Generating Utilization Vectors for the Systematic Evaluation of Schedulability Tests

David Griffin, Iain Bate, Robert I. Davis
Department of Computer Science, University of York, United Kingdom.
Email: {david.griffin, iain.bate, rob.davis}@york.ac.uk

***Abstract*—This paper introduces the Dirichlet-Rescale (DRS) algorithm. The DRS algorithm provides an efficient general-purpose method of generating n-dimensional vectors of components (e.g. task utilizations), where the components sum to a specified total, each component conforms to individual constraints on the maximum and minimum values that it can take, and the vectors are uniformly distributed over the valid region of the domain of all possible vectors, bounded by the constraints.**

The DRS algorithm can be used to improve the nuance and quality of empirical studies into the effectiveness of schedulability tests for real-time systems; potentially making them more realistic, and leading to new conclusions. It is efficient enough for use in large-scale studies where millions of task sets need to be generated. Further, the constraints on individual task utilizations can be used for fine-grained control of task set parameters enabling more detailed exploration of schedulability test behavior. Finally, the real power of the algorithm lies in the fact that it can be applied recursively, with one vector acting as a set of constraints for the next. This is particularly useful in task set generation for mixed criticality systems and multi-core systems, where task utilizations are either multi-valued or can be decomposed into multiple constituent parts.

I. Introduction

The research presented in this paper supports the empirical evaluation of schedulability tests for real-time systems. Two aspects are important here:

1) *Effectiveness* – measured in terms of how many task sets a particular schedulability test deems to be schedulable.
2) *Tractability* – measured in terms of how long the schedulability test takes to run, and how its runtime scales with increasing numbers of tasks.

Often there is a tradeoff between effectiveness and tractability. Exact tests may be intractable for relatively small numbers of tasks, whereas simple sufficient tests may be fast and work for large numbers of tasks, but not be particularly effective, classifying many task sets as unschedulable when they are not.

In this paper, we focus on the issue of *effectiveness*. A systematic and scientific study of the effectiveness of different schedulability tests requires a method of synthesising large numbers of task sets to which the tests can be applied. Further, the different task parameter settings used must cover in an unbiased way, the range of possible task sets that could potentially occur in practice.

Underpinning task set generation is the problem of generating individual task *utilization* values. The utilization U_i of a task τ_i is defined as the maximum bandwidth of a processing resource that the task can use. Thus, in the traditional sporadic task model, the utilization of a task is directly related to its maximum execution time C_i and its period or minimal inter-arrival time T_i, with $U_i = C_i/T_i$. The total utilization U of a task set is given by the sum of the individual task utilizations $U = \sum_{i=1}^{n} U_i$, where n is the number of tasks. Note that the total utilization cannot exceed the bandwidth that can be supplied by the resource, e.g. $U \leq 1$ for a single processor system, and $U \leq m$ for a homogeneous m processor system, otherwise the task set will necessarily be unschedulable. Further, each individual task utilization cannot exceed the bandwidth that can be supplied by a single processor, otherwise that task alone would be not be schedulable[1].

As noted by Davis [10], it is important that task set generation is done in a way that does not confound variables. An often employed technique that suffers from this pitfall involves constructing task sets by repeatedly adding a task until the desired utilization level is reached. The result, for each utilization level, is a mix of task sets with different cardinalities. This approach confounds task set cardinality and utilization, resulting in a strong correlation between the two, as tasks sets with low utilization have on average fewer tasks than those with higher utilization. This makes it impossible to determine if some aspect of schedulability test performance is impacted by the number of tasks or by the utilization level.

The two primary inputs to an algorithm that generates utilization values are the task set cardinality n and the total utilization required U. The output is a vector $\mathbf{u}$, containing n task utilization values U_i that sum to the desired total U. (Note, as is customary, we use a **bold** font to denote vectors).

There are three key requirements for any general-purpose algorithm that generates task utilization values:

1) *Efficiency* – The algorithm must be fast. To achieve statistically significant sample sizes across a wide-ranging systematic evaluation it is necessary to generate millions of task sets, with typically 1000 task sets required per data point [10].
2) *Uniformity* – The generated vectors of utilization values must be unbiased i.e. uniformly distributed within the domain. This is equivalent to selecting the utilization values for each vector at random from a uniform distribution in the range $[0, 1]$ and then discarding those vectors that do not match the constraints [13]. As shown by Bini and Buttazzo [5], bias in the sets of vectors generated can undermine the conclusions drawn from empirical studies.
3) *Flexibility* – The algorithm should be capable of handling constraints on individual utilization values for each of the n tasks, This enables the utilization values to be tailored to the specific requirements of the problem at hand (examples are given later), while still producing an unbiased distribution of vectors, i.e. uniformly distributed over the valid region determined by the constraints.

Existing approaches to the generation of unbiased task utilization vectors: UUnifast [5], UUnifast-Discard [13], and

[1] Assuming that parallel execution of a single task is not permitted.

RandFixedSum [16] provide only limited support for the three key requirements set out above.

The UUnifast method, introduced by Bini and Buttazzo in 2005 [5], is efficient, but takes no explicit constraints. Thus UUnifast(n, U) is a function of two parameters, with the constraint vectors effectively fixed at $\mathbf{u^{max}} = (U, U, \ldots, U)$ and $\mathbf{u^{min}} = \mathbf{u^0} = (0, 0, \ldots 0)$. As a consequence, UUnifast only provides valid output vectors when $U \leq 1$, i.e. for the single processor systems for which it was designed. We note that a naïve and inappropriate use of UUnifast for multiprocessor systems ($m > 1$) with values of $U > 1$, would produce vectors where some individual task utilizations are invalid ($U_i > 1$).

The problem with applying UUnifast to multiprocessor systems was partly addressed via the *UUnifast-Discard* method introduced by Davis and Burns in 2010 [13]. UUnifast-Discard employs UUnifast to generate vectors of utilization values, but then applies the constraint that $\mathbf{u^{max}} = \mathbf{u^1} = (1, 1, \ldots 1)$, and thus discards any vectors where some individual task utilization is invalid. Davis and Burns [13] showed that UUnifast-Discard is effective provided that there are more than approximately 2 tasks per processor. Below this level, for example from $n = 16$ to $n = 9$ tasks on an 8 processor system, with $U = 8$, the number of discards required to generate a single valid utilization vector quickly becomes very large, making the approach intractable. We observe that the basic idea of UUnifast-Discard can easily be adapted to apply vectors of maximum and minimum constraints, $\mathbf{u^{max}} = (U_1^{max}, U_2^{max}, \ldots, U_n^{max})$ and $\mathbf{u^{min}} = (U_1^{min}, U_2^{min}, \ldots, U_n^{min})$ on individual task utilizations, rather than simply $\mathbf{u^{max}} = \mathbf{u^1}$; however, tighter constraints would only increase the number of discards, exacerbating the tractability issues.

The RandFixedSum method, invented by Stafford [30] in 2006, and adapted to the problem of generating task utilization values by Emberson et al. [16] in 2010, provides an efficient solution for multiprocessor systems for any valid combination of n and U. (An open source Python implementation by Emberson et al. is available online[2]). RandFixedSum applies symmetric constraints[3], u^{max} and u^{min} on the maximum and minimum individual task utilizations that are applied to all tasks, typically $u^{max} = 1$ and $u^{min} = 0$. Unfortunately, the way in which RandFixedSum makes use of symmetric constraints means that adapting it to the more general case of non-symmetrical constraints would make the algorithm intractable, as explained in Section II-C.

The main contribution of this paper is the introduction of a general-purpose algorithm for generating task utilization values, called the Dirichlet-Rescale algorithm (DRS). The algorithm has the following signature:

$$\mathbf{u} = \text{DRS}(n, U, \mathbf{u^{max}}, \mathbf{u^{min}}) \tag{1}$$

where $\mathbf{u} = (U_1, U_2, \ldots, U_n)$ is the output vector of task utilization values, n is the cardinality of the task set, U is the total utilization required, and $\mathbf{u^{max}} = (U_1^{max}, U_2^{max}, \ldots, U_n^{max})$ and $\mathbf{u^{min}} = (U_1^{min}, U_2^{min}, \ldots, U_n^{min})$ are vectors of constraints indicating the maximum and minimum value that each task utilization may take. Note, $\mathbf{u^{max}}$ and $\mathbf{u^{min}}$ are optional arguments, which take the values $\mathbf{u^{max}} = \mathbf{u^1}$ and $\mathbf{u^{min}} = \mathbf{u^0}$ if not explicitly specified.

[2].See https://github.com/MaximeCheramy/simso/blob/master/simso/generator/task_generator.py

[3]Constraints are referred to as *symmetric* if they all take the same value.

The DRS algorithm meets all three key requirements for generating vectors of task utilization values: (i) efficiency, (ii) uniformity, and (iii) flexibility. It is efficient enough for use in large-scale empirical studies that need to generate millions of task sets, generates an unbiased distribution of utilization vectors, and can work with any valid set of constraints on individual task utilizations, where $\sum_{i=1}^{n} U_i^{max} \leq U \geq \sum_{i=1}^{n} U_i^{min}$ and $\forall i\ U_i^{max} \geq U_i^{min} \geq 0$. The constraints on individual task utilizations can be used for fine grained control of task set parameters, for example constraining some number of the tasks to be *small*, i.e. with $U_i \leq 0.5$. However, the real power of the algorithm, lies in the fact that it can be applied recursively on problems where task utilizations are multi-valued or can be decomposed into multiple constituent parts. Examples include modelling mixed criticality systems [8], multi-core systems [23], typical and worst-case execution times [2,26], self-suspensions [9], and resource locking.

Real-time systems have a variety of different distributions of task utilizations, with different distributions representative of different systems, and no single distribution representative of them all. For generic schedulability analysis experiments, using a uniform distribution of utilization vectors means that each possible vector that complies with the constraints has the same chance of being selected. Thus, the distribution is unbiased, provides full and fair coverage of the region of all valid possibilities, and is thus an appropriate one to use.

A. Motivation

The motivation for our work on the Dirichlet-Rescale algorithm comes from two fields that have continued to be hot topics of real-time systems research over the past 10 years: mixed criticality systems [8] and multi-core systems [23].

Mixed Criticality Systems: From a scheduling and analysis perspective, the key defining characteristic of Mixed Criticality Systems (MCS) is that at least one of the task parameters is not single valued, but rather two or more values (e.g. $C_i(LO)$ and $C_i(HI)$) need to be considered, corresponding to different *criticality levels* or behaviors $\{HI, LO\}$. Further, the timing constraints that the tasks must comply with depend on the *criticality mode* of the system. As noted in a survey of MCS research [8], these properties significantly undermine many of the standard scheduling results that were developed for single criticality systems. We note that they also undermine, or at least require consideration of extensions to, the techniques that are used in the evaluation of schedulability test effectiveness.

As an example, consider the approach taken to generating $C_i(LO)$ and $C_i(HI)$ values by Baruah et al. [3] in the paper on Adaptive Mixed Criticality (AMC) scheduling, and subsequently used in many related works [6,7,11,18]. Baruah et al. generated $U_i(LO)$ values using UUnifast, and then set $C_i(LO) = U_i(LO) \cdot T_i$ and $C_i(HI) = CF \cdot C_i(LO)$, where the Criticality Factor (CF), is a *fixed multiplier*. Hence $U_i(HI) = CF \cdot U_i(LO)$. This represents a small and somewhat unrealistic part of the domain of all possible MCS task sets where the HI and LO criticality execution times are in a fixed ratio. The authors seek to address this drawback by varying the value of CF and plotting the weighted schedulability measure [4] as a function of CF. Even so, there remains a perfect correlation between $C_i(HI)$ and $C_i(LO)$ and hence between $U_i(HI)$ and $U_i(LO)$.

Other authors, for example Ekberg and Yi [14,15], select values for $C_i(LO)$ and $C_i(HI)$ as uniform random variables in

the ranges $[1, C^{max}(LO)]$ and $[C_i(LO), CF \cdot C_i(LO)]$ respectively. While this provides additional variation for $C_i(HI)$, the utilization vectors produced are *not* uniformly distributed, as illustrated in Section II-B.

In MCS, there is a constraint that $\forall i\,(C_i(HI) \geq C_i(LO))$ and hence $\forall i\,(U_i(HI) \geq U_i(LO))$. Hence, what is required is a means of generating vectors of $U_i(HI)$ and $U_i(LO)$ values that meet those constraints, and where the individual utilizations sum to specified totals U^{HI} and U^{LO} respectively. The Dirichlet-Rescale algorithm introduced in this paper enables vectors of utilization values to be generated that address this problem, using an iterative approach. First, the vector $\mathbf{u}(HI)$ of $U_i(HI)$ values that sum to U^{HI} can be generated with the only constraints being that $\forall i\,(U_i(HI) \leq 1)$:

$$\mathbf{u}(HI) = \text{DRS}(n, U^{HI}, \mathbf{u^1}) \tag{2}$$

Then the constraint vector can be set to $\mathbf{u}(HI)$ and a vector $\mathbf{u}(LO)$ of $U_i(LO)$ values produced that sum to U^{LO} $(< U^{HI})$ via a second iteration of the algorithm:

$$\mathbf{u}(LO) = \text{DRS}(n, U^{LO}, \mathbf{u}(HI)) \tag{3}$$

Thus vectors of $U_i(HI)$ and $U_i(LO)$ values are obtained that comply with the constraints on both total utilizations U^{HI} and U^{LO}, ensure that the intra-task constraints $\forall i\,(U_i(HI) \geq U_i(LO))$ are met, and importantly are uniformly distributed over the domain.

Note that while this example considers only two criticality levels, the approach can easily be extended to an arbitrary number of levels, via additional calls to the Dirichlet-Rescale algorithm with the appropriate parameters.

Multi-core Systems: From a scheduling and analysis perspective, the execution time of a task in a multi-core system can be broken down into multiple constituent parts, corresponding to different shared resources that are accessed. For example, Altmeyer et al. [1,12] consider both the processor demand PD_i and the memory demand[4] MD_i, thus the overall task execution time, not including cross-core interference, is given by $C_i = PD_i + MD_i$. The processor (core) utilization of each task is given by $U_i = C_i/T_i$, and the bus utilization by $U_i^{BUS} = MD_i/T_i$. Note that $\forall i\, U_i > U_i^{BUS}$.

In the task set generation process used in their evaluation, Altmeyer et al. [1,12] randomly selected a trace from the Mäladarlen benchmark suite [21] to represent each task and define its pair of PD_i and MD_i values. They then generated utilization values (either U_i or U_i^{BUS}) via UUnifast, and scaled the task periods to achieve the desired total processor utilization or the desired total bus utilization, given the sets of values of PD_i and MD_i selected. While this approach provides a reasonable basis for evaluation, it is unable to control for both processor and bus utilization at the same time. It also fixes the relationship between PD_i and MD_i based on a limited number of examples from the benchmark suite. (A simpler scheme representing PD_i and MD_i as fixed proportions of C_i would be unrealistic here, since task behavior varies from compute-intensive to memory-intensive, as shown by the data from the benchmark suite, see Table 2 in [12]).

The Dirichlet-Rescale algorithm, introduced in this paper, enables task sets to be generated that address these problems. Both total processor and total bus utilization can be controlled for independently, while also ensuring that the distribution of the pairs of vectors of processor and bus utilizations are uniformly distributed over the domain of possible values that they could take. As an example, suppose we have a multi-core system with 4 cores, and require a task set that has a core utilization U^{CORE} of 2.8 and a bus utilization of U^{BUS} of 0.8. The DRS algorithm can be first used to produce the required vector $\mathbf{u^{CORE}}$ of processor utilizations values:

$$\mathbf{u^{CORE}} = \text{DRS}(n, U^{CORE}, \mathbf{u^1}) \tag{4}$$

The constraint vector $\mathbf{u^{max}}$ can then be set to the values of $\mathbf{u^{CORE}}$, and the vector $\mathbf{u^{BUS}}$ of U_i^{BUS} bus utilization values produced that sum to $U^{BUS} (< U^{CORE})$ via a second call of the algorithm:

$$\mathbf{u^{BUS}} = \text{DRS}(n, U^{BUS}, \mathbf{u^{CORE}}) \tag{5}$$

While this example considers only processor and bus utilization, the approach can easily be extended to an arbitrary number of resources, via additional calls to the DRS algorithm with the appropriate parameters.

B. Organization

The remainder of the paper is organized as follows. Section II outlines the mathematical concepts used in the DRS algorithm, discusses some common misconceptions about generating uniformly distributed utilization vectors, and reviews related work. Section III gives an overview of the algorithm, followed by a more detailed presentation of its operation. An empirical verification is provided in Section IV, examining the efficiency and correctness of the algorithm. Section V gives an example of how it can be used to improve schedulability test evaluation for mixed criticality systems. Section VI concludes with a summary and directions for future work.

II. Background and Related Work

In this section, we first provide an introduction to the mathematical concepts relevant to design of algorithms used to generate vectors of task utilization values in an unbiased way. We then discuss some common misconceptions related to the generation of such vectors, and finally provide a more detailed discussion of the existing approaches.

A. Mathematical Concepts

A *simplex* (plural simplices) is a generalization of a triangle, extended to an arbitrary number of dimensions. More specifically, an $(n-1)$-simplex is an $(n-1)$-dimensional flat-sided shape (or *polytope*) formed by the convex hull of its n vertices $\mathbf{V_i}$. In this paper, we are interested in $(n-1)$-dimensional simplices embedded in an n-dimensional space. To aid visualization of the concepts, we make use of examples of 2-simplices (triangles) embedded in a 3-dimensional space. (Note, here we deliberately overload n to mean both the number of dimensions and the number of tasks, since, for the purposes of this paper, the former derives from the latter). The vertices of the simplices that we consider are assumed to be *affinely independent*, meaning that no three vertices lie on a line. A simplex is *non-degenerate* if the vectors $(\forall j \neq i\ \mathbf{V_i} - \mathbf{V_j})$ are all independent, and hence cannot be constructed from a linear combination of each other. A simplex is referred to as *regular* if all of its edges are the same length.

[4]For simplicity, we have scaled MD_i to account for the bus access latency, whereas [1,12] includes this multiplier in the equation for C_i.

A simplex is referred to as *standard* if it is formed from the standard unit vectors.

An *Affine transformation* can be represented by a combination of a linear transformation and a translation. Affine transformations preserve a number of interesting properties following transformation [32]: (i) *co-linearity*: any set of points that are in a line remain in a line; (ii) *parallelism*: lines that are parallel remain parallel; (iii) *convexity*: a convex set of points remains convex; (iv) *ratios*: the ratio between the lengths of any two parallel line segments remains the same; and (v) *uniformity*: a uniform distribution of points remains uniform after transformation.

Triangles are *affine*, meaning that any triangle can be transformed into any other triangle via an affine transformation. Further, the non-degenerate simplices considered in this paper are affine. These simplices are embedded in n-dimensional space on the hyperplane given by $x_1 + x_2 + x_3 \ldots x_n = c$, where c is a constant. Thus they span $\mathbb{R}^n$. As a consequence, we can write any coordinate of $\mathbb{R}^n$ as a linear combination of the simplex coordinates, and vice versa. The translation between two sets of basis coordinates in this way is both isomorphous and Affine, as it can be written as a transformation matrix, constructed using the translate-scale-translate method.

While there are many ways to compute the volume of a simplex, there are only a limited number of methods that work when the simplex is embedded in a higher dimensional space. Hence for our purposes, the *Cayley-Menger* determinant method [22,24] is used, which computes the volume of an arbitrary $(n-1)$-dimensional simplex in n-dimensional space.

B. Common Misconceptions

In this section, we dispel a number of naïve conclusions and misconceptions regarding the generation of uniformly distributed utilization vectors. Each of these misconceptions may lead to the generation of vectors that are non-uniform, and hence potentially skewed results from their subsequent use in the evaluation of scheduling algorithms and analyses.

Misconception #1: The utilization values (i.e. the components of the vectors) themselves form a uniform distribution

In the context of generating vectors of task utilization values, the term *unbiased* or *uniformly distributed* means that the random vectors generated should be uniformly distributed over the *domain* of all possible vectors that are valid with respect to the constraints. This does not, however, imply that the utilization values will follow a uniform distribution.

A uniform distribution of vectors over the domain of interest is illustrated in Figure 1 for 1000 runs of UUnifast$(3, 1)$, i.e. for 3 tasks and a total required utilization of 1. In Figure 1, the (x, y, z) co-ordinates of each point correspond to the three utilization values of the vector that it represents. The large triangle shows the boundary of the domain of valid vectors, with all points on the plane within that triangle meeting the constraint $x + y + z = 1$. Points on the boundary represent vectors where one of the x, y, or z co-ordinates is zero, while the vertices are the points where two of these values are zero, and hence the other is 1. The points are color-coded: red indicates there is a co-ordinate value in the range $(0.6, 0.8]$ and blue where there is a value in the range $(0.8, 1.0]$. Notice that the size of regions where vectors have a value in the range $(0.6, 0.8]$ is much larger than the size of the regions for values in the range $(0.8, 1.0]$. The vectors are uniformly distributed within the domain; however, due to the way that

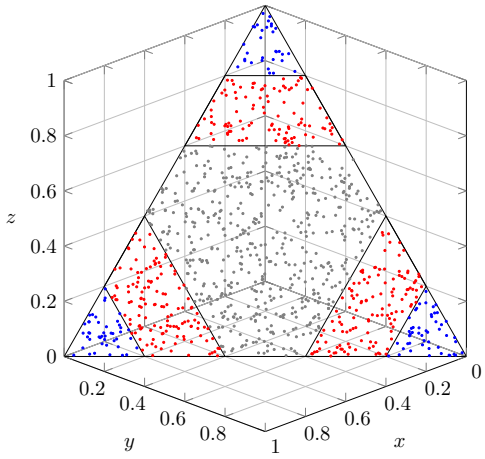

Fig. 1. The output of 1000 runs of UUnifast$(3, 1)$.

the constraint $x + y + z = 1$ shapes the domain, the individual utilization values within those vectors *do not* follow a uniform distribution. There are fewer values in the range $(0.8, 1.0]$ (130 values) than in the range $(0.6, 0.8]$ (361 values), since the former areas are much smaller.

Misconception #2: A uniform distribution of utilization vectors can be obtained by independently selecting individual U_i values from a uniform distribution in the range $[0, 1]$ and then scaling the values such that they sum to 1. This is the *UScale* approach discussed by Bini and Buttazzo [5]. As they noted, this method does not result in a set of vectors that are uniformly distributed over the required domain. Rather the vectors are clustered near the center where x, y, and z have similar values.

Misconception #3: Multiple related vectors of utilization values (e.g. the $\mathbf{u}(HI)$ and $\mathbf{u}(LO)$ vectors discussed in Section I-A) can be obtained by first independently generating the vector of smaller utilization values (e.g. $\mathbf{u}(LO)$) and a vector representing the differences (e.g. $\forall i\ U_i(HI) - U_i(LO)$), and then adding the two together to obtain the required vector of larger utilization values (e.g. $\mathbf{u}(HI)$). The problem with this approach, which we refer to as *UAdd*, is that the vectors produced via such addition are not uniformly distributed over the relevant domain. This is illustrated in Figure 2. Here, we called UUnifast$(3, 0.5)$ twice for each point and added the vectors together to obtain the coordinates. Notice the clustering of the vectors in the center of the domain

C. Related Work

In this section, we give a brief overview of existing methods that can be used to generate uniformly distributed vectors of utilization values.

In 1964, Olkin and Rubin [25] described the Multivariate Beta Distribution, also called the *Dirichlet distribution*, which we denote by Dir$(n, \boldsymbol{\alpha})$, where $\boldsymbol{\alpha}$ is a vector of n real numbers. The Dirichlet distribution is a generalization of the Beta distribution to multiple dimensions. A special case of the Beta distribution Beta$(1, 1)$ equates to the Uniform distribution over the interval $[0, 1]$. A special case of the Dirichlet distribution, where all elements of $\boldsymbol{\alpha}$ are 1, is called the *flat Dirichlet distribution*. This is equivalent to a uniform distribution over the standard $(n-1)$-simplex (defined by n vertices $\mathbf{V_i}$ with cartesian co-ordinates $(x_1, x_2, \ldots, x_n)$ with $x_i = 1$ and $x_{j \neq i} = 0$). We observe that the flat Dirichlet

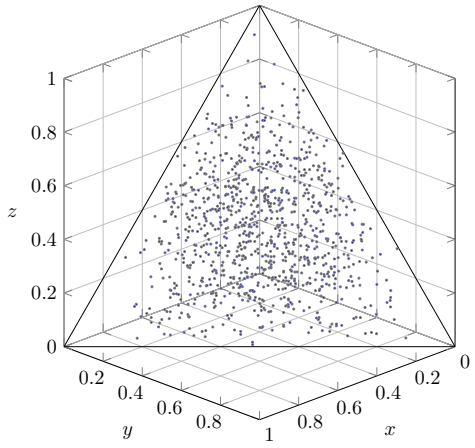

Fig. 2. The output of 1000 runs of UAdd$(3, 1)$ = UUnifast$(3, 0.5)$ + UUnifast$(3, 0.5)$.

distribution can therefore be used as a means of directly generating uniformly distributed vectors of utilization values that sum to 1. The Python code for a scaled version of the flat Dirichlet distribution, with parameters n and U, is shown below. This implementation (chosen for its efficiency), utilises random variates from the Gamma$(1, 1)$ distribution, and calculates these via inverse transform sampling[5].

```
def scaled_flat_dirichlet(n, U):
    intermediate =
    [-math.log(1 - random.random()) for _ in range(n)]
    divisor = sum(intermediate)
    return [(x/divisor)*U for x in intermediate]
```

In 2005, Bini and Buttazzo introduced the UUnifast(n, U) algorithm [5], which provides a solution to the problem of generating uniformly distributed vectors of utilization values, where the sum of the components of each vector is $U \leq 1$.

UUnifast works by iteratively sampling a value that represents the sum of $n-1, n-2, n-3, \ldots$ task utilizations. On each iteration i, the utilization for task τ_i is set to the difference between the sampled value and the remaining total utilization required. The remaining total utilization is then reduced to the sampled value. Iteration ends when there is one task left, and that task is assigned all of the remaining total utilization.

UUnifast has previously [16] been characterized as obtaining the sampled values via applying the inverse transform sampling method to obtain the random variates of the sum of $n-1$ independent uniformly distributed random variables. This description unfortunately obfuscates the true nature of the random variates used in UUnifast, which are drawn from the Beta distribution [17]. We observe that there is a direct equivalence between UUnifast and the Dirichlet distribution, since the UUnifast algorithm follows an intuitive definition of the Dirichlet distribution [17], and functions by drawing random variates via inverse transform sampling from the marginal Beta distributions of the Dirichlet distribution.

$$\text{UUnifast}(n, U) = \text{Dir}(n, \mathbf{u^1}) \times U \tag{6}$$

In 2010, Davis and Burns presented UUnifast-Discard [13], an adaptation of UUnifast to the multiprocessor case. UUnifast-Discard repeatedly calls UUnifast to generate a vector of utilization values, discarding any vector where $\exists i \mid U_i >$ 1 until a vector is found that meets all of the constraints. Recall that UUnifast-Discard can easily be adapted to apply vectors of constraints $\mathbf{u^{max}}$ and $\mathbf{u^{min}}$.

$$\text{UUnifast-Discard}(n, U, \mathbf{u^{max}}, \mathbf{u^{min}}) \tag{7}$$

Figure 3 illustrates the output of UUnifast-Discard for a multiprocessor system, with a required total utilization of $U = 1.4$, and constraints on the maximum utilization of the three tasks given by $x \leq 0.5$, $y \leq 0.8$, and $z \leq 0.9$. These constraints are shown as annotated colored lines. Observe that the constraint on the maximum utilization of each task eliminates a region that is itself a simplex, similar in shape to that of the simplex given by the constraint on total utilization, shown by the thick black line. The results shown are for 1000 calls to UUnifast, which produced 289 valid vectors that met the constraints and are therefore plotted. Unfortunately, UUnifast-Discard suffers from the *curse of dimensionality*. Thus, although the number of discarded vectors in 3-dimensional space is manageable, this is not the case in higher dimensions, since the ratio of the constrained volume to the total volume of the simplex becomes vastly smaller[6] for larger n and larger U.

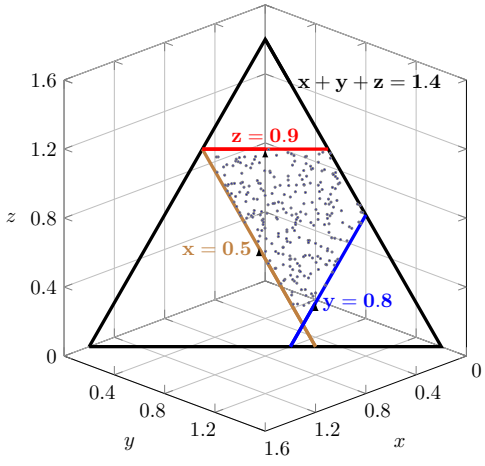

Fig. 3. The output of 1000 runs of UUnifast-Discard$(3, 1.4, (0.5, 0.8, 0.9))$.

In 2010, Emberson et al. [16] described the RandFixedSum method, invented by Stafford [30] in 2006, and adapted it to the problem of generating task utilization values for multiprocessor systems. RandFixedSum support symmetric constraints $[u^{min}, u^{max}]$ on the range of values permitted for each task utilization value U_i, subject to $nu^{min} \leq U \leq nu^{max}$, where U is the total utilization required.

The basic operation of RandFixedSum is as follows:

1) Divide the valid region into simplices, all starting from the center point at $(U/n, U/n, \ldots, U/n)$, see [16] for details of how this is done. The result is $\binom{n-1}{k}$ distinct types of simplex, where $k = \lfloor U \rfloor$, with $n!$ of each type of simplex needed to cover the entire valid region.
2) Compute the volume of each different type of simplex.
3) Randomly select a type of simplex to generate a point in, weighted by volume with respect to the set of all $\binom{n-1}{k}$ types of simplex.
4) Randomly select a point within that simplex, using an approach similar to UUnifast.

[5]Inverse transform sampling is a method of obtaining sample numbers at random from any probability distribution given its cumulative distribution function.

[6]For example, if each constraint halves the volume of the valid region, then the ratio of the two volumes is $1/2^n$.

5) Randomly permute the co-ordinates of the point to obtain coverage of the whole region.

Figure 4 illustrates the output of 1000 runs of RandFixedSum, with $n = 3$, $U = 1.4$, and $u^{max} = 1.0$. The red triangles indicate the $\binom{2}{1} = 2$ types of simplex, with $3! = 6$ of each needed to cover the entire valid region.

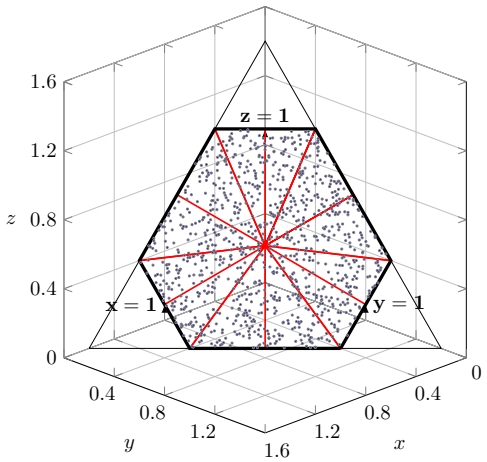

Fig. 4. The output of 1000 runs of RandFixedSum$(3, 1.4)$. Also shows the two types of simplex, 6 copies of each, required to cover the valid region.

The efficiency of RandFixedSum is due to the fact that it deals with a relatively small number of simplex types, given by the binomial coefficient $\binom{n-1}{\lfloor U \rfloor}$, rather than the actual number $(n!)$ of each type of simplex. Unfortunately, extending RandFixedSum to handle non-symmetric constraints is problematic, since it removes the symmetry that makes the co-ordinates interchangeable. This greatly increases the number of different types of simplex that need to be considered. With non-symmetric constraints, the number of different types of simplex depends on $n!$ [30], making the approach intractable. (To aid visualization of this problem, note that with 3 tasks in general the valid region becomes an irregular hexagon with no two sides equal. As a result, $3! = 6$ different types of simplex (shapes of triangle) need to be considered, 2 of each. In higher dimensions, this issue becomes much more acute).

III. Dirichlet-Rescale Algorithm

In designing an algorithm to solve the general problem of generating uniformly distributed utilization vectors containing values that sum to a required total and also meet a set of constraints on individual utilizations, we first considered the drawbacks of UUnifast-Discard and RandFixedSum. UUnifast-Discard may need to discard large numbers of points, which due to the curse of dimensionality, makes it intractable for many interesting cases. RandFixedSum on the other hand does not discard any points, but has the drawback that it only works for symmetrical constraints. It relies on symmetry to avoid intractability issues that would otherwise arise due to the need to generate points uniformly within an irregular n-dimensional polytope. The Dirichlet-Rescale Algorithm introduced in this paper overcomes these issues by generating points within a standard simplex, then rather than discarding points, it performs a series of transformations that shifts the coordinates of the points into the valid region. Crucially, these transformations preserve the property of uniform distribution over the valid region. Further, the DRS algorithm operates by transforming the problem into a *canonical form*, whereby the total utilization required is one, and the minimum constraints are zero, solving the transformed problem, and then transforming the solution back to the original problem that has non-zero minimum constraints.

We note there is a *duality* between the *standard simplex* that provides lower bounds of zero for all co-ordinates, and the *constraints simplex* that provides upper bounds for all co-ordinates, and is derived from the transformed constraints. These simplices lie on the same hyperplane, and provide upper and lower bounds, such that the valid region is given by the intersection of the two simplices. Figure 5 illustrates the standard simplex defined by the vertices $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$ and the constraints simplex defined by the constraints, i.e. $x \leq 0.5$, $y \leq 0.45$, $z \leq 0.7$. The valid region can be considered as either: (i) the region of the standard simplex that lies within the constraints simplex, or (ii) the region of the constraints simplex that lies within the standard simplex.

The DRS algorithm makes use of this duality to transpose the two simplices, when necessary to improve performance. It always selects the smaller of the two simplices as the *reference* simplex to generate initial points in, and treats the larger simplex as the constraints simplex. This ensures that initial points generated within the reference simplex break no more than $n - 1$ of the n constraints. (If all n constraints were broken, then that would imply that the constraints simplex was entirely enclosed within the reference simplex and therefore smaller). Generating points within the smaller simplex results in substantial efficiency gains when the two simplices differ greatly in volume. In practice, the use of the reference simplex is achieved by detecting when the constraints simplex is smaller, and transforming the problem accordingly. This may be different to how the problem is posed; however, it has no impact on the validity of the output.

A. Intuition and Overview

Below, we describe the intuition behind the DRS algorithm, and give an overview of its operation. A more detailed mathematical description is given in the following subsection.

The outline operation of the DRS algorithm is as follows:

1) Transform the input parameters so that the rest of the algorithm can operate on a canonical form of the problem, whereby the sum of the co-ordinate values is required to be one, and the lower bound on each co-ordinate is zero.
2) If the standard simplex has a larger volume than the constraints simplex, switch them around and further transform the problem so that the new reference simplex is again the standard simplex.
3) Generate a point on the standard simplex, using the Dirichlet distribution. Assign this point to the vector $\mathbf{P}$.
4) If $\mathbf{P}$ satisfies the constraints then return $\mathbf{P}$ after reversing the transformations applied to the original problem (exit).
5) Otherwise, define a simplex S based on the constraints that have been broken. By construction, simplex S contains point $\mathbf{P}$. (See Figure 5 which shows an initial point $\mathbf{P^0}$ that breaks the constraint $x \leq 0.5$, and the simplex S constructed using that broken constraint).
6) Map simplex S onto the standard simplex via a matrix transformation that performs the necessary translation and rescaling. This transformation alters the coordinates of point $\mathbf{P}$, making it more likely that the point will now be in the valid region. (In the example shown in Figure 5 $\mathbf{P^0}$ at $(0.9, 0.05, 0.05)$ first moves to $\mathbf{P^1}$ at $(0.8, 0.1, 0.1)$,

and then on subsequent iterations to $\mathbf{P^2}$ at $(0.6, 0.2, 0.2)$, and $\mathbf{P^3}$ at $(0.2, 0.4, 0.4)$, which is inside the valid region).

7) Go to Step 4.

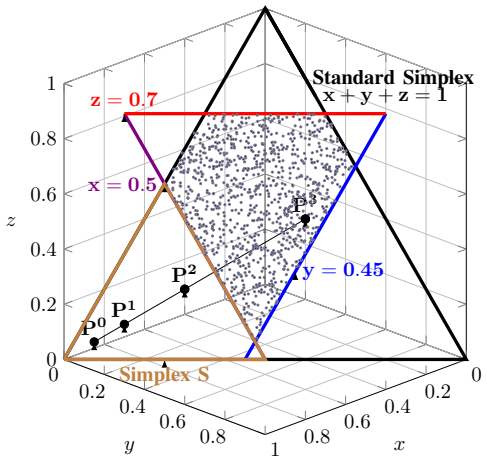

Fig. 5. The output of 1000 runs of Dirichlet-Rescale$(3, 1, (0.5, 0.45, 0.7))$. The points $\mathbf{P^0}$, $\mathbf{P^1}$, $\mathbf{P^2}$, and $\mathbf{P^3}$, show the transformation of a point via the repeated rescaling of simplex $\mathbf{S}$ onto the standard simplex.

Note that the example shown in Figure 5 is for the simplest case where a single constraint is broken; however, in general the matrix transformation in step 6 addresses multiple broken constraints at once. For example if the initial point were at $(0.52, 0.47, 0.01)$ then both the $x \leq 0.5$ and $y \leq 0.45$ constraints would be broken. In that case, the simplex S would be given by the very small triangle at the bottom of the picture, bounded by the lines $x = 0.5$, $y = 0.45$, and $z = 0$. The transformation of that simplex onto the standard simplex involves both translation and scaling.

Since the distribution of points over the standard simplex produced by the flat Dirichlet distribution is uniform, it follows that the distribution of points over simplex S is also uniform. Importantly, the matrix transformation in Step 6 that maps simplex S onto the standard simplex is an *Affine transformation* (see Section II-A), and hence the points that are uniformly distributed over S become uniformly distributed over the standard simplex and hence uniformly distributed over the valid region.

B. Detailed Description

In this subsection, we give a detailed description of the Dirichlet-Rescale algorithm. The algorithm is decomposed into a number of sub-functions, which are elaborated below, starting with the lowest level sub-functions, and ending with the DRS function itself.

Rescale Matrix to Standard Simplex: RMSS(S)

Summary: This sub-function returns a transformation matrix that rescales a regular simplex S, so that it becomes the standard simplex.

Definitions:

- Let $S = (\mathbf{s_1}, \mathbf{s_2}, ...\mathbf{s_n})$ be an $(n-1)$-dimensional simplex embedded in $\mathbb{R}^n$, with $\forall i \ \sum_{j=1}^{n} s_{i,j} = 1$. Further, let $\mathbf{o}$ be a point with all $\mathbf{o} - \mathbf{s_i}$ orthogonal to all $\mathbf{o} - \mathbf{s_k}$ for $k \neq i$.
- Let $\mathbf{b_k^n}$ be the standard basis vectors of $\mathbb{R}^n$, with the non-zero component at the k-th position (e.g. $\mathbf{b_2^3} = (0, 1, 0)$).
- Let $B^n = (\mathbf{b_1^n}, \mathbf{b_2^n}, ...\mathbf{b_n^n})$ be the $(n-1)$-dimensional standard simplex embedded in $\mathbb{R}^n$.

As S and B^n are regular simplices, it follows that S can be transformed into B^n via a standard translate-scale-translate method[7], that works as follows:

- Without loss of generality, pick $\mathbf{s_1}$ to have the greatest first component (i.e. $\forall \mathbf{s_i}, i > 1, \quad s_{1,1} > s_{i,1}$).
- Translate the set S by $-\mathbf{s_1}$ to $S' = \{\mathbf{s'_i} = \mathbf{s_i} - \mathbf{s_1}\}$. Due to the orthogonality of the components of S with respect to $\mathbf{o}$, each component of S that is not $\mathbf{s_1}$ will have exactly two non-zero components once translated. (This can be seen as follows: Each $\mathbf{s_i}$ is equal to $\mathbf{o} + k_i\mathbf{I_i}$ where $\mathbf{I_i}$ is the i-th unity vector (i.e the i-th row of the identity matrix), and k_i is some constant. $\mathbf{s'_i} = \mathbf{s_i} - \mathbf{s_1} = \mathbf{o} + k_i\mathbf{I_i} - \mathbf{o} - k_1\mathbf{I_1} = k_i\mathbf{I_i} - k_1\mathbf{I_1}$, hence two non-zero components).
- Without loss of generality, we assume that for $i > 1$, $\mathbf{s'_i} = \mathbf{s_i} - \mathbf{s_1}$ has $s'_{i,1} = x_1$ and $s'_{i,i} = x_i$ with all other components of the vector equal to zero.
- Next, scale S in each dimension as follows:
  - For $i = 1$, scale by a factor of $-\frac{1}{x_1}$
  - For $i > 1$, scale by a factor of $\frac{1}{x_i}$
- Applying this to the $\mathbf{s'_i}$, gives $\mathbf{s''_i}$, where for $i > 1$, $s''_{i,1} = -1$, $s''_{i,i} = 1$, and all other components of the vector are zero.
- Finally, translate all the $\mathbf{s''_i}$ by $\mathbf{b_1^n}$ to obtain $\mathbf{s'''_i} = \mathbf{b_i^n}$.

As all of the transformations are Affine, they can be combined into a single matrix transformation. This sub-function returns that matrix.

Constraints to Simplex: CtS$(\mathbf{r})$

Summary: This sub-function converts a vector of constraints $\mathbf{r} = \{(r_1, r_2, \ldots r_n) \mid r_i \geq 0, \ i = 1, \ldots n\}$, into a *constraints simplex* S, which is returned. This constraints simplex, on the hyperplane of the standard simplex, bounds the region in which the maximum constraints are met, while the standard simplex bounds the region in which the minimum constraints, already transformed to zero, are met.

The simplex $S = \{\mathbf{s_i}\}$, which describes the constraints $\mathbf{r}$, is defined by:

$$s_{i,j} = \begin{cases} 1 - \sum_{k=1}^{n} r_k + r_i & i = j \\ r_j & otherwise \end{cases}$$

Note, some components $s_{i,j}$ may have negative values (for example, the intersection of the $z = 0.7$ and $x = 0.5$ constraint boundaries in Figure 5 is at $(0.5, -0.2, 0.7)$).

Rescale: Rescale$(\mathbf{r}, \mathbf{P})$

Summary: This sub-function takes as its input parameters a vector of constraints $\mathbf{r}$, and a point $\mathbf{P}$ within the standard simplex. It checks if $\mathbf{P}$ breaks any of the constraints in $\mathbf{r}$. If not, then it returns $\mathbf{P}$. Otherwise, it constructs a simplex using the broken constraints and uses a matrix transformation to translate and rescale that simplex onto the standard simplex, transforming the coordinates of point $\mathbf{P}$.

1) If $\mathbf{P}$ satisfies all of the constraints in $\mathbf{r}$, then return $\mathbf{P}$.
2) Otherwise, construct the vector $\mathbf{b}$ such that:

$$\mathbf{b}_i = \begin{cases} \mathbf{r}_i & \text{if constraint } \mathbf{r}_i \text{ was broken} \\ 0 & \text{otherwise} \end{cases}$$

[7]This is true for any pair of regular simplices in $\mathbb{R}^n$.

3) Let $S = \text{CtS}(\mathbf{b})$, where S is the simplex constructed from the broken constraints. (Note that by construction, $\mathbf{P}$ is within S).
4) Let $M = \text{RMBS}(S)$, where M is the transformation matrix that translates and rescales simplex S onto the standard simplex.
5) Return $\text{Rescale}(\mathbf{r}, M\mathbf{P})$. (Note $M\mathbf{P}$ requires $\mathbf{P}$ to be embedded into an $(n+1)$-dimensional vector, multiplied, and then de-embedded[8]).

SmallestSimplexRescale - $SSR(\mathbf{r}, \mathbf{P})$

Summary: This sub-function takes as its input parameters a vector of constraints $\mathbf{r}$, and a point $\mathbf{P}$ within the standard simplex. It checks if the simplex corresponding to the constraints $\mathbf{r}$ is smaller than the standard simplex. If so, then the problem is transformed, switching the standard simplex for the constraints simplex, and vice-versa.

1) Let $C = \text{CtS}(\mathbf{r})$, where C is the simplex corresponding to the constraints $\mathbf{r}$.
2) If the volume of C is greater than the volume of the standard simplex, return $\text{Rescale}(\mathbf{r}, \mathbf{P})$.
3) Otherwise, let $M = \text{RMBS}(C)$, where M is the matrix that transforms simplex C onto the standard simplex.
4) Let $\mathbf{q} = M\mathbf{0}$ be the new constraints. Note that $M\mathbf{0}$ is the boundary of the standard simplex transformed by the matrix transformation that maps the simplex representing the original constraints to the standard simplex, and hence provides the constraints of the transformed problem.
5) Let $\mathbf{t} = \text{Rescale}(\mathbf{q}, \mathbf{P})$ be the solution to the transformed problem.
6) Let M^{-1} be the inverse of M, calculated via $M^{-1} = \text{RMBS}(\text{CtS}(\mathbf{q}))$.
7) Return $M^{-1}\mathbf{t}$, inverting the original transformation by M that was applied.

Dirichlet-Rescale: $DRS(n, U, \mathbf{u^{max}}, \mathbf{u^{min}})$

Summary: This is the main function. It takes as input parameters the number of tasks n, the desired total utilization U, a vector of maximum constraints $\mathbf{u^{max}}$ (default $\mathbf{u^1}$), and a vector of minimum constraints $\mathbf{u^{min}}$ (default $\mathbf{u^0}$). It first checks that the problem is valid. It then transforms the parameters converting the problem into a *canonical form*, whereby the sub-functions can operate on a standard simplex (i.e. minimum constraints of $\mathbf{u^0}$). It generates the initial point $\mathbf{P}$ in the standard simplex, and calls SmallestSimplexRescale(). Finally, it applies the reverse transformation on the output.

1) Check that the problem is valid (i.e. $\sum_{k=1}^{n} U_k^{min} \leq U \leq \sum_{k=1}^{n} U_k^{max}$ and $\forall k\, U_k^{max} \geq U_k^{min}$).
2) If $\mathbf{u^{min}} \neq \mathbf{u^0}$, then convert the problem into a form with $\mathbf{u^{min}} = \mathbf{u^0}$ as follows. First modify the constraints such that $\mathbf{u'^{max}} = \mathbf{u^{max}} - \mathbf{u^{min}}$, $U' = U - \sum_{k=1}^{n} U_k^{min}$, then return $(\text{DRS}(n, U', \mathbf{u'^{max}}, \mathbf{u^0})) + \mathbf{u^{min}}$.
3) Let $\mathbf{P}$ be an initial point in the standard simplex generated by $\text{Dirichlet}(n, 1)$.
4) Return $U \cdot \text{SmallestSimplexRescale}(\mathbf{u^{max}}/U, \mathbf{P})$.

C. Convergence and Complexity

To prove that the Dirichlet-Rescale algorithm converges, we effectively "paint" the regions of the standard simplex that converge after q iterations, where an iteration represents a call to the Rescale() sub-function.

[8]See https://en.wikipedia.org/wiki/Translation_(geometry) for an overview of translation using a matrix representation.

Consider a uniformly distributed, randomly chosen point $\mathbf{P}$ on the standard simplex. Let

$$p = \frac{\text{volume(valid region)}}{\text{volume(standard simplex)}} \tag{8}$$

Trivially, after no iterations, p is the probability that point $\mathbf{P}$ will be accepted, and $1 - p$ is the probability that it will not. If $\mathbf{P}$ is not accepted, then that is because it lies outside of the valid region. In which case, a new simplex S is formed from the broken constraints. By construction, $\mathbf{P}$ is within S. The ratio of the volume of simplex S to the volume of the standard simplex can be no greater than $(1 - p)$, since by construction, the entire volume of S lies within the standard simplex, and does not intersect with the valid region. The rescale operation maps S onto the standard simplex, and hence an upper bound on the probability that $\mathbf{P}$ is not accepted on any one subsequent iteration is given by $(1 - p)$. It follows that an upper bound on the probability of $\mathbf{P}$ not being accepted after q iterations is given by $(1 - p)^q$. This leads to the following formula for the minimum converged volume c after q iterations:

$$c \geq 1 - (1 - p)^q \tag{9}$$

Thus, as $q \to \infty$, $c \to 1$, and hence the algorithm converges.

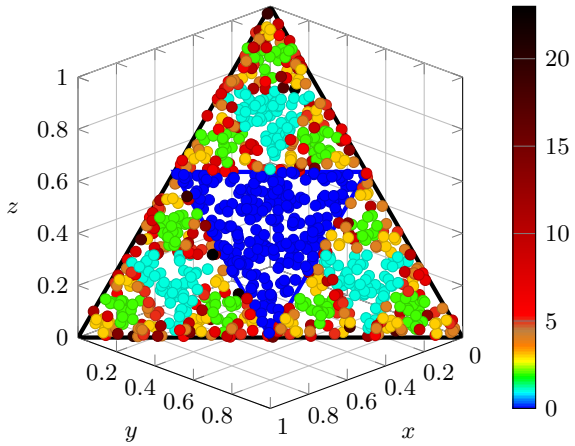

Fig. 6. Number of iterations (basic rescale operations) required to converge for various initial points with the constraints $x \leq 0.5$, $y \leq 0.5$, $z \leq 0.5$, and $x + y + z = 1$.

Figure 6 illustrates the areas of the standard simplex[9] that converge after 0 (blue), 1 (cyan), 2 (green), 3 (yellow), 4 (orange), 5 (brown), and 6 (red) basic rescale operations, assuming symmetrical constraints of $x \leq 0.5$, $y \leq 0.5$, $z \leq 0.5$ and $x + y + z = 1$. The standard simplex is shown in black and the constraints simplex in blue. Observe the fractal pattern (equating to the volume removed from a Sierpinski triangle [28]) created by the areas that have converged after a fixed number of iterations, and that on each iteration the remaining unconverged area is reduced by a factor of approximately 0.75. Hence, after 24 iterations, only an area approximately 0.001003 times that of the standard simplex remains. (In this particular example, all 1000 initial points converged within 24 iterations).

[9]For illustration purposes, no optimization was performed leveraging the duality between the constraints simplex and the standard simplex to switch them around.

We note that convergence could be slow when the value of p is very small, i.e. when the valid region is a tiny fraction of the volume of the standard simplex. This can potentially occur in two ways, the first of which is addressed by the algorithm design. If the constraints are tight i.e. $\sum_i U_i^{max} = U + \delta$ where δ is a small value, (for example, $\forall i, U_i^{max} = (1+\delta)/n$, $U = 1$), then the constraints simplex has a very small volume. However, in this scenario, the algorithm exploits the duality of the reference and constraints simplices to switch them around, leading to rapid convergence. The worst-case scenario thus occurs when there is no gain from exploiting duality and yet the volume of the valid region is still very small. This happens when the two simplices are of equal volume and overlap occurs in a region along a single edge[10]. In this case, two of the constraints take values of $1 - \delta/2$, while the remaining constraints are all $\delta/(n-2)$. While such pathological constraints are possible to program, they are highly unlikely to occur within the scope of the intended use of the algorithm, i.e. generating task utilization values.

D. Implementation Issues and Convergence

There are two implementation issues related to the repeated rescaling employed by the DRS algorithm: the accumulation of floating point error, and the finite (e.g. 64-bit) precision of the floating point representation. The former can result in a small non-uniform component to the values generated, while the latter effectively places a bound on the number of rescale operations that can occur before Shannon's Entropy [27] in the initial point is effectively exhausted. These issues manifest themselves as a divergence from the required total utilisation. The algorithm detects when this divergence exceeds a given constant ϵ. A default value of $\epsilon = 10^{-4}$ is used in our implementation, thus the sum of the utilization values is permitted to have an error of at most 0.01%. If divergence exceeds ϵ, then the DRS algorithm retries with a new initial point. Note, as this is an error term, there is no guarantee of uniformity when values are considered to a precision smaller than ϵ. If uniformity is required to a higher precision, then a smaller value of ϵ can be used. Although the implementation of the DRS algorithm rejects some points and retries, the regions affected are much smaller than with UUnifast-Discard, hence the algorithm remains tractable. In the following section, we detail an empirical investigation into the convergence of the algorithm in practice.

In cases where one or more constraints have very small values, repeated rescaling may be required to map the initial point onto the valid region. (A simple form of this problem is illustrated in Figure 5 where point $\mathbf{P}$ is subject to the same transformation four consecutive times). The implementation of the DRS algorithm is optimized to compute the maximum number of times k that the same matrix transformation M can be applied without the set of broken constraints changing[11]. The matrix M^k for the overall transformation is then obtained by combining M, M^2, M^4, $M^8 \ldots$, thus reducing the number of matrix operations needed to $O(log(k))$. We refer to this as a *power transformation*.

The complexity of the algorithm is effectively pseudo-polynomial $O(p^{-1}n^2)$, where p is defined by (8), with the $O(n^2)$ term coming from the matrix transformations.

[10]It is not possible to collapse the valid region further, i.e. towards a single point, without also decreasing the volume of the constraints simplex.

[11]A change in the set of broken constraints would imply that a different matrix transformation was required.

IV. Empirical Verification

This section provides an empirical investigation into the performance and correctness of the DRS algorithm. First, we examine the efficiency of the algorithm in terms of the number of re-scaling operations required to generate utilization vectors. Here, we investigate how runtime performance varies with the desired total utilization, and also with the number of tasks (i.e. the dimension of the vectors). Second, we verify empirically that the algorithm produces a set of vectors that are uniformly distributed within the valid region.

The evaluation covers the full range of possible inputs in terms of the canonical form of the problem. This is achieved by fixing the sum of the maximum constraints at 1, varying the total required utilization over the range $[0, 1]$, and employing no minimum constraints (i.e. $\mathbf{u^{min}} = \mathbf{u^0}$).

Experiment A: investigated how the number of re-scale operations and the number of retries required varies with the utilization of the vectors generated from $U = 0.05$ to 0.95 in steps of 0.05, for vectors of cardinality $n = 50$. The data for each utilization level is based on 10,000 runs. On each run, first a set of constraints were generated via $\mathbf{u^{max}} = \text{UUnifast}(n, 1)$, then $\text{DRS}(n, U, \mathbf{u^{max}})$ was called to generate a vector with the desired utilization, subject to those constraints. Figure 7 illustrates the distribution of the number of rescale operations required for each of the runs. Observe that the maximum number of rescale operations did not exceed 200, for those points that converged, and that the mean and upper percentiles took their worst-case values for $U = 0.5$.

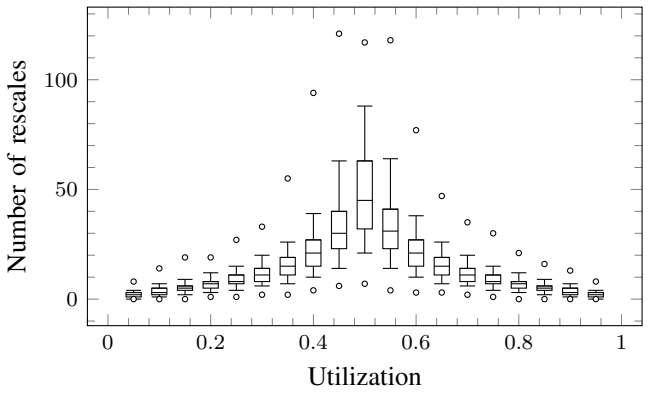

Fig. 7. Distributions of the number of rescale operations required by DRS to generate vectors of $n = 50$ utilization values, summing to a total of 0.05 to 0.95 (x-axis). Constraints generated as a vector of $n = 50$ maximum utilization values summing to 1. The plot shows: minimum (lower circle), 5-percentile (lower whisker), 25-percentile (bottom of box), median (line across box), 75-percentile (top of box), 95-percentile (upper whisker), and maximum (upper circle) of the distribution.

Figure 8 shows the average number of retries per vector over the 10,000 runs at each utilization level. Observe that the average number of retries per vector did not exceed 0.25 and the worst-case was again at $U = 0.5$. (Note, in Figures 8 and 9 we increased U in steps of 0.01 to capture the fine detail).

By way of comparison, we repeated *Experiment A* using UUnifast-Discard to generate the vectors. In this case, we limited the maximum number of discards on each run to 10,000. If that limit was exceeded, then we recorded that the algorithm had failed to generate a valid vector. Figure 9 shows the average number of discards over the 10,000 runs at each utilization level. Also shown is the number of valid vectors produced. The worst-case occurs for $U > 0.3$, where no valid

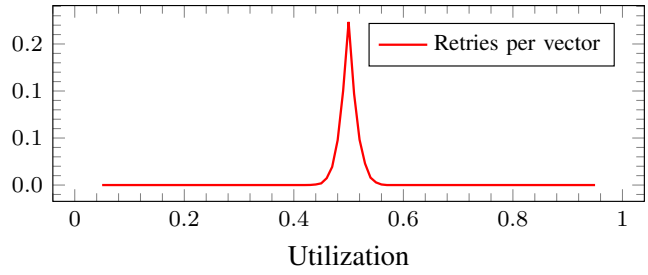

Fig. 8. Average number of retries per vector, required by DRS to generate vectors of $n = 50$ values for each utilization level (x-axis).

vectors are produced in 10^8 attempts per utilization level, and so the average number of discards per vector is 10,000. For $U > 0.15$, UUnifast-Discard produced less than half of the vectors within 10,000 discards, illustrating that the range of problem characteristics where it can provide a viable solution is severely restricted[12]. There are no such issues with the DRS algorithm, which produced valid vectors in *every* case.

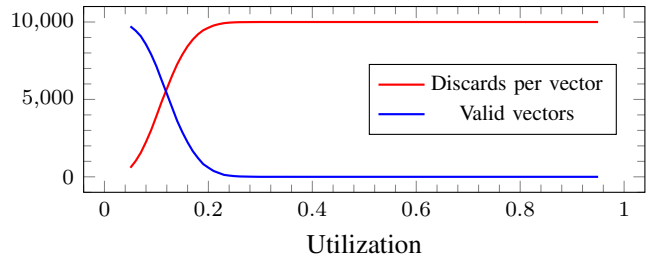

Fig. 9. Average number of discards per vector, required by UUnifast-Discard to generate vectors of $n = 50$ values for each utilization level (x-axis). Also, number of valid vectors generated in less than 10,000 discards each.

Experiment B: was similar to *Experiment A*, however, it fixed the total required utilization at $U = 0.5$ (the worst-case with respect to DRS algorithm efficiency), and varied the number of tasks from $n = 5$ to 100 in steps of 5. Figure 10 illustrates the distribution of the number of rescale operations required for each of the 10,000 runs at each task set cardinality value. Observe that the maximum number of rescale operations required does not exceed 200 for $n \leq 100$, for those points that converged. Further, the DRS algorithm succeeded in generating vectors in all cases. Figure 11 shows the average number of retries per vector over the 10,000 runs at each task set cardinality. Note the average number of retries per vector did not exceed 5, even for task sets with $n = 100$.

Experiment C: examined how the runtime of the DRS algorithm varies with task set cardinality n, based on the requirements of a de facto standard schedulability analysis experiment. Such an experiment typically requires 1000 task sets to be generated at each utilization level from 0.05 to 0.95 in steps of 0.05. To obtain each task utilization vector, first a set of constraints were generated via $\mathbf{u^{max}} = \text{UUnifast}(n, 1)$, then $\text{DRS}(n, U, \mathbf{u^{max}})$ was called to generate a vector with the desired utilization, subject to those constraints. Figure 12 shows how the overall processing time required to generated the 18,000 vectors needed for a standard schedulability anal-

[12]For smaller numbers of tasks e.g. $n = 20$, UUnifast-Discard is somewhat more effective, but still there were no valid vectors produced for $U > 0.6$, and less than half for $U > 0.3$.

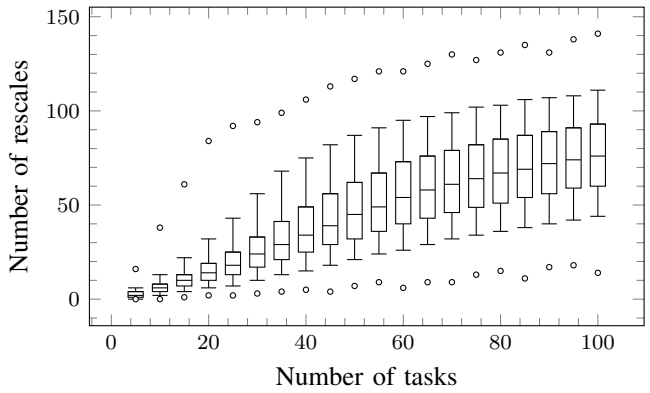

Fig. 10. Boxplot of the number of rescale operations required by DRS to generate a vector of n utilization values summing to a total of $U = 0.5$ for varying cardinality n (x-axis).

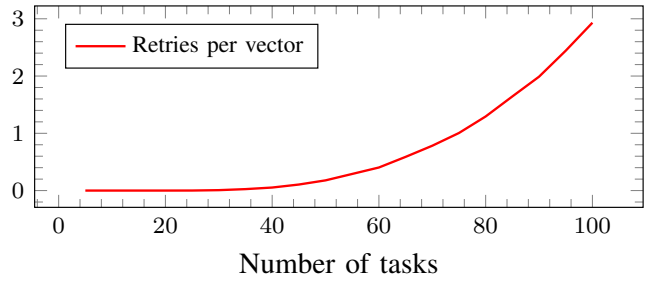

Fig. 11. Average number of retries per vector, required by DRS, to generate 10,000 vectors, $U = 0.5$, for varying cardinality n (x-axis).

ysis experiment[13] varies with task set cardinality. The results were obtained by running a Python/NumPy implementation of the DRS algorithm on all cores of a Raspberry Pi 4 (with a heatsink capable of avoiding thermal throttling), using Ubuntu Mate 20.04 Beta 1 AArch64. This hardware platform was used to obtain accurate runtimes, while avoiding any interference from co-running software. Note, in practice, the DRS algorithm would normally be run on a desktop or laptop PC or a server. Using a Dell XPS 13 with Intel® i7-1065G7 running at 3.5GHz, the time required to generate 18,000 vectors was approx. 6 seconds for 10 tasks, 1 minute for 50 tasks, and 6 minutes for 100 tasks (approx. 6 times faster than the Pi 4).

Experiment D: verified that the distribution of vectors produced by the DRS algorithm is unbiased. This was done by showing that the distribution is not statistically significantly different from that produced by UUnifast-Discard. (By construction, the latter produces an unbiased distribution of vectors). We note that while both algorithms in theory produce a uniform distribution of vectors when operating on real values, when a fixed-precision (e.g 64-bit) floating point implementation is used, then the distributions obtained are *not* precisely uniform. This issue is caused by the fixed-precision arithmetic rather than any issue with the algorithms. As this experiment aims to demonstrate uniformity as precisely as possible, the value of ϵ was set to 10^{-8} (see Section III-D). *Expt. D* was characterized as follows ($n = 10$, and $U = 0.5$):

[13]Note, weighted schedulability [4] experiments vary a additional parameter over perhaps 10 different values, but also combine results for different utilization levels into a single data point. Hence they typically only require 100 task sets per utilization level to achieve high quality results. The total number of vectors needed is therefore similar to a standard schedulability analysis experiment.

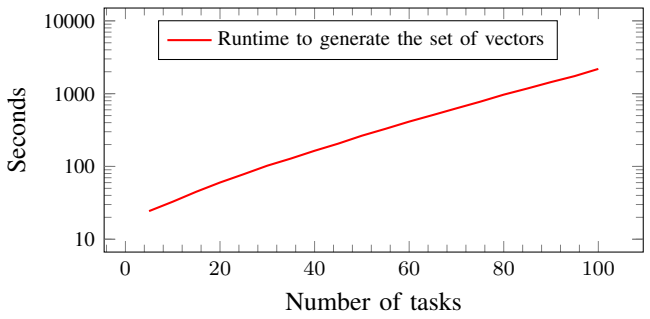

Fig. 12. Total processing time in seconds (y-axis) required by the DRS algorithm to generate the set of vectors for a de facto standard schedulability analysis experiment, for varying cardinality n (x-axis).

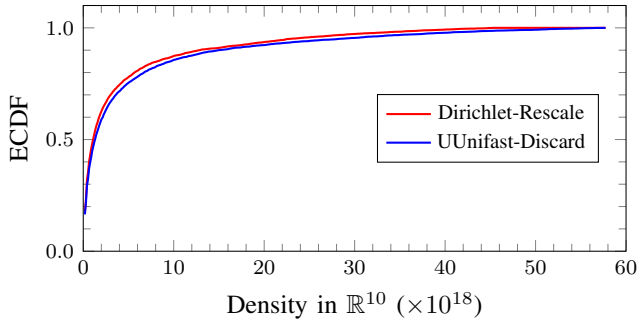

Fig. 13. Empirical Cumulative Distribution Function (ECDF) for the density of points in $\mathbb{R}^{10}$ generated by DRS and UUnifast-Discard.

1) $\mathbf{u^{max}}$ = UUnifast$(n, 1)$ was used to generate a constraints vector $\mathbf{u^{max}}$.
2) DRS$(n, U, \mathbf{u^{max}})$ and UUnifast-Discard$(n, U, \mathbf{u^{max}})$ were called 10^6 times each to generate sets G and H respectively, of vectors within the valid region.
3) 1000 reference simplices were generated as follows. First, 1000 sample simplices were created by calling UUnifast-Discard$(n, U, \mathbf{u^{max}})$, and the volume of each sample simplex calculated. After discarding outliers, the $98-$ and $95-$percentiles were used to calculate upper and lower bounds for acceptable reference simplex volumes. Reference simplices were then generated by the same method as sample simplices, and only accepted if their volume was within the bounds. This method was used to pick reasonably large simplices and therefore minimize the probability that a reference simplex would contain no points. Since the vertices were selected from points within the valid region, each reference simplex was also fully contained within the valid region.
4) For each reference simplex, the numbers of points from G and H contained within it were determined. This data was then used to form density distributions. Any reference simplex containing zero points was discarded. Further, only data within the $5 - 95$ percentiles were considered, since outliers, outside of this range, can have a disproportionate effect on statistical testing.
5) The density distributions for DRS and UUnifast-Discard, were then compared using a two sample Kolmogorov-Smirnov (KS) test [29,31].

The two density distributions are shown in Figure 13, as cumulative distribution functions. The two sample KS test on this data gave a KS-statistic of 0.04, and a p-value of 1.0, which is as high as it can be. Hence, there is absolutely no evidence that the distributions diverge, and we cannot reject the null hypothesis that the density values from UUniFast-Discard and DRS are drawn from the same distribution. Therefore we have a high degree of confidence that the two distributions are identical.

V. Mixed Criticality Systems Example

In this section, we give an example of how the Dirichlet-Rescale algorithm can be employed in the evaluation of schedulability tests for mixed criticality systems.

Below, we make use of the following additional notation: The sets of LO- and HI-criticality tasks are denoted by $\{LO\}$ and $\{HI\}$ respectively. $U^{LO} = \sum_{\forall i} U_i(LO)$ is the total LO-criticality utilization of all the tasks. Further, $U^{LO}_{LO} = \sum_{\forall i \in \{LO\}} U_i(LO)$ is the total LO-criticality utilization of the LO-criticality tasks, and $U^{HI}_{LO} = \sum_{\forall i \in \{HI\}} U_i(LO)$ is the total LO-criticality utilization of the HI-criticality tasks, hence, $U^{LO} = U^{LO}_{LO} + U^{HI}_{LO}$. Finally, $U^{HI}_{HI} = \sum_{\forall i \in \{HI\}} U_i(HI)$ is the total HI-criticality utilization of the HI-criticality tasks.

Our example is based on the first experiment described by Baruah et al. in [3]. As a baseline, we replicated the task set generation used in that paper, summarized as follows: Each task set contained $n = 20$ tasks. The number n^{HI} of HI-criticality tasks was determined by the *Criticality Proportion*[14], CP, hence $n^{HI} = CP \cdot n$, and $n^{LO} = n - n^{HI}$. With $CP = 0.5$, $n^{HI} = n^{LO} = 10$. Task LO-criticality utilizations, $U_i(LO)$, were generated using UUnifast [5], and task HI-criticality utilizations set to $U_i(HI) = CF \cdot U_i(LO)$, where the *Criticality Factor*, $CF = 2.0$. Task periods were selected at random from a log-uniform distribution, with a ratio of 100 between the minimum and maximum periods. Task executions times were given by $C_i(LO) = U_i(LO) \cdot T_i$ and $C_i(HI) = U_i(HI) \cdot T_i$. The deadlines of all tasks were implicit, i.e. equal to their periods.

Figure 14 shows the proportion of 1000 task sets generated for each total LO-criticality utilization level (from 0.05 to 0.95) that were schedulable according to the MCS schedulability tests: AMC-max, AMC-rtb, and SMC (described in [3]). Also shown is UB-H&L, an upper bound on the performance of any fixed priority mixed criticality scheduling algorithm (also described in [3]), and a further test, labelled Valid, which is a necessary condition for schedulability of a mixed criticality system [7]: (i) $\forall_i U_i(LO) \leq 1$, (ii) $\forall_{i \in \{HI\}} U_i(HI) \leq 1$, (iii) $U^{LO} \leq 1$, and (iv) $U^{HI}_{HI} \leq 1$.

Observe that in Figure 14, for utilization levels of $U^{LO} \geq 0.6$, not all of the task sets generated are valid, and at $U^{LO} = 0.95$ just over half are valid. The reason for this is that the approach used by Baruah et al. [3] ensures that $U^{LO} \leq 1$, but does not ensure that $U^{HI}_{HI} \leq 1$. In theory, U^{HI}_{HI} can take any value in the range $(0, CF \cdot U^{LO})$. Checking the data for the 1000 task sets generated with $U^{LO} = 0.95$, the 25-percentile, median, and 75-percentile values for U^{HI}_{HI} were $\{0.86, 0.96, 1.11\}$, with 425 of the 1000 values exceeding 1, and therefore invalid. Further, since $U_i(HI) = CF \cdot U_i(LO)$, individual HI-criticality tasks may also be invalid, with $U_i(HI) > 1$.

Figure 15 illustrates schedulability test performance when task sets are generated making use of the DRS algorithm introduced in this paper, summarized as follows: The number of tasks, and the methods of determining deadlines and periods

[14] Baruah et al. [3] used a probability of 50% of each task being HI-criticality; however, here we fix the number of HI-criticality tasks to avoid this additional source of variability.

were the same as in [3] (recapped above). The $U_i(HI)$ values for the n^{HI} HI-criticality tasks were first generated by calling DRS$(n^{HI}, U_{HI}^{HI}, \mathbf{u^1})$, where $U_{HI}^{HI} = CF \cdot CP \cdot U^{LO}$ and U^{LO} is the desired LO-criticality utilization level. Next, the constraint vector $\mathbf{u^{max}}$ of maximum utilization values was constructed using the $U_i(HI)$ values computed in the previous step for HI-criticality tasks and the value 1 (i.e. no constraint except for validity) for the LO-criticality tasks. The $U_i(LO)$ values for all n tasks were then generated by calling DRS$(n, U^{LO}, \mathbf{u^{max}})$, where U^{LO} is the desired LO-criticality utilization level. Task executions times were given by $C_i(LO) = U_i(LO) \cdot T_i$ and $C_i(HI) = U_i(HI) \cdot T_i$.

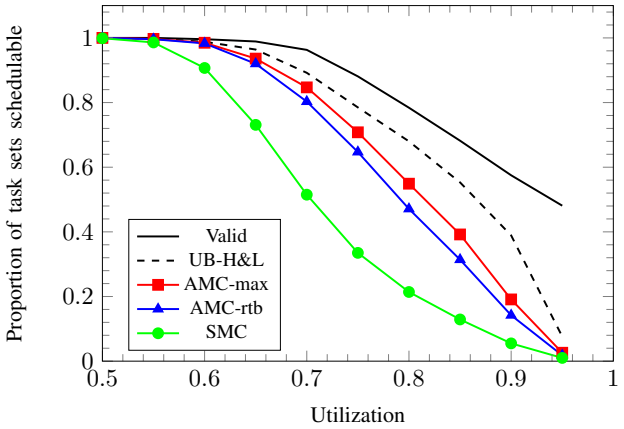

Fig. 14. Performance of AMC and SMC scheduling schemes with task sets generated following the approach of Baruah et al. [3], controlling only U^{LO}.

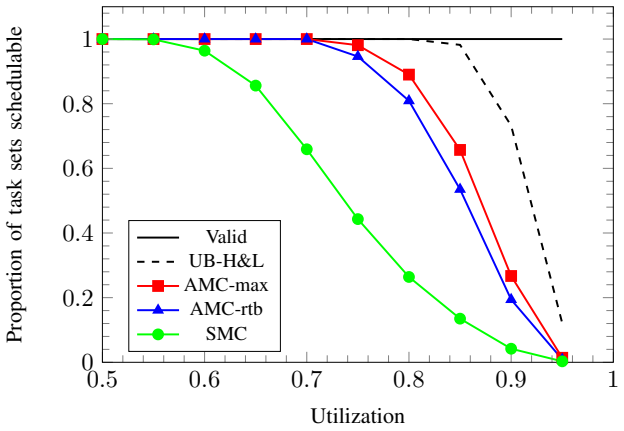

Fig. 15. Performance of AMC and SMC scheduling schemes with task sets generated using the DRS algorithm, controlling both U^{LO} and U_{HI}^{HI}.

This method controls both U^{LO} and U_{HI}^{HI}, the latter via the parameters CP and CF. Since $CP = 0.5$ and $CF = 2.0$, in this case $U_{HI}^{HI} = U^{LO}$ meaning that provided $U^{LO} \leq 1$, none of the task sets generated are invalid. Further, as $U_{HI}^{HI} = U^{LO}$, the y-axis of Figure 15 gives both the total LO-criticality utilization of the task set U^{LO} and the HI-criticality utilization of HI-criticality tasks, U_{HI}^{HI}.

We observe that there are significant differences between the apparent performance of the AMC and SMC scheduling schemes, dependent on which method of task set generation is used. Figure 15 reveals a *sharp utilization threshold* [19] for AMC, where the probability of a randomly generated task set being schedulable changes from close to 1 to close to zero, over a small increase in utilization. This behavior occurs with AMC because overall schedulability is mainly dependent on schedulability in the individual LO- and HI-criticality modes (as it is with the UB-H&L bound), with some impact from the mode change transition. Hence, the values of U^{LO} and U_{HI}^{HI}, which are precisely controlled for by using the DRS algorithm and necessarily cannot exceed 1 in a valid system, characterize the threshold. No such sharp threshold is apparent in Figure 14, since the method of task set generation used by Baruah et al. [3] does not control for U_{HI}^{HI}. By contrast, schedulability according to SMC does not exhibit a sharp threshold with either method of task set generation. The reason for this is that SMC still executes LO-criticality tasks in HI-criticality mode, even though there is no requirement for those tasks to meet their deadlines. This impinges on the schedulability of HI-criticality tasks in the HI-criticality mode, hence schedulability with SMC depends on the values of U^{LO} and $U_{HI}^{HI} + U_{LO}^{LO}$, and the latter can exceed 1 in valid systems.

Comparing Figures 14 and 15, it is clear that the advantage that the AMC scheme has over SMC is significantly larger than could be inferred from previous work [3], and this disparity has been hidden due to an artefact of the task set generation methods previously used.

To summarize, the Dirichlet-Rescale algorithm introduced in this paper can be used to improve the quality of performance evaluation for MCS scheduling schemes and schedulability tests. Using this algorithm, precise control can be exercised over multiple key utilization parameters (e.g. U^{LO} and U_{HI}^{HI}), while simultaneously ensuring that all of the task sets generated are valid, comply with the constraints $(U_i(LO) \leq U_i(HI))$, and the utilization vectors have an unbiased distribution.

VI. Conclusions

This paper introduced the Dirichlet-Rescale (DRS) algorithm, a general-purpose method of generating uniformly distributed n-dimensional vectors of components (e.g. task utilizations), where the components sum to a specified total, and each component conforms to individual maximum and minimum constraints.

The evaluation showed that the DRS algorithm can efficiently generate vectors with dimensions of up to $n = 100$ via a Python implementation that has been made publicly available [20]. (Note, this 64-bit floating point implementation of the DRS algorithm is able to generate vectors with dimensions of up to $n = 200$, with a commensurate slowdown in performance). Finally, we verified that the distribution of vectors generated is unbiased, by making a statistical comparison against UUnifast-Discard, which is known to produce a uniform distribution of vectors within the valid region defined by the constraints.

The DRS algorithm can be used to improve the nuance and quality of empirical studies into the effectiveness of schedulability tests for real-time systems; potentially making them more realistic, and leading to new conclusions. It is particularly useful in task set generation where task utilizations are either multi-valued or can be decomposed into multiple constituent parts. Examples include modelling mixed criticality systems, multi-core systems, typical and worst-case execution times, self-suspensions, and resource locking.

Acknowledgements

The research in this paper is partially funded by the Innovate UK HICLASS project (Ref. 113213) and the ESPRC grant STRATA (EP/N023641/1). EPSRC Research Data Management: No new primary data was created during this study.

REFERENCES

[1] S. Altmeyer, R. I. Davis, L. Indrusiak, C. Maiza, V. Nelis, and J. Reineke. A generic and compositional framework for multicore response time analysis. In *Proc. International Conference on Real Time and Networks Systems (RTNS)*, pages 129–138, 2015.

[2] S. Baruah. Rapid routing with guaranteed delay bounds. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 13–22, 2018.

[3] S. Baruah, A. Burns, and R. I. Davis. Response-time analysis for mixed criticality systems. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 34–43, 2011.

[4] A. Bastoni, B. Brandenburg, and J. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proc. International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 33–44, 2010.

[5] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Journal of Real-Time Systems*, 30(1-2):129–154, 2005.

[6] A. Burns, R. Davis, S. K. Baruah, and I. Bate. Robust mixed-criticality systems. *IEEE Transactions on Computers*, 67(10):1478–1491, 2018.

[7] A. Burns and R. I. Davis. Adaptive mixed criticality scheduling with deferred preemption. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 21–30, 2014.

[8] A. Burns and R. I. Davis. A survey of research into mixed criticality systems. *ACM Computer Surveys*, 50(6):1–37, 2017.

[9] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real-Time Systems*, 55(1):144–207, Jan 2019.

[10] R. I. Davis. On the evaluation of schedulability tests for real-time scheduling algorithms. In *Proc. International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, July 2016.

[11] R. I. Davis, S. Altmeyer, and A. Burns. Mixed criticality systems with varying context switch costs. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 140–151, 2018.

[12] R. I. Davis, S. Altmeyer, L. S. Indrusiak, C. Maiza, V. Nelis, and J. Reineke. An extensible framework for multicore response time analysis. *Real-Time Systems*, 54(3):607 – 661, July 2018.

[13] R. I. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47:1–40, 2010.

[14] P. Ekberg and W. Yi. Bounding and shaping the demand of mixed-criticality sporadic task systems. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, pages 135–144, 2012.

[15] P. Ekberg and W. Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Journal of Real-Time Systems*, 50:48–86, 2014.

[16] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proc. International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, July 2010.

[17] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian data analysis (2nd edn).*, volume 23. Chapman & Hall/CRC, 2003.

[18] O. Gettings, S. Quinton, and R. I. Davis. Mixed criticality systems with weakly-hard constraints. In *Proc. International Conference on Real-Time Networks and Systems (RTNS))*, pages 237–246, 2015.

[19] S. Gopalakrishnan. Sharp utilization thresholds for some realtime scheduling problems. *SIGMETRICS Perform. Eval. Rev.*, 39(4):12–22, Apr. 2012.

[20] D. Griffin, I. Bate, and R. I. Davis. Dirichlet-Rescale (DRS) algorithm software: dgdguk/drs: v1.0.0 available at https://doi.org/10.5281/zenodo.4118059, Dec. 2020.

[21] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The malardalen wcet benchmarks: Past, present and future. In *Proc. International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2010.

[22] L. Liberti and C. Lavor. Six mathematical gems from the history of distance geometry, 2015.

[23] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3), June 2019.

[24] K. Menger. Untersuchungen uber allgemeine metrik. *Mathematische Annalen*, 103, Dec 1930.

[25] I. Olkin and H. Rubin. Multivariate beta distributions and independence properties of the wishart distribution. *Annals of Mathematical Statistics*, 35(1):261–269, March 1964.

[26] S. Quinton, M. Hanke, and R. Ernst. Formal analysis of sporadic overload in real-time systems. In *Proc. Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 515–520, 2012.

[27] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[28] W. Sierpinski. Sur une courbe dont tout point est un point de ramification. *C. R. Acad. Sci.*, 160:302–305, 1915.

[29] N. Smirnov. Table for estimating the goodness of fit of empirical distributions. *Ann. Math. Statist.*, 19(2):279–281, 06 1948.

[30] R. Stafford. Random vectors with fixed sum. Technical Report Available at https://www.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum, MathWorks, 2006.

[31] L. J. Stephens. *Schaum's Outlines: Beginning Statistics*. McGraw-Hill, 2nd edition, 2006.

[32] D. Stirzaker. *Elementary Probability*. Cambridge University Press, 2 edition, 2003.