeting the completion jitter constraints of a set of independent, periodic, hard real-time tasks scheduled according to a preemptive fixed-priority scheme. Control over completion jitter is achieved by judiciously applying Dynamic Voltage Scaling (DVS). Through simulation, the proposed method is shown to be an effective tool to meet jitter constraints on a variety of systems.**

I. Introduction

Given the recent demand for portable and low-power computation devices designed for extended battery life, current interest in design techniques for power management and energy reduction is no surprise. One such technique that has received a considerable amount of attention in recent years is called Dynamic Voltage Scaling (DVS). DVS capitalizes on the inherent convex dependence of power on supply voltage in CMOS based circuitry to trade system performance for reduced power and energy consumption. DVS is particularly useful in embedded real-time applications, as task parameters and timing constraints are well defined in such an environment. Many real-time systems are composed of tasks that release jobs periodically, a topic that has been considered in the DVS literature [6], [10].

An important class of constraints that has not been considered by the DVS community is known as Jitter. In the context of real-time periodic tasks, jitter refers to the variation of the release time or completion time of a periodic task from its period. There are many possible sources of jitter, e.g., preemption due to priority-based scheduling or variation in execution times. Jitter can have a profound impact on the performance of a system. Marti *et. al.* show that jitter in a real-time control system can cause performance degradation or even instability [8]. Another area where jitter becomes problematic is the integration and synchronization of heterogeneous system components. This is a key problem for Computer-Aided Design (CAD) tools such as Symbolic Timing Analysis for Systems (SymTA/S) [12].

SymTA/S is a software tool for formal performance analysis of heterogeneous SoCs and distributed systems. The SymTA/S analysis approach couples local scheduling algorithms using event streams. Event streams describe the possible I/O timing of tasks and are characterized by appropriate event models such as periodic events with jitter or bursts and sporadic events. SymTA/S supports the combination and integration of different analysis techniques known from real-time research. For this purpose, it is essential to transition between the often incompatible event stream models resulting from the dissimilitude of the local techniques. Such incompatibilities appear, for instance, between an analysis technique assuming simple periodic (no jitter) events at the input and another that generates jitter at its output.

In this paper, we show that DVS has a large potential to both increase jitter when used aggressively, or to meet jitter constraints when used judiciously. Specifically, we consider meeting completion jitter constraints on a set of independent, periodic, hard real-time tasks scheduled according to a preemptive fixed-priority scheme. Fixed priority is often the scheduling method of choice due to its high predictability and low overhead [7]. To the best of our knowledge, this is the first work that considers utilizing DVS for any purpose other than power/energy reduction.

*This work is supported in part by the ARTIST Network of Excellence and by NSF under grant numbers CCR02-08992 and CNS-0410771.

There is a considerable amount of work on managing jitter in real-time systems. In [8], Marti *et. al.* present a method to make control systems more tolerant of jitter by directly modifying control algorithms, e.g., software implementations of PID or State Feedback controllers. For systems scheduled according to EDF, Kim *et. al.* present a linear program formulation to assign task deadlines [5]. Baruah *et. al.* present two deadline assignment algorithms that bound the completion jitter of EDF tasks in polynomial and pseudo-polynomial time respectively [1].

There has also been considerable work on jitter within the fixed-priority framework. Tindell *et. al.* present a method for calculating the worst case response time of periodic and sporadic fixed-priority tasks that exhibit input jitter and have arbitrary deadlines [13]. Bate and Burns present a heuristic method that reduces task deadlines and/or changes the offset of the initial task release times in order to meet jitter constraints [2]. Cervin also presents a heuristic that reduces deadlines and introduces offset [3]. Additionally, tasks are prioritized in deadline monotonic order, so a priority assignment is also considered with respect to the altered deadlines. David *et. al.* present a method that partitions tasks into jitter-constrained and non-jitter- constrained sets [4]. The tasks are then offset optimally using the Chinese Remainder Theorem to avoid the overlap of execution windows. Deadline assignment is also considered in this method. In general, deadline assignment is not the best method from a schedulability standpoint, as the methods presented in [2], [4] may not guarantee task deadlines after the deadlines are adjusted. Also, when considering DVS, decreasing task deadlines translates directly into less flexibility for energy minimization. Changing the priority assignment or introducing task offset may or may not be possible, depending on the application in question. If possible, DVS can still be applied to further reduce completion jitter.

An effective method for eliminating jitter is to buffer the input (for input jitter) and output (for completion jitter). Indeed, this is the approach used by Richter *et. al.* in [11]. However, this method is only applicable during system design. If, for example, a software update to an already designed and deployed system requires an additional task to be added, the current hardware buffers in the system may not be sufficient. In this case a software solution is more flexible. Another disadvantage of the hardware approach is the additional space and power required by the buffers. Given that DVS will already be included in many emerging low-power systems, the benefit of DVS on jitter is essentially a free fringe benefit that the designer can exploit.

The remainder of this paper is organized as follows. Section II presents definitions, notation and a motivational example. Section III describes the proposed method. Section IV presents the experimental results. Finally, we conclude with Section V and offer several future directions of research. Note that all material omitted due to the page limit (e.g., proofs) can be found in [9].

II. Preliminaries

We consider real-time systems composed of a set of n periodic tasks, $\mathcal{T} = \{T_1, T_2, ..., T_n\}$. Task T_i is said to have a higher priority than task T_j if $i < j$. Each task, $T_i = (wc_i, p_i, d_i)$, is described by its worst case execution cycles, wc_i, period, p_i, and relative deadline, d_i, with $d_i \leq p_i$. The system *hyperperiod*, is the least common multiple of all task periods. Each task is invoked periodically and we refer to

the k-th invocation of task T_i as job J_i^k. The set of all jobs within one hyperperiod is denoted by $\mathcal{J}$, while $\mathcal{J}_i$ represents the set of all jobs of task T_i in $\mathcal{J}$ and $\mathcal{J}_{hp(i)}$ represents the set of all jobs in $\mathcal{J}$ with a priority higher than T_i. Each job $J_i^k = (r_i^k, d_i^k, e_i^k)$ is composed of a release time, r_i^k, deadline, d_i^k, and end time, e_i^k. For any given job, the term *relative time* referrers to some time measurement minus the release time of the job. For example, the *relative deadline* is $d_i^k - r_i^k$ and the *relative end time* is $e_i^k - r_i^k$. For now we assume that the the DVS processor can operate at any voltage in the range $[V_{min}, V_{max}]$, with corresponding speeds in $[S_{min}, S_{max}]$ and incurs negligible transition overhead. A voltage schedule, denoted $\mathcal{SCH} = \{J_i^k, S_i^k\}$, is a set of job-speed pairs such that every job in $\mathcal{J}$ maps to exactly one speed. The impact of processor limitations such as transition overhead and discrete voltage levels is left for future work.

Release or *input jitter* refers to the variation in job release times, while completion jitter is the variation in job completion times. Completion jitter arises from several sources: (i) input jitter itself can cause completion times of a task to vary, (ii) preemption jitter, or jitter caused by task preemption, and (iii) execution jitter, when jobs require fewer cycles than the worst case to complete. Completion jitter can be measured in two ways. Inter-completion jitter is measured with respect to consecutive jobs, and is important for control applications. Absolute jitter refers to the difference between the minimum and maximum system variable in question. This work specifically targets completion jitter due to task preemptions.

Definition 1: **Inter-Completion Jitter-** The variation in relative end times of consecutive jobs of the same task, i.e.,

$$iJtr(T_i, k) = |(e_i^j - r_i^j) - (e_i^k - r_i^k)|, \quad (1)$$

$$j = k \bmod |\mathcal{J}_i| + 1$$

The modulus is used to compare the first and last jobs in the hyperperiod.

Definition 2: **Absolute Completion Jitter-** The difference between the maximum and minimum response times of jobs of the same task, i.e.,

$$aJtr(T_i) = \max(e_i^k - r_i^k) - \min(e_i^k - r_i^k) \quad (2)$$

$$k = 1..|\mathcal{J}_i|$$

Based on (1) and (2) it is trivial to show that the following lemma holds.

Lemma 1: $iJtr(T_i, k) \leq aJtr(T_i)$ for any task T_i and any value of k where $1 \leq k \leq |J_i|$.

We define a set of jitter constraints, $\mathcal{JC}$, as a set of 3-tuples $\{(i, type, value)\}$. The first element, i, indicates that the associated task is T_i. The second element, $type \in \{a, ic\}$, takes on the value a if the constraint is on the absolute jitter, and ic if the constraint is on the maximum inter-completion jitter. The final element, $value$, indicates the magnitude of allowable jitter. If $value = \infty$ then the task is unconstrained by the given type of jitter. $\mathcal{JC}$ is indexed by $\mathcal{JC}_i^{type}$, which returns $value$. If a given task set can meet all jitter constraints without deadline violations, we say that the task set is *schedulable* under the given constraints.

Next, the impact of aggressively applying DVS on completion jitter is introduced. Figure 1 gives an example task set with two tasks having $p_1 = 30$, $d_1 = 20$, $wc_1 = 10$, $p_2 = 40$, $d_2 = 40$ and $wc_2 = 10$. In (a), the tasks are executed using the maximum processor speed, resulting in $aJtr(T_1) = 0$ and $aJtr(T_2) = 10$. In (b), DVS is applied to minimize the energy consumption. However, this causes an increase in jitter for both tasks, with $aJtr(T_1) = 5$, and $aJtr(T_2) = 20$. Finally, in (c), DVS is applied less aggressively, resulting in $aJtr(T_2) = 1$. Note that there is still slack available in (c), which could be used to either minimize the jitter for lower priority tasks, if they were present, or to minimize energy consumption. The problem of interest is introduced formally in Problem 1. An important extension to Problem 1 is how to simultaneously minimize energy. We leave this extension to future work.

Problem 1: **Constrained-Jitter Voltage Scheduling-** Given a set of independent, fixed-priority, periodic tasks $\mathcal{T}$ and a corresponding set of jitter constraints $\mathcal{JC}$. Find a voltage schedule $\mathcal{SCH}$ such that all constraints in $\mathcal{JC}$ are met and no deadline in $\mathcal{T}$ is missed, or determine that it is impossible to do so.

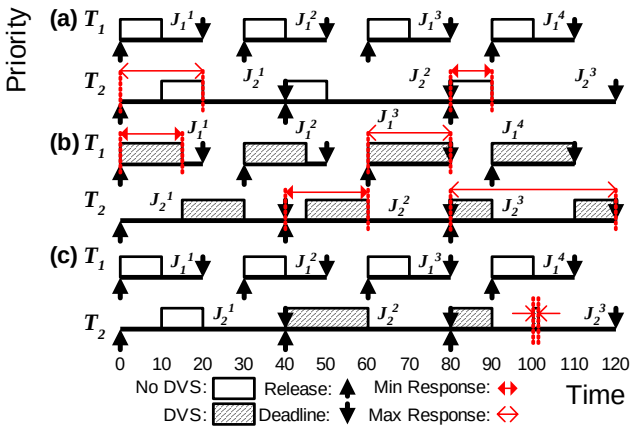

Fig. 1. An example two-task system: (a) Executing at the maximum processor speed, (b) scheduled using DVS for minimum energy, (c) scheduled using DVS for minimum jitter.

III. Proposed Method

There are several implications resulting from the focus on a fixed-priority system. First, in a fixed-priority system, lower-priority jobs cannot impact the execution of higher-priority jobs. Thus it makes sense to examine the tasks individually, in a decreasing order of priority. Next, it is important to understand exactly how DVS can alleviate the jitter problem. By its very nature, DVS is only capable of introducing delay, i.e., the processor speed is reduced to extend the execution time of a job. The result is that an individual job may end any time during its active interval (i.e., [release, deadline]). However, a job cannot end during an interval in which a higher priority job is being executed. This segments the active interval of a job into feasible completion regions, which we refer to as islands. Identifying the island in which a job should be completed is one critical step in solving Problem 1. Although many combinations of end times may meet the jitter constraints for an individual task, the earliest set should be selected to maximize the chance that lower priority tasks will also meet their jitter constraints.

Definition 3: An **island** $I(i, k, j)$ of job J_i^k refers to the j-th interval in time during which the worst case execution cycles of J_i^k can be completed, without missing d_i^k. The start and end of an island I is denoted by $\mathsf{start}(I)$ and $\mathsf{end}(I)$. The set of all such islands for jobs in $\mathcal{J}_i$ is $\mathcal{I}_i$, and for job J_i^k is $\mathcal{I}_i^k$. Island sets are sorted by their start time in a monotonically increasing order.

For the highest priority task, there is always a single island per job. For all lower priority tasks, there may be one or more islands per job, depending on the preemption pattern of higher priority jobs. This point is illustrated in Figure 2(a). Notice that job J_2^2 has two islands, i.e., $I(2, 2, 1)$ and $I(2, 2, 2)$, resulting from the possible preemption of J_2^2 by J_1^3. If J_2^2 was not completed by this preemption, then there must be at least one execution cycle left. This is accounted for by delaying the island $I(2, 2, 2)$ by one time step.

When comparing islands of the same task, it is sometimes helpful to draw a Relative-time Island Diagram (RID). A RID is constructed by first subtracting the release time r_i^k from the start and end times of all islands $I(i, k, j)$. Next, all islands are drawn along the x-axis, with a height on the y-axis corresponding to their job index k. Figure 2(b) gives the RID for T_1, and Figure 2(c) gives the RID for T_2. Although Figure 2 shows the locations of the islands of T_2, they cannot actually be identified until the end time of every job of T_1 is fixed.

The island-identifying algorithm, denoted *Identify_Islands*, finds the islands for jobs at a particular task level i given the end times of all higher priority jobs. Essentially, all idle times for each job of task T_i are identified, and the early idle times of each job are "filled up" with the worst case execution cycles of the corresponding job. For brevity, the pseudocode is omitted. Once the set of islands have been identified for task T_i, the next step is to identify a set of end times for each job of T_i that meets the jitter constraints in $\mathcal{JC}_i$.

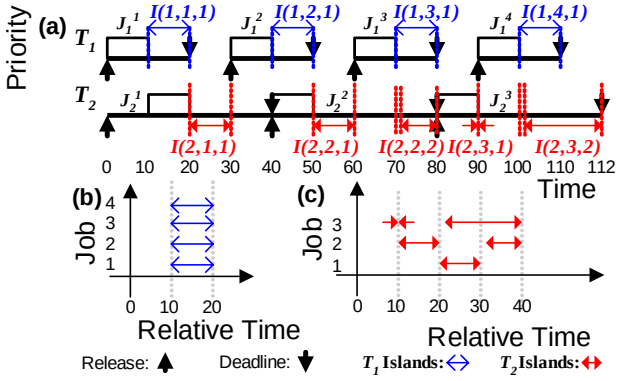

Fig. 2. (a) An Island example for a two-task system. (b) The RID of T_1. (c) The RID of T_2

The first step in assigning job end times is to identify a set of islands that will allow both the absolute jitter and inter-completion jitter constraints to be met. To help formalize the discussion, the concept of an island solution set is introduced in Definition 4.

Definition 4: **Island solution set-** A set of islands of task T_i, denoted by $\mathcal{IS}_i = \{IS_i^k\}$, $k = 1..|\mathcal{J}_i|$, one for each job $J_i^k \in \mathcal{J}_i$. An island solution set is said to be valid if there exists an end time e_i^k such that $\mathsf{start}(IS_i^k) \leq e_i^k \leq \mathsf{end}(IS_i^k)$ for $k = 1..|\mathcal{J}_i|$ and all jitter constraints in $\mathcal{JC}_i$ are met.

The goal of meeting the jitter constraints of a single task can be achieved if one can identify a valid island solution set for the task. How to identify this set becomes a key issue. A convenient method would scan the islands from the earliest to the latest, as ending early preserves slack time for lower priority tasks. Lemma 2 provides the basis on which a linear search method can be constructed.

Lemma 2: Consider a task, T_i, with corresponding jobs, $\mathcal{J}_i$, jitter constraints, $\mathcal{JC}_i$, and an arbitrary island solution set, $\mathcal{IS}_i$, from the set of possible islands, $\mathcal{I}_i$. Assume that any island $I \in \mathcal{I}_i^k$ that satisfies $\mathsf{end}(I) < \mathsf{start}(IS_i^k)$ for $k = 1..|\mathcal{J}_i|$ does not belong to any valid island solution set of T_i. Further, let $\mathcal{E}$ be the set of all end times e_i^k such that when the absolute (resp., inter-completion) jitter of T_i is minimized, $e_i^k = \mathsf{end}(IS_i^k)$, and reducing e_i^k will result in a larger jitter. If $\mathcal{IS}_i$ is not valid, then **(i)** $\mathcal{E} \neq \emptyset$ and **(ii)** all islands IS_i^k corresponding to $e_i^k \in \mathcal{E}$ do not belong to any valid island solution set of T_i.

Putting the complicated notation aside, Lemma 2 affirms that if e_i^k cannot both be inside IS_i^k and meet the given jitter constraints, then IS_i^k will never belong to any valid island solution set and can be removed from consideration.

Based on Lemma 2, a solution to Problem 1 begins to emerge. This method requires the repeated testing of island solution sets, beginning with the earliest set of islands. The solution set then hops from island to island, moving forward in time until a valid set is found. One difficulty with this approach is that Lemma 2 requires the jitter to be minimized for each island solution set. However, for the method presented here, it is sufficient to determine if the minimum jitter is less than or equal to the given jitter constraints. To facilitate the minimization process, the concepts of maximum island start and minimum island end times are introduced.

Definition 5: The **maximum relative island start time** of an island solution set, denoted $s_{max}(i)$, is given by:

$$s_{max}(i) = \max(\mathsf{start}(IS_i^k) - r_i^k), \quad k = 1..|\mathcal{IS}_i| \tag{3}$$

Definition 6: The **minimum relative island end time** of an island solution set, denoted $e_{min}(i)$, is given by:

$$e_{min}(i) = \min(\mathsf{end}(IS_i^k) - r_i^k), \quad k = 1..|\mathcal{IS}_i| \tag{4}$$

Definitions 5 and 6 can be used to directly compute the minimum absolute jitter, as shown in the following lemma.

Lemma 3: The minimum absolute jitter for an island solution set is given directly by the following equation:

$$\min(aJtr(\mathcal{IS}_i)) = \max(s_{max}(i) - e_{min}(i), 0) \tag{5}$$

If the minimum absolute jitter according to Equation 5 is greater than the absolute jitter constraint, then the job corresponding to e_{min}(i) belongs in set $\mathcal{E}$ according to Lemma 2, and can be replaced by the next island for that job. However, the method for determining which islands belong in $\mathcal{E}$ based on the inter-completion jitter constraint is more involved.

The unified method for identifying the validity of an island solution set is given in Algorithm 1. The algorithm is divided into two general phases, the first phase (Lines 3–5) checks and enforces the absolute jitter constraint, while the second phase (Lines 6–16) checks and enforces the inter- completion jitter constraint. Line 3 uses Equation 5 to identify the minimum absolute jitter and compare it with the absolute jitter constraint. Lines 4 and 5 duplicate the island solution set, convert it to relative time, and enforce the absolute jitter constraint. Line 6 uses Lemma 1 to skip the second phase if the inter-completion jitter constraint is greater than or equal to the absolute jitter constraint (i.e., the inter-completion jitter constraint is met automatically). Starting with the island after the one corresponding to $s_{max}(i)$, the previous island, $IS2_i^p$, is extended by the inter-completion jitter constraint and the intersection of the two intervals is computed (Line 9). Intuitively, the intersection represents every possible end time of J_i^k in the island IS_i^k that satisfies JC_i^{ic} with respect to J_i^p. The loop continues until the island of $s_{max}(i)$ is reached again.

Once it has been determined that the given island solution set is valid, the final consideration for the task at the current level is where the end times of each job should be placed. Lemma 4 shows that this occurs when the end times are placed as early as possible. Lines 15 and 17 serve this purpose by repeating the intersection process from the job of s_{max}(i), but proceeding instead in the opposite direction. The resulting islands in $IS2_i$ represent the intervals in which the end times of each job may be placed that will satisfy the inter-completion jitter constraint from both adjacent jobs, as well as satisfying the absolute jitter constraint. The minimum end time is simply the start of that island (plus the job release to transform the end time back from relative time). Theorem 1 states the correctness and complexity of Algorithm 1.

Lemma 4: Given a task, $T_i \in \mathcal{T}$, with jobs, $\mathcal{J}_i$, jitter constraints, $\mathcal{JC}_i$, and a valid island solution set, $\mathcal{IS}_i$, from the set of possible islands, $\mathcal{I}_i$. Assume that all islands in I_i^k less than the corresponding island IS_i^k for $k = 1..|\mathcal{J}_i|$ do not belong to any valid island solution set of T_i. If $\mathcal{T}$ is schedulable under $\mathcal{JC}$ given the current selection of end times for jobs in $\mathcal{J}_{hp(i)}$, then it is a sufficient (but not necessary) condition of a valid schedule that the end time of each job in $\mathcal{J}_i$ is minimized while still meeting the constraints in $\mathcal{JC}_i$.

Theorem 1: Algorithm 1 identifies one island in the island solution set that is not part of any valid island solution set, or determines that the island solution set is valid and minimizes the end times of jobs in $\mathcal{J}_i$, while meeting all jitter and deadline constraints. Algorithm 1 requires $\mathbf{O}(|\mathcal{J}_i|)$ time.

Using Algorithm 1, determining a schedule that meets all deadlines and jitter constraints is a relatively straightforward process. The process, denoted *Hop_Islands*, begins by identifying the islands and a valid island solution set with the corresponding end times for each task, in a decreasing order of priority. If a valid solution set exists for every task, then a voltage schedule is constructed using the islands and end times. This is done by identifying the idle time allocated to each job, J_i^k , which is equivalent to the sum of all island intervals in the range $[r_i^k, e_i^k]$. The speed for the J_i^k is then scaled according to the available idle time using $S_i^k = \frac{wc_i}{wc_i + idle}$. Theorem 2 states the correctness and complexity of this method.

Theorem 2: *Hop_Islands* solves Problem 1 in $\mathbf{O}(|\mathcal{T}| \times |\mathcal{J}|^2)$ time.

IV. Experimental Results

This section describes the experiments used to evaluate the effectiveness of the proposed method with respect to meeting jitter constraints. The processor was assumed to have continuous frequency levels available in the range of 10 to 100 MHz and negligible transition overhead.

Algorithm 1 $[I]$ = Is_Valid($\mathcal{J}_i$, $\mathcal{IS}_i$, $\mathcal{JC}_i$)

1: **INPUT:** The set of jobs $\mathcal{J}_i$, the island solution set $\mathcal{IS}_i$, and the jitter constraints $\mathcal{JC}_i$, of task T_i;
2: **OUTPUT:** I, an island in $\mathcal{IS}_i$ that is not part of any valid solution st of T_i, or $\emptyset$ if the island solution set is valid. *** If the solution set is valid, the end times of jobs in J_i are minimized;
3: **if** $\max(s_{max}(i) - e_{min}(i), 0) > JC_i^{ic}$ **then return** the island corresponding to $e_{min}(i)$;
4: $\mathcal{IS}2_i := \mathcal{IS}_i$;
5: Transform each island in $\mathcal{IS}2_i$ to relative time, and restrict relative start and end times to the range $[s_{max}(i) - JC_i^a$, $s_{max}(i)]$;
6: **if** $JC_i^a > JC_i^{ic}$ **then**
7: IS_i^{max} := the island corresponding to s_{max}(i);
8: **for** each island, starting from the island after IS_i^{max}, and looping back around until IS_i^{max} is reached **do**
9: $IS2_i^k := IS2_i^k \bigcap [\mathsf{start}(IS2_i^p) + JC_i^{ic}, \mathsf{end}(IS2_i^p) - JC_i^{ic}]$;
10: **if** $IS2_i^k = \emptyset$ **then**
11: **if** $\mathsf{end}(IS_i^k) - r_i^k < \mathsf{start}(IS_i^p) - r_i^q$ **then return** IS_i^k;
12: **else return** IS_i^p;
13: **end if**
14: **end for**
15: ***Repeat the intersection process for each island, looping in the reverse direction;
16: **end if**
17: ***for** every job J_i^k **do** set $e_i^k := \mathsf{start}(IS2_i^k) + r_i^k$;
18: **return** $\emptyset$;

The simulation was conducted on randomly generated task sets of 5 to 20 tasks with periods from 1 to 100 ms. The utilization was varied from 0.3 to 0.7 in order to gage the effect of an increased load on the results. The jitter was set for every task as a percentage of the period, 2% for a highly constrained jitter and 10% for a less constrained jitter.

Figures 3 and 4 show the effectiveness of the proposed method, called Jitter Aware DVS (JADVS) and rate-monotonic (RM) scheduling on absolute and inter-completion jitter, respectively. As expected, the general trend is that it becomes more difficult to meet jitter constraints as the utilization and the number of tasks increase. With a restrictive absolute jitter (Figures 3(a) and (b)) it is easy to see that JADVS wins out over RM. For example, with 5 tasks and a utilization of 0.3, JADVS can schedule 6 times as many task sets as RM. At higher utilizations RM cannot schedule any tasks. JADVS can schedule as many as 20% of task sets consisting of only 5 tasks even at high utilizations (0.7). With a less restrictive jitter (Figure 3(c) and (d)) both algorithms improve, but JADVS maintains a 40 to 60% improvement for 5 tasks, and can even schedule 70 to 95% of task sets of 20 tasks at utilizations of 0.4 and 0.3 respectively.

The story is very similar for inter-completion jitter (Figure 4). The difference is that inter-completion jitter only restricts adjacent tasks, so it is less restrictive than absolute completion jitter. RM does not benefit from the less restrictive nature of inter-completion jitter (Figure 4(b) and (d)) because in cases where tasks are released simultaneously, the best case response time happens directly after the worst case response time, making the absolute and inter-completion jitters equal. While applying DVS, however, this is not the case. This is particularly evident from the 15 and 20 task curves with 10% jitter (Figures 3(c) and 4(c)). Notice that a utilization of 0.5, JADVS can schedule 40% of sets with 15 tasks under inter-completion constraints, but only 20% under absolute jitter constraints, a 2× difference.

V. Summary

We have shown through simulations and detailed examples that DVS is a viable option for meeting absolute and inter-completion jitter constraints. This method can be readily incorporated into the SymTA/S framework to manage both jitter and energy for a distributed real-time system. However, much work remains. First, the effectiveness of this method on a larger range of systems should be explored and compared with previous methods of jitter management. Second, issues such as input jitter, variation in execution time, bursty

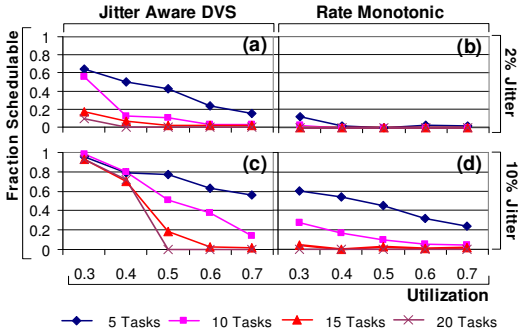

Fig. 3. Percentage of randomly generated task sets schedulable with various absolute jitter constraints using JADVS (a, c) and RM (b, d).

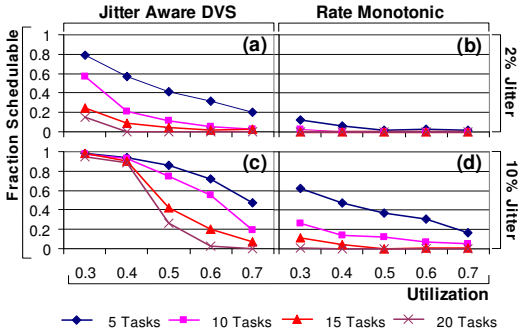

Fig. 4. Percentage of randomly generated task sets schedulable with various inter-completion jitter constraints using JADVS (a, c) and RM (b, d).

behavior, sporadic tasks, and voltage transition overhead must be considered for this method to be useful in practice.

References

[1] S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari. Scheduling periodic task systems to minimize output jitter. In *Proceedings of the International Conference on Real-Time Computing Systems and Applications*, pages 62–69, Dec. 1999.

[2] I. Bate and A. Burns. An approach to task attribute assignment for uniprocessor systems. In *In The 11th Euromicro Conference on Real-Time Systems (ECRTS99)*, pages 46–53, 1999.

[3] A. Cervin. Improved scheduling of control tasks. In *the 11th Euromicro Conference on Real-Time Systems (ECRTS99)*, pages 4–10, 1999.

[4] L. David, F. Cottet, , and N. Nissanke. Jitter control in on-line scheduling of dependent real-time tasks. In *Proceedings of the 22nd Real-Time Systems Symposium (RTSS)*, pages 49–58, Dec. 2001.

[5] T. Kim, H. Shin, and N. Chang. Deadline assignment to reduce output jitter of real-time tasks. In *Proceedings of the 16th IFAC workshop on Distributed Computer Control Systems*, pages 67–72, Nov. 2000.

[6] W. Kim, J. Kim, and S. L. Min. Dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In *Proceedings of the 2002 Design, Automation and Test in Euroope Conference and Exhibition (DATE)*, pages 788–794, Mar. 2002.

[7] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, Jan. 1973.

[8] P. Marti, J. M. Fuertes, G. Fohler, and K. Ramamritham. Jitter compensation for real-time control systems. In *Proceedings of the 22nd Real-Time Systems Symposium (RTSS)*, pages 39–48, Dec. 2001.

[9] B. C. Mochocki. Dynamic voltage scaling for the schedulability of jitter-constraind real-time embedded systems: Supplementary material. Technical Report TR-2005-12, University of Notre Dame: online-http://cse.nd.edu/research/tech_reports/, July 2005.

[10] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP)*, pages 89–102, 2001.

[11] K. Richter, R. Racu, and R. Ernst. Scheduling analysis integration for heterogeneous multiprocessor soc. In *Proceedings of the 24nd Real-Time Systems Symposium (RTSS)*, Dec. 2003.

[12] Symta/s - symbolic timing analysis for systems. online-http://www.symta.org/.

[13] K. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, Nov. 1994.