Parametric Design Synthesis of Distributed Embedded Systems *

Dong-In Kang, Richard Gerber
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742
{dikang, rich}@cs.umd.edu

Manas Saksena
Dept. of Computer Science
Concordia University
Montreal Quebec H3G 1M8
Canada
manas@cs.concordia.ca

Abstract

This paper presents a design synthesis method for distributed embedded systems. In such systems, computations can flow through long pipelines of interacting software components, hosted on a variety of resources, each of which is managed by a local scheduler. Our method automatically calibrates the local resource schedulers to achieve the system's global end-to-end performance requirements.

A system is modeled as a set of distributed task chains (or pipelines), where each task represents an activity requiring nonzero load from some CPU or network resource. Task load requirements can vary stochastically, due to second-order effects like cache memory behavior, DMA interference, pipeline stalls, bus arbitration delays, transient head-of-line blocking, etc. We aggregate these effects – along with a task's per-service load demand – and model them via a single random variable, ranging over an arbitrary discrete probability distribution. Load models can be obtained via profiling tasks in isolation, or simply by using an engineer's hypothesis about the system's projected behavior.

The end-to-end performance requirements are posited in terms of throughput and delay constraints. Specifically, a pipeline's delay constraint is an upper bound on the total latency a computatation can accumulate, from input to output. The corresponding throughput constraint mandates the pipeline's minimum acceptable output rate – counting only outputs which meet their delay constraints. Since per-component loads can be generally distributed, and since resources host stages from multiple pipelines, meeting all of the system's end-to-end constraints is a nontrivial problem.

Our approach involves solving two sub-problems in tandem: (A) finding an optimal proportion of load to allocate each task and channel; and (B) deriving the best combination of service intervals over which all load proportions can be guaranteed. The design algorithms use analytic approximations to quickly estimate output rates and propagation delays for candidate solutions.

When all parameters are synthesized, the estimated end-to-end performance metrics are rechecked by simulation. The per-component load reservations can then be increased, with the synthesis algorithms re-run to improve performance. At that point the system can be configured according to the synthesized scheduling parameters – and then re-validated via on-line profiling.

In this paper we demonstrate our technique on an example system, and compare the estimated performance to its simulated on-line behavior.

*This research is supported in part by ONR grant N00014-94-10228, NSF Young Investigator Award CCR-9357850, ARL Cooperative Agreement DAAL01-96-2-0002 and NSERC Operating Grant OGPO170345. A preliminary version of this work was reported in "Performance-Based Design of Distributed Real-Time Systems," at the *Proceedings of IEEE Real-Time Technology and Applications Symposium* (June 1997).

1 Introduction

An embedded system's intrinsic real-time constraints are imposed on its external inputs and outputs, from the perspective of its environment. At the same time, the computation paths between these end-points may flow through a large set of interacting components, hosted on a variety of resources – and managed by local scheduling and queuing policies. A crucial step in the design process involves calibrating and tuning the local resource management policies, so that the original real-time objectives are achieved.

Real-time scheduling analysis is often used to help make this problem more tractable. Using the approach, upper bounds are derived for processing times and communication delays. Then, using these worst-case assumptions, tasks and messages can be deterministically scheduled to *guarantee* that all timing constraints will get met. Such constraints might include an individual thread's processing frequency, a packet's deadline, or perhaps the rate at which a network driver is run. In this type of system, hard real-time analysis can be used to help predict (and then ensure) that the performance objectives will be attained.

This approach is becoming increasingly difficult to carry out. First, achieving near-tight execution-time bounds is virtually impossible due to architectural features like superscalar pipelining, hierarchies of cache memory, etc. – not to mention the nondeterminism inherent in almost any network. Given this, as well as the fact that a program's actual execution time is data-dependent, a worst-case timing estimate may be several orders of magnitude greater than the average case. If one incorporates worst-case costs in a design, the result will often lead to an extremely under-utilized system.

Moreover, parameters like processing periods and deadlines are used to help achieve acceptable end-to-end performance – i.e., *as a means to an end*, and not an end in itself. In reality, missing a deadline will rarely lead to failure; in fact, such an occurrence should be *expected*, unless the system is radically over-engineered. While hard real-time scheduling theory provides a sufficient way to build an embedded system, it is not strictly necessary, and it may not yield the most efficient design.

In this paper we explore an alternative approach, by using statistical guarantees to generate cost-effective system designs. We model a real-time system as a set of task chains, with each task representing some activity requiring a specific CPU or network link. For example, a chain may correspond to the data path from a camera to a display in a video conferencing system, or reflect a servo-loop in a distributed real-time control system. The chain's real-time performance requirements are specified in terms of (i) a maximum acceptable propagation delay from input to output, and (ii) a minimum acceptable average throughput. In designing the system, we treat the first requirement as a hard constraint, that is, any end-to-end computation that takes longer than the maximum delay is treated a failure, and does not contribute to the overall throughput. (Some applications may be able to use late outputs – yet within the system model we currently do not count them.) In contrast, the second requirement is viewed in a statistical sense, and we design a system to exceed, on average, its minimal acceptable rate. We assume that a task's cost (i.e., execution time for a program, or delay for a network link) can be specified with any arbitrary discrete probability distribution function.

Problem and Solution Strategy. Given a set of task chains, each with its own real-time performance requirements, our objective is to design a system which will meet the requirements for all of the chains. Our system model includes the following assumptions, which help make the solution tractable:

- We model a task's load requirements stochastically, in terms of a discrete probability distribution function (or PDF), whose random variable characterizes the resource time needed for one execution instance of the task. Successive task instances are modeled to be independent of each other.
- We assume a static partitioning of the system resources; in other words, a task has already been allocated to a specific resource.

It is true that embedded systems design often involves making some task-placement decisions. However, we note that tuning the resource schedulers is, by definition, subservient to the allocation phase – which often involves accounting for device-specific localities (e.g., IO ports, DMA channels, etc.), as well as system-level issues (e.g., services provided by each node). While this paper's focus is narrowed to the scheduling synthesis problem at hand, we note that a "holistic" design tool could integrate the two problems, and use our system-tuning algorithms as "subroutines."

Also, while our objective is to achieve an overall, statistical level of real-time performance, we can still use the tools provided by hard real-time scheduling to help solve this problem. Our method involves the following steps: (1) assigning to each task a fixed proportion of its resource's load; and (2) determining the reasonable service interval (or frame) over which the proportion can be guaranteed. Then, using some techniques provided by real-time CPU and network scheduling, we can guarantee that during any such frame, a task will get at least its designated share of the resource's time. When that share fails to be sufficient for a currently running task to finish executing, it runs over into the next frame – and gets *that* frame's share, etc. Given this model, the design problem may be viewed as the following interrelated sub-problems:

(1) How should the CPU and network load be partitioned among the tasks, so that every chain's performance requirements are met?

(2) Given a load-assignment, how should the frame-sizes be set to maximize the effective output rate?

As we discuss in the sequel, load proportions cannot be quantized over infinitesimal time-frames. Hence, when a task's frame gets progressively smaller, it starts paying a large price for its guarantees – in the form of wasted overhead.

In this paper we present algorithms to solve these problems. The algorithm for problem (1) uses a heuristic to compare the relative needs of tasks from different chains competing for the same resources. The algorithm for problem (2) makes use of connecting Markov chains, to estimate the effective throughput of a given chain. Since the analysis is approximate, we validate the generated solution using a simulation model.

2 Related Work

Like much of the work in real-time systems, our results extend from preemptive, uniprocessor scheduling analysis. There are many old and new solutions to this problem (e.g., [1, 15, 20, 21]); moreover, many of these methods come equipped with offline, analysis tests, which determine *a priori* whether the underlying system is schedulable. Some of these tests are *load-oriented* sufficiency conditions – they predict that the tasks will always meet their deadlines, provided that the system utilization does not exceed a certain pre-defined threshold.

The classical model has been generalized to a large degree, and there now exist analogous results for distributed systems, network protocols, etc. For example, the model has been applied to distributed hard real-time systems in the following straightforward manner (e.g., see [26, 31]): each network connection is abstracted as a real-time task (sharing a network resource), and the scheduling analysis incorporates worst-case blocking times potentially suffered when high-priority packets have to wait for transmission of lower-priority packets. Then, to some extent, the resulting global scheduling problem can be solved as a set of interrelated local resource-scheduling problems.

In [30], the classical model was extended to consider probabilistic execution times on uniprocessor systems. This is done by giving a nominal "hard" amount of execution time to each task instance, under the assumption that the task will usually complete within this time. But if the nominal time is exceeded, the excess requirement is treated like a sporadic arrival (via a method similar to that used in [19]).

In our previous work [8, 9] we relaxed the precondition that period and deadline parameters are always known ahead of time. Rather, we used the system's end-to-end delay and jitter requirements to automatically *derive* each task's constraints; these, in turn, ensure that the end-to-end requirements will be met on a uniprocessor system. A similar approach for uniprocessor systems was explored in [2], where execution time budgets were automatically derived from the end-to-end delay requirements; the method used an imprecise computation technique as a metric to help gauge the "goodness" of candidate solutions.

These concepts were later modified for use in various application contexts. Recent results adapted the end-to-end theory to both discrete and continuous control problems (e.g. [18, 27], where real-time constraints were derived from a set of control laws, and where the objectives were to optimize the system's performance index while satisfying schedulability. Our original approach (from [8, 9]) was also used to produce schedules for real-time traffic over fieldbus networks [6, 7], where the switch priorities are synthesized to ensure end-to-end rate and latency guarantees. A related idea was pursued for radar processing domains [11], where an optimization method produces per-component processing rates and deadlines, based on the system's input pulse rate and its prescribed allowed latency.

End-to-end design becomes significantly more difficult in distributed contexts. Solving this problem usually involves finding an answer to the following question: "Given an end-to-end latency budget, what is the optimal way to spend this budget on each pipeline hop?" Aside the complexity of the basic decision problem, a solution also involves the practical issue of getting the local runtime schedulers to guarantee their piece-wise latencies. Results presented in [25] address this problem in a deterministic context: they extend our original uniprocessor method from [8] to distributed systems, by statically partitioning the end-to-end delays via heuristic optimization metrics [25]. Similar approaches have been proposed for "soft" transactions in distributed systems [17], where each transaction's deadline is partitioned between the system's resources.

To our knowledge, this paper presents the first technique that achieves statistical real-time performance in a distributed system, by using end-to-end requirements to *assign* both periods and the execution time budgets. In this light, our method should be viewed less as a scheduling tool (it is not one), and more as an approach to the problem of real-time systems design.

To accomplish this goal, we assume an underlying runtime system that is capable of the following: (1) decreasing a task's completion time by increasing its resource share; (2) enforcing, for each resource, the proportional shares allocated for every task, up to some minimum quantization; and (3) within these constraints, isolating a task's real-time behavior from the other activities sharing its resource. In this regard, we build on many results that have been developed for providing OS-level reservation guarantees, and for rate-based, proportional-share queuing in networks. Since these concepts are integral to understanding the work in this paper, we treat them at some length.

In rate-based methods, tasks get allocated percentages of the available bandwidth. Obviously these percentages are cannot be maintained over infinitesimal time-intervals; rather, the proportional-shares are serviced in an approximate sense — i.e., within some margin of error. The magnitude of the error is usually due to the following factors: (1) *quantization* (i.e., the degree to which the underlying system can multiplex traffic); and (2) *priority-selection* (i.e., the order in which tasks are selected for service). At higher levels quantization, and as multiple streams share the same FIFO queues, the more service-orders depart from true proportional-sharing.

Our analytical results rely on perhaps the oldest known variant of rate-based scheduling – time-division multiplexing, or TDM. In our TDM abstraction, a task is guaranteed a fixed number of "time-slots" over pre-defined periodic intervals (which we call frames). Our analytical techniques assume tasks have their time-slots *reserved*; i.e., if a task does not claim its load, the load gets wasted. We appeal to TDM for a basic reason: we need to handle an inherently stochastic workload model, in which tasks internally "decide" how much load they will need for specific instances. These load demands can be be quite high for arbitrary instances; they may be minuscule for other instances. Moreover, once a task is started, we assume that its semantics mandate that it also needs to be finished. So, since unregulated workloads cannot simply be "re-shaped," and since end-to-end latency guarantees still must be guaranteed, TDM ensures a reasonable level of fairness between different tasks on a resource – and between successive instances of the same task.

The downside of this scheme is that TDM ends up wasting unused load. Other rate-based disciplines solve this problem by re-distributing service over longer intervals – at a cost of occasionally postponing the projected completion times of certain tasks. Most of these disciplines, however, were conceived for inherently regulated workload models, e.g., linear bounded arrival processes [3]. In such settings, transient unfairnessis is often smoothed out by simply "re-shaping" the departure process – i.e., by inserting delay stages.

Many algorithms have been developed to provide proportional-share service in high-speed networks, including the "Virtual Clock Method" [38], "Fair-Share Queuing" [4], "Generalized Processor Sharing" (or GPS) [23], and "Rate Controlled Static Priority Queuing" (or RCSP) [36]. These models have also been used to derive statistical delay guarantees; in particular, within the framework of RCSP (in [37]) and GPS (in [39]). Related results can be found in [5] (using a policy like "Virtual Clock" [38]), and in [34] (for FCFS, with a variety of traffic distributions). In [14], statistical service quality objectives are achieved via proportional-share queuing, in conjunction with server-guided backoff, where servers dynamically adjust their rates to help utilize the available bandwidth.

Recently, many of these rate-based disciplines have sprouted analogues for CPU scheduling. For example, Waldspurger et al. [32] proposed "Lottery Scheduling," which multiplexes available CPU load based on the relative throughput rates for the system's constituent tasks. The same authors also presented a deterministic variant of this, called "Stride Scheduling" [33]; this method provides an OS-level server for a method similar to the Weighted-Fair-Queuing (WFQ) discipline used in switches. WFQ – also known as "Packetized GPS" (or PGPS) – is a discrete, quantized version of the fluid-flow abstraction used in GPS. Scheduling decisions in WFQ are made via "simulating" proportional-sharing for the tasks on the ready-queue, under an idealized model of continuous-time multiplexing. The task which would hypothetically finish first under GPS gets the highest priority – and is put on the run queue (until the next scheduling round). Stoica et al. [29] proposed a related technique, which similarly uses a "virtual time-line" to determine the runtime dispatching order. This concept was also applied for hierarchical scheduling in [13], where multiple classes of tasks (e.g., hard and soft real-time applications) can coexist in the same system.

Several schemes have been proposed to guarantee processor capacity shares for the system's real-time tasks, and to simultaneously isolate them from overruns caused by other tasks in the system. For example, Mercer et al. [22] proposed a *processor capacity reservation* mechanism to achieve this, a method which enforces each task's reserved share within its reservation period, under micro-kernel control. Also, Shin et al. [28] proposed a reservation-based algorithm to guarantee the performance of periodic real-time tasks, and also to improve the schedulability of aperiodic tasks.

As noted above, many proportional-share methods have been subjected to response-time studies, for different types of arrival processes. This has been done for switches, CPUs, and for entire networks. Note that the problem of *determining* aggregate delay in a network is dual to the problem of *assigning* per-hop delays to achieve some end-to-end deadline. The latter is a "top-down" approach: the designer "tells" the network what its per-hop latencies *should be*, and then the network needs to guarantee those latencies. Delay analysis works in a "bottom-up" fashion: the network basically "tells" the user what the end-to-end delay *will be*, given the proportional-share allocations for the chain under observation.

While seemingly different, these problems are inextricably related. "Top-down" deadline-partitioning could not function without some way of getting "bottom-up" feedback. Similarly, the "bottom-up" method assumes a pre-allocated load for the chain – which, in reality, is negotiated to meet the chain's end-to-end latency and throughput requirements. In real-time domains, solving one problem requires solving the other.

Deriving the end-to-end latency involves answering the following question: "If my chain flows through N nodes, each of which is managed by a rate-based discipline, what will the end-to-end response time be?" This issue is quite simple when all arrival process are Poisson streams, service times are exponentially distributed, and all nodes use a FIFO service discipline. For a simple Jackson queuing network like this, many straightforward product-form techniques can be applied.

The question gets trickier for linearly regulated traffic, where each stream has a different arrival rate, with deviations bounded over different interval-sizes, and where each stream has different proportional service guarantees. Fortunately, compositional results do exist, and have been presented for various rate-based disciplines – for both deterministic [12, 24, 35] and statistical [39, 37] workloads. Deterministic, end-to-end per connection delays were considered in [24] for leaky-bucket regulated traffic, using the PGPS scheduling technique. In [35] a similar study was performed using a non-work-conserving service discipline. Also, as noted above, statistical treatments have been provided for the PGPS [39] and for RCSP [37].

In Section 4 we present an analytical approximation for our TDM abstraction – perhaps the extreme case of a non-work-conserving discipline. The method is used to estimates end-to-end delays over products of TDM queues; where a chain's load demand at a node can be generally distributed; where all tasks in a chain can have different PDFs; and where queue sizes are constrained to a single slot.

This problem is innately complex. Moreover, our design algorithm needs to test huge numbers of solution candidates before achieving the system objectives. Hence, the delay analysis should be fast – and will consequently be coarse. At the same time, it cannot be too coarse; after all, it must be sufficiently accurate to expose key performance trends over the entire solution space. As shown in Section 4, we approach this problem in a compositional, top-down fashion. Our algorithm starts by analyzing a chain's head task in isolation. The resulting statistics are then used to help analyze the second task, etc., down the line, until delay and throughput metrics are obtained for the chain's output task – and hence, for the chain as a whole.

3 Model and Solution Overview

As stated above, we model a system as set of independent, pipelined task chains, with every task mapped to a designated CPU or network resource. The chain abstraction can, for example, capture the essential data path structure of a video conferencing system, or a distributed process control loop. Formally, a system possesses the following structure and constraints.

Bounded-Capacity Resources: There is a set of resources $r_1, r_2, \ldots, r_m$, where a given resource r_i corresponds to one of the system's CPUs or network links. Associated with r_i is a maximum allowable capacity, or ρ_i^m, which is the maximum load the resource can multiplex effectively. The parameter ρ_i^m will typically be a function of its scheduling policy (as in the case of a workstation),

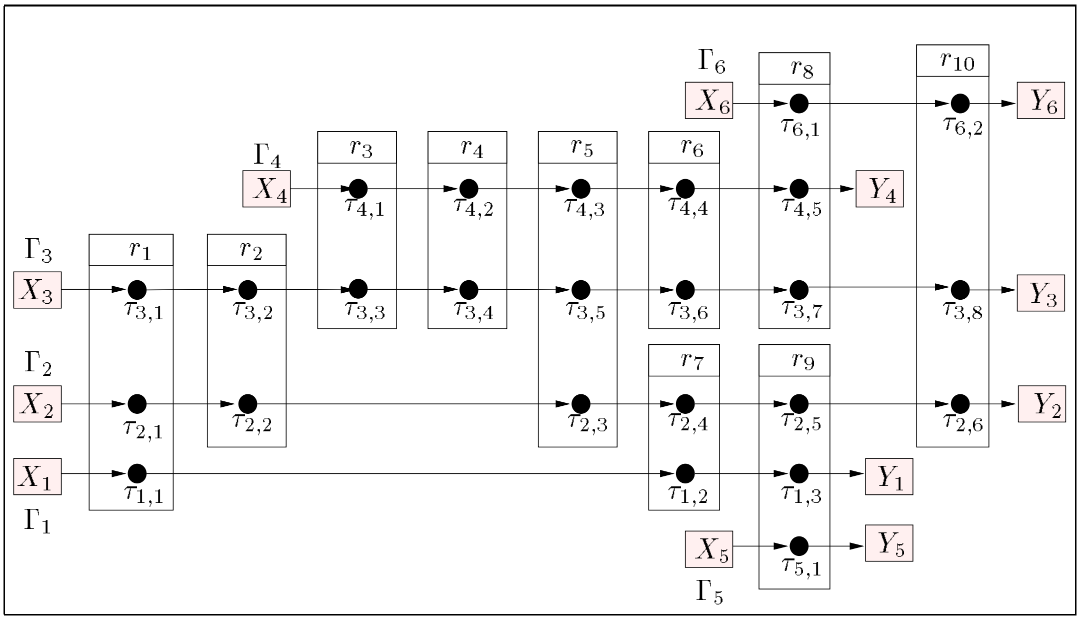


Figure 1: Example System Topology.

or its switching and arbitration policies (in the case of a LAN).

Task Chains: A system has n task chains, denoted $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$, where the j^{th} task in a chain Γ_i is denoted $\tau_{i,j}$. Each computation on Γ_i carries out an end-to-end transformation from its external input X_i to its output Y_i. Also, a producer/consumer relationship exists between each connected pair of tasks $\tau_{i,j-1}$ and $\tau_{i,j}$, and we assume a one-slot buffer between each such pair, since the queuing policy chooses only the newest data in the buffer. Hence a producer may overwrite buffered data which is not consumed.

Stochastic Processing Costs: A task's cost is modeled via a discrete probability distribution function, whose random variable characterizes the time it needs for one execution instance on its resource.

Maximum Delay Bounds: Γ_i's delay constraint, $\mathrm{MD_i}$, is an upper bound on the time it should take for a computation to flow through the system, and still produce a useful result. For example, if $\mathrm{MD_i} = 500ms$, it means that if Γ_i produces an output at time t, it will only be used if the corresponding input is sampled no earlier than $t - 500ms$. Computed results that exceed this propagation delay are dropped.

Minimum Output Rates: Γ_i's rate constraint, $\mathrm{MOR_i}$, specifies a minimum acceptable average rate for outputs which meet their delay constraints. For example, if $\mathrm{MOR_i} = 10Hz$, it means that the chain Γ_i must, on average, produce 10 outputs per second. Moreover, $\mathrm{MOR_i}$ implicitly specifies the maximum possible frame-size for the tasks in Γ_i; e.g., if $\mathrm{MOR_i} = 10Hz$, than Γ_i's maximum frame-size is 0.1 – which would suffice only if an output were produced during every frame.

An Example. Consider the example shown in Figure 1, which possesses six chains, labeled Γ_1-Γ_6. Here, rectangles denote shared resources, black circles denote tasks, and the shaded boxes are

Resource Usage PDFs

For Tasks	Derived From	E[t] (ms)	Var[t]	[Min,Max]	NumSteps
$\tau_{1,1}, \tau_{3,2}, \tau_{3,5}, \tau_{4,1}, \tau_{6,1}$	Normal	10	64	[4,35]	10
$\tau_{1,3}, \tau_{2,3}, \tau_{3,6}, \tau_{3,8}, \tau_{4,2}$	Normal	20	100	[10,50]	20
$\tau_{2,1}, \tau_{3,4}, \tau_{3,7}, \tau_{4,3}, \tau_{5,1}$	Exp	10		[0,100]	30
$\tau_{2,4}, \tau_{2,6}, \tau_{3,1}, \tau_{4,5}, \tau_{6,2}$	Exp	20		[0, 200]	50
$\tau_{1,2}, \tau_{2,2}, \tau_{2,5}, \tau_{3,3}, \tau_{4,4}$	Normal	8	144	[2,48]	20

End-To-End Constraints

Γ_i	$\mathrm{MD_i}$ (ms)	$\mathrm{MOR_i}$ (Hz)
Γ_1	240	10
Γ_2	1000	5
Γ_3	1000	5
Γ_4	700	5
Γ_5	100	5
Γ_6	300	5

Maximum Resource Utilizations:

ρ_1^m	ρ_2^m	ρ_3^m	ρ_4^m	ρ_5^m	ρ_6^m	ρ_7^m	ρ_8^m	ρ_9^m	ρ_{10}^m
0.9	0.7	0.9	0.6	0.9	0.7	0.7	0.8	0.9	0.9

Figure 2: Constraints in Example.

external inputs and outputs. The system's resource requirements and end-to-end constraints are shown in Figure 2.

In any system, a task's load demand varies stochastically, due to second-order effects like cache memory behavior, DMA interference, pipeline stalls, bus arbitration delays, transient head-of-line blocking, etc. By using one random variable to model a task's load, we essentially collapse all these residual effects into a single PDF, which also accounts for the task's idealized "best-case" execution-time. In our model, any discrete probability distribution can be used for this purpose.

Two points should be articulated here, the first of which is fairly obvious: If the per-task load models are ill-founded, then the synthesis results will be of little use. Indeed, in some situations one cannot just convolve all the abovementioned effects into a single PDF. While it may be tempting to stochastically bundle "driver processing interference" with a task's load model – and while often one can do just that – in other situations, this sort of factor needs to be represented explicitly, as a task. (Our method can easily accommodate this alternative.) The process of obtaining a good model abstraction is non-trivial; it requires accounting for matters like *causality* (i.e., charging load deviations to the tasks which cause them), *scale* (i.e., comparing the task's loading statistics – mean and variance – to those of the residual effects charged to it), and *sensitivity* (i.e., statistically gauging the effect of load quantization on end-process results). These problems are well outside the scope of this paper; interested readers should consult [16] for a decent introduction to statistical performance modeling. However, while none of these problems is trivial, given sufficient time, patience and statistical competence, one can employ some standard techniques for handling all of them.

The second point is a bit more subtle, though equally true: For the purposes of design, a coarse load model – represented with a single stationary distribution – is better than no load model. In this sort of systems design, one needs to start with some basic notion of the per-task performance. The alternative is designing the system based on blind guesswork.

Consider a task representing an inner-product function, where the vector-sizes can range between 10,000-20,000 elements, and where the actual vector-sizes change dynamically. Assume we profile the task on a representative data-set, and the resulting load demands appear to track the inner two quartiles of a normal distribution. Can we just quantize the resulting histogram, and use it to model the task's PDF? Assume we cannot explicitly model vector-size as a controlled variable. Perhaps its value is truly nondeterministic. Perhaps we just choose to treat it that way, since its addition contributes little marginal accuracy to the pipeline's estimated departure process statistics. In such cases, the quasi-normal distribution may, in fact, be the best model we can achieve for our inner-product. And, it may be sufficient for the purposes of the design, and lead to reasonable results in the real system. Our objective is to use these models as *a tool* – to abstractly predict trends in our synthesis algorithms. Our objective is not to directly represent the system itself.

To obtain a load model, one can often appeal to the method outlined above, a method commonly used in Hardware/Software Co-Design: a task is profiled in isolation, and the resulting histogram gets post-processed into a stationary response-time distribution. (We have experienced good results via this method in our multi-platform simulation work on digital video playout systems [10]). A second technique can be used at a more preliminary stage: the designer can coarsely estimate a task's average load, and use it to create a synthetic distribution – e.g., exponential, normal, chi-square, etc. Such load models would correspond to hypotheses on how a task might behave in the integrated system. When this is done for all tasks – with the PDFs quantized at some acceptable level – the results can be fed into the synthesis tool. If the system topology can handle a number of hypothetical per-task load profiles, a designer can gain a margin of confidence in the system's robustness to subtle changes in loading conditions.

We take the latter approach in our running example: we discretize two very different continuous distributions, and then re-use the results for multiple tasks. In Figure 2, "Derived From" denotes a base continuous distribution generated and quantized using the parameters Min, Max, E$[t]$ (mean), Var$[t]$ (variance) and "NumSteps" (number of intervals). In the case of an exponential distribution, the CDF curve is shifted up to "Min," and the probabilities given to values past "Max" are distributed to each interval proportionally. The granularity of discretization is controlled by "NumSteps," where we assume that execution time associated with an interval is the maximum possible value within that interval. For example, the time requirement for $\tau_{1,1}$ follows normal distribution, with 10ms mean, 8ms standard deviation, a minimum time of 4ms, and a maximum time of 35 ms. The continuous distribution is chopped into 10 intervals; since the first interval contains all times within $[4ms, 7ms]$, we associate a time of $7ms$ with this interval, and give it the corresponding portion of the continuous CDF.

Note that if we attempt a hard real-time approach here, no solution would be possible. For example, $\tau_{3,1}$ requires up to 200ms dedicated resource time to run, which means that $\tau_{3,1}$'s frame must be no less than 200ms. Yet $\mathrm{MOR}_3 = 5$, i.e., Γ_3 must produce a new output at least 5 times

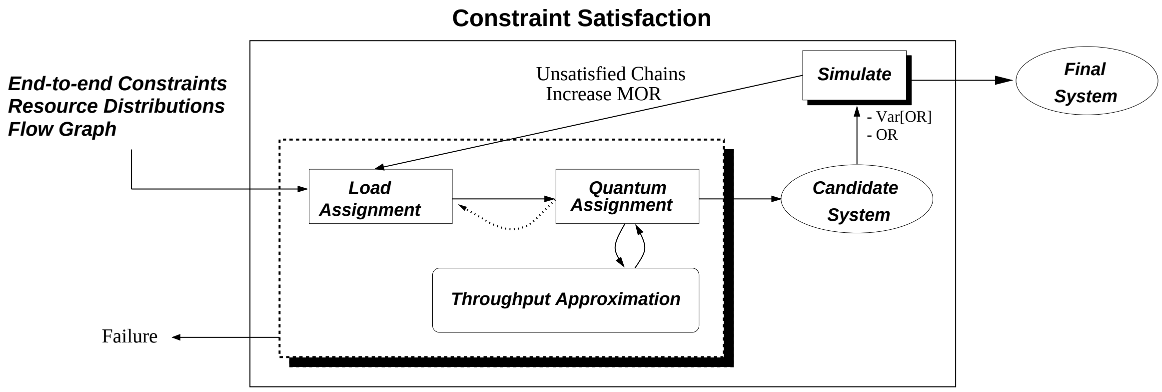


Figure 3: Design Overview

per second. This, in turn, means $\tau_{3,1}$'s frame can also be *no greater than* 200ms. But if the frame is exactly 200ms, the task induces a utilization of 1.0 on resource r_1 – exceeding the resource's intrinsic 0.9 limit, and disallowing any capacity for other tasks hosted on it.

3.1 Run-Time Model

Within the system model, all tasks in chain Γ_i are considered to be scheduled in a quasi-cyclic fashion, using a time-division multiplexing abstraction for resource-sharing, over $\mathrm{F_i}$-sized frames. That is, all of Γ_i's load-shares are guaranteed for $\mathrm{F_i}$ intervals on all constituent resources. Hence, the synthesis algorithm's job is to (1) assign each task $\tau_{i,j}$ a proportion of its resource's capacity (which we denote as $u_{i,j}$) and (2) assign a global $\mathrm{F_i}$ frame for Γ_i. Given this, $\tau_{i,j}$'s runtime behavior can be described as follows:

(1) Within every $\mathrm{F_i}$ frame, $\tau_{i,j}$ can use up to $u_{i,j}$ of its resource's capacity. This is policed by assigning $\tau_{i,j}$ an execution-time budget $E_{i,j} = \lfloor u_{i,j} \times \mathrm{F_i} \rfloor$; that is, $E_{i,j}$ is an upper bound on the amount of resource time provided within each $\mathrm{F_i}$ frame, truncated to discrete units. (We assume that the system cannot keep track of arbitrarily fine granularities of time.)

(2) A particular execution instance of $\tau_{i,j}$ may require multiple frames to complete, with $E_{i,j}$ of its running time expended in each frame.

(3) A new instance of $\tau_{i,j}$ will be started within a frame if no previous instance of $\tau_{i,j}$ is still running, and if $\tau_{i,j}$'s input buffer contains new data that has not already exceeded $\mathrm{MD_i}$. This is policed by a time-stamp mechanism, put on the computation when its input is sampled.

Due to a chain's pipeline structure, note that if there are n_i tasks in Γ_i, then we must have that $\mathrm{MD_i} \geq n_i \times \mathrm{F_i}$, since data has to flow through all n_i elements to produce a computation.

3.2 Solution Overview

A schematic of the design process is illustrated in Figure 3, where the main steps are as follows. (1) partitioning the CPU and network capacity between the tasks; (2) selecting each chain's frame to optimize its output rate; and (3) checking the solution via simulation, to verify the integrity of the approximations used, and to ensure that each chain's output profile is sufficiently smooth (i.e., not bursty).

The partitioning algorithm processes each chain Γ_i and finds a candidate load-assignment vector for it, denoted $\mathbf{u_i}$. (An element $u_{i,j}$ in $\mathbf{u_i}$ contains the load allocated to $\tau_{i,j}$ on its resource.) Given a load assignment for Γ_i, the synthesis algorithm attempts to find a frame $\mathrm{F_i}$ at which Γ_i achieves its optimal output rate. This computation is done approximately: For a given $\mathrm{F_i}$, a rate estimate is derived by (1) treating all of Γ_i's outputs uniformly, and deriving an i.i.d. per-frame "success probability" ξ_i; and (2) then simply multiplying $\xi_i \times \frac{1}{\mathrm{F_i}}$ to approximate the chain's output rate, $\mathrm{OR_i}$. If $\mathrm{OR_i}$ is lower than the $\mathrm{MOR_i}$ requirement, the load assignment vector is increased, and so on. Finally, if sufficient load is found for all chains, the resulting system is simulated to ensure that the approximations were sound – after which excess capacity can be given to any chain, with the hope of improving its overall rate.

4 Throughput Analysis

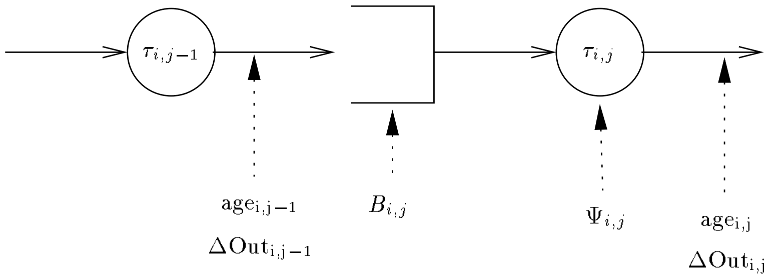
In this section we describe how we approximate Γ_i's output rate, $\mathrm{OR_i}$, given candidate load and frame parameters ($\mathrm{F_i}$ and $\mathbf{u}_\mathrm{i}$) for the chain. Then in Section 5, we show how we make use of this technique to derive all the system's $\mathrm{F_i}$ and $\mathbf{u}_\mathrm{i}$ parameters for every chain.

Assume we are currently processing Γ_i, which has some frame-size $\mathrm{F_i}$ and load vector $\mathbf{u}_\mathrm{i}$. How do we estimate its output probability ξ_i? Recall that outputs exceeding the maximum allowed delay $\mathrm{MD_i}$ are not counted – and hence, we need some way of determining latency through the system. One benefit of proportional-share queuing is as follows: Since each chain is effectively isolated from others over $\mathrm{F_i}$ intervals of observation, we can analyze the behavior of Γ_i independently, without worrying about head-of-line blocking effects from other components.

We use a discrete-time model, in which the time units are in terms of a chain's frame-size; i.e., our discrete domain $\{0, 1, 2, \ldots\}$ corresponds to the real times $\{0, \mathrm{F_i}, 2\mathrm{F_i}, \ldots\}$. Not only does this reduction make the analysis more tractable, it also corresponds to "worst-case" conditions: Since the underlying system may schedule a task execution at any time within a $\mathrm{F_i}$ frame, we assume that input may be read as early as the beginning of a frame, and output may be produced as late as the end of a frame. And with one exception, a chain's states of interest do occur at its frame boundaries. That exception is in modeling aggregate delay – which, in our discrete time domain we treat as $\lfloor \frac{\mathrm{MD_i}}{\mathrm{F_i}} \rfloor$. Hence the fractional part of the last frame is ignored, leading to a tighter notion of success (and consequently erring on the side conservatism).

Theoretically, we could model a computation's delay by constructing a stochastic process for the chain as a whole – and solving it for all possible delay probabilities. But this would probably be a futile venture for even smaller chains; after all, such a model would have to simultaneously keep track of each task's local behavior. And since a chain may hold as few as 0 ongoing computations, and as many as one computation per task, it's easy to see how the state-space would quickly explode.

Instead, we go about constructing a model in a compositional (albeit inexact) manner, by processing each task locally, and using the results for its successors. Consider the following diagram, which portrays the flow of a computation at a single task:



These random variables are defined as follows:

1. *Data-age* ($\mathrm{age_{i,j}}$): This variable charts a computation's total accumulated time, from entering Γ_i's head, to leaving $\tau_{i,j}$.

2. *Blocking time* ($B_{i,j}$): The duration of time an input is buffered, waiting for $\tau_{i,j}$ to complete its current computation.

3. *Processing time* ($\Psi_{i,j}$): If $\mathrm{t_{i,j}}$ is a random variable ranging over $\tau_{i,j}$'s PDF, then $\Psi_{i,j} \stackrel{\text{def}}{=} \lceil \frac{\mathrm{t_{i,j}}}{E_{i,j}} \rceil$ is the corresponding variable in units of frames.

4. *Inter-output time* ($\Delta\mathrm{Out_{i,j}}$): An approximation of $\tau_{i,j}$'s inter-output distribution, in terms of frames; it measures the time between two successive outputs.

We assume data is always ready at the chain's head; hence $\mathrm{age_{i,j}}$ can be *approximated* via the following recurrence relation:

$$\begin{aligned} \mathrm{age_{i,1}} &\cong \Psi_{i,1} \\ \mathrm{age_{i,j}} &\cong \mathrm{age_{i,j-1}} + B_{i,j} + \Psi_{i,j} \end{aligned}$$

And for $j > 1$, we approximate the entire $\mathrm{age_{i,j}}$ distribution by assuming the three variables to be independent, i.e.,

$$\Pr[\mathrm{age_{i,j}} = k] \cong \sum_{k_1+k_2+k_3=k} \Pr[\mathrm{age_{i,j-1}} = k_1] \times \Pr[B_{i,j} = k_2] \times \Pr[\Psi_{i,j} = k_3]$$

Note that $\tau_{i,j}$'s success probability, $\xi_{i,j}$, will then be $\frac{1}{E[\Delta\mathrm{Out_{i,j}}]}$, i.e., the probability that a (non-stale) output is produced during a random frame. After processing the final task in the chain, $\tau_{i,n}$, we can approximate the end-to-end success probability – which is just $\tau_{i,n}$'s output probability, appropriately scaled by the probability of excessive delay injected during $\tau_{i,n}$'s execution:

$$\xi_i \cong \xi_{i,n} \times \Pr[\mathrm{age_{i,n}} \le d_i]$$

At this point the end-to-end success rate is estimated as $\mathrm{OR_i} = \xi_i \times \frac{1}{\mathrm{F_i}}$.

Note that our method is "top-down," i.e., statistics are derived for $\tau_{i,1}$, then $\tau_{i,2}$ (using the synthesized metrics from $\tau_{i,1}$), then $\tau_{i,3}$, etc. Also note that when processing $\tau_{i,1}$, we already have

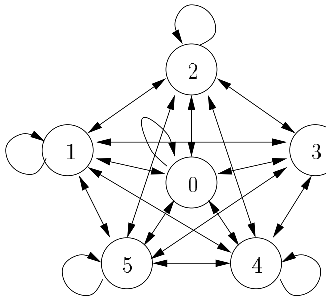


Figure 4: Chain $X_{i,j}(t)$, $max(\Psi_{i,j}) = 6$.

all the information we require – from its PDF. In other words, we trivially have $\Pr[\Delta\mathrm{Out_{i,1}} = t] = \Pr[\Psi_{i,1} = t]$, and thus $\xi_{i,1}$ is $\frac{1}{E[\Psi_{i,1}]}$. Since $\tau_{i,1}$ can retrieve fresh input whenever it is ready (and therefore incurs no blocking time), $\tau_{i,1}$ can execute a new phase whenever it finishes with the previous phase.

Blocking-Time. Obtaining reasonable blocking-time metrics at each stage is a non-trivial affair, especially when longer-tailed distributions are involved. In carrying out the analysis, we synthesize a stochastic process – whose states describe $\tau_{i,j}$'s remaining busy-time on a random input arrival. We describe the structure via a simple Markov Chain, $X_{i,j}(t)$, which describes random states of $\tau_{i,j}$. Specifically, we are interested in capturing (1) $\tau_{i,j}$'s remaining execution time until the next task instance; and (2) whether an input is to be processed or dropped.

$X_{i,j}(t)$'s transitions can be described using a simple Markov Chain, as shown in Figure 4 (where the maximum execution time is $6\mathrm{F_i}$.) The transitions are event-based, i.e., they are triggered by new inputs from $\tau_{i,j-1}$. On the other hand, states measure the remaining time left in a current execution, if there is any. In essence, moving from state k to l denotes that (1) a new input just received; and (2) it will induce blocking time of l frames. For the sake of our analysis, we distinguish between three different outcomes on moving from state k to state l. In the transition descriptions, we use the term d_i to denote the end-to-end delay bound, in terms of frames, i.e., $d_i = \lfloor \frac{\mathrm{MD_i}}{\mathrm{F_i}} \rfloor$.

(1) Dropping $[k \to l]$: The task is currently executing, and there is already another input queued up in its buffer – which was calculated to induce a blocking-time of k. The new input will overwrite it, and induce l blocking-time frames.

$$P^d[k \to l] \quad \cong \quad \Pr[\Delta\mathrm{Out_{i,j-1}} = k - l] \; (k > l \geq 0)$$

(2) Failure $[k \to 0]$: A new input arrives, but it will be too stale to get processed by the finish-time of the current execution.

$$P^f[k \to 0] \quad \cong \quad \Pr[\Delta\mathrm{Out_{i,j-1}} > k] \times \Pr[\mathrm{age_{i,j-1}} > d_i - k]$$

(3) Success $[k \to l]$: A new input arrives, and will get processed with blocking time l. Figure 5 illustrates a case of a successful transition.

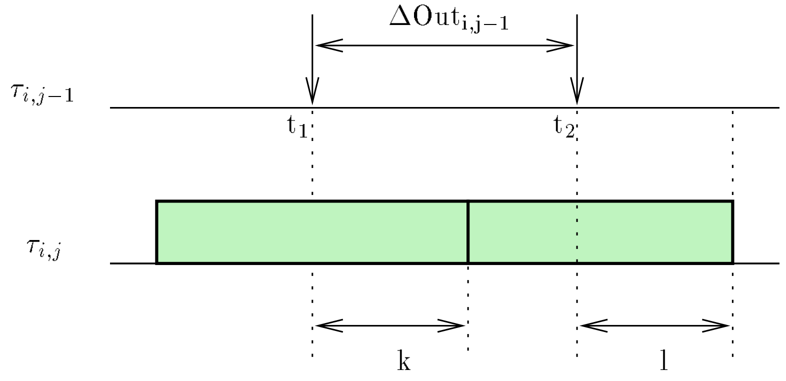


Figure 5: Transition $k \rightarrow l$, with $0 < k$, $0 < l$.

Case A: Destination state is 0.

$$P^s[k \rightarrow 0] \cong \sum_{t>0} \Pr[\Psi_{i,j} = t] \times \Pr[\Delta\mathrm{Out_{i,j-1}} \geq t + k] \times \Pr[\mathrm{age_{i,j-1}} \leq d_i - k]$$

Case B: Destination state $l > 0$:

$$P^s[k \rightarrow l] \cong \sum_{t>l} \Pr[\Psi_{i,j} = t] \times \Pr[\Delta\mathrm{Out_{i,j-1}} = t + k - l] \times \Pr[\mathrm{age_{i,j-1}} \leq d_i - k]$$

We distinguish between outcomes (1)-(3) via partitioning the state-transition matrix – i.e., $\mathbf{P}^d$, $\mathbf{P}^f$ and $\mathbf{P}^s$ denote the transition matrices for dropping, failure and success, respectively. Each is calculated in terms of parameters we discussed above – i.e., $\mathrm{age_{i,j-1}}$, $\Delta\mathrm{Out_{i,j-1}}$ and $\Psi_{i,j}$.

After getting the complete transition matrix, $\mathbf{P} = \mathbf{P}^d + \mathbf{P}^f + \mathbf{P}^s$, we solve for steady-state probabilities in the usual fashion, i.e., for $\mathbf{x}_{i,j} = \mathbf{x}_{i,j} \times \mathbf{P}$. In turn, the steady-state probabilities are used to derive $\tau_{i,j}$'s per-frame success probability

$$\xi_{i,j} \cong \xi_{i,j-1} \times \left(\sum_{k \geq 0} (\mathbf{x}_{i,j} \times \mathbf{P}^s)[k] \right)$$

where the k index denotes the kth element in the resulting vector. In essense, the calculation just computes the probability of (1) having an input to read at a random frame, and (2) successfully processing it – which is obtained by summing up the successful out-flow probabilities. The same simple Bayesean method is used to achieve a stationary blocking-time PDF:

$$\Pr[B_{i,j} = k] \cong \frac{\sum_{l \geq 0} (\mathbf{x}_{i,j}[k] \times \mathbf{P}^s[k, l])}{\sum_{k \geq 0} (\mathbf{x}_{i,j} \times \mathbf{P}^s)[k]}$$

The final ingredient is to derive task $\tau_{i,j}$'s inter-output distribution, $\Delta\mathrm{Out_{i,j}}$. To do this we use a coarse mean-value analysis: After $\tau_{i,j}$ produces an output, we know that it goes through an idle phase (waiting for fresh input from its producer), followed by a busy phase, culminating in another output. Let $\mathrm{Idle_{i,j}}$ be a random variable which counts the number of idle cycles before the busy phase. Then we have:

$$\begin{aligned} E[\Psi_{i,j} + \mathrm{Idle_{i,j}}] &= \frac{1}{\xi_{i,j}} \\ \Longrightarrow \qquad E[\mathrm{Idle_{i,j}}] &= \frac{1}{\xi_{i,j}} - E[\Psi_{i,j}] \end{aligned}$$

Using this information, we model the event denoting "compute-start" as a pure Bernoulli decision in probability $\mathrm{ST}_{i,j}$, where $\mathrm{ST}_{i,j} = \frac{1}{E[\mathrm{Idle_{i,j}}]+1}$, i.e., after an output has been delivered, $\mathrm{ST}_{i,j}$ is the probability that a random cycle starts the next busy phase. We then approximate the idle durations via a modified geometric distribution:

$$\Pr[\mathrm{Idle_{i,j}} = l] \cong (1 - \mathrm{ST}_{i,j})^l \times \mathrm{ST}_{i,j}$$

Then we derive the distribution for $\Pr[\Delta\mathrm{Out_{i,j}}]$ as:

$$\Pr[\Delta\mathrm{Out_{i,j}} = k] \cong \sum_{0 \le k_1 < k} \Pr[\mathrm{Idle_{i,j}} = k_1] \times \Pr[\Psi_{i,j} = k - k_1]$$

Finally, we approximate the end-to-end success probability, which is just $\tau_{i,n}$'s output probability, appropriately scaled to account for excessive delay injected during $\tau_{i,n}$'s execution:

$$\xi_i \stackrel{\mathrm{def}}{=} \xi_{i,n} \times \Pr[\mathrm{age_{i,n}} \le d_i]$$

Hence, by definition, the end-to-end output rate is given as follows:

$$\mathrm{OR_i} = \xi_i \times \frac{1}{\mathrm{F_i}}$$

Example. As an example, we perform throughput estimation on Γ_6, assuming system parameters of $\mathrm{F}_6 = 60\mathrm{ms}$, $E_{6,1} = 6\mathrm{ms}$, and $E_{6,2} = 30\mathrm{ms}$. Recall that the delay bound for the chain is $\mathrm{MD}_6 = 300$; thus $d_6 = \lfloor \mathrm{MD}_6/\mathrm{F}_6 \rfloor = 5$. Within our head-to-tail approach, we first have to consider task $\tau_{6,1}$. Recall, however, that the distributions for both $\mathrm{age}_{6,1}$ and $\Delta\mathrm{Out}_{6,1}$ are identical to $\Psi_{6,1}$, the quantized load distribution. Moreover, we also have that $\xi_{6,1} = \frac{1}{E[\Psi_{6,1}]} = 0.3291$.

Next we consider the second (and last) task $\tau_{6,2}$. The following tables show the PDFs for $\Psi_{6,2}$, for the Markov Chain's steady states, the blocking times, and for $\mathrm{age}_{6,2}$.

k	$\Pr[\Psi_{6,2}]$	$\mathbf{x}_{6,2}[0]$	$\Pr[B_{6,2}]$	$\Pr[\text{age}_{6,2}]$
0	0.0	0.975	0.980	0.0
1	0.7543	0.019	0.017	0.0
2	0.1968	0.0045	0.002	0.0
3	0.03751	0.0009	0.00009	0.2658
4	0.0098	0.0002	0.0	0.3400
5	0.0019	0.00003	0.0	0.2446
6	0.0005	0.000001	0.0	0.1153
7	0.00008	0.0	0.0	0.0269
8	0.0	0.0	0.0	5.7×10^{-3}
9	0.0	0.0	0.0	1.3×10^{-3}
10	0.0	0.0	0.0	2.7×10^{-4}
11	0.0	0.0	0.0	5.3×10^{-5}
12	0.0	0.0	0.0	5.8×10^{-6}

Now, summing up the successful out-flow probabilities, we have

$$\sum_{k \geq 0} (\mathbf{x}_{6,2} \times \mathbf{P}^s)[k] = 0.9804$$

and hence, the chain's i.i.d success probability is defined as follows:

$$\begin{aligned} \xi_{6,2} &= \xi_{6,1} \times 0.9804 = 0.3228 \\ \Pr[\text{age}_{6,2} \leq d_6] &= 0.850 \\ \xi_6 &= \xi_{6,2} \times \Pr[\text{age}_{6,2} \leq d_6] = 0.2745 \end{aligned}$$

Multiplying by the frame-rate, we get $\text{OR}_6 = 0.2745 \times \frac{1000}{60} = 4.574$.

5 System Design Process

We now revisit the "high-level" problem of determining the system's parameters, with the objective of satisfying each chain's performance requirements. As stated in the introduction, the design problem may be viewed as two inter-related sub-problems:

1. **Load Assignment.** Given a set of chains, how should the CPU and network load be partitioned among the set of tasks, so that the performance requirements are met?

2. **Frame Assignment.** Given a load-assignment to the tasks in the chain, what is the optimal frame for the chain, such that the effective throughput is maximized?

Note that load-allocation is the main "inter-chain" problem here, while frame-assignment can be viewed strictly as an "intra-chain" issue. With our time-division abstraction, altering a chain's frame-size will not effect the (average) rates of other chains in the system.

Consider the synthesis algorithm in Figure 6, and note that the F_i's (expressed in milliseconds) are initialized to the largest frame-sizes that could achieve the desired output rates. Here ΔF

Synthesize(): returns $\{(\mathbf{u_i}, \mathrm{F_i}, \mathrm{OR_i}) : 1 \le i \le n\}$

(1) $\mathrm{F_i} \leftarrow \lceil \frac{\Delta \mathrm{F}}{\mathrm{MOR_i}} \rceil$ (for all $1 \le i \le n$)

(2) $u_{i,j} \leftarrow \frac{E[t_{i,j}]}{\mathrm{F_i}}$ (for all $\tau_{i,j}$)

(3) $\rho_k \leftarrow \sum_{resource(\tau_{i,j})=k} u_{i,j}$ (for all $1 \le k \le m$)

(4) $\mathrm{S} \leftarrow \{\Gamma_i : 1 \le i \le n\}$

(5) **while** $(\mathrm{S} \neq \emptyset)$

$\{$

(6) Find the $\tau_{i,j}$ in $\Gamma_i \in S$ s.t. $r_k = resource(\tau_{i,j})$ which maximizes

(7) $w_{i,j} \leftarrow H(\mathrm{MOR_i}, \mathrm{OR_i}, \rho_k^m, \rho_k, u_{i,j})$

(8) **if**$(\rho_k \ge \rho_k^m - \delta)$

(9) **return** Failure

(10) $u_{i,j} \leftarrow u_{i,j} + \delta$

(11) $\rho_k \leftarrow \rho_k + \delta$

(12) $(\mathrm{OR_i}, \mathrm{F_i}) \leftarrow$ Get_Frame(Γ_i, $\mathrm{F_i}$, $\mathbf{u_i}$)

(13) **if** ($\mathrm{OR_i} \ge \mathrm{MOR_i}$)

(14) $\mathrm{S} \leftarrow \mathrm{S} - \{\Gamma_i\}$

$\}$

(15) **return**($\{(\mathbf{u_i}, \mathrm{F_i}, \mathrm{OR_i}) : 1 \le i \le n\}$)

Get_Frame$(\Gamma_i, \mathrm{F_i}, \mathbf{u_i})$: returns $(\mathrm{OR_i}, \mathrm{F_i})$

(1) $\mathrm{OR_i} \leftarrow$ *get_rate*$(\Gamma_i, \mathrm{F_i}, \mathbf{u_i})$

(2) **for** $(t \leftarrow \mathrm{F_i} - 1;\ t > 0\ ;\ t \leftarrow t - 1)$

$\{$

(3) **if** $((\mathrm{MD_i}$ **mod** $t) < (\alpha \times \mathrm{MD_i}))$

$\{$

(4) $\mathrm{OR_i'} \leftarrow$ get_rate(Γ_i, t, $\mathbf{u}_i$)

(5) **if** $(\mathrm{OR_i'} > \mathrm{OR_i})$

$\{$

(6) $\mathrm{OR_i} \leftarrow \mathrm{OR_i'}$

(7) $\mathrm{F_i} \leftarrow t$

$\}$

$\}$

$\}$

(8) **return** ($\mathrm{OR_i}$, $\mathrm{F_i}$)

Figure 6: Synthesis Algorithm

denotes global time-scale; in our example it was chosen as 1000, since all units were in milliseconds. Also note that for any task $\tau_{i,j}$, its resource share $u_{i,j}$ is initialized to accommodate the corresponding mean response-time time $E[t_{i,j}]$. (The system could only be solved with these initial parameters if all execution times were constant.)

Load Assignment. Load-assignment works by iteratively refining the load vectors (the $\mathbf{u_i}$'s), until a feasible solution is found. The entire algorithm terminates when the output rates for all chains meet their performance requirements – or when it discovers that no solution is possible. We do not employ backtracking, and task's load is never reduced. This means the solution space is not searched *totally*, and in some tightly-constrained systems, potential feasible solutions may not be found.

Load-assignment is task-based, i.e., it is driven by assigning additional load to the task estimated to need it the most. The heart of the algorithm can be found on lines (6)-(7), where all of the remaining unsolved chains are considered, with the objective of assigning additional load to the "most deserving" task in one of those chains. This selection is made using a heuristic weight $w_{i,j}$, reflecting the potential benefit of increasing $\tau_{i,j}$'s utilization, in the quest of increasing the chain's end-to-end performance.

The weight actually combines three factors, each of which plays a part in achieving feasibility: (1) additional output rate required, normalized via range-scaling to the interval [0,1]; (2) the current/maximum capacities for the task's resource (where current capacity is denoted as ρ_k for resource r_k); and (3) the task's current load assignment. The idea is that a high load assignment indicates diminishing returns are setting in, and working on a chain's other tasks would probably be more beneficial. For the results in this paper, the heuristic we used was:

$$H(\mathrm{MOR_i}, \mathrm{OR_i}, \rho_k^m, \rho_k, u_{i,j}) = \frac{\mathrm{MOR_i} - \mathrm{OR_i}}{\mathrm{MOR_i}} \times (\rho_k^m - \rho_k) \times \frac{1}{u_{i,j}}$$

Then the selected task gets its utilization increased by some tunable increment. Smaller increments will obviously lead to a greater likelihood of finding feasible solutions; however, they also incur a higher cost. (For the results presented in this paper, we set $\delta = .05$.)

After additional load is given to the selected task, the chain's new frame-size and rate parameters are determined; if it meets its minimum output requirements, it can removed from further consideration.

Frame Assignment. "Get_Frame" derives a feasible frame (if one exists). While the problem of frame-assignment seems straightforward enough, there are a few non-linearities to surmount: First, the *true, usable* load for a task is given by $\lfloor u_{i_j} \times \mathrm{F_i} \rfloor / \mathrm{F_i}$, due to the fact that the system cannot multiplex load at arbitrarily fine granularities of time. Second, in our analysis, we assume that the effective maximum delay is rounded up to the nearest frame, which errs on the side of conservatism.

The negative effect of the second factor is likely to be higher at larger frames, since it results in truncating the fractional part of a computation's final frame. On the other hand, the first factor becomes critical at smaller frames. Hence, the approximation utilizes a few simple rules. Since loads are monotonically increased, we restrict the search to frames which are lower than the current one. Further, we restrict the search to situations where our frame-based delay estimate truncates no more than $\alpha \times 100$ percent of the continuous-time deadline, where $\alpha << 1$. Subject to these guidelines, frames are evaluated via the throughput analysis presented in Section 4 – when determines the current $\mathrm{OR_i}$ metric.

Design Process - Solution of the Example. We ran the algorithm in Figure 6 to find a feasible solution, and the result is presented in Figure 7.

On a SPARC Ultra, the algorithm synthesized parameters for the example in approximately 30 minutes of wall-clock time. The results are presented in Figure 7. Note that the resources have capacity to spare; r_1 is has highest load in this configuration (at 0.72), r_7 has the lowest (at 0.35); most others are around 50% loaded. The spare load can be used to increase any chain's output rate, if desired – or held for other chains to be designed-in later.

A. Synthesized Solutions for Chains.

Γ_i	$\mathrm{F_i}$	$\mathrm{OR_i}$	$\mathbf{u_i}$
Γ_1	3	11.33	$[u_{1,1} = 0.33, u_{1,2} = 0.33, u_{1,3} = 0.33]$
Γ_2	16	5.50	$[u_{2,1} = 0.188, u_{2,2} = 0.188, u_{2,3} = 0.25, u_{2,4} = 0.188, u_{2,5} = 0.188, u_{2,6} = 0.25]$
Γ_3	5	5.26	$[u_{3,1} = 0.2, u_{3,2} = 0.2, u_{3,3} = 0.2, u_{3,4} = 0.2, u_{3,5} = 0.2, u_{3,6} = 0.2, u_{3,7} = 0.2, u_{3,8} = 0.2]$
Γ_4	20	5.47	$[u_{4,1} = 0.25, u_{4,2} = 0.2, u_{4,3} = 0.2, u_{4,4} = 0.15, u_{4,5} = 0.25]$
Γ_5	10	5.39	$[u_{5,1} = 0.10]$
Γ_6	5	6.91	$[u_{6,1} = 0.20, u_{6,2} = 0.20]$

B. Resource Capacity Used by System.

ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	ρ_6	ρ_7	ρ_8	ρ_9	ρ_{10}
0.72	0.39	0.45	0.4	0.65	0.52	0.35	0.62	0.65	0.65

Figure 7: Synthesized Solution of the Example System.

6 Simulation

Since the throughput analysis uses some key simplifying approximations, we validate the resulting solution via a simulation model.

Recall that the analysis injects imprecision for the following reasons. First, it tightens all (end-point) output delays by rounding up the fractional part of the final frame. Analogously, it assumes that a chain's state-changes always occur at its frame boundaries; hence, even intermediate output times are assumed to take place at the frame's end. A further approximation is inherent in our compositional data-age calculation – i.e., we assume the per-frame output ratios from predecessor tasks are i.i.d., allowing us to solve the resulting Markov chains in a quasi-independent fashion.

The simulation model discards these approximations, and keeps track all tagged computations through the chain, as well as the "true" states they induce in their participating tasks. Also, the clock progresses along the real-time domain; hence, if a task ends in the middle of a frame, it gets placed in the successor's input buffer at that time. Also, the simulation model schedules resources using a modified deadline-monotonic dispatcher (where a deadline is considered the end of a frame), so higher-priority tasks will get to run earlier than the analytical method assumes. Recall that the analysis implicitly assumes that computations may take place as late as possible, within a given frame.

On the other hand, the simulator does inherit some other simplifications used in our design. For example, input-reading is assumed to happen when a task gets released, i.e, at the start of a frame. As in the analysis, context switch overheads are not considered; rather, they are charged to the load distributions. Figure 8 summarizes the differences between the two models.

Validation of the Design. The following table compares the analytical throughput estimates with those derived via simulation. Simulated rates are displayed with 95% confidence-intervals

	Simulation	Analysis
Maximum delay	$\mathrm{MD_i}$	$\lfloor \mathrm{MD_i}/\mathrm{F_i} \rfloor \times \mathrm{F_i}$
State change	Measured	Frame boundary
Output Rate	Measured	i.i.d
Scheduling	Frame-based deadline monotonic	As late as possible
Data reading time	start of a frame	start of a frame

Figure 8: Comparison between Analysis model and Simulation model

over 100 trials, where each trial ran over 100,000 frames (for the largest frame in the system). The last column shows the standard deviation for output-rates calculated over t-time moving averages w.r.t. the simulated $\mathrm{OR_i}$, where $t \in \{0.5, 1.0\}$. This means that for 1-second intervals our tool does the following: (1) Γ_i's output rate is charted over all 1-second intervals; (2) each interval's deviation from $\mathrm{OR_i}$ is calculated; (3) the sum-of-squared deviations is obtained, and then divided by the degrees of freedom in the sample (4) the square-root of the result is produced.

Γ_i	$\mathrm{F_i}$ (ms)	$\mathrm{MOR_i}$ (ms)	$\mathrm{OR_i}$ (Analysis) (Hz)	$\mathrm{OR_i}$ (Simulated) (Hz)	$\sigma^t_{\mathrm{OR_i}}$ for moving averages over t: $t = 1$ sec.	$t = 0.5$ sec.
Y_1	3	10	11.33	11.44 ± 0.023	2.06	3.06
Y_2	16	5	5.50	5.47 ± 0.015	1.59	2.38
Y_3	5	5	5.26	5.35 ± 0.014	1.47	2.11
Y_4	20	5	5.47	5.76 ± 0.015	1.60	2.36
Y_5	10	5	5.39	5.39 ± 0.027	2.76	3.85
Y_6	5	5	6.91	6.90 ± 0.028	2.63	3.66

Note that the resulting (simulated) system satisfies minimum throughput requirements of all chains; hence we have a satisfactory solution. If desired, we can improve it by doling out the excess resource capacity, i.e., by simply iterating through the design algorithm again.

Figure 9 compares the simulated and analytic results for multiple frames, assigned to three selected chains. The frame-sizes are changed for one chain at a time, and system utilization remains fixed at the synthesized values. The simulation runs displayed here ran for $10,000$ frames, for the largest frame under observation (e.g, on the graph, if the largest frame tested is denoted as 200 ms, then that run lasted for 2000 seconds). The graphs show average output-rates for the chains over an entire simulation trial, along with the corresponding standard deviation, computed over all one-second interval samples vs. the mean.

The combinatorial comparison allows us to make the following observations. First, output rates generally increase as frame-sizes decrease – up to the point where the the system starts injecting a significant amount of truncation overhead (due to multiplexing). Recall that the runtime system cannot multiplex infinitesimal granularities of execution time; rather, a task's utilization is allocated in integral units over each frame.

Second, the relationship between throughput and burstiness is not direct. Note that for some chains (e.g., Γ_2), deviations tend to increase at both higher and lower frames. This reflects two separate facts, the first of which is fairly obvious, and is an artifact of our measurement process.

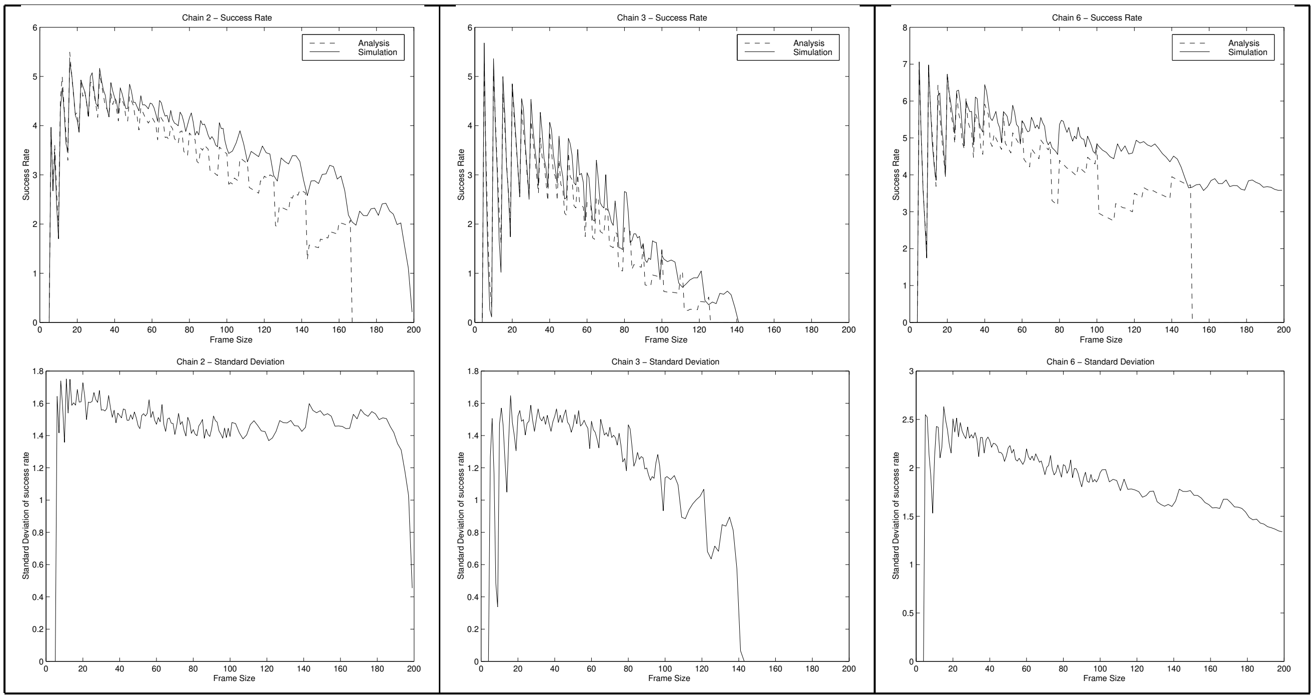


Figure 9: Analysis vs. simulation: $\mathrm{OR_i}$ for Γ_2,Γ_3,Γ_6; standard deviations for $\mathrm{OR_i}$ computed over 1 second moving averages.

We calculate variance statistics over 1-second intervals, yields deviations computed on the basis of a single reference time-scale. This method inserts some bias at low frames – since these processes get sampled more frequently. In turn, this can lead to artificially higher deviations.

At very large frames, another effect comes into play. Recall that tasks are only dispatched at frame boundaries, and only when they have input waiting from their predecessors. Hence, if a producer overruns its frame by even a slight amount, its consumer will have to wait for the next frame to use the data – which consequently adds to the data's age.

On the whole, however, we note that the simulated deviations are not particularly high, especially considering the fact that they include the one-second intervals where measured output rates actually *exceed* the mean.

Also, while output rates decrease as frame-sizes increase, the curves are not smooth, and small spikes can be observed in both the simulated and analytical results. Again, this effect is due to the multiplexing overhead discussed above, injected since execution-time budgets are integral. For example if a task's assigned utilization is 0.11, and if its frame-size is 10, then its execution-time budget will be 1 – resulting in an actual, *usable* utilization of 0.1. However, the output rates *do* monotonically decrease for larger frames when we only consider candidates that result in integral execution-time budgets.

Finally, we note that the difference between analysis and simulation is larger when a frame-size does not divide maximum permissible delay. Again, this is the result of rounding up the fractional part of a computation's final frame, which will cause us to overestimate the output's age.

Remarks. Coarse analytical estimations are essential at the synthesis stage. As we showed in Section 3, a deterministic real-time approach would fail to work for our small example, and the problems associated with stochastic timing deviations would only increase in larger systems. Note also that we could not rely exclusively on simulation during the synthesis phase, i.e., as a substitute for analysis. Based on our timing information for single-run simulations, such an approach would require over *three months* to synthesize our small example.

Hence, since we require coarse analytical estimates at the design stage, validating the solution is essential. As a first pass, the obvious choice is discrete-event simulation. A set of simulation runs – requiring perhaps several minutes – will often be significantly cheaper than going directly to integration. Indeed, discovering a severe design flaw after implementation can be a nasty proposition for a development team – particularly when the hardware is found to be insufficient for the application hosted on it. Also, the simulation model's underlying assumptions are fundamentally different from those used in the analysis; thus the results can provide a margin of confidence in the design's robustness. Simply put, for the sake of validity, two performance models are better than one.

However, simulation is not the end of the story. After all, our objective is to build the application, by calibrating the kernels and drivers to use our analytically-derived parameters. At that point, one subjects the system to the most important validity test of all: on-line profiling. Even with a careful synthesis strategy, testing usually leads to some additional system tuning – to help compensate for the imprecise modeling abstractions used during static design.

7 Conclusion

We presented a design scheme to achieve stochastic real-time performance guarantees, by adjusting system utilizations and processing rates. The solution strategy uses several approximations to avoid modeling the entire system; for example, in estimating end-to-end delay we use a combination of queuing analysis, real-time scheduling theory, and simple probability theory. Our search algorithm makes use of two heuristics, which help to significantly reduce the number of feasible solutions checked.

In spite of the approximations our simulation results are promising – they show that the approximated solutions are sufficiently close to be dependable; also, the resulting second-moment statistics show that output-rates are relatively smooth.

Much work remains to be done. First, we plan on extending the model to include a more rigid characterization of system overhead, due to varying degrees of multiplexing. Currently we assume that execution-time can be allocated in integral units; other than that, no specific penalty functions are included for context-switching, or for network-switch overhead. We also plan on getting better approximations for the "handover-time" between tasks in a chain, which will result in tighter analytical results.

We are also investigating new ways to achieve faster synthesis results. To speed up convergence, one needs a metric which approximates "direction of improvement" over the solution space as a whole. This would let the synthesis algorithm shoot over large numbers of incremental improvements – and hopefully attain a quicker solution. However, we note that the problem is not trivial; the solution space contains many non-linearities, with no ready-made global metric to unconditionally predict monotonic improvement. Yet, we can potentially take advantage of some relatively easy optimization strategies, such as hill-climbing and simulated annealing. The key to success lies in finding a reasonable "energy" or "attraction" function – and not necessarily one that is exact.

Finally, we are currently deploying our design technique in several large-scale field tests, on distributed applications hosted on SP-2 and Myrinet systems. Part of this project requires extending the scheme to handle dynamic system changes – where online arrivals and departures are permitted, and where the component-wise PDFs may vary over time. Hence, we are working on self-tuning mechanisms which get invoked when a chain's throughput degrades below a certain threshold – which can trigger an on-line adjustment in a chain's allocated load, as well as its associated frame size. Note that this is similar a common problem studied in the context of computer networks: handling on-the-fly QoS renegotiations, to help smooth out fluctuating service demands. Hence, we are investigating various strategies proposed for that problem, to determine if they can be modified for the more arcane (but equally challenging) domain of embedded real-time systems.

References

[1] Alan Burns. Preemptive Priority Based Scheduling: An Appropriate Engineering Approach. In Sang Son, editor, *Principles of Real-Time Systems*. Prentice Hall, 1994.

[2] Wu chun Feng and Jane W.-S. Liu. Algorithms for scheduling real-time tasks with input error and end-to-end deadlines. *IEEE Transactions on Software Engineering*, 23(2):93–106, February 1997.

[3] R.L. Cruz. A calculus for network delay, part i : Network elements in isloation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[4] Alan Demers. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM*, pages 1–12. ACM Press, September 1989.

[5] Norival R. Figueira and Joseph Pasquale. Leave-in-Time: A New Service Discipline for Real-Time Communications in a Packet-Switching Network. In *Proceedings of ACM SIGCOMM*, pages 207–218. ACM Press, October 1995.

[6] Lucia Franco. Communication configurator for fieldbus: An algorithm to schedule tran smission of data and messages. In *Proceedings of IFAC/IFIP Workshop on Real Time Programming*. IFIP, November 1996.

[7] Lucia Franco. Transmission scheduling for fieldbus: A strategy to schedule data and messages on fieldbus with end-to-end constraints. In *Proceedings of IEEE International Symposium on*

Intelligent Systems /Automation and Robotics (IAR). IEEE Computer Society Press, December 1996.

[8] R. Gerber, S. Hong, and M. Saksena. Guaranteeing Real-Time Requirements with Resource-Based Calibration of Periodic Processes. *IEEE Transactions on Software Engineering*, 21, July 1995.

[9] R. Gerber, Dong-In Kang, Seongsoo Hong, and Manas Saksena. *End-to-End Design of Real-Time Systems*, chapter 10, pages 237–265. Wiley, 1996. In *Formal Methods for Real-Time Computing*, edited by Constance Heitmeyer and Dino Mandrioli.

[10] Ladan Gharai and Richard Gerber. Multi-platform simulation of video playout performance. In *Proceedings of SPIE/IS&T Multimedia Computing and Networking (MCMN98)*, 1998.

[11] S. Goddard and Kevin Jeffay. Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application. In *Proceedings of IEEE Real-Time Technology and Applications Symposium.* IEEE Computer Society Press, June 1997.

[12] S. J. Golestani. Congestion-free communication in high-speed packet networks. *IEEE Transactions on Commmunication*, 39(12):1802–1812, 1991.

[13] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 107–121, October 1996.

[14] Pawan Goyal and Harrick M. Vin. Network Algorithms and Protocol for Multimedia Servers. In *Proceedings of IEEE INFOCOM.* IEEE Computer Society Press, March 1993.

[15] M. Harbour, M. Klein, and J. Lehoczky. Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority. In *Proceedings, IEEE Real-Time Systems Symposium*, pages 116–128, December 1991.

[16] Raj Jain. *The Art of Computer Systems Performance Analysis Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons, 1991.

[17] Ben Kao and Hector Garcia-Molina. Deadline Assignment in a Distributed Soft Real-Time System. In *Proceedings of International Conference on Distributed Computing systems*, pages 428–437. IEEE Computer Society Press, May 1993.

[18] Namyun Kim, Minsoo Ryu, Seongsoo Hong, Manas Saksena, Chong-Ho Choi, and Heonshik Shin. Visual assessment of a real-time system design : A case study on a cnc controller. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 300–310. IEEE Computer Society Press, December 1996.

[19] J. P. Lehoczky and S. Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 110–123. IEEE Computer Society Press, December 1992.

[20] J. Leung and M. Merill. A Note on the Preemptive Scheduling of Periodic, Real-Time Tasks. *Information Processing Letters*, 11(3):115–118, November 1980.

[21] C. Liu and J. Layland. Scheduling Algorithm for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[22] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*. IEEE Computer Society Press, May 1994.

[23] A. K. Parekh and G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks - The Single Node Case. In *Proceedings of IEEE INFOCOM*, pages 915–924. IEEE Computer Society Press, March 1992.

[24] A. K. Parekh and G. Gallager. A generalized processor sharing approach to flow control in integrated services networks - the multiple node case. In *Proceedings of IEEE INFOCOM*, pages 521–530. IEEE Computer Society Press, March 1993.

[25] M. Saksena and S. Hong. Resource Conscious Design of Real-Time Systems: An End-to-End Approach. In *IEEE International Conference of Engineering Complex Computer Systems*. IEEE Computer Society Press, October 1996.

[26] S. Sathaye and J. Strosnider. A Real-Time Scheduling Framework for Packet-Switched Networks. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 182–191. IEEE Computer Society Press, December 1994.

[27] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin. On Task Schedulability in Real-Time Control System. In *Proceedings of IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1996.

[28] Kang G. Shin and Yi-Chieh Chang. A Reservation-Based Algorithm for Scheduling Both Periodic and Aperiodic Real-Time Tasks. *IEEE Transactions on Computers*, 44:1405–1419, December 1995.

[29] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A Proportional Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 288–299. IEEE Computer Society Press, December 1996.

[30] T. S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W.-S Liu. Probabilistic Performance Guarantee for Real-Time Tasks with Varying Computation Times. In *Proceedings*

of IEEE Real-Time Technology and Applications Symposium, pages 164–173. IEEE Computer Society Press, May 1995.

[31] K. Tindell, A. Burns, and A. Wellings. Analysis of hard real-time communication. *The Journal of Real-Time Systems*, 9:147–171, September 1995.

[32] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Management. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI '94)*, November 1994.

[33] Carl A. WAldspurger and William E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.

[34] David Yates, James Kurose, Don Towsley, and Michael G. Hluchyj. On Per-session End-to-end Delay Distributions and the Call Admission Problem for Real-time Applications with QOS Requirements. In *Proceedings of ACM SIGCOMM*. ACM Press, September 1993.

[35] H. Zhang. Providing End-to-End Performance Guarantees Using Non-Work-Conserving Disciplines. *Computer Communications: Special Issue on System Support for Multimedia Computing*, 18, October 1995.

[36] Hui Zhang and D. Ferrari. Rate-controlled static-priority queueing. In *Proceedings of IEEE INFOCOM*, pages 227–236. IEEE Computer Society Press, September 1993.

[37] Hui Zhang and Edward W. Knightly. Providing End-to-End Statistical Performance Guarantees with Bounding Interval Dependent Stochastic Models. In *ACM SIGMETRICS*. ACM Press, May 1994.

[38] Lixia Zhang. VirtualClock : A New Traffic control Algorithm for Packet Switching Networks. In *Proceedings of ACM SIGCOMM*, pages 19–29. ACM Press, September 1990.

[39] Zhi-Li Zhang, Don Towsley, and Jim Kurose. Statistical Analysis of Generalized Processor Sharing Scheduling Discipline. In *Proceedings of ACM SIGCOMM*, pages 68–77. ACM Press, August 1994.